TOWARD LANGUAGE-INDEPENDENT PROGRAM VERIFICATION

BY

ANDREI ȘTEFĂNESCU

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Abstract

Recent years have seen a renewed interest in the area of deductive program verification, with focus on verifying real-world software components. Success stories include the verification of operating system kernels and of compilers.

This dissertation describes techniques for automatically building efficient correct-by-construction program verifiers for real-world languages from operational semantics. In particular, reachability logic is proposed as a foundation for achieving language-independent program verification. Reachability logic can express both operational semantics and program correctness properties, and has a sound and (relatively) complete proof systems that derives the program correctness properties from the operational semantics. These techniques have been implemented in the $\mathbb{K}$ verification infrastructure, which in turn yielded automatic program verifiers for C, JAVA, and JAVASCRIPT. These verifiers are evaluated by checking the full functional correctness of challenging heap manipulation programs implementing the same data-structures in these languages (e.g. AVL trees). This dissertation also describes the natural proof methodology for automated reasoning about heap properties.

To my family

Acknowledgments

I would like to thank my family for discovering and nurturing my passion for mathematics and computer science, and for encouraging me to pursue a career in this field and in particular this PhD. To my father Gheorghe for giving me math puzzles since I could barely speak, for his idealistic view of the world, and for being a role model I hope to emulate one day; to my mother Elena for training me when I was a school and I was participating in the mathematical and computer science olympiads, and for always encouraging me and believing in me; to my sister Cristina for always supporting me and being proud of me; and to everyone else for being so loving and caring, thank you!

My entire PhD research, including the results presented in this dissertation, would not have been possible without my advisor Grigore Roșu. I would like to especially thank him for providing countless research ideas, as well as for his help, guidance, and readiness in working together to overcome any research obstacle. In particular, his introduction of $\mathbb{K}$ and matching logic established the foundations on which our joint work on reachability logic presented in this dissertation builds.

I would also like to thank the rest of my thesis committee, made up of Madhusudan Parthasarathy, José Meseguer, Elsa Gunter, and Nikolaj Bjørner, for providing valuable feedback and insightful suggestions which strengthen this dissertation. In particular, I would like to thank Madhusudan Parthasarathy for his direct collaboration in developing the Dryad logic.

I would also like to thank all my colleagues and collaborators for all the discussions and ideas that we shared, and for all their advice and contributions. In particular, thanks are due to Michael Adams, Andrei Arusoaie, Kyungmin Bae, Denis Bogdănaș, Feng Chen, Andrew Cholewa, Ștefan Ciobâcă, Chucky Ellison, Yliès Falcone, Pranav Garg, Milos Gligoric, Dennis Griffith, Dwight Guth, Alex Gyori, Jeff Huang, Dongyun Jin, Mike Katelman, David Lazăr, Choonghwan Lee, Owolabi Legunsen, Liyi Li, Yilong Li, Pat Meredith, Radu Mereuță, Brandon Moore, Daejun Park, Edgar Pek, Lucas Peña, Andrei Popescu, Xiaokang Qiu, Camilo Rocha, Ralf Sasse, Manasvi Saxena, Stephen Skeirik, and Shijiao Yuwen. I would especially like to thank Cosmin Rădoi and Traian Șerbănuță, for all the time we worked together, and the many things they taught me.

Thanks are also due to my non-research related friends Alex, Andrei, Claudia, Codruța, Cristi, Mihnea, Thyago, and Yukiko, for being good friends.

The research in this dissertation has been supported in part by NSA contract H98230-10-C-0294, by NSF contract CCF-1218605, by Rockwell Collins contract 4504813093, by DARPA HACMS program as SRI subcontract 19-000222, and by Boeing contract on "Formal Analysis Tools for Cyber Security" 2016-2017.

Table of Contents

Chapter 1 Introduction 1

Chapter 2 Background 9
- 2.1 Operational Semantics 9
  - 2.1.1 Operational Semantics with Unconditional Rules 9
  - 2.1.2 Operational Semantics with Conditional Rules 12
- 2.2 $\mathbb{K}$ Semantics 14
- 2.3 Matching Logic 19
  - 2.3.1 Patterns and General Notions 19
  - 2.3.2 An Instance 22

Chapter 3 Reachability Logic 28
- 3.1 Specifying Reachability 28
- 3.2 Reachability Proof System 32
  - 3.2.1 Example Reachability Logic Proofs 35
- 3.3 Conditional One-Path Reachability Logic 39
  - 3.3.1 Conditional Reachability Rules 39
  - 3.3.2 Validity and Well-Definedness 43
  - 3.3.3 Proof System 44
  - 3.3.4 Nontermination and ω-Closure 47
- 3.4 Relationship with Hoare Logic 49
  - 3.4.1 Example Hoare Logic Proof 49
  - 3.4.2 From Hoare Logic Proofs to Reachability Logic Proofs . . 51
  - 3.4.3 Adding Recursion 60
- 3.5 Relationship with Separation Logic 62
- 3.6 Soundness 67
  - 3.6.1 All-Path and One-Path Reachability Logic 67
  - 3.6.2 Conditional One-Path Reachability Logic 72
- 3.7 Relative Completeness 78
  - 3.7.1 All-Path Reachability Logic 81
  - 3.7.2 Conditional One-Path Reachability Logic 89

Chapter 4 Implementation and Evaluation 119
4.1 MatchC 120
4.1.1 Examples using MatchC 120
4.1.2 Implementation and Evaluation 126
4.2 KVI 131
4.2.1 Motivating Example 131
4.2.2 Implementation 136
4.2.3 Evaluation 141

Chapter 5 Dryad 149
5.1 Motivating Example 154
5.2 The Logic Dryad 158
5.2.1 Syntax 158
5.2.2 Semantics 161
5.2.3 Examples 165
5.3 Translation to a Logic over the Global Heap 167
5.4 Natural Proofs for Dryad 171
5.4.1 Programs and Hoare-triples 171
5.4.2 Unfolding Across the Footprint 173
5.4.3 Formula Abstraction 174
5.5 Experimental Evaluation 175

Chapter 6 Related Work and Conclusion 181

References 186

Chapter 1

Introduction

Poor software quality can lead to financial loses and loss of life as demonstrated by numerous recent incidents. To achieve the high level of quality desired for critical software components, we need to formally verify these components. In particular, we need techniques that allow for easy construction of verification tools, and automatic reasoning about program correctness properties.

In this thesis, we present techniques for automatically building efficient correct-by-construction program verifiers from operational semantics. We have implemented these techniques in the $\mathbb{K}$ verification infrastructure (KVI), which in turn has yielded automatic program verifiers for C, Java, and JavaScript. We also present techniques for automated reasoning with a focus on proving data-structure properties. This work has resulted in the first automatic proofs of full functional correctness for a wide variety of data-structures (red-black trees, AVL trees, binomial heaps, B-trees, etc).

Building program verification tools for real-world languages is hard. Arguably the most popular theoretical foundation for program verification is axiomatic semantics, introduced by Floyd [36] and Hoare [45]. Intuitively, an axiomatic semantics defines a programming language as a proof system that derives Hoare triples of the form {*precondition*} `code` {*postcondition*}. The semantics of each program construct is given by one or more proof rules deriving such Hoare triples. For example, the proof rule below gives semantics to that `while` loop in a simple imperative language:

$$\frac{\mathcal{H} \vdash \{\psi \land \mathtt{e} \neq 0\}\, \mathtt{s}\, \{\psi\}}{\mathcal{H} \vdash \{\psi\}\, \mathtt{while(e)}\, \mathtt{s}\, \{\psi \land \mathtt{e} = 0\}}$$

Here we assume a C-like language, where zero means false and non-zero means true. This is the standard "loop invariant" proof rule associated with an axiomatic semantics. However, this rule assumes an idealized imperative language. In the case of real-world languages, concepts such as control flow, side effects, types, etc, complicate such a rule. A more realistic proof rule, from a separation logic [85] based axiomatic semantics for JavaScript [37], is shown here:

$$\frac{\begin{array}{l}\{P\}\,\mathtt{e1}\,\{S * \mathtt{r} = V_1\} \qquad S = R * \gamma(Ls, V_1, V_2) \\ \{S * \mathtt{True}(V_2)\}\,\mathtt{e2}\,\{P\} \\ Q = S * \mathtt{False}(V_2) * \mathtt{r} = \mathtt{undefined} \qquad \mathtt{r} \notin fv(R)\end{array}}{\{P\}\,\mathtt{while(e1)\{e2\}}\,\{Q\}}$$

This is one of the several proof rules for `while`, and is considerably more complex then the idealized one above. More importantly, we would argue that it is hard to correlate this rule with the English semantics of `while` as described in the official ECMAScript 5.1 standard. Moreover, such a proof rule cannot be turned easily into an interpreter, and thus it cannot be easily tested on existing benchmarks of programs. For this reason, an axiomatics semantics is not considered a very trusted model of the programming language it defines.

Operational semantics are an alternative style of formal semantics. They are easy to define and understand, similarly to implementing an interpreter. They require little formal training, scale up well, and, being executable, can be tested against existing implementations for faithfulness. Thus, operational semantics typically serve as the trusted models of programming languages, acting both as documentation and reference implementations for the defined languages.

Despite these advantages, they are rarely used directly for program verification, because proofs tend to be low-level and tedious, as they involve formalizing and working directly with the corresponding transition system. The state-of-the-art in mechanical program verification is to develop both an operational semantics (as a trusted model of the programming language) and axiomatic semantics (as a foundation for program verification) and to prove such axiomatic semantics sound with respect to the operational semantics [69, 47, 3]. Unfortunately, this needs to be done for each language separately and is very labor intensive. Such a semantics (either operational or axiomatic) typically consists of tens of thousands of lines of code and take several man-years to complete. Moreover, the soundness proofs need to be updated as the languages evolve.

For these reasons, many program verification tools forgo defining an operational semantics or an axiomatic semantics altogether, and instead they implement ad-hoc strongest-postcondition/weakest-precondition generation. For example, tools for C like VCC [22] and Frama-C [35], and for Java like jStar [28] take this approach.

Sometimes this is a two step process: first translate the high-level source code to a low-level intermediate verification language (IVL), and then perform the verification condition (VC) generation for the IVL. This leads to some re-usability: implementing a new program verifier for a language reduces to implementing a translator to the IVL, and then reusing the VC generation already implemented for the IVL. For example, VCC translates to Boogie [8] and Frama-C to Why3 [35].

However, defining correct language translations is not easy. Consider VCC. The translator consists of 5000 lines of F# [1] and has to be correct with respect to the 650 page ISO C11 Standard. There is the added difficulty that the translation cannot be easily tested. Due to limitations in the translation to Boogie, VCC both misses behaviors and verifies incorrect programs. Consider the following snippet:

```
1  unsigned x = UINT_MAX;
2  unsigned y = x + 1;
3  _(assert y == 0)
```

VCC fails to verify it, reporting an overflow in line 2. However, according to the C11 standard, the result of operations on unsigned integers does not overflow, it is reduced modulo `UINT_MAX` + 1, and thus the assertion in line 3 holds. Due to this bug in the translation to Boogie, VCC reports a false positive. Consider another snippet:

```
1  int foo(int *p, int x)
2  _(ensures *p == x)
3  _(writes p)
4  { return (*p = x); }
5
6  void main() {
7    int r;
8    foo(&r, 0) == foo(&r, 1);
9    _(assert r == 1)
10 }
```

According to the C11 Standard, this program is well-defined but non-deterministic: the arguments of == can evaluate in any order, so `r` could be either 0 or 1 on line 9. We have witnessed both behaviors by using different compilation options of the GCC compiler. However, VCC reports no error for the assertion on line 9. These issues are caused solely by limitations in the translation from C to Boogie.

The purpose of these examples is not to bash VCC, but to illustrate a less glamorous aspects of program verification, namely handling the semantics of real-world languages. VCC is a state-of-the-art program verifier, able to efficiently reason about very complex aspects of C programs, like threads, and has been used to verify software components like the Microsoft Hyper-V hypervisor [53]. In particular, it should have no problem handling the examples above. Moreover, what makes VCC an effective program verifier are its reasoning capabilities (support for modular reasoning about concurrency, axiomatizations of different mathematical domains, integration with SMT solvers, etc) and its conventions for writing the correctness properties, both of which are orthogonal to the tricky language features. Unfortunately, in general, with the current state-of-the-art, the only way to ensure the absence of such false positives and false negatives is to prove the underlying axiomatic semantics, or VC generation, or IVL translation sound with respect to a trusted reference model of the language, typically an operational semantics. This is a very tedious task for real-world languages.

In this work we propose to go back to the ideal approach of leveraging existing operation semantics for program verification. We build correct-by-construction program verifiers directly from operational semantics, without defining any other semantics or verification condition generator or translator. We view operational semantics as a mathematical models of programming languages which should exist independently of any program analysis. Thus, we aim to use the semantics unchanged, and do not count the effort of defining the operational semantics towards the total effort of building the program verifiers.

Our insight is that many of the tricky language-specific details (like type systems, scoping, implicit conversions, etc) are orthogonal to features that make program verification hard (reasoning about heap-allocated mutable data structures, integers/bit-vectors/floating-points, etc). As such, we propose a methodology to separate the two:

- define an operational semantics, and
- implement reasoning in a language-independent infrastructure.

Our approach has two advantages over the traditional approaches:

- it provides a way to obtain semantics-based verifiers without a need for multiple semantics, equivalence proofs, or translators, and
- it separates reasoning from language-specific operational details.

On the theoretical side We introduce reachability logic as a foundation for achieving language-independent program verification. Specifically, we introduce

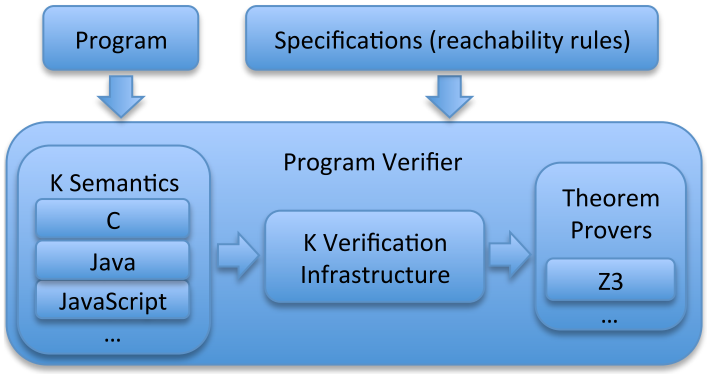

Figure 1.1: Architecture of Semantic-Based Verification

one-path reachability rules $\varphi \Rightarrow^{\exists} \varphi'$, which generalize operational semantics reduction rules, and all-path reachability rules $\varphi \Rightarrow^{\forall} \varphi'$, which generalize Hoare triples. A reachability rule is a pair of formulae capturing the partial correctness intuition: for every pair (code, state) γ satisfying φ, one path ($\exists$), respectively each path ($\forall$) derived using the operational semantics from γ either diverges or otherwise reaches a pair γ' satisfying φ'. The formulae φ and φ' are expressed using matching logic [94]. Intuitively, matching logic specifies structural properties of the program configuration by means of special predicates, namely configuration terms with variables, whose satisfaction is given by "matching".

Then, we give a language-independent proof system that derives new reachability rules (program properties) from a set of given reachability rules (the language operational semantics), at the same proof granularity and compositionality as a language-specific axiomatic semantics. The proof system consists of only 8 proof rules. We prove that the proof system is sound and relatively complete. In effect, the proof system subsumes all the language specific proof rules for loop invariants, recursive functions, etc from Hoare logic.

On the practical side We have implemented the $\mathbb{K}$ verification infrastructure (KVI) based on the language-independent proof system we propose here. We developed it as part of the open-source $\mathbb{K}$ semantic framework [89] (`http://kframework.org`). The framework takes an operational semantics defined in $\mathbb{K}$ as a parameter and uses it to automatically derive program correctness properties. In other words, the verification infrastructure *automatically* generates a program verifier from the semantics, which is *correct-by-construction* with respect to the semantics.

Figure 1.1 describes the architecture of our verification framework. Internally, the verifier uses the operational semantics to perform symbolic execution. Also, it has an internal matching logic prover for reasoning about implication between patterns (states) that generates queries to external theorem provers (for example, Z3 [26]). The most complex part of matching logic prover is handling heap abstractions. The program correctness properties are given as reachability rules between matching logic patterns.

A major difficulty in a language-independent setting is that standard language features relevant to verification, like control flow or memory access, are not explicit, but rather implicit (defined through the semantics). Thus, we adapt existing techniques to a language-independent setting. In particular, we notice that symbolic execution is captured by narrowing (instead of rewriting which captures concrete execution). For reasoning about heap-manipulating data-structures, we adapted our own work on natural proofs. The generated program verifiers are fully automated. The user only provides the program correctness specifications. KVI is implemented in JAVA. It consists of approximately 30,000 non-blank lines of code, and it took about 3 man-years to complete.

Prior, we have build MATCHC, a preliminary program verifier based on the language-independent proof system, which is a prototype hand-crafted for KERNELC, a toy language. MATCHC mixes the language-independent reasoning with the operational semantics of KERNELC, e.g., it hardcodes when to perform CASE ANALYSIS (for constructs like `if`), and when to perform heap abstractions folding/unfolding.

To ascertain the practicality of our approach, we have instantiated the KVI with the operational semantics of C [30, 43], JAVA [14], and JAVASCRIPT [75] (all developed independently from this project), thus obtaining program verifiers for these complex real-world languages. Then, we have evaluated these verifiers by checking the full functional correctness of challenging heap manipulation programs implementing the same data-structures in these languages (e.g. AVL trees). These programs have been used before to evaluate verification approaches. The verification time is competitive with other state-of-the-art verifiers. The time is dominated by symbolic execution, which reflects the complexity of the operational semantics and the languages themselves. Reasoning about the mathematical properties of the data-structures is similar in all three languages. Regarding the number of user annotations, our approach is comparable to the state-of-the-art language-specific approaches that do not infer invariants. Thus, our approach is effective both in terms of verification capabilities and user effort. Our works has resulted in the first time that verifiers for C, Java, and JavaScript are sharing the same core infrastructure. We believe these experiments validate our hypothesis that separating the tricky language details from the main verification process is practical.

Next, we turn our attention to the question of automated reasoning about state properties, especially in the case of heaps. Entirely decidable logics are too restrictive to support the verification of the complex specifications of heap manipulating programs implementing data-structures. On the other hand, logics requiring manual/semi-automatic reasoning put too much burden on the user (in the form of proof tactics and lemmas).

To address these limitations, we have developed the natural proofs approach, which combines the two methodologies above. It

- identifies a class of simple proofs for verifying heap manipulating programs, founded on how people prove these conditions manually, and
- builds terminating procedures that efficiently and thoroughly search this class of proofs.

This results in a sound, incomplete, but terminating procedure that finds natural proofs automatically and efficiently.

Specifically, the program specifications are expressed using recursively defined predicates/functions. During the symbolic execution of the code, the recursive definitions are unfolded precisely across the memory footprint (the memory locations accessed by the code). Thus, the verification reduces to checking the satisfiability of a quantifier-free formula depending only on the values of predicates/functions on the frontier of the footprint. The recursive definitions are abstracted as uninterpreted functions and the resultant formula is sent to an automatic logic solver.

We have evaluated our approach by verifying the full functional correctness of data-structures ranging from sorted linked lists, binary search trees, max-heaps, treaps, AVL trees, red-black trees, B-trees, and binomial heaps. These benchmarks are an almost exhaustive list of algorithms on tree-based data-structures covered in a first undergraduate course on data-structures.

Contributions. This thesis makes the following contributions:

- Reachability logic as a unifying formalism for both operational semantics and axiomatic semantics. Its sentences consist of the reachability rules with one-path semantics and with all-path semantics.

- A language-independent proof system for reachability logic that can derive program correctness properties (specified as reachability rules) directly from the operational semantics of a programming language (also specified as reachability rules). The proof system comes with proofs of soundness and relative completeness.

- KVI, a language-independent verification infrastructure, which can be instantiated with a $\mathbb{K}$ semantics to obtain a program verifier for the respective language.

- Program verifiers for C, JAVA, and JAVASCRIPT generated from their existing $\mathbb{K}$ semantics, and an evaluation of the development cost of building these verifiers from the operational semantics.

- Empirical evaluation of these verifiers on challenging heap manipulation programs implementing data-structure.

- The natural proofs methodology for automated reasoning about heap properties.

- Empirical evaluation of the natural proofs methodology on a large number of algorithms on tree-based data-structures.

Chapter 2

Background

In this chapter we recall basic notions of operational semantics, and then we focus more on the $\mathbb{K}$ framework. The theoretical results presented in this thesis work with any operational semantics style, while the implementation works with $\mathbb{K}$ semantics. Then we present matching logic, as the foundation of our language-independent verification approach.

2.1 Operational Semantics

In this section we give a brief overview of operational semantics, using a simple imperative language IMP as a running example. We first cover operational semantics styles that use unconditional reduction rules, and then we move on to styles that use conditional reduction rules.

2.1.1 Operational Semantics with Unconditional Rules

Here we recall basic notions of operational semantics, reduction rules, and transition systems, and introduce our notation and terminology for these. We do so by means of a simple parallel imperative language, IMP. Figure 2.1 shows its syntax and an operational semantics based on evaluation contexts. IMP has only integer expressions. When used as conditions of `if` and `while`, zero means false and any non-zero integer means true (like in C). Expressions are formed with integer constants, program variables, and conventional arithmetic constructs. For simplicity, we only assume a generic binary operation, `op`. IMP statements are assignment, `if`, `while`, sequential composition and parallel composition. IMP has shared memory parallelism without explicit synchronization.

Various operational semantics styles define programming languages (or calculi, or systems, etc.) as sets of rewrite or reduction rules of the form "$l \Rightarrow r$ if b", where l and r are program configurations with variables constrained by the boolean

IMP language syntax

$$
\begin{array}{rcl}
\mathit{PVar} & ::= & \text{program variables} \\
\mathit{Exp} & ::= & \mathit{PVar} \mid \mathit{Int} \mid \mathit{Exp}\ \texttt{op}\ \mathit{Exp} \\
 & \mid & \texttt{--}\mathit{PVar} \\
\mathit{Stmt} & ::= & \texttt{skip} \mid \mathit{PVar}\ \texttt{:=}\ \mathit{Exp} \\
 & \mid & \mathit{Stmt}\ \texttt{;}\ \mathit{Stmt} \mid \mathit{Stmt} \parallel \mathit{Stmt} \\
 & \mid & \texttt{if(}\mathit{Exp}\texttt{)}\ \mathit{Stmt}\ \texttt{else}\ \mathit{Stmt} \\
 & \mid & \texttt{while(}\mathit{Exp}\texttt{)}\ \mathit{Stmt}
\end{array}
$$

IMP evaluation contexts syntax

$$
\begin{array}{rcl}
\mathit{Context} & ::= & \blacksquare \\
 & \mid & \langle \mathit{Context},\ \mathit{State} \rangle \\
 & \mid & \mathit{Context}\ \texttt{op}\ \mathit{Exp} \mid \mathit{Int}\ \texttt{op}\ \mathit{Context} \\
 & \mid & \mathit{PVar}\ \texttt{:=}\ \mathit{Context} \mid \mathit{Context}\texttt{;}\ \mathit{Stmt} \\
 & \mid & \mathit{Context} \parallel \mathit{Stmt} \ \mid\ \mathit{Stmt} \parallel \mathit{Context} \\
 & \mid & \texttt{if(}\mathit{Context}\texttt{)}\ \mathit{Stmt}\ \texttt{else}\ \mathit{Stmt}
\end{array}
$$

IMP operational semantics

$$
\begin{array}{ll}
\textbf{lookup} & \langle C,\ \sigma\rangle[X] \Rightarrow \langle C,\ \sigma\rangle[\sigma(X)] \\
\textbf{op} & I_1\ \texttt{op}\ I_2 \Rightarrow I_1\ \mathit{op}_{\mathit{Int}}\ I_2 \\
\textbf{dec} & \langle C,\ \sigma\rangle[\texttt{--}X] \Rightarrow \langle C,\ \sigma[X \leftarrow (\sigma(X) -_{\mathit{Int}} 1)]\rangle[\sigma(X) -_{\mathit{Int}} 1] \\
\textbf{asgn} & \langle C,\ \sigma\rangle[X\ \texttt{:=}\ I] \Rightarrow \langle C,\ \sigma[X \leftarrow I]\rangle[\texttt{skip}] \\
\textbf{seq} & \texttt{skip}\ \texttt{;}\ S \Rightarrow S \\
\textbf{cond}_1 & \texttt{if(}I\texttt{)}\ S_1\ \texttt{else}\ S_2 \Rightarrow S_1 \qquad \text{if } I \neq 0 \\
\textbf{cond}_2 & \texttt{if(0)}\ S_1\ \texttt{else}\ S_2 \Rightarrow S_2 \\
\textbf{while} & \texttt{while(}E\texttt{)}\ S \Rightarrow \texttt{if(}E\texttt{)}\ S\texttt{;}\ \texttt{while(}E\texttt{)}\ S\ \texttt{else}\ \texttt{skip} \\
\textbf{finish} & \texttt{skip} \parallel \texttt{skip} \Rightarrow \texttt{skip}
\end{array}
$$

Figure 2.1: IMP language syntax and operational semantics based on evaluation contexts.

condition b. One of the most popular such operational approaches is reduction semantics with evaluation contexts [32], with rules "$C[t] \Rightarrow C'[t']$ if b", where C is the evaluation context that reduces to C' (typically $C = C'$), t is the redex that reduces to t', and b is a side condition. Another approach is the chemical abstract machine [12], where l is a chemical solution that reacts into r under condition b. The $\mathbb{K}$ framework [89] is another, based on plain (no evaluation contexts) rewrite rules. Several large languages have been given such semantics, including C [30].

Here we chose to define IMP using reduction semantics with evaluation contexts. Note, however, that our subsequent results work with any of the aforementioned operational approaches. The program configurations of IMP are pairs

$\langle \texttt{code},\ \sigma\rangle$, where code is a program fragment and σ is a state term mapping program variables into integers. As usual, we assume appropriate definitions for the integer and map domains available, together with associated operations like arithmetic operations ($i_1\ op_{Int}\ i_2$, etc.) on the integers and lookup ($\sigma(x)$) or update ($\sigma[x \leftarrow i]$) on the maps. We also assume a context domain that can both decompose a configuration into a context and a redex ("$C[t]$"), and compose it back. A configuration context consists of a code context and a state.

The IMP definition in Figure 2.1 consists of nine reduction rules between program configurations, which make use of first-order variables: X is a variable of sort *PVar*; E is a variable of sort *Exp*; S, S_1, S_2 are variables of sort *Stmt*; I, I_1, I_2 are variables of sort *Int*; σ is a variable of sort *State*; C is a variable of sort *Context*. A rule reduces a configuration by splitting it into a context and a redex, rewriting the redex and possibly the context, and then plugging the resulting term into the resulting context. As an abbreviation, a context is not mentioned if not used; e.g., the rule **op** is in full $\langle C,\ \sigma\rangle[I_1 \texttt{ op } I_2] \Rightarrow \langle C,\ \sigma\rangle[I_1\ op_{Int}\ I_2]$. For example, configuration $\langle \texttt{x := (2 + 5) - 4},\ \sigma'\rangle$ reduces to $\langle \texttt{x := 7 - 4},\ \sigma\rangle$ by applying the $\mathbf{op}_+$ rule with $C \equiv \texttt{x := } \blacksquare \texttt{ - 4}$, $\sigma \equiv \sigma'$, $I_1 \equiv 2$ and $I_2 \equiv 5$. We can therefore regard the operational semantics of IMP above as a set of reduction rules of the form "$l \Rightarrow r$ if b", where l and r are configurations with variables constrained by the boolean condition b. The subsequent results presented in this thesis work with such sets of reduction rules and are agnostic to the particular underlying operational semantics style.

Let $\mathcal{S}$ (for "semantics") be a set of reduction rules like above, and let Σ be the underlying signature; also, let *Cfg* be a distinguished sort of Σ (for "configurations"). $\mathcal{S}$ yields a transition system on any Σ-algebra/model $\mathcal{T}$, no matter whether $\mathcal{T}$ is a term model or not. Let us fix an arbitrary model $\mathcal{T}$, which we may call a *configuration model*; as usual, $\mathcal{T}_{Cfg}$ denotes the elements of $\mathcal{T}$ of sort *Cfg*, which we call *configurations*:

Definition 1. *$\mathcal{S}$ induces a* ***transition system*** *$(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$ as follows: $\gamma \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma'$ for any $\gamma, \gamma' \in \mathcal{T}_{Cfg}$ iff there is some rule "$l \Rightarrow r$ if b" in $\mathcal{S}$ and some $\rho : Var \to \mathcal{T}$ such that $\rho(l) = \gamma$, $\rho(r) = \gamma'$ and $\rho(b)$ holds (Var is the set of variables appearing in rules in $\mathcal{S}$ and we used the same ρ for its homomorphic extension to terms l, r and predicates b).*

$(\mathcal{T}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$ is a conventional transition system, i.e., a set together with a binary relation on it (in fact, $\Rightarrow_{\mathcal{S}}^{\mathcal{T}} \subseteq \mathcal{T}_{Cfg} \times \mathcal{T}_{Cfg}$), and captures precisely how the language defined by $\mathcal{S}$ operates. We use it in Section 3.2 to define and prove the soundness of our proof system.

2.1.2 Operational Semantics with Conditional Rules

A common denominator of typical syntaxes for big-step, small-step, and reduction semantics is *conditional term rewriting*. The general form of a rewrite rule is

$$p \Rightarrow p' \text{ if } b \wedge p_1 \Rightarrow p'_1 \wedge b_1 \wedge \ldots \wedge p_n \Rightarrow p'_n \wedge b_n$$

where each p_i is a pattern and each b_i is a boolean expression over variables bound in earlier patterns. A set of rules generates a transition system which includes a step between a pair of configurations if there is some rule and some environment mapping variables to subterms so that the first term matches p, the second term matches p', all of the b_i are true, and for each $1 \leq i \leq n$ the term obtained by instantiating p_i can take zero or more steps in the transition system to reach p'_i. A rule is called unconditional if $n = 0$, whether or not b is trivial. For example,

$$\langle \texttt{if}\,\texttt{(}E\texttt{)}\ S_1\ \texttt{then}\ S_2,\ \sigma\rangle \Rightarrow \langle\sigma'\rangle \text{ if } \langle E,\ \sigma\rangle \Rightarrow \langle I\rangle \wedge I \neq 0 \wedge \langle S_1,\ \sigma\rangle \Rightarrow \langle\sigma'\rangle$$

handles the semantics of the positive case of `if` in a big-step style. Here configurations are pairs $\langle \texttt{code},\ \sigma\rangle$ of a statement or expression `code` to evaluate, and a state/store σ, as well as singleton stores $\langle\sigma\rangle$ or integers $\langle i\rangle$ for the result of executing statements and expressions, respectively. This rule says that an `if` statement executed under store σ transitions in one step to the result σ', provided the expression S evaluates in zero or more steps to an integer result I, the result is nonzero, and σ' is the result of executing the statement from the then-branch of the `if` statement. In this particular rule, $n = 2$ and b and b_2 are *true*, so not written.

Conditional term rewriting is sufficient to express every style of operational semantics, perhaps through a translation adding some auxiliary configurations and rules (e.g., to capture the one-step reduction $\Rightarrow^1$). This is covered in detail in [97] for small-step and big-step semantics, reduction semantics [105], the chemical abstract machine [12], and continuation-based semantics [31]. The representations are strongly faithful in the sense that two configurations are related by a single step in the original system iff appropriate injections into the domain of the term rewriting system are related by a single step.

Figure 2.2 shows a small-step and a big-step semantics of a simple imperative

IMP language syntax

PVar	::=	program variables
Exp	::=	$\mathit{PVar} \mid \mathit{Int} \mid \mathit{Exp}\ \texttt{op}\ \mathit{Exp}$
Stmt	::=	$\mathit{skip} \mid \mathit{PVar}\ \texttt{:=}\ \mathit{Exp} \mid \mathit{Stmt}\ \texttt{;}\ \mathit{Stmt}$
	|	$\texttt{if(}\mathit{Exp}\texttt{)}\ \mathit{Stmt}\ \texttt{else}\ \mathit{Stmt} \mid \texttt{while(}\mathit{Exp}\texttt{)}\ \mathit{Stmt}$

IMP small-step semantics

$\mathbf{op}_1$	$\langle E_1\ \texttt{op}\ E_2,\ \sigma\rangle \Rightarrow^1 \langle E_1'\ \texttt{op}\ E_2,\ \sigma\rangle$ if $\langle E_1,\ \sigma\rangle \Rightarrow^1 \langle E_1',\ \sigma\rangle$
$\mathbf{op}_2$	$\langle E_1\ \texttt{op}\ E_2,\ \sigma\rangle \Rightarrow^1 \langle E_1\ \texttt{op}\ E_2',\ \sigma\rangle$ if $\langle E_2,\ \sigma\rangle \Rightarrow^1 \langle E_2',\ \sigma\rangle$
$\mathbf{op}_3$	$\langle I_1\ \texttt{op}\ I_2,\ \sigma\rangle \Rightarrow^1 \langle I_1\ \texttt{op}_{\mathit{Int}}\ I_2,\ \sigma\rangle$
lookup	$\langle X,\ \sigma\rangle \Rightarrow^1 \langle \sigma(X),\ \sigma\rangle$
$\mathbf{asgn}_1$	$\langle X\ \texttt{:=}\ E,\ \sigma\rangle \Rightarrow^1 \langle X\ \texttt{:=}\ E',\ \sigma\rangle$ if $\langle e,\ \sigma\rangle \Rightarrow^1 \langle e',\ \sigma\rangle$
$\mathbf{asgn}_2$	$\langle X\ \texttt{:=}\ I,\ \sigma\rangle \Rightarrow^1 \langle \texttt{skip},\ \sigma[X \leftarrow I]\rangle$
$\mathbf{seq}_1$	$\langle S_1\ \texttt{;}\ S_2,\ \sigma\rangle \Rightarrow^1 \langle S_1'\ \texttt{;}\ S_2,\ \sigma'\rangle$ if $\langle S_1,\ \sigma\rangle \Rightarrow^1 \langle S_1',\ \sigma'\rangle$
$\mathbf{seq}_2$	$\langle \texttt{skip ;}\ S_2,\ \sigma\rangle \Rightarrow^1 \langle S_2,\ \sigma\rangle$
$\mathbf{cond}_1$	$\langle \texttt{if(}E\texttt{)}\ S_1\ \texttt{else}\ S_2,\ \sigma\rangle \Rightarrow^1 \langle \texttt{if(}E'\texttt{)}\ S_1\ \texttt{else}\ S_2,\ \sigma\rangle$ if $\langle E,\ \sigma\rangle \Rightarrow^1 \langle E',\ \sigma\rangle$
$\mathbf{cond}_2$	$\langle \texttt{if(}I\texttt{)}\ S_1\ \texttt{else}\ S_2,\ \sigma\rangle \Rightarrow^1 \langle S_1,\ \sigma\rangle$ if $I \neq 0$
$\mathbf{cond}_3$	$\langle \texttt{if(0)}\ S_1\ \texttt{else}\ S_2,\ \sigma\rangle \Rightarrow^1 \langle S_2,\ \sigma\rangle$
while	$\texttt{while(}E\texttt{)}\ S \Rightarrow^1 \texttt{if(}E\texttt{)}\ S\texttt{;}\ \texttt{while(}E\texttt{)}\ S\ \texttt{else skip}$

IMP big-step semantics

op	$\langle E_1\ \texttt{op}\ E_2,\ \sigma\rangle \Rightarrow \langle I_1 \texttt{op}_{\mathit{Int}} I_2\rangle$ if $\langle E_1,\ \sigma\rangle \Rightarrow \langle I_1\rangle, \langle E_2,\ \sigma\rangle \Rightarrow \langle I_2\rangle$
int	$\langle I,\ \sigma\rangle \Rightarrow \langle I\rangle$
lookup	$\langle X,\ \sigma\rangle \Rightarrow \langle \sigma(X)\rangle$
skip	$\langle \texttt{skip},\ \sigma\rangle \Rightarrow \langle \sigma\rangle$
asgn	$\langle X\ \texttt{:=}\ E,\ \sigma\rangle \Rightarrow \langle \sigma[X \leftarrow I]\rangle$ if $\langle E,\ \sigma\rangle \Rightarrow \langle I\rangle$
seq	$\langle S_1\ \texttt{;}\ S_2,\ \sigma\rangle \Rightarrow \langle \sigma_2\rangle$ if $\langle S_1,\ \sigma\rangle \Rightarrow \langle \sigma_1\rangle\ ,\ \langle S_2,\ \sigma_1\rangle \Rightarrow \langle \sigma_2\rangle$
$\mathbf{cond}_1$	$\langle \texttt{if(}E\texttt{)}\ S_1\ \texttt{else}\ S_2,\ \sigma\rangle \Rightarrow \langle \sigma'\rangle$ if $\langle E,\ \sigma\rangle \Rightarrow \langle I\rangle, I \neq 0, \langle S_1,\ \sigma\rangle \Rightarrow \langle \sigma'\rangle$
$\mathbf{cond}_2$	$\langle \texttt{if(}E\texttt{)}\ S_1\ \texttt{else}\ S_2,\ \sigma\rangle \Rightarrow \langle \sigma'\rangle$ if $\langle E,\ \sigma\rangle \Rightarrow \langle 0\rangle\ ,\ \langle S_2,\ \sigma\rangle \Rightarrow \langle \sigma'\rangle$
$\mathbf{while}_1$	$\langle \texttt{while(}E\texttt{)}\ S,\ \sigma\rangle \Rightarrow \langle \sigma\rangle$ if $\langle E,\ \sigma\rangle \Rightarrow \langle 0\rangle$
$\mathbf{while}_2$	$\langle \texttt{while(}E\texttt{)}\ S,\ \sigma\rangle \Rightarrow \langle \sigma'\rangle$ if $\langle E,\ \sigma\rangle \Rightarrow \langle I\rangle\ ,\ I \neq 0\ ,\ \langle S\ \texttt{;}\ \texttt{while(}E\texttt{)}\ S,\ \sigma\rangle \Rightarrow \langle \sigma'\rangle$

Figure 2.2: The IMP language: syntax, a small-step and a big-step operational semantics.

language, called IMP, using rewrite rules. The operational semantics contain rewrite rules making use of ordinary first-order variables: $E, E', E_1, E_1', E_2, E_2'$ are variables of sort *Exp*; σ, σ' are variables of sort *State*; I, I_1, I_2 are variables of sort *Int*; X is a variable of sort *PVar*; S, S_1, S_1', S_2 are variables of sort *Stmt*. The underlying mathematical domain is assumed to provide all the needed operations, for example $+_{Int}$, $*_{Int}$, $<_{Int}$, etc., for integers, and $\sigma(x)$, $\sigma[x \leftarrow i]$, $x \in Dom(\sigma)$, etc., for maps. Note that IMP here does not include the decrement $--X$ and parallel composition || features of IMP in the previous section, in part to keep the definitions simple and in part because our results for semantics with conditional rules in Section 3.3 do not support non-determinism.

In what follows, we will refer to three IMP programs:

```
SUM   ≡  s := 0; while (n > 0) (s := s+n; n := n-1)
SUM′  ≡  s := 0; while (n > 0)  s := s+n
SUM∞  ≡  n := 1; while (n > 0)  s := s+n
```

`SUM` always terminates, `SUM′` only terminates when `n` ≤ 0, and $\texttt{SUM}_\infty$ never terminates. Nontermination occurs in small-step semantics by infinite "horizontal" computation: each rule application terminates, but there are infinitely many rule applications, and in big-step by infinite "vertical" computation: a rule application does not terminate as it requires another rule application to solve one of its premises, which requires another rule application to solve one of its premises, and so on.

In conclusion, rewrite rules can be used to formally and uniformly define operational semantics.

2.2 $\mathbb{K}$ Semantics

We illustrate how to write $\mathbb{K}$ definitions by means of defining IMP++, a simple concurrent imperative language (also part of the $\mathbb{K}$ distribution and tutorial). Figure 2.3 shows the IMP++ syntax and Figure 2.4 the semantics. $\mathbb{K}$ definitions are structured using modules, which can contain **imports** statements to include other modules (line 31), **syntax** definitions (e.g., lines 2–27), **configuration** declarations (lines 33–44), and $\mathbb{K}$ rules.

Syntax definitions describe a CFG in a BNF-style extended with priorities and associativity filters used for disambiguation. Terminals are enclosed in quotes, > separates priority levels, and | separates productions with the same priority. Productions can have comma-separated tags; e.g., **bracket** on line 14 says that parentheses are to be used only for grouping purposes, and **left** on line 6 that addition is left associative. $\mathbb{K}$ generates parsers from syntax definitions, which

```
 1  module IMP−SYNTAX
 2    syntax AExp ::=  Int  |  String  |  Id
 3                  |  "++" Id
 4                  |  "read"  "("  ")"
 5                  >  AExp "/"  AExp                    [left , strict ]
 6                  >  AExp "+" AExp                     [left , strict ]
 7                  >  "spawn" Block
 8                  >  Id  "=" AExp                         [strict (2)]
 9                  |  "("  AExp ")"                          [bracket]
10    syntax BExp ::=  Bool
11                  |  AExp "≤" AExp                          [strict ]
12                  |  "!"  BExp                              [strict ]
13                  >  BExp "&&" BExp                [left , strict (1)]
14                  |  "("  BExp ")"                          [bracket]
15    syntax Block ::=  "{"  Stmts  "}"
16    syntax Stmt  ::=  Block
17                  |  AExp ";"                               [strict ]
18                  |  "if "  "("  BExp ")" Block "else "  Block  [strict (1)]
19                  |  "while"  "("  BExp ")" Block
20                  |  "int "  Ids  ";"
21                  |  "print "  "("  AExps ")"  ";"          [strict ]
22                  |  "halt "  ";"
23                  >  "join "  AExp ";"                      [strict ]
24
25    syntax Ids     ::=  List {Id,","}                       [strict ]
26    syntax AExps ::=  List {AExp,","}                       [strict ]
27    syntax Stmts ::=  List {Stmt,""}
28  endmodule
```

Figure 2.3: IMP++ language syntax

are used to parse both programs and rules. IMP++ has arithmetic (AExp) and boolean (BExp) expressions, statements (Stmt), and blocks (Block). The builtin List construct (lines 25–27) specifies lists of elements of certain types (here Id, AExp, or Stmt) separated by a certain terminal (comma, for lines 25 and 26) or just whitespace (the empty terminal on line 27).

Computations extend syntax with a task sequentialization operation, "$\curvearrowright$" (having unit $\cdot$). A task can be either a fragment of syntax to be processed or a semantic task, such as the recovery of an environment. Most of the manipulation of the computation is abstracted away from the language designer via intuitive syntax annotations.

```
module IMP
  imports IMP−SYNTAX
  syntax KResult ::=  Int  |  Bool |  String
  configuration  <T color="yellow">
                  <threads  color ="orange">
                    <thread  multiplicity   ="∗" color ="blue">
                      <k color ="green"> $PGM:Stmts </k>
                      <env color ="LightSkyBlue"> .Map </env>
                      <id color ="black"> 0  </id>
                    </thread>
                  </threads>
                  <store  color ="red"> . Map </store>
                  <in color ="magenta" stream="stdin">.List</in>
                  <out color ="Orchid" stream="stdout">.List</out>
                </T>
  rule  <k> X:Id => I ...</k> <env>... X ↦N ...</env> <store>... N ↦I ...</store>
  rule  <k> ++X => I +Int 1 ...</k>
        <env>... X ↦N ...</env> <store>... N ↦(I => I +Int 1) ...</store>
  rule  <k> read()  => I ...</k> <in> ListItem(I: Int )  => .  ...</in>
  rule  I1: Int  /  I2: Int   => I1 /Int   I2  when I2 =/=Int  0
  rule  I1: Int   + I2: Int   => I1  +Int  I2
  rule  Str1 : String   + Str2 : String   => Str1  +String  Str2
  rule  I1: Int   ≤ I2: Int   => I1  ≤Int  I2
  rule  !  T:Bool => notBool T
  rule  true   && B => B
  rule  false   && _ => false
  rule  <k> {Ss} => Ss ↷Rho ...</k> <env> Rho </env>
  rule  <k> Rho => . ...</k> <env> _ => Rho </env>
  rule  _: Int ;  => .
  rule  <k> X = I:Int   => I  ...</k>
        <env>... X ↦N ...</env> <store>... N ↦(_ => I) ...</store>
  rule  if ( true )    S  else   _  => S
  rule  if ( false  )  _  else   S  => S
  rule  while( B)  S  => if( B)  {S while( B)  S}  else   {}
  rule  <k> int  ( X:Id, Xs => Xs); ...</k>
        <env> Rho => Rho[!N/X] </env> <store>... . => N ↦0 ...</store>
  rule  int   . Ids ;  => .
  syntax Printable   ::=  Int  |  String
  syntax AExp ::= Printable
  rule  <k> print ( P:Printable  , AEs => AEs); ...</k> <out>... . => ListItem(P)  </out>
  rule  print  (. AExps); => .
  rule  <k> halt;  ↷_ => .  </k>
  rule  <k> spawn S => T ...</k> <env> Rho </env>
    (.   => <thread>... <id> T </id> <k> S </k> <env> Rho </env> ...</thread>)
  rule  <k> join( T);  => .  ...</k> <thread>... <k>.</k> <id>T</id> ...</thread>
  rule  . Stmts  => .
  rule  S  Ss  => S ↷Ss
endmodule
```

Figure 2.4: IMP++ language semantics

Strictness annotations. The **strict** tag specifies the evaluation strategy of the corresponding construct. E.g., **strict** on line 6 says that the arguments of + must be evaluated before + itself is evaluated, and **strict** (2) on line 8 says that only the second argument of the assignment is to be evaluated. 𝕂 generates rules from these strictness annotations transforming the syntax into tasks (and back) to ensure the proper order of evaluation.

Configuration declarations describe the initial state of the execution environment as a nested multiset/bag of cells. The nested nature of cells resembles that of molecules and membranes in the CHAM soup [12], albeit our cells are named. Cells are written using an XML-like notation and can contain list/sets/maps/bags as well as computations.

The IMP++ configuration consists of a top cell T (lines 33–44), which contains a cell holding the execution threads (lines 34–40), the shared store (line 41), and cells for I/O (lines 42–43). The threads cell holds multiple thread cells, each containing a computation cell k (line 36), an environment (line 37) mapping local variables to store locations, and a thread identifier cell id. The k cell usually appears in all definitions and has a special status among cells, holding the computation and effectively directing the execution. Variables in the configuration declaration (e.g., $PGM:Exp on line 36) must be set by the 𝕂 tool when initializing the execution environment (e.g., $PGM is initialized with the AST of the program to be executed). The cell attribute color is used for displaying purposes, the stream attribute links the contents of the cell to the specified buffer for interactive I/O, and the multiplicity attribute specifies that multiple instances of that cell can coexist. The IMP++ configuration is relatively simple, containing only 9 cells and three nesting levels. The configuration of C has 100 cells and 5 nesting levels [30].

𝕂 **rules** use configuration patterns with variables to describe transitions between configurations, together with an aggressive configuration abstraction mechanism to minimize the size of rules and to increase their modularity. First, rules not mentioning any cell are assumed to take place at the top of the k cell, modeling that evaluation takes place only at the current redex. Second, to account for the fact that most rules collect data from several cells but only make small changes, 𝕂 rules specify the change *in-place* by defining the matching pattern and using the rewrite symbol => *inside* that pattern to locally specify what rewrites into what. This also enables a comprehension mechanism for cells: non-changing parts of cells are abstracted away using ellipses. For example, the rule on line 45 specifies the lookup of an identifier in the store: if X is the first task in the computation cell (... says there might be other subsequent tasks), and if X is mapped to a location N in the environment (the other mappings are abstracted by ...), and if N is mapped to a value I in the store, then X changes to I, leaving the rest unchanged. Finally, $\mathbb{K}$ allows users to only mention the relevant cells in each rule, the missing cells and cell fragments being automatically inferred from the fixed configuration structure. Hence, configuration abstraction allows existing rules to stay unchanged when the configuration is extended or reorganized to accommodate new language features.

The lookup rule was explained above, but now note that it makes full use of configuration abstraction: the k, env, and store cells reside at different levels in the configuration. Variable increment (lines 46–47) and assignment (lines 59–60) are similar, but note that each performs two local rewrites. The read rule on line 48 consumes one integer from the input stream and uses it as a value for the read. Rules on lines 49-55 define the semantics for arithmetic and boolean expressions. Note that + is defined for both integers and strings, and that && is short-circuited. The block rule (line 56) saves the environment on the computation stack, to be recovered upon executing the block statements by the rule on line 57. An expression statement is discarded once the expression was evaluated (line 58). Lines 61–62 define the conditional statement, and line 63 the semantics of while through loop unrolling. The semantics of variable declarations (lines 64–66) adds a new location (N) to the store for each variable, and the variable is to that location in the environment. Note that the N variable is unbound on the left-hand side of the rule, which means that a symbolic value of the specified type will be introduced. print (lines 67–70) appends printable items to the output stream cell (lines 69–70). Halt (line 71) simply voids the computation cell. Spawn creates a new thread holding the spawned statement, the parent's environment, and a fresh (integer) identifier which is also returned to the parent thread (lines 72–73). The join rule (line 74) dissolves the join (T) statement when the thread identified by T has an empty computation. Finally, the rules on lines 75–76 desugar statement sequences into task sequences.

$\mathbb{K}$ rules can introduce symbolic variables in the configuration to rewrite, and are *unconditional*, i.e., there are no premises that need to be recursively reduced to apply a rule. Boolean side conditions are allowed, like in the rule for division, but those are moved into constraints on the rule left and right patterns when regarding the $\mathbb{K}$ rule as a reachability logic rule (see Section 3.1) and are handled by the underlying SMT solver.

2.3 Matching Logic

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 in non-conventional ways, or sometimes both. In contrast, matching logic [94] takes program configurations at its core.

We first recall general matching logic notions and notations (Section 2.3.1) with emphasis on its patterns, and then give an instance of it for configurations corresponding to a fragment of the C language (Section 2.3.2).

2.3.1 Patterns and General Notions

Matching logic 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 thesis, 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 aprogram 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 3.5) 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 [30], which contains more than 100 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 [66] and Maude [21] 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*.

Definition 2. *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 $\exists$):*

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

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. In the simple imperative language IMP with configurations $\langle$`code`, $\sigma\rangle$, described in Section 2.1, let `SUM` be the code "`s:=0; while(n>0)(s:=s+n; n:=n-1)`". Then the following 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 3. *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 3 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 2) iff $(\gamma, \rho) \models \varphi^{\square}$ (notation in Definition 3). 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 $\rho \models \varphi^{\square}[c/\square]$.*

Proof. We have that $(\rho(c), \rho) \models \varphi^{\square}$ iff $\rho^{\rho(c)} \models \varphi^{\square}$. Notice that if a valuation agrees on two variables, then it satisfies a formula iff it satisfies the formula obtained by substituting one of the two variables for the other. In particular, since $\rho^{\rho(c)}(\square) = \rho^{\rho(c)}(c)$, it follows that $\rho^{\rho(c)} \models \varphi^{\square}$ iff $\rho^{\rho(c)} \models \varphi^{\square}[c/\square]$. We notice that $\square$ does not occur in $\varphi^{\square}[c/\square]$, thus $\rho^{\rho(c)} \models \varphi^{\square}[c/\square]$ iff $\rho \models \varphi^{\square}[c/\square]$, and we are done. $\square$

Not all patterns are equally meaningful. For example, the pattern *true* is matched by all configurations, the pattern *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 4. *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, which we use extensively in Section 3.7:

Lemma 2. *If φ is well-defined, then $\models \varphi^\square \wedge \varphi^\square[c/\square] \to \square = c$.*

Proof. Let $\gamma \in \mathcal{T}_{Cfg}$ and $\rho : Var \to \mathcal{T}$. It suffices to prove that if $(\gamma, \rho) \models \varphi^\square$ and $(\gamma, \rho) \models \varphi^\square[c/\square]$ then $(\gamma, \rho) \models \square = c$. Since $\varphi^\square[c/\square]$ is structureless, we have that $(\gamma, \rho) \models \varphi^\square[c/\square]$ iff $\rho \models \varphi^\square[c/\square]$. By Lemma 1 that is iff $(\rho(c), \rho) \models \varphi^\square$. Further, since φ is well-defined, by Definition 4 there exists precisely one γ such that $(\gamma, \rho) \models \varphi^\square$, thus $\gamma = \rho(c)$. Then we can conclude that $(\gamma, \rho) \models \square = c$, and we are done. □

2.3.2 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 thesis, 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.

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

Figure 2.5: Sample configuration

Figure 2.5 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 4.1.2, 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.:

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

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

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

while the pattern

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

says that the location $\mathtt{x}$ 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 $\langle\langle e\rangle_{\mathsf{env}} \ldots\rangle_{\mathsf{cfg}} \wedge \psi$.*

With these notational conventions, the patterns above become:

$$\begin{array}{l}
\langle \mathtt{x} \mapsto 5 \ \ldots\rangle_{\mathsf{env}} \\
\langle \mathtt{x} \mapsto ?i \ \ldots\rangle_{\mathsf{env}} \wedge ?i \geq 0 \\
\langle \mathtt{x} \mapsto ?p \ \ldots\rangle_{\mathsf{env}} \ \langle ?p \mapsto \_ \ \ldots\rangle_{\mathsf{heap}} \\
\langle \mathtt{x} \mapsto ?p \ \ldots\rangle_{\mathsf{env}} \ \langle ?p \mapsto \_\rangle_{\mathsf{heap}}
\end{array}$$

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:

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

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:

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

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

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

The following pattern is more complex:

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

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:

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

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 4.1.2. It abstracts heap subterms into list terms and captures two cases, one in which the list is empty and the other in which it has at least one element.

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

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

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

Chapter 3

Reachability Logic

In this chapter we present reachability logic (Section 3.1) as a unifying framework for specifying operational semantics defined with unconditional rules (as one-path reachability rules) and program correctness properties (as all-path or one-path reachability rules), and then we give a language-independent proof system (Section 3.2) for deriving the program correctness properties from the operational semantics. We extend our results to operational semantics defined with conditional rules for one-path program correctness properties (Section 3.3). We discuss Hoare logic (Section 3.4) and separation logic (Section 3.5) in the context of the work presented in this dissertation. We prove that our proof systems are sound (Section 3.6) and relatively complete in the sense of Cook (Section 3.7).

Much of the work in this chapter comes from Roșu and Ștefănescu [90], Roșu and Ștefănescu [92], Roșu and Ștefănescu [91], Roșu and Ștefănescu [93], Roșu et al. [88], and Ștefănescu et al. [99].

3.1 Specifying Reachability

In this section we define the notion of a one-path reachability rule and an all-path reachability rule. These are pairs of matching logic patterns, written $\varphi \Rightarrow^{\exists} \varphi'$ and, respectively, $\varphi \Rightarrow^{\forall} \varphi'$ to distinguish them, capturing the partial correctness intuition: for any program configuration γ that matches φ, one path ($\exists$), respectively each path ($\forall$), derived using the operational semantics from γ either diverges or otherwise reaches a configuration γ' that matches φ'.

We begin by introducing some basic notions that we need for specifying reachability. Let us fix the following: (1) an algebraic signature Σ, associated to some desired configuration syntax, with a distinguished sort *Cfg*, (2) a sort-wise infinite set *Var* of variables, and (3) a Σ-algebra $\mathcal{T}$, the *configuration model*, which may but need not be a term algebra. As usual, $\mathcal{T}_{Cfg}$ denotes the elements of $\mathcal{T}$ of sort *Cfg*,

Definition 5. *A (one-path)* ***reachability rule*** *is a pair $\varphi \Rightarrow^{\exists} \varphi'$, where φ and φ' are patterns (which can have free variables). Rule $\varphi \Rightarrow^{\exists} \varphi'$ is* ***weakly well-defined*** *iff for any $\gamma \in \mathcal{T}_{Cfg}$ and $\rho : Var \to \mathcal{T}$ with $(\gamma, \rho) \models \varphi$, there exists $\gamma' \in \mathcal{T}_{Cfg}$ with $(\gamma', \rho) \models \varphi'$. A* ***reachability system*** *is a set of reachability rules. Reachability system $\mathcal{S}$ is* ***weakly well-defined*** *iff each rule is weakly well-defined. $\mathcal{S}$ induces a* ***transition system*** *$(\mathcal{T}, \Rightarrow^{\mathcal{T}}_{\mathcal{S}})$ on the configuration model: $\gamma \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma'$ for $\gamma, \gamma' \in \mathcal{T}_{Cfg}$ iff there is some rule $\varphi \Rightarrow^{\exists} \varphi'$ in $\mathcal{S}$ and some valuation $\rho : Var \to \mathcal{T}$ with $(\gamma, \rho) \models \varphi$ and $(\gamma', \rho) \models \varphi'$. A* ***$\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$-path*** *is a finite sequence $\gamma_0 \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma_1 \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \ldots \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma_n$ with $\gamma_0, \ldots, \gamma_n \in \mathcal{T}_{Cfg}$. A $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$-path is* ***complete*** *iff it is not a strict prefix of any other $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$-path.*

We assume an operational semantics is a set of (unconditional) reduction rules "$l \Rightarrow r$ if b", where $l, r \in T_{\Sigma,Cfg}(Var)$ are program configurations with variables and $b \in T_{\Sigma,Bool}(Var)$ is a condition constraining the variables of l, r. Styles of operational semantics using only such (unconditional) rules include evaluation contexts [32], the chemical abstract machine [12] and $\mathbb{K}$ [89] (see Section 2.1.1 for an evaluation contexts semantics). Several large languages have been given semantics in such styles, including C [30] (about 2500 rules) and R5RS Scheme [64]. The reachability proof system works with any set of rules of this form, being agnostic to the particular style of semantics.

Such a rule "$l \Rightarrow r$ if b" states that a ground configuration γ which is an instance of l and satisfies the condition b reduces to an instance γ' of r. Matching logic can express terms with constraints: $l \wedge b$ is satisfied by exactly the γ above. Thus, we can regard such a semantics as a particular weakly well-defined reachability system $\mathcal{S}$ with rules of the form "$l \wedge b \Rightarrow^{\exists} r$". The weakly well-defined condition on $\mathcal{S}$ guarantees that if γ matches the left-hand-side of a rule in $\mathcal{S}$, then the respective rule induces an outgoing transition from γ. The transition system induced by $\mathcal{S}$ describes precisely the behavior of any program in any given state. Such reachability rules capture one-path reachability properties and Hoare triples for deterministic languages.

Figure 2.1 shows a reduction-style executable semantics of a simple imperative language. With the notation explained in Section 2.1, the semantics consists of reduction rules between configuration terms. Each of these rules can be regarded as a one-path reachability rule, with side conditions as constraints on the left-hand-side pattern of the rule. For example, the second rule for the conditional statement becomes the following one-path reachability rule:

$$\langle C,\ \sigma\rangle[\texttt{if}\,(I)\ S_1\,\texttt{else}\,S_2]\ \wedge\ I \neq_{Int} 0 \Rightarrow^{\exists} \langle C,\ \sigma\rangle[S_1]$$

Mathematical domain operations ($+_{Int}$, etc.) are subscripted with *Int* to distinguish them from the language constructs.

A generic, language-independent notion of termination is needed for the partial correctness

Definition 6. *Configuration $\gamma \in \mathcal{T}_{Cfg}$* ***terminates*** *in $(\mathcal{T}, \Rightarrow^{\mathcal{T}}_{S})$ iff there is no infinite $\Rightarrow^{\mathcal{T}}_{S}$-sequence starting with γ.*

We also introduce the following notion

Definition 7. *A reachability rule $\varphi \Rightarrow^{\exists} \varphi'$ is* ***well-defined****, iff φ' is well-defined (recall Definition 4). well-defined.*

Operational semantics defined with rules "$l \Rightarrow r$ if b" are particular well-defined reachability systems with rules of the form $l \wedge b \Rightarrow^{\exists} r$, 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 $\varphi \Rightarrow^{\exists} \varphi_1 \vee \varphi_2$, 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 $\varphi \Rightarrow^{\exists} \varphi'$ 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 thesis 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.

Reachability rules can specify not only operational semantics of languages, but also program properties. Semantic validity in matching logic reachability captures the same intuition of *partial correctness* as Hoare logic, but in more general terms of reachability. Formally, let us fix an operational semantics given as a reachability system $\mathcal{S}$. Then, we can specify reachability in the transition system induced by $\mathcal{S}$

Definition 8. *An* ***all-path reachability rule*** *is a pair $\varphi \Rightarrow^{\forall} \varphi'$ of patterns φ and φ'.*

An all-path reachability rule $\varphi \Rightarrow^{\forall} \varphi'$ is satisfied, $\mathcal{S} \models \varphi \Rightarrow^{\forall} \varphi'$, iff for all complete $\Rightarrow^{\mathcal{T}}_{S}$-paths τ starting with $\gamma \in \mathcal{T}_{Cfg}$ and for all $\rho : Var \to \mathcal{T}$ such that $(\gamma, \rho) \models \varphi$, there exists some $\gamma' \in \tau$ such that $(\gamma', \rho) \models \varphi'$.

A one-path reachability rule $\varphi \Rightarrow^{\exists} \varphi'$ is satisfied, $\mathcal{S} \models \varphi \Rightarrow^{\exists} \varphi'$, iff for all $\gamma \in \mathcal{T}_{Cfg}$ and $\rho : Var \to \mathcal{T}$ such that $(\gamma,\rho) \models \varphi$, there is either a $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$-path from γ to some γ' such that $(\gamma',\rho) \models \varphi'$, or there is a non-terminating execution $\gamma \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma_1 \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma_2 \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \cdots$ from γ.

The correctness property of a racing increment program in the context of a simple imperative language with configurations of the form $\langle \texttt{code},\ \sigma \rangle$ can be specified by

$$\langle \texttt{x:=x+1} \,\|\, \texttt{x:=x+1},\ \texttt{x}\mapsto m\rangle \Rightarrow^{\forall} \exists n\,(\langle \texttt{skip},\ \texttt{x}\mapsto n\rangle \wedge (n = m +_{Int} 1 \vee n = m +_{Int} 2)$$

which states that every terminating execution reaches a state where execution of both threads is complete and the value of x has increased by 1 or 2 (this code has a race). For deterministic programs, the one-path and the all-path reachability coincide. For example, the property of the SUM program mentioned in Section 2.3.1 in the context of the same language would be

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

A Hoare triple describes the resulting state after execution finishes, so it corresponds to a reachability rule where the right side contains no remaining code. However, reachability rules are strictly more expressive than Hoare triples, as they can also specify intermediate configurations (the code in the right-hand-side need not be empty) Reachability rules provide a unified representation for both language semantics and program specifications: $\varphi \Rightarrow^{\exists} \varphi'$ for semantics or one-path reachability, and $\varphi \Rightarrow^{\forall} \varphi'$ for all-path reachability specifications. This makes them perfectly suitable for our goal to obtain program verifiers from operational semantics. Note that, like Hoare triples, reachability rules can only specify properties of complete paths (that is, terminating execution paths). One can use existing Hoare logic techniques to break reasoning about a non-terminating program into reasoning about its terminating components.

We would like to point out that in Definition 5 we used one-path reachability rules $\varphi \Rightarrow^{\exists} \varphi'$ to define the (one-step) transition relation $(\mathcal{T}, \Rightarrow^{\mathcal{T}}_{\mathcal{S}})$, while in Definition 8 we give a (more general) multi-step semantics for one-path rules. We decided to use $\Rightarrow^{\exists}$ for both one-step and multi-step because, as we discuss next in Section 3.2, we do not use further the fact that a rule is one-step or multi-step, other than to notice that, with the definitions above, $\mathcal{S} \models \varphi \Rightarrow^{\exists} \varphi'$, for each $\varphi \Rightarrow^{\exists} \varphi' \in \mathcal{S}$.

We would also like to point a subtle difference between the semantics of the all-path rules $\varphi \Rightarrow^{\forall} \varphi'$ and one-path rules $\varphi \Rightarrow^{\exists} \varphi'$, namely that $\varphi \Rightarrow^{\forall} \varphi'$ is semantically valid iff all terminating execution paths starting at any configuration γ satisfying φ respect the property, while $\varphi \Rightarrow^{\exists} \varphi'$ is semantically valid iff all terminating configurations γ satisfying φ respect the property. One might expect $\varphi \Rightarrow^{\exists} \varphi'$ to be semantically valid iff for all γ satisfying φ, one terminating execution path starting at γ respects the property. For deterministic transition systems, the two definitions are equivalent, since a configuration γ is terminating iff the unique path starting at γ is terminating. However, for non-deterministic systems, the second definition would be stronger, and it would need a more complicated proof system. Thus, we do not explore this possible alternative definition in this thesis.

3.2 Reachability Proof System

Figure 3.1 shows our proof system for both one-path and all-path reachability, which we refer to as *reachability logic*. The target language is given as a weakly well-defined reachability system $\mathcal{S}$. The soundness result (Theorem 3) guarantees that $\mathcal{S} \models \varphi \Rightarrow^{Q} \varphi'$ if $\mathcal{S} \vdash \varphi \Rightarrow^{Q} \varphi'$ is derivable, where $Q \in \{\forall, \exists\}$. The proof system derives more general sequents "$\mathcal{S}, \mathcal{A} \vdash_{C} \varphi \Rightarrow^{Q} \varphi'$", where $\mathcal{A}$ and C are sets of reachability rules. The rules in $\mathcal{A}$ are called *axioms* and rules in C are called *circularities*. If $\mathcal{A}$ or C does not appear in a sequent, it is empty: $\mathcal{S} \vdash_{C} \varphi \Rightarrow^{Q} \varphi'$ is shorthand for $\mathcal{S}, \emptyset \vdash_{C} \varphi \Rightarrow^{Q} \varphi'$, and $\mathcal{S}, \mathcal{A} \vdash \varphi \Rightarrow^{Q} \varphi'$ is shorthand for $\mathcal{S}, \mathcal{A} \vdash_{\emptyset} \varphi \Rightarrow^{Q} \varphi'$. Initially, $\mathcal{A}$ and C are empty. Note that "$\rightarrow$" in Step and Consequence denotes implication.

We assume the free variables of $\varphi_l \Rightarrow^{\exists} \varphi_r$ in the Step proof rule are disjoint from those of $\varphi \Rightarrow^{\forall} \varphi'$. This can be achieved by renaming all the free variables in the rules in $\mathcal{S}$ before beginning a proof, and by noticing that the free variables in the rules in $\mathcal{S}$ do not spill into any sequent during the proof (Axiom applies a renaming substitution).

The intuition is that the reachability rules in $\mathcal{A}$ can be assumed valid, while those in C have been postulated but not yet justified. After making progress from φ (at least one derivation by Step or by Axiom), the rules in C become (coinductively) valid and can be used in derivations by Axiom. During the proof, circularities can be added to C via Circularity, flushed into $\mathcal{A}$ by Transitivity, and used

STEP :

$$\frac{\begin{array}{l}\models \varphi \to \bigvee_{\varphi_l \Rightarrow^{\exists} \varphi_r \,\in\, \mathcal{S}} \exists \mathit{FreeVars}(\varphi_l).\varphi_l \\ \models (\exists c\ (\varphi^{\square}[c/\square] \wedge \varphi_l^{\square}[c/\square]) \wedge \varphi_r) \to \varphi' \qquad \text{for each } \ \varphi_l \Rightarrow^{\exists} \varphi_r \ \in \ \mathcal{S}\end{array}}{\mathcal{S}, \mathcal{A} \ \vdash_C \varphi \Rightarrow^{\forall} \varphi'}$$

AXIOM :

$$\frac{\varphi \Rightarrow^{Q} \varphi' \in \mathcal{S} \cup \mathcal{A} \qquad \psi \text{ is FOL formula (logical frame)} \qquad \sigma : \mathit{Var} \to \mathit{Var}}{\mathcal{S}, \mathcal{A} \vdash_C \varphi\sigma \wedge \psi \Rightarrow^{Q} \varphi'\sigma \wedge \psi}$$

REFLEXIVITY :

$$\frac{\cdot}{\mathcal{S}, \mathcal{A} \vdash \varphi \Rightarrow^{Q} \varphi}$$

TRANSITIVITY :

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

CONSEQUENCE :

$$\frac{\models \varphi_1 \to \varphi_1' \qquad \mathcal{S}, \mathcal{A} \vdash_C \varphi_1' \Rightarrow^{Q} \varphi_2' \qquad \models \varphi_2' \to \varphi_2}{\mathcal{S}, \mathcal{A} \vdash_C \varphi_1 \Rightarrow^{Q} \varphi_2}$$

CASE ANALYSIS :

$$\frac{\mathcal{S}, \mathcal{A} \vdash_C \varphi_1 \Rightarrow^{Q} \varphi \qquad \mathcal{S}, \mathcal{A} \vdash_C \varphi_2 \Rightarrow^{Q} \varphi}{\mathcal{S}, \mathcal{A} \vdash_C \varphi_1 \vee \varphi_2 \Rightarrow^{Q} \varphi}$$

ABSTRACTION :

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

CIRCULARITY :

$$\frac{\mathcal{S}, \mathcal{A} \vdash_{C \cup \{\varphi \Rightarrow^{Q} \varphi'\}} \varphi \Rightarrow^{Q} \varphi'}{\mathcal{S}, \mathcal{A} \vdash_C \varphi \Rightarrow^{Q} \varphi'}$$

Figure 3.1: Proof system for reachability. Here $Q \in \{\forall, \exists\}$.

via AXIOM. The semantics of sequent $\mathcal{S}, \mathcal{A} \vdash_C \varphi \Rightarrow^{Q} \varphi'$ (read "$\mathcal{S}$ with axioms $\mathcal{A}$ and circularities C proves $\varphi \Rightarrow^{Q} \varphi'$") is: $\varphi \Rightarrow^{Q} \varphi'$ holds if the rules in $\mathcal{A}$ hold and those in C hold after taking at least one step from φ in the transition system $(\Rightarrow_{\mathcal{S}}^{\mathcal{T}}, \mathcal{T})$. Moreover, if $C \neq \emptyset$ then φ reaches φ' after at least one step on all complete paths when $Q = \forall$ and on at least one path when $Q = \exists$. As a consequence of this definition, any rule $\varphi \Rightarrow^{Q} \varphi'$ derived by CIRCULARITY has the property that φ reaches φ' after at least one step, due to CIRCULARITY having a prerequisite $\mathcal{S}, \mathcal{A} \vdash_{C \cup \{\varphi \Rightarrow^{Q} \varphi'\}} \varphi \Rightarrow^{Q} \varphi'$ (with a non-empty set of circularities). We next discuss the proof rules.

STEP derives a sequent where φ reaches φ' in one step on all paths. The first premise ensures any configuration matching φ matches the left-hand-side φ_l of some rule in $\mathcal{S}$ and thus, as $\mathcal{S}$ is weakly well-defined, can take a step: if $(\gamma, \rho) \models \varphi$ then there is a $\varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}$ and a valuation ρ' of the free variables of φ_l s.t. $(\gamma, \rho') \models \varphi_l$, and thus γ has at least one $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$-successor generated by $\varphi_l \Rightarrow^{\exists} \varphi_r$. The second premise ensures that each $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$-successor of a configuration matching φ matches φ': if $\gamma \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma'$ and γ matches φ then there is some rule $\varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}$ and $\rho : \mathit{Var} \to \mathcal{T}$ such that $(\gamma, \rho) \models \varphi \wedge \varphi_l$ and $(\gamma', \rho) \models \varphi_r$; then the second part implies γ' matches φ'.

Designing a proof rule for deriving an execution step along all paths is non-trivial. For instance, one might expect Step to require as many premises as there are transitions going out of φ, as is the case for the examples presented later in this section. However, that is not possible, as the number of successors of a configuration matching φ may be unbounded even if each matching configuration has a finite branching factor in the transition system. Step avoids this issue by requiring only one premise for each rule by which some configuration φ can take a step, even if that rule can be used to derive multiple transitions. To illustrate this situation, consider a language defined by $\mathcal{S} \equiv \{\langle n_1 \rangle \wedge n_1 >_{\mathit{Int}} n_2 \Rightarrow^{\exists} \langle n_2 \rangle\}$, with n_1 and n_2 non-negative integer variables. A configuration in this language is a singleton with a non-negative integer. Intuitively, a positive integer transits into a strictly smaller non-negative integer, in a non-deterministic way. The branching factor of a non-negative integer is its value. Then it follows that $\mathcal{S} \models \langle m \rangle \Rightarrow^{\forall} \langle 0 \rangle$. Deriving this rule reduces (by Circularity and other proof rules) to deriving $\langle m_1 \rangle \wedge m_1 >_{\mathit{Int}} 0 \Rightarrow^{\forall} \exists m_2\, (\langle m_2 \rangle \wedge m_1 >_{\mathit{Int}} m_2)$. The left-hand-side is matched by any positive integer, and thus its branching factor is infinity. Deriving this rule with Step requires only two premises, $\models (\langle m_1 \rangle \wedge m_1 >_{\mathit{Int}} 0) \to \exists n_1 n_2\, (\langle n_1 \rangle \wedge n_1 >_{\mathit{Int}} n_2)$ and $\models \exists c\, (c = \langle m_1 \rangle \wedge m_1 >_{\mathit{Int}} 0 \wedge c = \langle n_1 \rangle \wedge n_1 >_{\mathit{Int}} n_2) \wedge \langle n_2 \rangle \to \exists m_2\, (\langle m_2 \rangle \wedge m_1 >_{\mathit{Int}} m_2)$. A similar situation arises in the case of real-world languages with thread pools of arbitrary size.

Axiom applies a trusted one-path or all-path rule. It also derives a one-path sequent where φ reaches φ' in one step on some path. Notice that while a variable x cannot be instantiated with a term t by σ, one can simply add $x = t$ to ψ, and use Consequence to get the same effect.

The logical frame ψ is formalized as a patternless formula, as it is meant to only add logical but no structural constraints. Since reachability logic keeps a clear separation between program variables and logical variables the logical constraints are persistent, that is, they do not interfere with the dynamic nature of the operational rules and can therefore be safely used for framing. This is not the case for structural constraints.

Reflexivity and Transitivity capture the closure properties of the reachability relation. Reflexivity requires C empty to ensure that rules derived with non-empty C take at least one step. Transitivity enables the circularities as axioms for the second premise, since if C is not empty, the first premise is guaranteed to take a step. Consequence, Case Analysis and Abstraction are adapted from Hoare logic. Ignoring circularities, these seven proof rules are the formal infrastructure for symbolic execution.

Circularity has a coinductive nature, allowing us to make new circularity claims. We typically make such claims for code with repetitive behaviors, such as loops, recursive functions, jumps, etc. If there is a derivation of the claim using itself as a circularity, then the claim holds. This would obviously be unsound if the new assumption was available immediately, but requiring progress (taking at least on step before circularities can be used) ensures that only diverging executions correspond to endless invocation of a circularity.

One important aspect of concurrent program verification, which we do not address in this dissertation, is proof compositionality. Our focus here is limited to establishing a sound and complete language-independent proof system for all-path reachability rules, to serve as a foundation for further results and applications, and to discuss our current implementation of it.

From here on, we make the following convention: we can use $\mathcal{S} \cup \mathcal{A} \vdash_C \varphi \Rightarrow^{\exists} \varphi$ as a shortcut for $\mathcal{S}, \mathcal{A} \vdash_C \varphi \Rightarrow^{\exists} \varphi'$. Essentially, we merge the one-path reachability rules in $\mathcal{S}$ and $\mathcal{A}$, since the one-path part of the proof system only uses the rules in these sets in the Axiom proof rule. The union set is sometimes refered as $\mathcal{S}$ or as $\mathcal{A}$.

3.2.1 Example Reachability Logic Proofs

Sum Example

Consider the following snippet in a simple imperative language IMP (Section 2.1), say `SUM`, summing up the natural numbers smaller than `n`:

```
s = 0;
while(--n)
  s = s + n;
```

Let us now verify the `SUM` program using the generic reachability logic instantiated with the executable semantics of the language. Let $\mathcal{S}$ be the reachability logic system in Figure 2.1, where each rule is regarded as a one-path rule as explained in Section 3.1. We first make some notations:

$$\begin{array}{lll}
\varphi_{pre} & \equiv & \langle \texttt{SUM},\ \texttt{n} \mapsto n,\ \texttt{s} \mapsto s\rangle \ \wedge\ n \geq_{Int} 1 \\
\varphi_{post} & \equiv & \langle \texttt{skip},\ \texttt{n} \mapsto 0,\ \texttt{s} \mapsto n *_{Int} (n -_{Int} 1) /_{Int} 2\rangle \\
\texttt{LOOP} & \equiv & \texttt{while(--n) (s = s + n;)} \\
\varphi_1 & \equiv & \langle \texttt{LOOP},\ \texttt{n} \mapsto n,\ \texttt{s} \mapsto 0\rangle \ \wedge\ n \geq_{Int} 1 \\
\varphi_{inv} & \equiv & \langle \texttt{LOOP},\ \texttt{n} \mapsto n',\ \texttt{s} \mapsto \Sigma_{n'}^{n -_{Int} 1}\rangle \ \wedge\ n' \geq_{Int} 1 \\
\texttt{IF} & \equiv & \texttt{if(--n) (s = s + n; LOOP) else skip} \\
\varphi_2 & \equiv & \langle \texttt{IF},\ \texttt{n} \mapsto n',\ \texttt{s} \mapsto \Sigma_{n'}^{n -_{Int} 1}\rangle \ \wedge\ n' \geq_{Int} 1 \\
\varphi_3 & \equiv & \langle \texttt{LOOP},\ \texttt{n} \mapsto n' -_{Int} 1, \texttt{s} \mapsto \Sigma_{n' -_{Int} 1}^{n -_{Int} 1}\rangle \wedge\ n' >_{Int} 1
\end{array}$$

The reachability logic rule stating the correctness of `SUM` is

$$\varphi_{pre} \Rightarrow^{\exists} \varphi_{post}$$

which can be derived as follows:

$$\begin{array}{rlr}
1. & \mathcal{S}, \emptyset \vdash_{\emptyset} \varphi_{pre} \Rightarrow^{\exists} \varphi_{post} & \textsc{Trans}(2,3) \\
2. & \mathcal{S}, \emptyset \vdash_{\emptyset} \varphi_{pre} \Rightarrow^{\exists} \exists n'.\varphi_{inv} & \textsc{Consequence}(4) \\
3. & \mathcal{S}, \emptyset \vdash_{\emptyset} \exists n'.\varphi_{inv} \Rightarrow^{\exists} \varphi_{post} & \textsc{Abstraction}(5) \\
4. & \mathcal{S}, \emptyset \vdash_{\emptyset} \varphi_{pre} \Rightarrow^{\exists} \varphi_1 & \textsc{Axiom} \\
5. & \mathcal{S}, \emptyset \vdash_{\emptyset} \varphi_{inv} \Rightarrow^{\exists} \varphi_{post} & \textsc{Circularity}(6) \\
6. & \mathcal{S}, \emptyset \vdash_{\{\varphi_{inv} \Rightarrow^{\exists} \varphi_{post}\}} \varphi_{inv} \Rightarrow^{\exists} \varphi_{post} & \textsc{Trans}(7,8) \\
7. & \mathcal{S}, \emptyset \vdash_{\{\varphi_{inv} \Rightarrow^{\exists} \varphi_{post}\}} \varphi_{inv} \Rightarrow^{\exists} \varphi_2 & \textsc{Axiom} \\
8. & \mathcal{S}, \{\varphi_{inv} \Rightarrow^{\exists} \varphi_{post}\} \vdash_{\emptyset} \varphi_2 \Rightarrow^{\exists} \varphi_{post} & \textsc{Case}(9,10) \\
9. & \mathcal{S}, \{\varphi_{inv} \Rightarrow^{\exists} \varphi_{post}\} \vdash_{\emptyset} \varphi_2 \wedge n' >_{Int} 1 \Rightarrow^{\exists} \varphi_{post} & \textsc{Trans}(11,12) \\
10. & \mathcal{S}, \{\varphi_{inv} \Rightarrow^{\exists} \varphi_{post}\} \vdash_{\emptyset} \varphi_2 \wedge n' \leq_{Int} 1 \Rightarrow^{\exists} \varphi_{post} & \textsc{Axiom}^{+} \\
11. & \mathcal{S}, \{\varphi_{inv} \Rightarrow^{\exists} \varphi_{post}\} \vdash_{\emptyset} \varphi_2 \wedge n' >_{Int} 1 \Rightarrow^{\exists} \varphi_3 & \textsc{Axiom}^{+} \\
12. & \mathcal{S}, \{\varphi_{inv} \Rightarrow^{\exists} \varphi_{post}\} \vdash_{\emptyset} \varphi_3 \Rightarrow^{\exists} \varphi_{post} & \textsc{Axiom}(\mu)
\end{array}$$

Step (1) factors the proof using the loop invariant existentially quantified in all its new (mathematical) variables. To show that the invariant holds when the loop is reached (2), we "execute" the initial pattern φ_{pre} with the operational semantics rule of assignment (4), reaching pattern φ_1, which implies (in matching

$$\begin{tikzcd}
\left\langle\begin{matrix} \mathtt{x} := \mathtt{x} + 1 \\ \| \, \mathtt{x} := \mathtt{x} + 1, \\ \mathtt{x} \mapsto m \end{matrix}\right\rangle \arrow[r] \arrow[d] & \left\langle\begin{matrix} \mathtt{x} := \mathtt{x} + 1 \\ \| \, \mathtt{x} := m + 1, \\ \mathtt{x} \mapsto m \end{matrix}\right\rangle \arrow[r] \arrow[d] & \left\langle\begin{matrix} \mathtt{x} := \mathtt{x} + 1 \\ \| \, \mathtt{x} := m +_{Int} 1, \\ \mathtt{x} \mapsto m \end{matrix}\right\rangle \arrow[r] \arrow[d] & \left\langle\begin{matrix} \mathtt{x} := \mathtt{x} + 1 \\ \| \, \mathtt{skip}, \\ \mathtt{x} \mapsto m +_{Int} 1 \end{matrix}\right\rangle \arrow[dr, bend left] & \\
\left\langle\begin{matrix} \mathtt{x} := m + 1 \\ \| \, \mathtt{x} := \mathtt{x} + 1, \\ \mathtt{x} \mapsto m \end{matrix}\right\rangle \arrow[r] \arrow[d] & \left\langle\begin{matrix} \mathtt{x} := m + 1 \\ \| \, \mathtt{x} := m + 1, \\ \mathtt{x} \mapsto m \end{matrix}\right\rangle \arrow[r] \arrow[d] & \left\langle\begin{matrix} \mathtt{x} := m + 1 \\ \| \, \mathtt{x} := m +_{Int} 1, \\ \mathtt{x} \mapsto m \end{matrix}\right\rangle \arrow[r] \arrow[d] & \left\langle\begin{matrix} \mathtt{x} := m + 1 \\ \| \, \mathtt{skip}, \\ \mathtt{x} \mapsto m +_{Int} 1 \end{matrix}\right\rangle \arrow[d] & \left\langle\begin{matrix} \mathtt{x} := m +_{Int} 1 + 1 \\ \| \, \mathtt{skip}, \\ \mathtt{x} \mapsto m +_{Int} 1 \end{matrix}\right\rangle \arrow[d] \\
\left\langle\begin{matrix} \mathtt{x} := m +_{Int} 1 \\ \| \, \mathtt{x} := \mathtt{x} + 1, \\ \mathtt{x} \mapsto m \end{matrix}\right\rangle \arrow[r] \arrow[d] & \left\langle\begin{matrix} \mathtt{x} := m +_{Int} 1 \\ \| \, \mathtt{x} := m + 1, \\ \mathtt{x} \mapsto m \end{matrix}\right\rangle \arrow[r] \arrow[d] & \left\langle\begin{matrix} \mathtt{x} := m +_{Int} 1 \\ \| \, \mathtt{x} := m +_{Int} 1, \\ \mathtt{x} \mapsto m \end{matrix}\right\rangle \arrow[r] \arrow[d] & \left\langle\begin{matrix} \mathtt{x} := m +_{Int} 1 \\ \| \, \mathtt{skip}, \\ \mathtt{x} \mapsto m +_{Int} 1 \end{matrix}\right\rangle \arrow[d] & \left\langle\begin{matrix} \mathtt{x} := m +_{Int} 2 \\ \| \, \mathtt{skip}, \\ \mathtt{x} \mapsto m +_{Int} 1 \end{matrix}\right\rangle \arrow[dd] \\
\left\langle\begin{matrix} \mathtt{skip} \\ \| \, \mathtt{x} := \mathtt{x} + 1, \\ \mathtt{x} \mapsto m +_{Int} 1 \end{matrix}\right\rangle \arrow[dr, bend right] & \left\langle\begin{matrix} \mathtt{skip} \\ \| \, \mathtt{x} := m + 1, \\ \mathtt{x} \mapsto m +_{Int} 1 \end{matrix}\right\rangle \arrow[r] & \left\langle\begin{matrix} \mathtt{skip} \\ \| \, \mathtt{x} := m +_{Int} 1, \\ \mathtt{x} \mapsto m +_{Int} 1 \end{matrix}\right\rangle \arrow[r] & \left\langle\begin{matrix} \mathtt{skip} \\ \| \, \mathtt{skip}, \\ \mathtt{x} \mapsto m +_{Int} 1 \end{matrix}\right\rangle & \\
 & \left\langle\begin{matrix} \mathtt{skip} \\ \| \, \mathtt{x} := m +_{Int} 1 + 1, \\ \mathtt{x} \mapsto m +_{Int} 1 \end{matrix}\right\rangle \arrow[r] & \left\langle\begin{matrix} \mathtt{skip} \\ \| \, \mathtt{x} := m +_{Int} 2, \\ \mathtt{x} \mapsto m +_{Int} 1 \end{matrix}\right\rangle \arrow[rr] & & \left\langle\begin{matrix} \mathtt{skip} \\ \| \, \mathtt{skip}, \\ \mathtt{x} \mapsto m +_{Int} 2 \end{matrix}\right\rangle
\end{tikzcd}$$

Figure 3.2: State space of the racing increment example

logic) the existentially quantified invariant. To prove the existentially quantified invariant, thanks to Abstraction we first eliminate the existential quantifier (3) and then, expecting a circular behavior of the loop, we add the proof obligation as a circularity (5). The rest is just symbolic execution of the loop body using the executable semantics and giving priority to the circularity when it matches. Specifically, the loop is unrolled using the executable semantics of `while` (2), then a case analysis is initiated on whether the value held by `n` is larger than 1 or not (8), and φ_{post} is indeed reached on both paths (9,10). The circularity is used on the positive branch only, as expected. For brevity, we do not mention the use of Consequence when changing a formula into an equivalent formula. For example, when deriving step (8) by Case Analysis from (9, 10), φ_2 implicitly becomes $\varphi_2 \wedge n' >_{Int} 1 \vee \varphi_2 \wedge n' \leq_{Int} 1$.

Parallel Increment

The first example shows that our proof system enables exhaustive state exploration, similar to symbolic model-checking but based on the operational semantics. Although humans prefer to avoid such explicit proofs and instead methodologically use abstraction or compositional reasoning whenever possible (and such methodologies are not excluded by our proof system), a complete proof system must nevertheless support them. Thus, model-checking techniques for reducing the space size (like partial order reduction or abstraction) should apply in our setting as well. The code $\mathtt{x:=x+1} \parallel \mathtt{x:=x+1}$ exhibits a race on $\mathtt{x}$: the value of $\mathtt{x}$ increases by 1 when both reads happen before either write, and by 2 otherwise. The all-path rule that captures this behavior is

$$\langle \mathtt{x:=x+1} \parallel \mathtt{x:=x+1},\ \mathtt{x}{\mapsto}m\rangle \Rightarrow^{\forall} \exists n\,(\langle \mathtt{skip},\ \mathtt{x}{\mapsto}n\rangle \wedge (n = m +_{Int} 1 \vee n = m +_{Int} 2)$$

We show that the program has exactly these behaviors by deriving this rule in the proof system. Call the right-hand-side pattern G. The proof contains subproofs of $c \Rightarrow^{\forall} G$ for every reachable configuration c, tabulated in Figure 3.2. The subproofs for c matching G use Reflexivity and Consequence, while the rest use Transitivity, Step, and Case Analysis to reduce to the proofs for the next configurations. For example, the following sequent

$$\langle \mathtt{x} := m + 1 \parallel \mathtt{x} := \mathtt{x} + 1,\ \mathtt{x} \mapsto m\rangle \Rightarrow^{\forall} G$$

can be inferred from the following two sequents

$$\langle \mathtt{x} := m +_{Int} 1 \parallel \mathtt{x} := \mathtt{x{+}1},\ \mathtt{x} \mapsto m\rangle \Rightarrow^{\forall} G$$
$$\langle \mathtt{x} := m{+}1 \parallel \mathtt{x} := m{+}1,\ \mathtt{x} \mapsto m\rangle \Rightarrow^{\forall} G$$

as shown by the proof fragment below

$$\dfrac{\textsc{Step}\ \dfrac{\cdots}{\left\langle\begin{array}{c}\mathtt{x} := m+1\\ \parallel \mathtt{x} := \mathtt{x}+1,\\ \mathtt{x}\mapsto m\end{array}\right\rangle \Rightarrow^{\forall} \left\langle\begin{array}{c}\mathtt{x} := m+_{Int} 1\\ \parallel \mathtt{x} := \mathtt{x}+1,\\ \mathtt{x}\mapsto m\end{array}\right\rangle \vee \left\langle\begin{array}{c}\mathtt{x} := m+1\\ \parallel \mathtt{x} := m+1,\\ \mathtt{x}\mapsto m\end{array}\right\rangle} \qquad \dfrac{\dfrac{\cdots}{\left\langle\begin{array}{c}\mathtt{x} := m+_{Int} 1\\ \parallel \mathtt{x} := \mathtt{x}+1,\\ \mathtt{x}\mapsto m\end{array}\right\rangle \Rightarrow^{\forall} G} \quad \dfrac{\cdots}{\left\langle\begin{array}{c}\mathtt{x} := m+1\\ \parallel \mathtt{x} := m+1,\\ \mathtt{x}\mapsto m\end{array}\right\rangle \Rightarrow^{\forall} G}}{\left\langle\begin{array}{c}\mathtt{x} := m+_{Int} 1\\ \parallel \mathtt{x} := \mathtt{x}+1,\\ \mathtt{x}\mapsto m\end{array}\right\rangle \vee \left\langle\begin{array}{c}\mathtt{x} := m+1\\ \parallel \mathtt{x} := m+1,\\ \mathtt{x}\mapsto m\end{array}\right\rangle \Rightarrow^{\forall} G}\ \text{CA}}{\langle \mathtt{x} := m+1 \parallel \mathtt{x} := \mathtt{x}+1,\ \mathtt{x}\mapsto m\rangle \Rightarrow^{\forall} G}\ \textsc{Trans}$$

For the rule hypotheses of Step above, note that all rules but **lookup** and $\mathbf{op}_{+}$ make the overlap condition $\exists c \left(\left\langle\begin{array}{c}\mathtt{x} := m+1\\ \parallel \mathtt{x} := \mathtt{x}+1,\\ \mathtt{x}\mapsto m\end{array}\right\rangle[c/\square] \wedge \varphi_l[c/\square]\right)$ unsatisfiable, and only one choice of free variables works for the **lookup** and $\mathbf{op}_{+}$ rules. For **lookup**, φ_l is $\langle C,\sigma\rangle[x]$ and the overlap condition is only satisfiable if the logical variables C, σ and x are equal to $(\mathtt{x} := m + 1 \parallel \mathtt{x} := \blacksquare + 1)$, $(\mathtt{x} \mapsto m)$, and $\mathtt{x}$, resp. Under this assignment, the pattern $\varphi_r = \langle C,\sigma\rangle[\sigma(x)]$ is equivalent to $\langle \mathtt{x} := m + 1 \parallel \mathtt{x} := m + 1,\ \mathtt{x} \mapsto m\rangle$, the right branch of the disjunction. The $\mathbf{op}_{+}$ rule is handled similarly. The assignment for **lookup** can also witness the existential in the progress hypothesis of Step. Subproofs for other states in Figure 3.2 can be constructed similarly.

3.3 Conditional One-Path Reachability Logic

In this section, we extend the results from the previous two sections to work with semantics formalized with conditional rewrite rules. Specifically, we introduce conditional one-path reachability rules, and a one-path proof system that derives program correctness properties expressed as unconditional one-path reachability rules. Unfortunately, we do not have a similar result for program correctness properties expressed as unconditional all-path reachability rules.

3.3.1 Conditional Reachability Rules

As discussed before, unconditional reachability rules can express particular operational semantics that do not require rule premises. Here we introduce *conditional* one-path reachability rules, a generalization capturing as special instances the rules used in conventional operational semantics with rule premises.

Definition 9. *A **conditional one-path reachability rule** is a sentence*

$$\varphi \Rightarrow^{\exists} \varphi' \text{ if } \varphi_1 \Rightarrow^{\exists} \varphi'_1 \bigwedge \ldots \bigwedge \varphi_n \Rightarrow^{\exists} \varphi'_n$$

*with $n \geq 0$ and with φ, φ', φ_1, φ'_1, ..., φ_n, φ'_n matching logic patterns. We call φ the **left-hand side (LHS)** and φ' the **right-hand side (RHS)** of the rule. A rule is **unconditional** when $n = 0$. A **reachability system** is a set of reachability rules.*

As discussed in Section 2.1.2, in here we assume that operational semantics are defined with rewrite rules of the form

$$cfg \Rightarrow cfg' \text{ if } b \bigwedge cfg_1 \Rightarrow cfg'_1 \bigwedge b_1 \bigwedge \ldots \bigwedge cfg_n \Rightarrow cfg'_n \bigwedge b_n$$

which can now be seen as syntactic sugar for reachability rules

$$cfg \wedge b \wedge b_1 \wedge \ldots \wedge b_n \Rightarrow^{\exists} cfg' \text{ if } cfg_1 \Rightarrow^{\exists} cfg'_1 \bigwedge \ldots \bigwedge cfg_n \Rightarrow^{\exists} cfg'_n$$

Here the boolean side conditions have been all conjuncted with the LHS pattern. The above is a correct reachability rule, where φ is $cfg \wedge b \wedge b_1 \wedge \ldots \wedge b_n$. For example, the rule **cond**$_1$ in the big-step semantics of IMP in Figure 2.2 is syntactic sugar for the reachability rule

$$\langle \texttt{if}\ e\ s_1\ s_2,\ \sigma\rangle \wedge i \neq 0 \Rightarrow^{\exists} \langle\sigma'\rangle \text{ if } \langle e,\ \sigma\rangle \Rightarrow^{\exists} \langle i\rangle \bigwedge \langle s_1,\ \sigma\rangle \Rightarrow^{\exists} \langle\sigma'\rangle$$

In this section, we assume that a language/calculus/system is defined as a reachability system and, unless otherwise specified, fix an arbitrary reachability system $\mathcal{S}$. It is irrelevant for the subsequent developments whether such rules represent a small-step, a big-step, or any other particular operational semantics.

An operational semantics typically describes program behaviors by generating a transition system over program configurations, which can associate a behavior to any given program in any given state. In some cases, e.g., small-step semantics, the transition system comprises all the atomic computational steps; in other cases, e.g., big-step semantics, the transition system consists of a binary relationship mapping configurations holding (fragments of) programs to their resulting configurations after evaluation. Recall (Definition 2) that matching logic comes equipped with a model of configurations, $\mathcal{T}$. We next show how $\mathcal{S}$ yields a transition system over the configurations of $\mathcal{T}$.

Definition 10. *The* ***transition relation induced by*** $\mathcal{S}$*,* $\Rightarrow_{\mathcal{S}}^{\mathcal{T}} \subseteq \mathcal{T}_{Cfg} \times \mathcal{T}_{Cfg}$ *(written infix), is the least fixpoint of the following condition:* $\gamma \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma'$ *if there exists a reachability rule*

$$\varphi \Rightarrow^{\exists} \varphi' \text{ if } \varphi_1 \Rightarrow^{\exists} \varphi_1' \bigwedge \ldots \bigwedge \varphi_n \Rightarrow^{\exists} \varphi_n'$$

in $\mathcal{S}$ *and some valuation* $\rho : Var \to \mathcal{T}$ *such that:*

1. $(\gamma,\rho) \models \varphi$ *and* $(\gamma',\rho) \models \varphi'$*; and*
2. *for all* $\gamma_1, \ldots, \gamma_n \in \mathcal{T}_{Cfg}$ *with* $(\gamma_i, \rho) \models \varphi_i$ *for all* $1 \leq i \leq n$ *there exist* $\gamma_1', \ldots, \gamma_n'$ *with* $(\gamma_i', \rho) \models \varphi_i'$ *and* $\gamma_i \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma_i'$ *for all* $1 \leq i \leq n$ *(*$\Rightarrow_{\mathcal{S}}^{\star\mathcal{T}}$ *is the transitive/reflexive closure of* $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$*).*

Then $(\mathcal{T}_{Cfg}, \Rightarrow_{\mathcal{S}}^{\mathcal{T}})$ *is the* ***transition system induced by*** $\mathcal{S}$*.*

Intuitively, $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$ is the least relation compatible with all the rules in $\mathcal{S}$, with all rule conditions interpreted as $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$-reachability. The existence of a least fixpoint is guaranteed by the Knaster-Tarski theorem: the set of binary relations on $\mathcal{T}_{Cfg}$ with inclusion forms a complete lattice, and the condition is monotonic.

If $\mathcal{S}$ contains only rewrite rules, that is, rules whose patterns are all basic, then all the configurations γ, γ', γ_1, γ'_1, $\ldots$, γ_n, γ'_n in Definition 10 are uniquely determined by ρ, since $(\gamma, \rho) \models \pi$ iff $\gamma = \rho(\pi)$ for any basic pattern π (by Definition 2). In this case, $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$ becomes the usual transition relation induced by a (top-most) term rewrite system ($\mathcal{S}$) on a Σ-algebra ($\mathcal{T}$). For example, if $\mathcal{S}$ is the IMP small-step semantics in Figure 2.2 then the following are valid transitions (`LOOP` is the loop of `SUM` in Section 2.1.2; for notational simplicity, we make no distinction between ground terms and their interpretation in $\mathcal{T}$):

$$
\begin{array}{r}
\langle \texttt{SUM},\ (\texttt{s} \mapsto 7, \texttt{n} \mapsto 10)\rangle \ \Rightarrow^{\mathcal{T}}_{\mathcal{S}}\ \langle \texttt{LOOP},\ (\texttt{s} \mapsto 0, \texttt{n} \mapsto 10)\rangle \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \\
\langle \texttt{if (n>0) (s:=s+n;n:=n-1;LOOP) skip},\ (\texttt{s} \mapsto 0, \texttt{n} \mapsto 10)\rangle \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \\
\langle \texttt{if(10>0) (s:=s+n;n:=n-1;LOOP) skip},\ (\texttt{s} \mapsto 0, \texttt{n} \mapsto 10)\rangle \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \\
\ldots \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \langle \texttt{LOOP},\ (\texttt{s} \mapsto 10, \texttt{n} \mapsto 9)\rangle \ \Rightarrow^{\mathcal{T}}_{\mathcal{S}}\ \ldots \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \\
\langle \texttt{LOOP},\ (\texttt{s} \mapsto 55, \texttt{n} \mapsto 0)\rangle \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \ldots \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \langle \texttt{skip},\ (\texttt{s} \mapsto 55, \texttt{n} \mapsto 0)\rangle
\end{array}
$$

In computing the transitions above, we need to go up to 3 nested conditional rules in Definition 10. On the other hand, if $\mathcal{S}$ is the big-step semantics in Figure 2.2, then we have

$$\langle \texttt{SUM},\ (\texttt{s} \mapsto 7, \texttt{n} \mapsto 10)\rangle \ \Rightarrow^{\mathcal{T}}_{\mathcal{S}}\ \langle \texttt{s} \mapsto 55, \texttt{n} \mapsto 0\rangle$$

in one transition step, but in order to compute that we need to apply more than 40 nested conditional rules.

To define rule validity with the sense of partial correctness we need to say which configurations terminate. In some cases, e.g., small-step semantics, nontermination is captured by the ability to take an infinite sequence of transitions starting with the given configuration; in other cases, e.g., big-step semantics, nontermination is captured by the ability to make an infinite sequence of nested attempts to fulfill conditions of rules while trying to take a step—which is not the same as a stuck configuration which cannot take a step because no rules apply.

We define a novel notion of termination of configurations with respect to $\mathcal{S}$, which captures both cases above. Our definition is based on a preorder on configurations, which will be well-founded under terminating configurations. This order is inspired by quasi-decreasing orders for conditional term rewriting systems [39]. Our definition is also somewhat related to operational termination of conditional term rewrite systems [59], although the latter is a property of a rewrite system as whole, while our notion of termination refers to a particular configuration in a particular model.

Definition 11. *Let $(\mathcal{T}_{Cfg}, \succ)$ be the* ***termination dependence*** *relation defined as follows:*

- *$\gamma \succ \gamma'$ if $\gamma \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma'$; and*
- *$\gamma \succ \gamma'$ if there is a rule $\varphi \Rightarrow^{\exists} \varphi'$ if $\varphi_1 \Rightarrow^{\exists} \varphi'_1 \wedge \ldots \wedge \varphi_n \Rightarrow^{\exists} \varphi'_n$ in $\mathcal{S}$, valuation $\rho : Var \to \mathcal{T}$, and index $1 \leq i \leq n$ so that:*
  1. *$(\gamma, \rho) \models \varphi$*
  2. *$(\gamma', \rho) \models \varphi_i$*
  3. *For each $1 \leq j < i$, $\varphi_j \Rightarrow^{\exists} \varphi'_j$ is "strongly ρ-valid": for any γ_j such that $(\gamma_j, \rho) \models \varphi_j$ there exists γ'_j such that $\gamma_j \Rightarrow^{\star\mathcal{T}}_{\mathcal{S}} \gamma'_j$ and $(\gamma'_i, \rho) \models \varphi'_j$*

$\gamma \in \mathcal{T}_{Cfg}$ ***terminates*** *iff there are no infinite decreasing $\succ$ chains starting at γ; γ* ***diverges*** *otherwise. We let $\succeq$ denote the partial order associated to $\succ$, i.e., its reflexive and transitive closure.*

Our definition of termination above mimics the application of conditional rules in the configuration model, in that conditions are solved in order and a condition is considered only if all the previous conditions are successfully solved. Taking into account the order of conditions is essential to get the correct notion of termination. If condition 3 were dropped, then any while loop could be said to diverge in the big-step semantics by using the **while**$_2$ rule and recursing into the second condition which executes the body again, without first checking that the test of the loop actually passes. Termination dependence is essential in the proof of soundness, which justifies circularity by well-founded induction on $\succ$ under terminating configurations.

Let us consider IMP again. In Section 2.1.2 we informally claimed that `SUM` always terminates, `SUM'` only terminates when `n` ≤ 0, and `SUM`$_\infty$ never terminates. We can now make these claims formal. For `SUM`, we can show that any configuration γ of the form $\langle$`SUM`, $\sigma\rangle$ terminates with any of the two semantics in Figure 2.2, for any state σ (including such σ which lacks `s` or `n`). For `SUM'`, any configuration $\langle$`SUM'`, (`n`$\mapsto n$, σ)$\rangle$ with $n \leq 0$ terminates in both semantics, whether or not σ binds `s`. However, our informal claim "`SUM'` only terminates when `n` ≤ 0" in Section 2.1.2 was (purposely) imprecise. Indeed, configurations $\langle$`SUM'`, $\sigma\rangle$ with `n` or `s` undefined in σ also terminate. Finally, our informal claim "`SUM`$_\infty$ never terminates” was also imprecise for similar reasons. Stated precisely, configurations of the form $\langle \texttt{SUM}_\infty,\ (\texttt{n} \mapsto n,\ \texttt{s} \mapsto s,\ \sigma)\rangle$ diverge. Interestingly, such configurations diverge for different reasons in the two semantics, descending by the first bullet of Definition 11 in small-step semantics, and by the second in big-step semantics.

Definition 12. *A pattern φ **terminates** (resp. **diverges**), written $\mathcal{S} \models \varphi{\downarrow}$ (resp. $\mathcal{S} \models \varphi{\uparrow}$), iff for all $\gamma \in \mathcal{T}_{Cfg}$ and for all $\rho : Var \to \mathcal{T}$, if $(\gamma, \rho) \models \varphi$ then γ terminates (resp. diverges).*

In the case of IMP with $\mathcal{S}$ either its small-step or its big-step semantics, from the discussion above we can conclude

$$\mathcal{S} \models \langle \texttt{SUM},\ \sigma\rangle{\downarrow}$$
$$\mathcal{S} \models (\langle \texttt{SUM}',\ (\texttt{n} \mapsto n, \sigma)\rangle \wedge n \leq_{Int} 0 \vee \langle \texttt{SUM}',\ \sigma\rangle \wedge (\texttt{n} \notin Dom(\sigma) \vee \texttt{s} \notin Dom(\sigma)){\downarrow}$$
$$\mathcal{S} \models (\langle \texttt{SUM}',\ (\texttt{n} \mapsto n, \texttt{s} \mapsto s, \sigma)\rangle \wedge n >_{Int} 0){\uparrow}$$
$$\mathcal{S} \models \langle \texttt{SUM}_\infty,\ (\texttt{n} \mapsto n, \texttt{s} \mapsto s, \sigma)\rangle{\uparrow}$$

3.3.2 Validity and Well-Definedness

In Hoare logic, $\{pre\}$ code $\{post\}$ is (semantically) valid, in the sense of partial correctness, iff for any state satisfying *pre*, if `code` terminates then the resulting state satisfies *post*. This elegant definition has the luxury of relying on another formal semantics of the target language which provides the language-specific notions of “state”, “satisfaction”, and “termination”. Since here everything happens in a single language-independent framework, we generalize the notion of validity as follows:

Definition 13. *Given valuation $\rho : Var \to \mathcal{T}$, an unconditional reachability rule $\varphi \Rightarrow^{\exists} \varphi'$ is ρ-**valid**, written $\mathcal{S}, \rho \models \varphi \Rightarrow^{\exists} \varphi'$, iff for any $\gamma \in \mathcal{T}_{Cfg}$ with $(\gamma, \rho) \models \varphi$, if γ terminates then there is a $\gamma' \in \mathcal{T}_{Cfg}$ such that $(\gamma', \rho) \models \varphi'$ and $\gamma \Rightarrow^{\star\mathcal{T}}_{\mathcal{S}} \gamma'$. Rule $\varphi \Rightarrow^{\exists} \varphi'$ is **valid**, written $\mathcal{S} \models \varphi \Rightarrow^{\exists} \varphi'$, iff it is ρ-valid for each $\rho : Var \to \mathcal{T}$.*

Intuitively, $\mathcal{S} \models \varphi \Rightarrow^{\exists} \varphi'$ specifies reachability: any terminating configuration matching φ transits, on some execution path, to a configuration matching φ'. This notion of validity becomes the usual Hoare logic validity when the reachability rule $\varphi \Rightarrow^{\exists} \varphi'$ corresponds to a Hoare triple and $\mathcal{S}$ is deterministic. Both IMP definitions in Figure 2.2 are deterministic. A major difference between our validity and Hoare validity is that the language-specific “state” and “code” are replaced by the language-independent “configuration”.

Recall that $\mathcal{S}$ is an arbitrary reachability system, thought of as a "semantics". However, not all reachability systems are meaningful as semantics in all situations. Consider a reachability system containing a rule of the form $\varphi \Rightarrow^{\exists} \mathit{false}$. Such a rule is semantically useless (because it generates no transitions), but also makes reachability reasoning unsound, because there are no transitions in the generated transition system which would validate $\varphi \Rightarrow^{\exists} \mathit{false}$. Some of the subsequent results require that $\mathcal{S} \models \mu$ for any unconditional $\mu \in \mathcal{S}$, which can be ensured by simple conditions on $\mathcal{S}$ such as:

Definition 14. *Rule $\varphi \Rightarrow^{\exists} \varphi'$ if $\varphi_1 \Rightarrow^{\exists} \varphi'_1 \bigwedge \ldots \bigwedge \varphi_n \Rightarrow^{\exists} \varphi'_n$ is* ***(weakly) well-defined*** *iff φ', φ_1, ..., φ_n are (weakly) well-defined. $\mathcal{S}$ is* ***(weakly) well-defined*** *iff all its rules are.*

Rules of the form $\varphi \Rightarrow^{\exists} \mathit{false}$ are *not* (weakly) well-defined. Since operational semantics rules contain only configuration terms except possibly for their LHSs (see discussion at beginning of Section 3.3.1), and since configuration terms are basic patterns, which are always well-defined, it is safe to say that the reachability systems of interest are expected to be well-defined. Nevertheless, weak well-definedness suffices for the soundness of conditional one-path reachability logic, although we need full well-definedness for completeness.

3.3.3 Proof System

Figure 3.3 shows the reachability logic proof system. The target language is given as a weakly well-defined reachability system $\mathcal{S}$. The soundness result (Theorem 4) guarantees that $\mathcal{S} \models \varphi \Rightarrow^{\exists} \varphi'$ if $\mathcal{S}, \mathcal{A} \vdash \varphi \Rightarrow^{\exists} \varphi'$ is derivable. Note that the proof system derives more general sequents of the form $\mathcal{S}, \mathcal{A} \vdash_C \varphi \Rightarrow^{\exists} \varphi'$, where $\mathcal{A}$ and C are sets of reachability rules, like the proof system in Figure 3.1 (Section 3.2).

The proof system generalizes the one-path part ($Q = \exists$) of the proof system in Figure 3.1 to work with a reachability system $\mathcal{S}$ given with conditional rules. Specifically, it generalizes the Axiom proof rule to the following form

$$\textsc{Axiom}: \quad \frac{\begin{array}{c}(\varphi \Rightarrow^{\exists} \varphi' \text{ if } \varphi_1 \Rightarrow^{\exists} \varphi'_1 \bigwedge \cdots \bigwedge \varphi_n \Rightarrow^{\exists} \varphi'_n) \in \mathcal{A} \cup \mathcal{S} \\ \psi \text{ is a structureless pattern} \qquad \sigma : \mathit{Var} \to \mathit{Var} \\ \mathcal{S}, \mathcal{A} \cup C \vdash \varphi_i\sigma \wedge \psi \Rightarrow^{\exists} \varphi'_i\sigma \quad \text{for } i \in 1, ..., n\end{array}}{\mathcal{S}, \mathcal{A} \vdash_C \varphi\sigma \wedge \psi \Rightarrow^{\exists} \varphi'\sigma \wedge \psi}$$

$$\textsc{Axiom}: \quad \frac{\begin{array}{c}(\varphi \Rightarrow^\exists \varphi' \text{ if } \varphi_1 \Rightarrow^\exists \varphi'_1 \wedge \cdots \wedge \varphi_n \Rightarrow^\exists \varphi'_n) \in \mathcal{A} \cup \mathcal{S} \\ \psi \text{ is a structureless pattern} \qquad \sigma : Var \to Var \\ \mathcal{S}, \mathcal{A} \cup C \vdash \varphi_i\sigma \wedge \psi \Rightarrow^\exists \varphi'_i\sigma \quad \text{for } i \in 1, ..., n\end{array}}{\mathcal{S}, \mathcal{A} \vdash_C \varphi\sigma \wedge \psi \Rightarrow^\exists \varphi'\sigma \wedge \psi}$$

$$\textsc{Reflexivity}: \quad \frac{\cdot}{\mathcal{S}, \mathcal{A} \vdash \varphi \Rightarrow^\exists \varphi}$$

$$\textsc{Transitivity}: \quad \frac{\mathcal{S}, \mathcal{A} \vdash_C \varphi_1 \Rightarrow^\exists \varphi_2 \qquad \mathcal{S}, \mathcal{A} \cup C \vdash \varphi_2 \Rightarrow^\exists \varphi_3}{\mathcal{S}, \mathcal{A} \vdash_C \varphi_1 \Rightarrow^\exists \varphi_3}$$

$$\textsc{Consequence}: \quad \frac{\models \varphi_1 \to \varphi'_1 \quad \mathcal{S}, \mathcal{A} \vdash_C \varphi'_1 \Rightarrow^\exists \varphi'_2 \quad \models \varphi'_2 \to \varphi_2}{\mathcal{S}, \mathcal{A} \vdash_C \varphi_1 \Rightarrow^\exists \varphi_2}$$

$$\textsc{Case Analysis}: \quad \frac{\mathcal{S}, \mathcal{A} \vdash_C \varphi_1 \Rightarrow^\exists \varphi \qquad \mathcal{S}, \mathcal{A} \vdash_C \varphi_2 \Rightarrow^\exists \varphi}{\mathcal{S}, \mathcal{A} \vdash_C \varphi_1 \vee \varphi_2 \Rightarrow^\exists \varphi}$$

$$\textsc{Abstraction}: \quad \frac{\mathcal{S}, \mathcal{A} \vdash_C \varphi \Rightarrow^\exists \varphi' \quad \text{where} \;\; X \cap FV(\varphi') = \emptyset}{\mathcal{S}, \mathcal{A} \vdash_C \exists X\, \varphi \Rightarrow^\exists \varphi'}$$

$$\textsc{Circularity}: \quad \frac{\mathcal{S}, \mathcal{A} \vdash_{C \cup \{\varphi \Rightarrow \varphi'\}} \varphi \Rightarrow^\exists \varphi'}{\mathcal{S}, \mathcal{A} \vdash_C \varphi \Rightarrow^\exists \varphi'}$$

Figure 3.3: Proof system for (one-path) reachability using conditional rules.

and it keeps the rest of the proof rules unchanged. Notice that this proof rule has one premise for each condition of the conditional reachability rule, and in each premise the circularities from C are enabled. Incorporating framing into the axiom rule is necessary to make logical constraints available while proving the conditions of the axiom hold.

Figure 3.4 shows detailed formal proofs that the `SUM` program (Section 2.1.2) indeed calculates the sum of the first `n` natural numbers in `s`, for the small-step and big-step semantics of IMP from Figure 2.2. In the small-step case (left column) the circularity corresponding to the loop is used via the Transitivity rule, while in the big-step case (right column) the circularity is used via the Axiom rule. Below we discuss these proofs informally.

In small-step, the specification $\varphi_{\text{SUM}} \Rightarrow^\exists \varphi$ (sequent 14) is

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

General macros

$\texttt{SUM} \equiv \texttt{s := 0; while (n>0) (s := s+n; n := n-1)}$

$\texttt{S}_1 \equiv \texttt{s := s + n; n := n - 1; LOOP}$

$\texttt{IF} \equiv \texttt{if (n > 0) then S}_1\ \texttt{else skip}$

$\varphi_{\texttt{SUM}} \equiv \langle \texttt{SUM}, (\texttt{s} \mapsto s, \texttt{n} \mapsto n)\rangle \wedge n \geq_{Int} 0$

$\varphi_{\texttt{INV}} \equiv \langle \texttt{LOOP}, (\texttt{s} \mapsto \texttt{sum}_{\texttt{inv}}(n,n'), \texttt{n} \mapsto n')\rangle \wedge n' \geq_{Int} 0$

$\varphi_{\texttt{S}_1} \equiv \langle \texttt{S}_1, (\texttt{s} \mapsto \texttt{sum}_{\texttt{inv}}(n,n'), \texttt{n} \mapsto n')\rangle$

$\texttt{LOOP} \equiv \texttt{while (n>0) (s := s+n; n := n-1)}$

$\texttt{S}_2 \equiv \texttt{n := n - 1; LOOP}$

$\texttt{sum}_{\texttt{inv}}(n,n') \equiv (n -_{Int} n') *_{Int} (n +_{Int} n' +_{Int} 1) /_{Int} 2$

$\varphi_{\texttt{IF}} \equiv \langle \texttt{IF}, (\texttt{s} \mapsto \texttt{sum}_{\texttt{inv}}(n,n'), \texttt{n} \mapsto n')\rangle$

$\varphi^{after}_{\texttt{LOOP}} \equiv \langle \texttt{LOOP}, (\texttt{s} \mapsto \texttt{sum}_{\texttt{inv}}(n,n' -_{Int} 1), \texttt{n} \mapsto n' -_{Int} 1)\rangle$

$\varphi_{\texttt{S}_2} \equiv \langle \texttt{S}_2, (\texttt{s} \mapsto \texttt{sum}_{\texttt{inv}}(n,n' -_{Int} 1), \texttt{n} \mapsto n')\rangle$

Small-step macros

$\varphi \equiv \langle \texttt{skip}, (\texttt{s} \mapsto n *_{Int} (n +_{Int} 1) /_{Int} 2, \texttt{n} \mapsto 0)\rangle$

$\mu \equiv \exists n'\, \varphi_{\texttt{INV}} \Rightarrow^{\exists} \varphi$

Big-step macros

$\varphi \equiv \langle (\texttt{s} \mapsto n *_{Int} (n +_{Int} 1) /_{Int} 2, \texttt{n} \mapsto 0)\rangle$

$\mu \equiv \exists n'\, \varphi_{\texttt{INV}} \Rightarrow^{\exists} \varphi$

Small-step proof derivation ($\mathcal{S}$ is IMP's small-step semantics)

1. $\mathcal{S} \vdash \varphi_{\texttt{SUM}} \Rightarrow^{\exists} \varphi_{\texttt{INV}} \wedge n' =_{Int} n$	[**asgn**$_2$, **seq**$_1$, **seq**$_2$]
2. $\mathcal{S} \vdash_{\{\mu\}} \varphi_{\texttt{INV}} \Rightarrow^{\exists} \varphi_{\texttt{IF}} \wedge n' \geq_{Int} 0$	[**while**]
3. $\mathcal{S} \vdash_{\{\mu\}} \varphi_{\texttt{IF}} \wedge n' >_{Int} 0 \Rightarrow^{\exists} \varphi_{\texttt{S}_1} \wedge n' >_{Int} 0$	[**lookup**, **>**$_1$, **>**$_3$, **cond**$_1$, **cond**$_2$]
4. $\mathcal{S} \vdash_{\{\mu\}} \varphi_{\texttt{S}_1} \wedge n' >_{Int} 0 \Rightarrow^{\exists} \varphi_{\texttt{S}_2} \wedge n' >_{Int} 0$	[**lookup**,**+**$_1$,**+**$_2$,**+**$_3$,**asgn**$_1$,**asgn**$_2$,**seq**$_1$,**seq**$_2$]
5. $\mathcal{S} \vdash_{\{\mu\}} \varphi_{\texttt{S}_2} \wedge n' >_{Int} 0 \Rightarrow^{\exists} \varphi^{after}_{\texttt{LOOP}} \wedge n' >_{Int} 0$	[**lookup**,**-**$_1$,**-**$_3$,**asgn**$_1$,**asgn**$_2$,**seq**$_1$,**seq**$_2$]
6. $\mathcal{S} \vdash_{\{\mu\}} \varphi_{\texttt{S}_2} \wedge n' >_{Int} 0 \Rightarrow^{\exists} \exists n'\, \varphi_{\texttt{INV}}$	[**Consequence**(5)]
7. $\mathcal{S} \cup \{\mu\} \vdash \exists n'\, \varphi_{\texttt{INV}} \Rightarrow^{\exists} \varphi$	[μ]
8. $\mathcal{S} \cup \{\mu\} \vdash \varphi_{\texttt{IF}} \wedge n' >_{Int} 0 \Rightarrow^{\exists} \varphi$	[**Transitivity**(3, 4, 6, 7)]
9. $\mathcal{S} \cup \{\mu\} \vdash \varphi_{\texttt{IF}} \wedge n' =_{Int} 0 \Rightarrow^{\exists} \varphi$	[**lookup**, **>**$_1$, **>**$_3$, **cond**$_1$, **cond**$_3$]
10. $\mathcal{S} \cup \{\mu\} \vdash \varphi_{\texttt{IF}} \wedge n' \geq_{Int} 0 \Rightarrow^{\exists} \varphi$	[**Case Analysis**(8, 9)]
11. $\mathcal{S} \vdash_{\{\mu\}} \exists n'\, \varphi_{\texttt{INV}} \Rightarrow^{\exists} \varphi$	[**Transitivity**(2, 10); **Abstraction**]
12. $\mathcal{S} \vdash \exists n'\, \varphi_{\texttt{INV}} \Rightarrow^{\exists} \varphi$	[**Circularity**(11)]
13. $\mathcal{S} \vdash \varphi_{\texttt{SUM}} \Rightarrow^{\exists} \exists n'\, \varphi_{\texttt{INV}}$	[**Consequence**(1)]
14. $\mathcal{S} \vdash \varphi_{\texttt{SUM}} \Rightarrow^{\exists} \varphi$	[**Transitivity**(13, 12)]

Big-step proof derivation ($\mathcal{S}$ is IMP's big-step semantics)

1. $\mathcal{S} \cup \{\mu\} \vdash \exists n'\, \varphi_{\texttt{INV}} \Rightarrow^{\exists} \varphi$	[μ]
2. $\mathcal{S} \cup \{\mu\} \vdash \varphi^{after}_{\texttt{LOOP}} \wedge n' >_{Int} 0 \Rightarrow^{\exists} \varphi$	[**Consequence**(1)]
3. $\mathcal{S} \vdash_{\{\mu\}} \varphi_{\texttt{S}_2} \wedge n' >_{Int} 0 \Rightarrow^{\exists} \varphi$	[**lookup**, **int**, **-**, **asgn**, **seq**(2)]
4. $\mathcal{S} \vdash_{\{\mu\}} \varphi_{\texttt{S}_1} \wedge n' >_{Int} 0 \Rightarrow^{\exists} \varphi$	[**lookup**, **int**, **+**, **asgn**, **seq**(3)]
5. $\mathcal{S} \vdash_{\{\mu\}} \varphi_{\texttt{INV}} \wedge n' >_{Int} 0 \Rightarrow^{\exists} \varphi$	[**lookup**, **int**, **>**, **while**$_2$(4)]
6. $\mathcal{S} \vdash_{\{\mu\}} \varphi_{\texttt{INV}} \wedge n' =_{Int} 0 \Rightarrow^{\exists} \varphi$	[**lookup**, **int**, **>**, **while**$_1$]
7. $\mathcal{S} \vdash_{\{\mu\}} \varphi_{\texttt{INV}} \Rightarrow^{\exists} \varphi$	[**Case Analysis**(5, 6)]
8. $\mathcal{S} \vdash_{\{\mu\}} \exists n'\, \varphi_{\texttt{INV}} \Rightarrow^{\exists} \varphi$	[**Abstraction**(7)]
9. $\mathcal{S} \vdash \exists n'\, \varphi_{\texttt{INV}} \Rightarrow^{\exists} \varphi$	[**Circularity**(8)]
10. $\mathcal{S} \vdash \varphi_{\texttt{INV}} \wedge n' =_{Int} n \Rightarrow^{\exists} \varphi$	[**Consequence**(9)]
11. $\mathcal{S} \vdash \varphi_{\texttt{SUM}} \Rightarrow^{\exists} \varphi$	[**int**, **asgn**, **skip**, **seq**(10)]

Figure 3.4: Formal reachability logic proofs for SUM. Simple Consequence rules used to perform domain reasoning are elided for readability.

We begin by Transitivity (12,13) through $\exists n' \, \varphi_{\text{INV}}$, with φ_{INV}

$$\langle \texttt{LOOP}, \ (\texttt{s} \mapsto (n -_{Int} n') *_{Int} (n +_{Int} n' +_{Int} 1) /_{Int} 2, \ \texttt{n} \mapsto n') \rangle \wedge n' \geq_{Int} 0.$$

$\varphi_{\text{SUM}} \Rightarrow^{\exists} \exists n' \, \varphi_{\text{INV}}$ (13) holds by running the operational semantics on `SUM` (1), and abstracting this as $\exists n' \, \varphi_{\text{INV}}$ by Consequence. The property $\mu \equiv \exists n' \, \varphi_{\text{INV}} \Rightarrow^{\exists} \varphi$ (12) is proved by Circularity (from 11). Sequent 11 is proven by using Abstraction to remove the quantifier and fix an arbitrary n', and using Transitivity between 2 and 10. Sequent 2 holds by applying the `while` rule to unroll the loop into a conditional. This progress releases the circularity μ in 10. We continue by Case Analysis on $n' =_{Int} 0 \vee n' >_{Int} 0$, running the operational semantics in each case (the two cases are described by sequents 8 and 9). When $n' =_{Int} 0$ the goal is reached directly (sequent 9), and when $n' >_{Int} 0$ we reach a configuration implying $\exists n' \, \varphi_{\text{INV}}$ and finish by applying the recently-added axiom μ (sequent 7).

In the big-step case the specification (11) $\varphi_{\text{SUM}} \Rightarrow^{\exists} \varphi$ is now

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

As before, we prove $\mu \equiv \exists n' \, \varphi_{\text{INV}} \Rightarrow^{\exists} \varphi$, with the same φ_{INV} as before. We reach $\exists n' \, \varphi_{\text{INV}}$ from φ_{SUM} by applying the big-step semantics of assignment and sequential composition (10) and then Consequence (9). The difference is that this is reached in a premise of applications of conditional axioms, rather than a premise of Transitivity. Property $\exists n' \, \varphi_{\text{INV}} \Rightarrow^{\exists} \varphi$ is also proved by Circularity (8), but this time the circularity is released (in 1) by applying the **while**$_2$ axiom, and used in one of its conditions.

3.3.4 Nontermination and ω-Closure

In Hoare logic divergence can be indirectly specified using Hoare triples with postcondition *false*. We can similarly reduce proving divergence to proving a reachability rule whose RHS is *false*, provided our arbitrary matching logic $\mathcal{L}$ has a pattern *false* matched by no configurations (in first-order matching logic we have the FOL *false* formula): $\mathcal{S} \models \varphi\uparrow$ iff $\mathcal{S} \models \varphi \Rightarrow^{\exists} \mathit{false}$. Therefore, any complete proof system for reachability can also prove divergence. It turns out, however, that it is necessary (for the completeness theorem) and convenient to refer to divergence directly.

Definition 15. *We let $\mathcal{S}^\omega$, called the* ***ω-closure*** *of $\mathcal{S}$, be the reachability system extending $\mathcal{S}$ as follows:*

- *Add to Σ a new constant ω of sort Cfg;*
- *Add to $\mathcal{T}_{Cfg}$ a new element $\mathcal{T}_\omega$;*
- *Add to $\mathcal{S}$ a new rule, $\omega \Rightarrow^\exists \omega$;*
- *For each rule $\varphi \Rightarrow^\exists \varphi' \mathsf{if} \varphi_1 \Rightarrow^\exists \varphi'_1 \wedge \ldots \wedge \varphi_n \Rightarrow^\exists \varphi'_n$ in $\mathcal{S}$ and each $1 \leq i \leq n$, add to $\mathcal{S}$ a conditional reachability rule*

$$\varphi \Rightarrow^\exists \omega \ \mathsf{if}\ \varphi_1 \Rightarrow^\exists \varphi'_1 \wedge \ldots \wedge \varphi_{i-1} \Rightarrow^\exists \varphi'_{i-1} \wedge \varphi_i \Rightarrow^\exists \omega.$$

By convention $(\mathcal{S}^\omega)^\omega = \mathcal{S}^\omega$ and we call $\mathcal{S}$ ***ω-closed*** *iff $\mathcal{S} = \mathcal{S}^\omega$.*

The ω-closure operation is algorithmic and easy to implement. Since $\mathcal{T}_\omega$ is the only configuration that matches ω, we conclude that ω is well-defined. In fact, the ω-closure operation does not affect well-definedness: $\mathcal{S}$ is (weakly) well-defined iff $\mathcal{S}^\omega$ is (weakly) well-defined. Moreover, the additional rules are semantically irrelevant:

Proposition 2. *The following equivalences hold for all configurations $\gamma, \gamma' \in \mathcal{T}_{Cfg}$ and for all patterns φ, φ':*

- *$\gamma \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma'$ iff $\gamma \rightarrow_{\mathcal{S}^\omega} \gamma'$;*
- *γ terminates for $\mathcal{S}$ iff γ terminates for $\mathcal{S}^\omega$;*
- *$\mathcal{S} \models \varphi \Rightarrow^\exists \varphi'$ iff $\mathcal{S}^\omega \models \varphi \Rightarrow^\exists \varphi'$.*

Therefore, the ω-closure has no semantic effect. It only has proof-theoretical merit, ensuring we can prove divergence as follows:

Proposition 3. *If $\mathcal{S}$ is ω-closed, then $\mathcal{S} \models \varphi\uparrow$ iff $\mathcal{S} \models \varphi \Rightarrow^\exists \omega$.*

For an ω-closed system of rules, our proof system can prove divergence:

Corollary 1. *If $\mathcal{S}$ is also ω-closed, then $\mathcal{S} \vdash \varphi \Rightarrow^\exists \omega$ implies $\mathcal{S} \models \varphi\uparrow$.*

3.4 Relationship with Hoare Logic

Here we briefly compare reachability logic with Hoare logic, aiming to convey the message that verification using reachability logic is not harder than using Hoare logic, even when done manually. First we look at the Hoare logic proof of correctness for the `SUM` example, the same example that we verified earlier using reachability logic. Then, we show that how we can mechanically translate any Hoare logic proof tree for IMP into a reachability logic proof tree, which has size linear in the size of the Hoare logic proof tree. Thus, we would argue that reachability logic is a practical alternative for Hoare logic.

We assume the reader is familiar with Hoare logic. For a detailed survey of Hoare logic, we recommend Apt [4, 5].

3.4.1 Example Hoare Logic Proof

The Hoare logic precondition ψ_{pre} is $\texttt{n} =_{Int} n \ \wedge \ n \geq_{Int} 1$, and the postcondition ψ_{post} is $\texttt{n} =_{Int} 0 \ \wedge \ \texttt{s} =_{Int} n *_{Int} (n -_{Int} 1) \ /_{Int} 2$. The variable n using italic font is introduced to capture the original value of the program variable `n`, so that we can use it to express the value of `s` in the post-condition (the loop changes the value of `n`). A typical (over-)simplification in hand proofs using Hoare logic is to collapse expression constructs in the language with operations in the underlying domain, e.g., + with $+_{Int}$. Tools, however, distinguish the two and implement translations from the former to the latter; e.g., + may be 32-bit while $+_{Int}$ may be arbitrary precision, or + may have a concurrent semantics allowing all the interleavings of its arguments' behaviors, etc. Since our language is simple, we do this translation by hand on the fly, but for clarity we use mathematical operations in formulae.

To derive the Hoare triple $\{\psi_{pre}\}$ `SUM` $\{\psi_{post}\}$, we need to find a loop invariant ψ_{inv} and then use the invariant proof rule:

$$\frac{\{\psi_{inv} \wedge E \neq_{Int} 0\} \ \ S \ \ \{\psi_{inv}\}}{\{\psi_{inv}\} \ \texttt{while(}E\texttt{)}\, S \ \ \{\psi_{inv} \wedge E =_{Int} 0\}} \quad \text{(HL-W{\small HILE})}$$

The loop condition is inserted within formulae. Thus, when verifying programs using Hoare logic, expressions cannot have side effects; programs need to be modified to isolate side effects from computed values of expressions, which is an inherently language-specific operation.

For example, VCC [22] expands the loop above into one having more than a dozen statements in its translation to Boogie [8]. To keep it human readable, we manually modify `SUM` in a minimal (but adhoc) way to the equivalent `SUM′` below, which can be verified using conventional Hoare logic:

```
s = 0;
n = n - 1;
while(n) {
  s = s + n;
  n = n - 1;
}
```

Recall the remaining Hoare logic rules required for this proof:

$$\{\psi[E/X]\}\ X = E\texttt{;}\ \{\psi\} \qquad \text{(HL-Asgn)}$$

$$\frac{\{\psi_1\}\ S_1\ \{\psi_2\} \qquad \{\psi_2\}\ S_2\ \{\psi_3\}}{\{\psi_1\}\ S_1\, S_2\ \{\psi_3\}} \qquad \text{(HL-Seq)}$$

$$\frac{\models \psi_1' \to \psi_1 \qquad \{\psi_1\}\ S\ \{\psi_2\} \qquad \models \psi_2 \to \psi_2'}{\{\psi_1'\}\ S\ \{\psi_2'\}} \qquad \text{(HL-Cnsq)}$$

Using the following notations,

$$\begin{aligned}
\psi_{pre} &\equiv \texttt{n} =_{Int} n \ \land\ n \geq_{Int} 1 \\
\psi_{post} &\equiv \texttt{n} =_{Int} 0 \ \land\ \texttt{s} =_{Int} n *_{Int} (n -_{Int} 1) /_{Int} 2 \\
\psi_1 &\equiv \texttt{n} =_{Int} n -_{Int} 1 \ \land\ n \geq_{Int} 1 \ \land\ \texttt{s} =_{Int} 0 \\
\Sigma_i^j &\equiv (j +_{Int} i) *_{Int} (j -_{Int} i +_{Int} 1) /_{Int} 2 \\
\psi_{inv} &\equiv \texttt{n} \geq_{Int} 0 \ \land\ s =_{Int} \Sigma_{\texttt{n}+_{Int}1}^{n-_{Int}1} \\
\texttt{LOOP'} &\equiv \texttt{while(n)\{s = s + n; n = n - 1;\}} \\
\psi_2 &\equiv \texttt{n} >_{Int} 0 \ \land\ \texttt{s} =_{Int} \Sigma_{\texttt{n}+_{Int}1}^{n-_{Int}1} \\
\psi_3 &\equiv \texttt{n} >_{Int} 0 \ \land\ \texttt{s} =_{Int} \Sigma_{\texttt{n}}^{n-_{Int}1}
\end{aligned}$$

the proof proceeds as follows (we follow the program order):

1.	$\{\psi_{pre}\}$ `SUM'` $\{\psi_{post}\}$	HL-SEQ(2, 6)
2.	$\{\psi_{pre}\}$ `s=0;n=n-1;` $\{\psi_{inv}\}$	HL-CNSQ(3)
3.	$\{\psi_{pre}\}$ `s=0;n=n-1;` $\{\psi_1\}$	HL-SEQ(4, 5)
4.	$\{\psi_{pre}\}$ `s=0;` $\{\psi_{pre} \wedge \texttt{s} =_{Int} 0\}$	HL-ASGN
5.	$\{\psi_{pre} \wedge \texttt{s} =_{Int} 0\}$ `n=n-1;` $\{\psi_1\}$	HL-ASGN
6.	$\{\psi_{inv}\}$ `LOOP'` $\{\psi_{post}\}$	HL-WHILE(7), HL-CNSQ
7.	$\{\psi_2\}$ `s=s+n;n=n-1;` $\{\psi_{inv}\}$	HL-SEQ(8,9)
8.	$\{\psi_2\}$ `s=s+n;` $\{\psi_3\}$	HL-ASGN
9.	$\{\psi_3\}$ `n=n-1;` $\{\psi_{inv}\}$	HL-ASGN, HL-CNSQ

Therefore, step (1) factors the proof using the loop invariant ψ_{inv}. First we show using HL-ASGN twice (4,5) followed by HL-SEQ (4) that ψ_1 is reachable before the loop (3), which implies the invariant holds when the loop is reached (2). To prove the invariant, we use HL-WHILE at (6), which generates the proof obligation (7) for the loop body, noticing that ψ_2 is logically equivalent to $\psi_{inv} \wedge \texttt{n} \neq_{Int} 0$. The rest follows by two applications of HL-ASGN at (8,9), followed by an HL-SEQ which concludes the proof.

3.4.2 From Hoare Logic Proofs to Reachability Logic Proofs

Here we show how proof derivations using the IMP-specific Hoare logic proof system in Figure 3.5 can be translated into proof derivations using the language-independent matching logic reachability proof system in Figure 3.1 with the IMP operational semantics in Figure 2.1 as axioms. The sizes of the two proof derivations are within a linear factor.

Translating Hoare Triples into Reachability Rules

Without restricting the generality, we make the following simplifying assumptions about the Hoare triples $\{\psi\}\,\texttt{code}\,\{\psi'\}$ that appear in the Hoare logic proof derivation that we translate into a matching logic reachability proof: (1) the variables appearing in `code` belong to an arbitrary but fixed finite set $\texttt{X} \subset \mathit{PVar}$; (2) the additional variables appearing in ψ and ψ' but not in `code` belong to an arbitrary but fixed finite set $\texttt{Y} \subset \mathit{PVar}$ such that $\texttt{X} \cap \texttt{Y} = \emptyset$. In other words, we fix the finite disjoint sets $\texttt{X}, \texttt{Y} \subset \mathit{PVar}$, and they have the properties above for all Hoare triples

Generic

HL-csq $$\frac{\models \psi_1 \rightarrow \psi_3 \quad \{\psi_3\}\,\mathtt{s}\,\{\psi_4\} \quad \models \psi_4 \rightarrow \psi_2}{\{\psi_1\}\,\mathtt{s}\,\{\psi_2\}}$$

IMP axiomatic semantics

HL-skip $$\frac{\cdot}{\{\psi\}\,\mathtt{skip}\,\{\psi\}}$$

HL-asgn $$\frac{\cdot}{\{\psi[\mathtt{e}/\mathtt{x}]\}\,\mathtt{x}\ :=\ \mathtt{e}\,\{\psi\}}$$

HL-seq $$\frac{\{\psi_1\}\,\mathtt{s}_1\,\{\psi_2\} \qquad \{\psi_2\}\,\mathtt{s}_2\,\{\psi_3\}}{\{\psi_1\}\,\mathtt{s}_1;\ \mathtt{s}_2\,\{\psi_3\}}$$

HL-cond $$\frac{\begin{array}{c}\{\psi_1 \wedge \mathtt{e} \neq 0\}\,\mathtt{s}_1\,\{\psi_2\} \\ \{\psi_1 \wedge \mathtt{e} = 0\}\,\mathtt{s}_2\,\{\psi_2\}\end{array}}{\{\psi_1\}\,\mathtt{if(e)}\,\mathtt{s}_1\,\mathtt{else}\,\mathtt{s}_2\,\{\psi_2\}}$$

HL-while $$\frac{\{\psi \wedge \mathtt{e} \neq 0\}\,\mathtt{s}\,\{\psi\}}{\{\psi\}\,\mathtt{while(e)}\,\mathtt{s}\,\{\psi \wedge \mathtt{e} = 0\}}$$

Figure 3.5: IMP axiomatic semantics

that we consider in this section. Note that we used a `typewriter` font to write these sets, which is consistent with our notation for variables in *PVar*. We need these disjointness restrictions because, Hoare logic makes no theoretical distinction between program and mathematical variables, while matching logic does. These restrictions do not limit the capability of Hoare logic, since we can always pick X to be the union of all the variables appearing in the program about which we want to reason and Y to be the union of all the remaining variables occurring in all the state specifications in any triple anywhere in the Hoare logic proof, making sure that the names of the variables used for stating mathematical properties of the state are always chosen different from those of the variables used in programs.

Definition 16. *Given a Hoare triple* $\{\psi\}\,\mathtt{code}\,\{\psi'\}$*, we define*

$$H2M(\{\psi\}\,\mathtt{code}\,\{\psi'\}) \quad \overset{def}{\equiv} \quad \exists X\,(\langle \mathtt{code},\ \sigma_X\rangle \wedge \psi_{X,Y}) \ \Rightarrow^{\exists}\ \exists X\,(\langle \mathtt{skip},\ \sigma_X\rangle \wedge \psi'_{X,Y})$$

where:

1. *$X, Y \subset Var$ (written using italic font) are finite sets of variables corresponding to the sets $\mathtt{X}, \mathtt{Y} \subset PVar$ fixed above, one variable x or y in Var (written using italic font) for each variable* x *or* y *in PVar (written using typewriter font);*

2. *σ_X is the state mapping each* $\mathtt{x} \in \mathtt{X}$ *to its corresponding* $x \in X$*; and*

3. *$\psi_{X,Y}$ and $\psi'_{X,Y}$ are ψ and respectively ψ' with* $\mathtt{x} \in \mathtt{X}$ *or* $\mathtt{y} \in \mathtt{Y}$ *replaced by its corresponding $x \in X$ or $y \in Y$, respectively, and each expression construct* `op` *replaced by its mathematical correspondent op_{Int}.*

The *H2M* mapping in Definition 16 is quite simple and mechanical, and can be implemented by a linear traversal of the Hoare triple. In fact, we have implemented it as part of the MatchC program verifier, to allow users to write program specifications in a Hoare style when possible (see Section 4.1.2).

It is important to note that, like $\mathtt{X}, \mathtt{Y} \subset PVar$, the sets of variables $X, Y \subset Var$ in Definition 16 are also fixed and thus the same for all Hoare triples considered in this section. For example, suppose that $\mathtt{X} = \{\mathtt{s}, \mathtt{n}\}$ and $\mathtt{Y} = \{\mathtt{oldn}, \mathtt{z}\}$. Then the Hoare triple

$$\{\mathtt{n} = \mathtt{oldn} \land \mathtt{n} \geq 0\}\,\mathtt{SUM}\,\{\mathtt{s} = \mathtt{oldn*(oldn+1)/2} \land \mathtt{n} = 0\}$$

from Section 3.4.1 is translated into the following reachability rule:

$$\begin{array}{ll} & \exists s, n\,(\langle \mathtt{SUM},\ (\mathtt{s} \mapsto s,\ \mathtt{n} \mapsto n)\rangle \land n = oldn \land n \geq_{Int} 0) \\ \Rightarrow^{\exists} & \exists s, n\,(\langle \mathtt{skip},\ (\mathtt{s} \mapsto s,\ \mathtt{n} \mapsto n)\rangle \land s = oldn *_{Int} (oldn +_{Int} 1) /_{Int} 2 \land n = 0) \end{array}$$

We also show an (artificial) example where the original Hoare triple contains a quantifier. Consider the same $\mathtt{X} = \{\mathtt{s}, \mathtt{n}\}$ and $\mathtt{Y} = \{\mathtt{oldn}, \mathtt{z}\}$ as above. Then

$$H2M(\{true\}\,\mathtt{n:=4*n+3}\,\{\exists \mathtt{z}\,(\mathtt{n} = \mathtt{2*z+1})\})$$

is the reachability rule

$$\begin{array}{ll} & \exists s, n\,(\langle \mathtt{n:=4*n+3},\ (\mathtt{s} \mapsto s, \mathtt{n} \mapsto n)\rangle \land true) \\ \Rightarrow^{\exists} & \exists s, n\,(\langle \mathtt{skip},\ (\mathtt{s} \mapsto s, \mathtt{n} \mapsto n)\rangle \land \exists z\,(n = 2 *_{Int} z +_{Int} 1)) \end{array}$$

Using FOL reasoning and Consequence, this rule is equivalent to

$$\exists s, n\,\langle \mathtt{n:=4*n+3},\ (\mathtt{s} \mapsto s, \mathtt{n} \mapsto n)\rangle \Rightarrow^{\exists} \exists s, z\,\langle \mathtt{skip},\ (\mathtt{s} \mapsto s, \mathtt{n} \mapsto 2 *_{Int} z +_{Int} 1)\rangle$$

Helping Lemmas

The following holds for matching logic in general:

Lemma 3. *If $\mathcal{S} \vdash \varphi \Rightarrow^{\exists} \varphi'$ is derivable then $\mathcal{S} \vdash \exists X\, \varphi \Rightarrow^{\exists} \exists X\, \varphi'$ is also derivable.*

Proof. We have $\models \varphi' \to \exists X\, \varphi'$. By Consequence, we derive $\mathcal{S} \vdash \varphi \Rightarrow^{\exists} \exists X\, \varphi'$. Since $X \cap \mathit{FreeVars}(\exists X\, \varphi') = \emptyset$, by Abstraction we get that $\mathcal{S} \vdash \exists X\, \varphi \Rightarrow^{\exists} \exists X\, \varphi'$ is also derivable. □

The following lemma states that symbolic evaluation of IMP expressions is actually formally derivable using the matching logic reachability proof system:

Lemma 4. *If $\mathtt{e} \in \mathit{Exp}$ is an expression, $C \in \mathit{Context}$ an appropriate context, and $\sigma \in \mathit{State}$ a state term binding each program variable in PVar of $\mathtt{e}$ to a term of sort Int (possibly containing variables in Var), then the following sequent is derivable:*

$$\mathcal{S}_{\mathrm{IMP}} \ \vdash\ \langle C,\ \sigma\rangle[\mathtt{e}] \Rightarrow^{\exists} \langle C,\ \sigma\rangle[\sigma(\mathtt{e})]$$

where $\sigma(\mathtt{e})$ replaces each $\mathtt{x} \in \mathit{PVar}$ in $\mathtt{e}$ by $\sigma(\mathtt{x})$ (i.e., a term of sort Int) and each operation symbol $\mathtt{op}$ by its mathematical correspondent in the Int domain, op_{Int}.

Proof. By induction on the structure of $\mathtt{e}$. If $\mathtt{e}$ is a variable $\mathtt{x} \in \mathit{PVar}$, then the result follows by Axiom with **lookup** in Figure 2.1. If $\mathtt{e}$ is of the form $\mathtt{e}_1\,\mathtt{op}\,\mathtt{e}_2$, then let C_1, C_2 be the contexts obtained from C by replacing $\square$ with "$\square\,\mathtt{op}\,\mathtt{e}_2$" and respectively "$\sigma(\mathtt{e}_1)\,\mathtt{op}\,\square$". Then, by the induction hypothesis, the following are derivable

$$\begin{aligned}\mathcal{S}_{\mathrm{IMP}} \ &\vdash\ \langle C_1,\ \sigma\rangle[\mathtt{e}_1] \Rightarrow^{\exists} \langle C_1,\ \sigma\rangle[\sigma(\mathtt{e}_1)]\\ \mathcal{S}_{\mathrm{IMP}} \ &\vdash\ \langle C_2,\ \sigma\rangle[\mathtt{e}_2] \Rightarrow^{\exists} \langle C_2,\ \sigma\rangle[\sigma(\mathtt{e}_2)]\end{aligned}$$

We also have the following pattern identities

$$\begin{aligned}\langle C,\ \sigma\rangle[\mathtt{e}] &= \langle C_1,\ \sigma\rangle[\mathtt{e}_1]\\ \langle C_1,\ \sigma\rangle[\sigma(\mathtt{e}_1)] &= \langle C_2,\ \sigma\rangle[\mathtt{e}_2]\\ \langle C_2,\ \sigma\rangle[\sigma(\mathtt{e}_2)] &= \langle C,\ \sigma\rangle[\sigma(\mathtt{e}_1)\,\mathtt{op}\,\sigma(\mathtt{e}_2)]\end{aligned}$$

Thus, by Transitivity, we derive

$$\mathcal{S}_{\mathrm{IMP}} \ \vdash\ \langle C,\ \sigma\rangle[\mathtt{e}] \Rightarrow^{\exists} \langle C,\ \sigma\rangle[\sigma(\mathtt{e}_1)\,\mathtt{op}\,\sigma(\mathtt{e}_2)]$$

and then the result follows by Axiom with $\mathtt{op}$ and by noticing the fact that $\sigma(\mathtt{e}) = \sigma(\mathtt{e}_1)\, op_{Int}\, \sigma(\mathtt{e}_2)$. □

Intuitively, the following lemma states that if we append some extra statement to the code of φ, then the execution of the original code is still possible, making abstraction of the appended statement. This holds because of the specific (simplistic) nature of IMP and may not hold in more complex languages (for example in ones with support for reflection or self-generation of code). A direct consequence is that we can (symbolically) execute a compound statement $\texttt{s}_1; \texttt{s}_2$ by first executing $\texttt{s}_1$ until we reach `skip` and then continuing from there with $\texttt{s}_2$.

Lemma 5. *If* $\mathcal{S}_{\text{IMP}} \vdash \varphi \Rightarrow^{\exists} \varphi'$ *is derivable and* $\texttt{s} \in \mathit{Stmt}$ *then*

$$\mathcal{S}_{\text{IMP}} \vdash \textsc{append}(\varphi, \texttt{s}) \Rightarrow^{\exists} \textsc{append}(\varphi', \texttt{s})$$

is also derivable, where $\textsc{append}(\varphi, \texttt{s})$ *is the pattern obtained from* φ *by replacing each basic pattern* $\langle \texttt{code}, \sigma\rangle$ *with the basic pattern* $\langle (\texttt{code}; \texttt{s}), \sigma\rangle$*.*

Proof. (sketch) Let $\textsc{append}(\mathcal{A}, \texttt{s})$ be the set of rules obtained from $\mathcal{A}$ by replacing each rule $\varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{A} \setminus \mathcal{S}_{\text{IMP}}$ by the rule $\textsc{append}(\varphi_l, \texttt{s}) \Rightarrow^{\exists} \textsc{append}(\varphi_r, \texttt{s})$, that is

$$\begin{aligned}&\textsc{append}(\mathcal{A}, \texttt{s})\\ &= (\mathcal{A} \cap \mathcal{S}_{\text{IMP}}) \cup \{\textsc{append}(\varphi_l, \texttt{s}) \Rightarrow^{\exists} \textsc{append}(\varphi_r, \texttt{s}) \mid \varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{A} \setminus \mathcal{S}_{\text{IMP}}\}\end{aligned}$$

Recall that $\mathcal{A} \setminus \mathcal{S}_{\text{IMP}}$ contains all the rules added by Circularity. Let $\mathcal{P}$ be a proof tree deriving $\mathcal{S}_{\text{IMP}} \vdash \varphi \Rightarrow^{\exists} \varphi'$. We prove the more general result that for each sequent $\mathcal{A} \vdash \varphi_l \Rightarrow^{\exists} \varphi_r$ in $\mathcal{P}$, we can also derive the sequent $\textsc{append}(\mathcal{A}, \texttt{s}) \vdash \textsc{append}(\varphi_l, \texttt{s}) \Rightarrow^{\exists} \textsc{append}(\varphi_r, \texttt{s})$. The lemma follows as particular case. The proof goes by induction on the structure of $\mathcal{P}$. If the last step is Reflexivity, the result trivially holds. If the last step is one of Substitution, Transitivity, Case Analysis, Logic Framing, Consequence, Abstraction or Circularity, then the result holds by applying the induction hypothesis, and by noticing that since s does not have any logical variables, then $\textsc{append}(\theta(\varphi), \texttt{s}) = \theta(\textsc{append}(\varphi, \texttt{s}))$ (Substitution), $\models \varphi_1 \rightarrow \varphi_1'$ iff $\models \textsc{append}(\varphi_1, \texttt{s}) \rightarrow \textsc{append}(\varphi_1', \texttt{s})$ (Consequence) and $\mathit{FreeVars}(\textsc{append}(\varphi, \texttt{s})) = \mathit{FreeVars}(\varphi)$ (Abstraction). If the last step is Axiom with a rule in $\mathcal{A} \setminus \mathcal{S}_{\text{IMP}}$, again the result trivially holds. If the last step is Axiom with a rule in $\mathcal{S}_{\text{IMP}}$, then the redex always goes to the left of "`;`", and we can conclude that $\varphi \Rightarrow^{\exists} \varphi' \in \mathcal{S}_{\text{IMP}}$ implies that $\textsc{append}(\varphi, \texttt{s}) \Rightarrow^{\exists} \textsc{append}(\varphi', \texttt{s}) \in \mathcal{S}_{\text{IMP}}$. □

The Main Result

Theorem 1 below states that, for the IMP language, any Hoare logic proof derivation of a Hoare triple $\{\psi\}\,\texttt{code}\,\{\psi'\}$ yields a matching logic reachability proof derivation of the corresponding reachability rule $H2M(\{\psi\}\,\mathsf{code}\,\{\psi'\})$. This proof correspondence is constructive and the resulting proof derivation is linear in the size of the original proof derivation. For example, to generate the matching logic reachability proof corresponding to a proof step using the Hoare logic proof rule for `while` loop, **HL-while**, we do the following:

1. We inductively assume a proof for the reachability rule corresponding to the Hoare triple for the `while` loop body;

2. We apply the Axiom step with **while** (Figure 2.1), followed by Substitution, Logic Framing, and Lemma 3, and this way we "unroll" the `while` loop into its corresponding conditional statement (in the logical context set by the Hoare triple);

3. Since the conditional statement contains the original `while` loop in its true branch and since 2. above does not use Reflexivity, we issue a Circularity proof obligation and thus add the claimed reachability rule for `while` to the set of axioms;

4. We "evaluate" symbolically the condition, by virtue of Lemma 4;

5. We apply a Case Analysis for the conditional, splitting the proof task in two subtasks, the one corresponding to the false condition being trivial to discharge;

6. To discharge the care corresponding to the true condition, we use the proof given by 1. by virtue of Lemma 5, then the Axiom for **seq**, and then the reachability rule added by Circularity and we are done.

Theorem 1. *Let $\mathcal{S}_{\text{IMP}}$ be the operational semantics of* IMP *in Figure 2.1 viewed as a matching logic reachability system, and let $\{\psi\}\,\mathsf{code}\,\{\psi'\}$ be a triple derivable with the* IMP*-specific Hoare logic proof system in Figure 3.5. Then we have that $\mathcal{S}_{\text{IMP}} \vdash H2M(\{\psi\}\,\mathsf{code}\,\{\psi'\})$ is derivable with the language-independent matching logic proof system in Figure 3.1.*

Proof. We prove that for any Hoare logic proof of $\{\psi\}\,\mathsf{code}\,\{\psi'\}$ one can construct a matching logic proof of $\mathcal{S}_{\text{IMP}} \vdash H2M(\{\psi\}\,\mathsf{code}\,\{\psi'\}))$. The proof goes by structural induction on the formal proof derived using the Hoare logic proof system in Figure 3.5. We consider each proof rule in Figure 3.5 and show how corresponding matching logic proofs for the hypotheses can be composed into a matching logic proof for the conclusion.

HL-skip $$\frac{\cdot}{\{\psi\}\,\texttt{skip}\,\{\psi\}}$$

Reflexivity (Figure 3.1) derives

$$\mathcal{S}_{\mathrm{IMP}} \vdash \exists X\,(\langle \texttt{skip},\ \sigma_X\rangle \wedge \psi_{X,Y}) \ \Rightarrow^{\exists}\ \exists X\,(\langle \texttt{skip},\ \sigma_X\rangle \wedge \psi_{X,Y})$$

and we are done.

HL-asgn $$\frac{\cdot}{\{\psi[\texttt{e}/\texttt{x}]\}\,\texttt{x} := \texttt{e}\,\{\psi\}}$$

We have to derive $\mathcal{S}_{\mathrm{IMP}} \vdash \exists X\,(\langle \texttt{x} := \texttt{e},\ \sigma_X\rangle \wedge \psi[\texttt{e}/\texttt{x}]_{X,Y}) \ \Rightarrow^{\exists}\ \exists X\,(\langle \texttt{skip},\ \sigma_X\rangle \wedge \psi_{X,Y})$. By using Lemma 4, Logical Framing and Lemma 3, we derive

$$\mathcal{S}_{\mathrm{IMP}} \vdash \exists X\,(\langle \texttt{x} := \texttt{e},\ \sigma_X\rangle \wedge \psi[\texttt{e}/\texttt{x}]_{X,Y}) \ \Rightarrow^{\exists}\ \exists X\,(\langle \texttt{x} := \sigma_{\texttt{X}}(\texttt{e}),\ \sigma_X\rangle \wedge \psi[\texttt{e}/\texttt{x}]_{X,Y})$$

Further, by using Axiom with **asgn** in Figure 2.1, Substitution and Logic Framing, followed by Lemma 3, we derive

$$\mathcal{S}_{\mathrm{IMP}} \vdash$$
$$\exists X\,(\langle \texttt{x} := \sigma_{\texttt{X}}(\texttt{e}),\ \sigma_X\rangle \wedge \psi[\texttt{e}/\texttt{x}]_{X,Y}) \Rightarrow^{\exists} \exists X\,(\langle \texttt{skip},\ \sigma_X[\texttt{x} \leftarrow \sigma_X(\texttt{e})]\rangle \wedge \psi[\texttt{e}/\texttt{x}]_{X,Y})$$

Then, the result follows by Transitivity with the rules above and by Consequence with

$$\models \exists X\,(\langle \texttt{skip},\ \sigma_X[\texttt{x} \leftarrow \sigma_X(\texttt{e})]\rangle \wedge \psi[\texttt{e}/\texttt{x}]_{X,Y}) \to \exists X\,(\langle \texttt{skip},\ \sigma_X\rangle \wedge \psi_{X,Y}),$$

which holds because $\sigma_X[\texttt{x} \leftarrow \sigma_X(\texttt{e})]$ and $\psi[\texttt{e}/\texttt{x}]_{X,Y}$ are nothing but σ_X and respectively $\psi_{X,Y}$ with $x \in X$ replaced by $\sigma_X(\texttt{e})$.

HL-seq $$\frac{\{\psi_1\}\,\texttt{s}_1\,\{\psi_2\} \qquad \{\psi_2\}\,\texttt{s}_2\,\{\psi_3\}}{\{\psi_1\}\,\texttt{s}_1;\ \texttt{s}_2\,\{\psi_3\}}$$

We have to derive $\mathcal{S}_{\mathrm{IMP}} \vdash \exists X\,(\langle \texttt{s}_1;\ \texttt{s}_2,\ \sigma_X\rangle \wedge \psi_{1X,Y}) \ \Rightarrow^{\exists}\ \exists X\,(\langle \texttt{skip},\ \sigma_X\rangle \wedge \psi_{3X,Y})$. By the induction hypothesis, the following sequents are derivable

$$\mathcal{S}_{\mathrm{IMP}} \ \vdash\ \exists X\,(\langle \texttt{s}_1,\ \sigma_X\rangle \wedge \psi_{1X,Y}) \ \Rightarrow^{\exists}\ \exists X\,(\langle \texttt{skip},\ \sigma_X\rangle \wedge \psi_{2X,Y})$$
$$\mathcal{S}_{\mathrm{IMP}} \ \vdash\ \exists X\,(\langle \texttt{s}_2,\ \sigma_X\rangle \wedge \psi_{2X,Y}) \ \Rightarrow^{\exists}\ \exists X\,(\langle \texttt{skip},\ \sigma_X\rangle \wedge \psi_{3X,Y})$$

By applying Lemma 5 with the former rule, we derive

$$\mathcal{S}_{\text{IMP}} \ \vdash \ \exists X\, (\langle \mathtt{s}_1; \mathtt{s}_2,\ \sigma_X\rangle \wedge \psi_{1X,Y}) \ \Rightarrow^{\exists} \ \exists X\, (\langle \mathtt{skip}; \mathtt{s}_2,\ \sigma_X\rangle \wedge \psi_{2X,Y})$$

Further, Axiom with **seq** (Figure 2.1), Substitution and Logic Framing, followed by Lemma 3, imply $\mathcal{S}_{\text{IMP}} \ \vdash \ \exists X\, (\langle \mathtt{s}_1; \mathtt{s}_2,\ \sigma_X\rangle \wedge \psi_{1X,Y}) \ \Rightarrow^{\exists} \ \exists X\, (\langle \mathtt{s}_2,\ \sigma_X\rangle \wedge \psi_{2X,Y})$. Then, the result follows by Transitivity with the rule above and the second induction hypothesis.

HL-cond $$\frac{\{\psi_1 \wedge \mathtt{e} \neq 0\}\, \mathtt{s}_1\, \{\psi_2\} \qquad \{\psi_1 \wedge \mathtt{e} = 0\}\, \mathtt{s}_2\, \{\psi_2\}}{\{\psi_1\}\, \mathtt{if(e)}\, \mathtt{s}_1\, \mathtt{else}\, \mathtt{s}_2\, \{\psi_2\}}$$

We have to derive

$$\mathcal{S}_{\text{IMP}} \ \vdash \ \exists X\, (\langle \mathtt{if(e)}\, \mathtt{s}_1\, \mathtt{else}\, \mathtt{s}_2,\ \sigma_X\rangle \wedge \psi_{1X,Y}) \ \Rightarrow^{\exists} \ \exists X\, (\langle \mathtt{skip},\ \sigma_X\rangle \wedge \psi_{2X,Y})$$

By the induction hypothesis, the following sequents are derivable

$$\begin{aligned}
\mathcal{S}_{\text{IMP}} \ &\vdash \ \exists X\, (\langle \mathtt{s}_1,\ \sigma_X\rangle \wedge (\psi_1 \wedge \mathtt{e} \neq 0)_{X,Y}) \ \Rightarrow^{\exists} \ \exists X\, (\langle \mathtt{skip},\ \sigma_X\rangle \wedge \psi_{2X,Y})\\
\mathcal{S}_{\text{IMP}} \ &\vdash \ \exists X\, (\langle \mathtt{s}_2,\ \sigma_X\rangle \wedge (\psi_1 \wedge \mathtt{e} = 0)_{X,Y}) \ \Rightarrow^{\exists} \ \exists X\, (\langle \mathtt{skip},\ \sigma_X\rangle \wedge \psi_{2X,Y})
\end{aligned}$$

By using Lemma 4, Logical Framing, and Lemma 3, we derive

$$\begin{aligned}
\mathcal{S}_{\text{IMP}} \quad \vdash \quad & \exists X\, (\langle \mathtt{if(e)}\, \mathtt{s}_1\, \mathtt{else}\, \mathtt{s}_2,\ \sigma_X\rangle \wedge \psi_{1X,Y})\\
& \Rightarrow^{\exists} \ \exists X\, (\langle \mathtt{if}(\sigma_X(\mathtt{e}))\, \mathtt{s}_1\, \mathtt{else}\, \mathtt{s}_2,\ \sigma_X\rangle \wedge \psi_{1X,Y})
\end{aligned}$$

By using Axiom with **cond₁** and **cond₂** in Figure 2.1, each followed by Substitution, Logic Framing and by Lemma 3, we also derive

$$\begin{aligned}
\mathcal{S}_{\text{IMP}} \quad \vdash \quad & \exists X\, (\langle \mathtt{if}(\sigma_X(\mathtt{e}))\, \mathtt{s}_1\, \mathtt{else}\, \mathtt{s}_2,\ \sigma_X\rangle \wedge (\psi_1 \wedge \mathtt{e} \neq 0)_{X,Y})\\
& \Rightarrow^{\exists} \ \exists X\, (\langle \mathtt{s}_1,\ \sigma_X\rangle \wedge (\psi_1 \wedge \mathtt{e} \neq 0)_{X,Y})\\
\mathcal{S}_{\text{IMP}} \quad \vdash \quad & \exists X\, (\langle \mathtt{if}(\sigma_X(\mathtt{e}))\, \mathtt{s}_1\, \mathtt{else}\, \mathtt{s}_2,\ \sigma_X\rangle \wedge (\psi_1 \wedge \mathtt{e} = 0)_{X,Y})\\
& \Rightarrow^{\exists} \ \exists X\, (\langle \mathtt{s}_2,\ \sigma_X\rangle \wedge (\psi_1 \wedge \mathtt{e} = 0)_{X,Y})
\end{aligned}$$

Further, by Transitivity with the rules above and the induction hypotheses, we derive

$$\begin{aligned}
\mathcal{S}_{\text{IMP}} \quad \vdash \quad & \exists X\, (\langle \mathtt{if}(\sigma_X(\mathtt{e}))\, \mathtt{s}_1\, \mathtt{else}\, \mathtt{s}_2,\ \sigma_X\rangle \wedge (\psi_1 \wedge \mathtt{e} \neq 0)_{X,Y})\\
& \Rightarrow^{\exists} \ \exists X\, (\langle \mathtt{skip},\ \sigma_X\rangle \wedge \psi_{2X,Y})\\
\mathcal{S}_{\text{IMP}} \quad \vdash \quad & \exists X\, (\langle \mathtt{if}(\sigma_X(\mathtt{e}))\, \mathtt{s}_1\, \mathtt{else}\, \mathtt{s}_2,\ \sigma_X\rangle \wedge (\psi_1 \wedge \mathtt{e} = 0)_{X,Y})\\
& \Rightarrow^{\exists} \ \exists X\, (\langle \mathtt{skip},\ \sigma_X\rangle \wedge \psi_{2X,Y})
\end{aligned}$$

Then the result follows by Case Analysis, Consequence and Transitivity.

HL-while $$\frac{\{\psi \wedge \mathtt{e} \neq 0\}\,\mathtt{s}\,\{\psi\}}{\{\psi\}\,\mathtt{while(e)}\,\mathtt{s}\,\{\psi \wedge \mathtt{e} = 0\}}$$

Let μ be the matching logic rule that we have to derive, namely

$$\mathcal{S}_{\mathrm{IMP}} \vdash \exists X\,(\langle \mathtt{while(e)\,s},\ \sigma_X\rangle \wedge \psi_{X,Y}) \Rightarrow^{\exists} \exists X\,(\langle \mathtt{skip},\ \sigma_X\rangle \wedge (\psi \wedge \mathtt{e} = 0)_{X,Y})$$

By the induction hypothesis, the following sequent is derivable

$$\mathcal{S}_{\mathrm{IMP}} \vdash \exists X\,(\langle \mathtt{s},\ \sigma_X\rangle \wedge (\psi \wedge \mathtt{e} \neq 0)_{X,Y}) \Rightarrow^{\exists} \exists X\,(\langle \mathtt{skip},\ \sigma_X\rangle \wedge \psi_{X,Y})$$

We derive μ by Circularity. First, by Axiom with **while** (Figure 2.1), Substitution, Logic Framing, and Lemma 3, we derive (note the $\Rightarrow^{+}$, as this derivation does not use Reflexivity)

$$\begin{aligned}\mathcal{S}_{\mathrm{IMP}} \quad \vdash \quad & \exists X\,(\langle \mathtt{while(e)\,s},\ \sigma_X\rangle \wedge \psi_{X,Y})\\ & \Rightarrow^{+} \exists X\,(\langle \mathtt{if(e)\,s;\ while(e)\,s\,else\,skip},\ \sigma_X\rangle \wedge \psi_{X,Y})\end{aligned}$$

Therefore, all we need to do now is to derive

$$\begin{aligned}\mathcal{S}_{\mathrm{IMP}} \cup \{\mu\} \quad \vdash \quad & \exists X\,(\langle \mathtt{if(e)\,s;\ while(e)\,s\,else\,skip},\ \sigma_X\rangle \wedge \psi_{X,Y})\\ & \Rightarrow^{\exists} \exists X\,(\langle \mathtt{skip},\ \sigma_X\rangle \wedge (\psi \wedge \mathtt{e} = 0)_{X,Y})\end{aligned}$$

Further, by Lemma 4, Logical Framing, Lemma 3 and Transitivity, we are left with

$$\begin{aligned}\mathcal{S}_{\mathrm{IMP}} \cup \{\mu\} \quad \vdash \quad & \exists X\,(\langle \mathtt{if}(\sigma_{\mathtt{X}}(\mathtt{e}))\,\mathtt{s;\ while(e)\,s\,else\,skip},\ \sigma_X\rangle \wedge \psi_{X,Y})\\ & \Rightarrow^{\exists} \exists X\,(\langle \mathtt{skip},\ \sigma_X\rangle \wedge (\psi \wedge \mathtt{e} = 0)_{X,Y})\end{aligned}$$

We apply Case Analysis with $\sigma_X(\mathtt{e}) = 0 \vee \sigma_X(\mathtt{e}) \neq 0$. The case $\sigma_X(\mathtt{e}) = 0$ follows by Axiom with **cond$_2$**, Substitution, Logic Framing and Lemma 3. By Axiom with **cond$_1$**, Substitution, Logic Framing, Lemma 3 and Transitivity, the other case reduces to

$$\begin{aligned}\mathcal{S}_{\mathrm{IMP}} \cup \{\mu\} \quad \vdash \quad & \exists X\,(\langle \mathtt{s;\ while(e)\,s},\ \sigma_X\rangle \wedge (\psi \wedge \mathtt{e} \neq 0)_{X,Y})\\ & \Rightarrow^{\exists} \exists X\,(\langle \mathtt{skip},\ \sigma_X\rangle \wedge (\psi \wedge \mathtt{e} = 0)_{X,Y})\end{aligned}$$

By using the induction hypothesis and Lemma 5 with `s` and `while(e) s` followed by Axiom with **skip**, Substitution, Logic Framing and Lemma 3 we derive

$$\begin{aligned}\mathcal{S}_{\text{IMP}} \cup \{\mu\} \quad \vdash \quad & \exists X\,(\langle \mathtt{s};\ \mathtt{while(e)\,s},\ \sigma_X\rangle \wedge (\psi \wedge \mathtt{e} \neq 0)_{X,Y}) \\ & \Rightarrow^{\exists}\ \exists X\,(\langle \mathtt{while(e)\,s},\ \sigma_X\rangle \wedge \psi_{X,Y})\end{aligned}$$

Then the result follows by using Axiom with μ and Transitivity with the rule above. □

Theorem 1 thus tells us that anything that can be proved using Hoare logic can also be proved using the matching logic reachability proof system. Furthermore, it gives us a novel way to prove soundness of Hoare logic proof systems, where the low-level details of the transition system corresponding to the target programming language, including induction on path length, are totally avoided and replaced by an abstract, small and fixed proof system, which is sound for all languages.

3.4.3 Adding Recursion

In this section we add procedures to IMP, which can be mutually recursive, and show that proof derivations done with the corresponding Hoare logic proof rule can also be done using the generic matching logic proof system, with the straightforward operational semantics rule as an axiom. We consider the following syntax for procedures:

$$\begin{aligned}&\mathit{ProcedureName} ::= \mathtt{proc} \mid \ldots \\ &\mathit{Procedure} ::= \mathit{ProcedureName}()\,\mathit{Stmt} \\ &\mathit{Stmt} ::= \ldots \mid \mathit{ProcedureName}()\end{aligned}$$

Our procedures therefore have the syntax "`proc() body`", where `proc` is the name of the procedure and `body` the body statement. Procedure invocations are statements of the form "`proc()`". For simplicity, and to capture the essence of the relationship between recursion and the Circularity rule of matching logic, we assume only no-argument procedures.

The operational semantics of procedure calls is trivial:

$$\textbf{call} \quad \mathtt{proc()} \Rightarrow^{\exists} \mathtt{body} \qquad \text{where "}\mathtt{proc()\,body}\text{" is a procedure}$$

The Hoare logic proof rule needs to take into account that procedures may be recursive:

$$\frac{\mathcal{H} \cup \{\psi\}\,\mathtt{proc()}\,\{\psi'\} \vdash \{\psi\}\,\mathtt{body}\,\{\psi'\}}{\mathcal{H} \vdash \{\psi\}\,\mathtt{proc()}\,\{\psi'\}} \quad \text{where "}\mathtt{proc()\ body}\text{" is a procedure}$$

This rule states that if the body of a procedure is proved to satisfy its contract while assuming that the procedure itself satisfies it, then the procedure's contract is indeed valid. If one has more mutually recursive procedures, then one needs to apply this rule several times until all procedure contracts are added to the hypothesis $\mathcal{H}$, and then each procedure body proved. The rule above needs to be added to the Hoare logic proof system in Figure 3.5, but in order for that to make sense we need to first replace each Hoare triple $\{\psi\}\,\mathtt{code}\,\{\psi'\}$ in Figure 3.5 by a sequent "$\mathcal{H} \vdash \{\psi\}\,\mathtt{code}\,\{\psi'\}$".

Theorem 2. *Let $\mathcal{S}_{\text{IMP}}$ be the operational semantics of* IMP *in Figure 2.1 extended with the rule* **call** *for procedure calls above, and let $\mathcal{H} \vdash \{\psi\}\,\mathtt{code}\,\{\psi'\}$ be a sequent derivable with the extended Hoare logic proof system. Then we have that $\mathcal{S}_{\text{IMP}} \cup H2M(\mathcal{H}) \vdash H2M(\{\psi\}\,\mathtt{code}\,\{\psi'\})$ is derivable with the matching logic reachability proof system in Figure 3.1.*

Proof. Like in Theorem 1, we prove by structural induction that for any Hoare logic proof of $\mathcal{H} \vdash \{\psi\}\,\mathtt{code}\,\{\psi'\}$ one can construct a matching logic proof of $\mathcal{S}_{\text{IMP}} \cup H2M(\mathcal{H}) \vdash H2M(\{\psi\}\,\mathtt{code}\,\{\psi'\}))$, by showing for each Hoare logic proof rule how corresponding matching logic proofs for the hypotheses can be composed into a matching logic proof for the conclusion. The proofs for the (extended) Hoare rules in Figure 3.5 are similar to those in Theorem 1, so we only discuss the new Hoare rule for procedure calls:

$$\frac{\mathcal{H} \cup \{\psi\}\,\mathtt{proc()}\,\{\psi'\} \vdash \{\psi\}\,\mathtt{body}\,\{\psi'\}}{\mathcal{H} \vdash \{\psi\}\,\mathtt{proc()}\,\{\psi'\}}$$

Let μ be the matching logic reachability rule $H2M(\{\psi\}\,\mathtt{proc()}\,\{\psi'\})$, that is,

$$\exists X\,(\langle \mathtt{proc()},\ \sigma_X\rangle \wedge \psi_{X,Y}) \ \Rightarrow^{\exists}\ \exists X\,(\langle \mathtt{skip},\ \sigma_X\rangle \wedge \psi'_{X,Y}).$$

The induction hypothesis gives us that the matching logic sequent

$$\mathcal{S}_{\text{IMP}} \cup H2M(\mathcal{H}) \cup \{\mu\} \vdash \exists X\,(\langle \mathtt{body},\ \sigma_X\rangle \wedge \psi_{X,Y}) \ \Rightarrow^{\exists}\ \exists X\,(\langle \mathtt{skip},\ \sigma_X\rangle \wedge \psi'_{X,Y})$$

is derivable with the generic proof system in Figure 3.1. Using Axiom with **call**, Logic Framing with $\psi_{X,Y}$, and then Lemma 3, we derive (note the $\Rightarrow^+$, as this derivation does not use Reflexivity):

$$\mathcal{S}_{\text{IMP}} \cup H2M(\mathcal{H}) \ \vdash \ \exists X\, (\langle \texttt{proc}(),\ \sigma_X\rangle \wedge \psi_{X,Y}) \Rightarrow^+ \ \exists X\, (\langle \texttt{body},\ \sigma_X\rangle \wedge \psi_{X,Y})$$

Circularity with the two rules above now derives $\mathcal{S}_{\text{IMP}} \cup H2M(\mathcal{H}) \ \vdash \ \mu$. □

3.5 Relationship with Separation Logic

Separation logic [73, 85] 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 in Chapter 4 we show many such properties and thus we investigate here the 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 [73], 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 [73, 85], 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:

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

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 $\bot$ in $\mathcal{T}$ standing for "error". All operations with at least one argument $\bot$ evaluate to $\bot$. Equality between $\bot$ and any other element does not hold. Valuations do not take any variables into $\bot$. The model of heaps has the important property that the concatenation of two maps with non-disjoint domains is $\bot$. 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:

- $\mathit{S2M}(\forall x P)$: let $\mathit{S2M}(P)$ be $\exists\sigma(\langle\sigma\rangle_{\mathsf{heap}} \wedge \psi)$. Then

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

- $\mathit{S2M}(P_1 \to P_2)$: let $\mathit{S2M}(P_1)$ be $\exists\sigma_1(\langle\sigma_1\rangle_{\mathsf{heap}} \wedge \psi_1)$ and similarly let $\mathit{S2M}(P_2)$

be $\exists\sigma_2(\langle\sigma_2\rangle_{\mathsf{heap}} \wedge \psi_2)$. Then we define

$$\begin{aligned}&S2M(P_1 \to P_2)\\ &\equiv \exists\sigma(\langle\sigma\rangle_{\mathsf{heap}} \wedge \exists\sigma_1\exists\sigma_2(\sigma = \sigma_1 \wedge \sigma = \sigma_2 \wedge (\psi_1 \to \psi_2)))\end{aligned}$$

Notice that the equalities $\sigma = \sigma_1$ and $\sigma = \sigma_2$ ensure that ψ_1 and ψ_2 constrain the same heap.

- $S2M(p(t_1, ..., t_n))$, where p is a "pure" predicate (one which is not interpreted over the heap, like "=" or "≤"):

  $$S2M(p(t_1, ..., t_n)) \equiv \exists\sigma(\langle\sigma\rangle_{\mathsf{heap}} \wedge p(t_1, ..., t_n))$$

  We used the same notation for the pure predicate and the corresponding Boolean algebraic operator.

- $S2M(\mathit{false})$: we define

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

- $S2M(P_1 * P_2)$: let $S2M(P_1)$ be $\exists\sigma_1(\langle\sigma_1\rangle_{\mathsf{heap}} \wedge \psi_1)$ and similarly let $S2M(P_2)$ be $\exists\sigma_2(\langle\sigma_2\rangle_{\mathsf{heap}} \wedge \psi_2)$. Then we define

  $$\begin{aligned}&S2M(P_1 * P_2)\\ &\equiv \exists\sigma(\langle\sigma\rangle_{\mathsf{heap}} \wedge \exists\sigma_1\exists\sigma_2(\sigma = (\sigma_1, \sigma_2) \wedge \psi_1 \wedge \psi_2))\end{aligned}$$

  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.

- $S2M(x \mapsto y)$: we define

  $$S2M(t_1 \mapsto t_2) \equiv \exists\sigma(\langle\sigma\rangle_{\mathsf{heap}} \wedge \sigma = (t_1 \mapsto t_2))$$

  We use the same notation for the separation logic predicate $t_1 \mapsto t_2$ and the algebraic map constructor $t_1 \mapsto t_2$.

- $S2M(\texttt{emp})$: we define (recall that $\cdot$ is the map unit)

  $$S2M(\texttt{emp}) \equiv \exists\sigma(\langle\sigma\rangle_{\mathsf{heap}} \wedge \sigma = \cdot)$$

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

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

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

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

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

Proof. Let $\rho[\sigma \leftarrow h]$ be the valuation which agrees with ρ on $Var \setminus \{\sigma\}$ and with $\rho[\sigma \leftarrow h](\sigma) = h$. We begin by noticing that $(\langle h\rangle_{\mathsf{heap}}, \rho) \models \exists\sigma(\langle\sigma\rangle_{\mathsf{heap}} \land \psi)$ iff $\rho[\sigma \leftarrow h] \models \psi$. Indeed, $(\langle h\rangle_{\mathsf{heap}}, \rho) \models \exists\sigma(\langle\sigma\rangle_{\mathsf{heap}} \land \psi)$ iff there exists a ρ' agreeing with ρ on $Var \setminus \{\sigma\}$ such that $(\langle h\rangle_{\mathsf{heap}}, \rho') \models \langle\sigma\rangle_{\mathsf{heap}}$ and $(\langle h\rangle_{\mathsf{heap}}, \rho') \models \psi$. By the satisfaction of matching logic formulae, $(\langle h\rangle_{\mathsf{heap}}, \rho') \models \langle\sigma\rangle_{\mathsf{heap}}$ iff $\rho'(\sigma) = h$, that is, iff $\rho' = \rho[\sigma \leftarrow h]$. Further, since ψ is a patternless formula, $(\langle h\rangle_{\mathsf{heap}}, \rho') \models \psi$ iff $\rho' \models \psi$. We conclude that such a ρ' exists iff $\rho[\sigma \leftarrow h] \models \psi$. Thus, to prove the proposition, it suffices to show that $(\overline{\rho}, h) \models P$ iff $\rho[\sigma \leftarrow h] \models \psi$, where ψ is the patternless formula in $S2M(P)$. The proof goes by induction on the structure of P. We distinguish the following cases:

- $\forall xP$: let ρ' be a valuation which agrees with ρ on $Var \setminus \{x\}$. Then, by the satisfaction of separation logic formulae, $(\overline{\rho}, h) \models \forall xP$ iff $(\overline{\rho}', h) \models P$. By the induction hypothesis, $(\overline{\rho}', h) \models P$ iff $\rho'[\sigma \leftarrow h] \models \psi$. Since σ is different from x, $\rho'[\sigma \leftarrow h] \models \psi$ iff $\rho[\sigma \leftarrow h] \models \forall x\, \psi$, and we are done.

- $P_1 \to P_2$: by the satisfaction of separation logic formulae, $(\overline{\rho}, h) \models P_1 \to P_2$

iff $(\overline{\rho}, h) \models P_1$ implies $(\overline{\rho}, h) \models P_2$. By the induction hypothesis, $(\overline{\rho}, h) \models P_1$ iff $\rho[\sigma_1 \leftarrow h] \models \psi_1$ and $(\overline{\rho}, h) \models P_2$ iff $\rho[\sigma_2 \leftarrow h] \models \psi_2$. Since σ does not occur in ψ_1 or ψ_2, σ_1 does not occur in ψ_2 and σ_2 does not occur in ψ_1, we have that $\rho[\sigma_1 \leftarrow h] \models \psi_1$ iff $\rho[\sigma \leftarrow h][\sigma_1 \leftarrow h][\sigma_2 \leftarrow h] \models \psi_1$ and $\rho[\sigma_2 \leftarrow h] \models \psi_2$ iff $\rho[\sigma \leftarrow h][\sigma_1 \leftarrow h][\sigma_2 \leftarrow h] \models \psi_2$. Thus, we conclude that $\rho[\sigma_1 \leftarrow h] \models \psi_1$ implies $\rho[\sigma_2 \leftarrow h] \models \psi_2$ iff $\rho[\sigma \leftarrow h] \models \exists\sigma_1\exists\sigma_2(\sigma = \sigma_1 \wedge \sigma = \sigma_2 \wedge \psi_1 \wedge \psi_2)$, and we are done.

- $p(t_1, ..., t_n)$: since p is pure and σ does not occur in $p(t_1, ..., t_n)$, we have that $(\overline{\rho}, h) \models p(t_1, ..., t_n)$ iff $\overline{\rho} \models p(t_1, ..., t_n)$ iff $\rho \models p(t_1, ..., t_n)$ iff $\rho[\sigma \leftarrow h] \models p(t_1, ..., t_n)$ and we are done.

- *false*: we notice that $(\overline{\rho}, h) \not\models \mathit{false}$ and $\rho[\sigma \leftarrow h] \not\models \mathit{false}$, and we are done.

- $P_1 * P_2$: by the satisfaction of separation logic formulae, $(\overline{\rho}, h) \models P_1 * P_2$ iff there exist mapping h_1, h_2 with disjoint domains such that h is the disjoint union of h_1 and h_2, $(\overline{\rho}, h_1) \models P_1$ and $(\overline{\rho}, h_2) \models P_1$. By the induction hypothesis, $(\overline{\rho}, h_1) \models P_1$ iff $\rho[\sigma_1 \leftarrow h_1] \models \psi_1$ and $(\overline{\rho}, h_2) \models P_2$ iff $\rho[\sigma_2 \leftarrow h_2] \models \psi_2$. Since σ does not occur in ψ_1 or ψ_2, σ_1 does not occur in ψ_2 and σ_2 does not occur in ψ_1, we have that $\rho[\sigma_1 \leftarrow h_1] \models \psi_1$ iff $\rho[\sigma \leftarrow h][\sigma_1 \leftarrow h_1][\sigma_2 \leftarrow h_2] \models \psi_1$ and $\rho[\sigma_2 \leftarrow h_2] \models \psi_2$ iff $\rho[\sigma \leftarrow h][\sigma_1 \leftarrow h_1][\sigma_2 \leftarrow h_2] \models \psi_2$. Notice that $\rho[\sigma \leftarrow h][\sigma_1 \leftarrow h_1][\sigma_2 \leftarrow h_2] \models \sigma = (\sigma_1, \sigma_2)$ iff h is the concatenation of h_1 and h_2 and the concatenation is different from $\perp$, that is, if h_1 and h_2 have disjoint domains. Thus, we conclude that $(\overline{\rho}, h) \models P_1 * P_2$ iff $\rho[\sigma \leftarrow h] \models \exists\sigma_1\exists\sigma_2(\sigma = (\sigma_1, \sigma_2) \wedge \psi_1 \wedge \psi_2)$, and we are done.

- $t_1 \mapsto t_2$: by the satisfaction of separation logic formulae, $(\overline{\rho}, h) \models t_1 \mapsto t_2$ iff h is the singleton heap mapping $\overline{\rho}(t_1)$ into $\overline{\rho}(t_2)$. On the other hand, $\rho[\sigma \leftarrow h] \models \sigma = t_1 \mapsto t_2$ iff h consists of exactly one entry mapping $\rho(t_1)$ into $\rho(t_2)$, and we are done.

- `emp`: by the satisfaction of separation logic formulae, $(\overline{\rho}, h) \models$ `emp` iff h is the empty heap, or equivalently, the map unit. On the other hand, $\rho[\sigma \leftarrow h] \models \sigma = \cdot$ iff h is the map unit, and we are done.

□

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.

3.6 Soundness

Soundness states that a syntactically derivable sequent holds semantically. First we prove the soundness the proof system in Figure 3.1 and then the soundness of the proof system in Figure 3.3. Note that, unlike the soundness of Hoare logic which is shown for each language separately, the soundness of reachability logic is proved only once, for all languages. Because of the utmost importance of the result below, we have also mechanized its proof in Coq `http://fsl.cs.uiuc.edu/RL`.

3.6.1 All-Path and One-Path Reachability Logic

Here we prove the soundness of the proof system in Figure 3.1.

Theorem 3 (**Soundness**). *If $\mathcal{S} \vdash \varphi \Rightarrow^{Q} \varphi'$ then $\mathcal{S} \models \varphi \Rightarrow^{Q} \varphi'$ (for $Q \in \{\exists, \forall\}$).*

Proof. Follows from Lemma 6 and Proposition 5. □

Definition 17. *Let $\mathcal{S}$ be a reachability system, $\varphi \Rightarrow^{Q} \varphi'$ (with $Q \in \{\forall, \exists\}$) a reachability rule and $n \in \mathbb{N}$ a natural number.*

- *We write $\mathcal{S} \models_n^* \varphi \Rightarrow^{\exists} \varphi'$ (respectively $\mathcal{S} \models_n^+ \varphi \Rightarrow^{\exists} \varphi'$) when for any γ and ρ such that $(\gamma, \rho) \models \varphi$ and γ terminates in strictly less than n steps in $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$, we have that there exists γ' such that $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$ (respectively $\gamma \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma'$) and $(\gamma', \rho) \models \varphi'$.*

- *We write $\mathcal{S} \models_n^* \varphi \Rightarrow^\forall \varphi'$ (respectively $\mathcal{S} \models_n^+ \varphi \Rightarrow^\forall \varphi'$) when for any complete path $\tau = \gamma_1 ... \gamma_k$ of length $k \leq n$ and for any $\rho : Var \to \mathcal{T}$ such that $(\gamma_1, \rho) \models \varphi$ there exists $i \in \{1, ..., n\}$ (resp. $i \in \{2, ..., n\}$) such that $(\gamma_i, \rho) \models \varphi$.*

We extend the previous notations to sets of formulae: for any $\delta \in \{'+', '*'\}$, we write $S \models_n^\delta D$ to mean $S \models_n^\delta \varphi \Rightarrow^Q \varphi'$ for all $\varphi \Rightarrow^Q \varphi' \in D$.

If C is a set of reachability rules, we write "Δ_C" for "+" when C is not empty and for "$*$" when C is empty. Therefore, the relation $\models_n^{\Delta_C}$ should be understood as $\models_n^*$ when C is empty and $\models_n^+$ when C is not empty.

Proposition 5. *We have that $\mathcal{S} \models \varphi \Rightarrow^Q \varphi'$ iff for all n, $\mathcal{S} \models_n^* \varphi \Rightarrow^Q \varphi'$.*

Proof. By case analysis. □

Lemma 6. *For any derivation tree, for any sets $\mathcal{A}$ and C of reachability rules, if the sequent $\mathcal{S}, \mathcal{A} \vdash_C \varphi \Rightarrow^Q \varphi'$ is the last sequent in the tree then for any natural number $n \in \mathbb{N} \setminus \{0\}$, if $\mathcal{S} \models_n^+ \mathcal{A}$ and $\mathcal{S} \models_{n-1}^+ C$, then $\mathcal{S} \models_n^{\Delta_C} \varphi \Rightarrow^Q \varphi'$.*

Proof. By induction on the proof tree for $\mathcal{S}, \mathcal{A} \vdash_C \varphi \Rightarrow^Q \varphi'$:

1. Step.

   If the last rule in the proof tree is Step, then Q must be $\forall$.

   Let $n \in \mathbb{N} \in \{0\}$ be an arbitrary positive natural number such that $\mathcal{S} \models_n^+ \mathcal{A}$ and $\mathcal{S} \models_{n-1}^+ C$. We assume that $\models \varphi \to \bigvee_{\varphi_l \Rightarrow^\exists \varphi_r \in \mathcal{S}} \exists FreeVars(\varphi_l)\varphi_l$, that $\models \exists c\ (\varphi[c/\square] \wedge \varphi_l[c/\square]) \wedge \varphi_r \to \varphi'$ for each $\varphi_l \Rightarrow^\exists \varphi_r \in \mathcal{S}$ and we show that $\mathcal{S} \models_n^{\Delta_C} \varphi \Rightarrow^\forall \varphi'$.

   Let $\tau = \gamma_1 ... \gamma_k$ be a complete path of length $k \leq n$ and let $\rho : Var \to \mathcal{T}$ be a valuation such that $(\gamma_1, \rho) \models \varphi$. We show that there exists $i \in \{1, ..., n\}$ such that $(\gamma_i, \rho) \models \varphi$.

   As we have $\models \varphi \to \bigvee_{\varphi_l \Rightarrow^\exists \varphi_r \in \mathcal{S}} \exists FreeVars(\varphi_l)\varphi_l$ it follows that $\tau = \gamma_1$ is not a complete $(\mathcal{T}, \Rightarrow_\mathcal{S}^\mathcal{T})$-path and therefore $n \neq 1$. Therefore $i = 2 \in \{1, ..., n\}$ and $\gamma_2 \in \tau$. We will show that $(\gamma_2, \rho) \models \varphi'$.

   By the definition of τ, we have that $\gamma_1 \Rightarrow_\mathcal{S}^\mathcal{T} \gamma_2$. By the definition of $\Rightarrow_\mathcal{S}^\mathcal{T}$, there exists $\varphi_l \Rightarrow^\exists \varphi_r \in S$ and a valuation $\rho' : Var \to \mathcal{T}$ such that $(\gamma_1, \rho') \models \varphi_l$ and $(\gamma_2, \rho') \models \varphi_r$.

   We have that $(\gamma_1, \rho) \models \varphi$, $(\gamma_1, \rho') \models \varphi_l$ and $(\gamma_2, \rho') \models \varphi_r$. Let $X = FreeVars(\varphi, \varphi')$ and $Y = FreeVars(\varphi_l, \varphi_r)$. We assume without loss of generality that $X \cap Y = \emptyset$. We have that $(\gamma_1, \rho[X]) \models \varphi$, $(\gamma_1, \rho'[Y]) \models \varphi_l$.

Therefore $(\gamma_1, \rho[X] \uplus \rho'[Y]) \models \varphi \wedge \varphi_l$. Therefore we have that for all γ_0, $(\gamma_0, \rho[X] \uplus \rho'[Y]) \models \exists c.(\varphi[c/\square] \wedge \varphi_l[x/\square])$. But $(\gamma_2, \rho[X] \uplus \rho'[Y]) \models \varphi_r$ and therefore $(\gamma_2, \rho[X] \uplus \rho'[Y]) \models \exists c.(\varphi[c/\square] \wedge \varphi_l[x/\square]) \wedge \varphi_r$. But $\models \exists c\ (\varphi[c/\square] \wedge \varphi_l[c/\square]) \wedge \varphi_r \rightarrow \varphi'$ and therefore $(\gamma_2, \rho[X] \uplus \rho'[Y]) \models \varphi'$. But $\mathit{FreeVars}(\varphi') \subseteq X$ and therefore $(\gamma_2, \rho) \models \varphi'$. But this is exactly what we had to prove.

2. Axiom.

   We have that $\varphi \Rightarrow^{Q} \varphi' \in \mathcal{A}$. Let $n \in \mathbb{N} \setminus \{0\}$ be an arbitrary positive natural number such that $\mathcal{S} \models^{+}_{n} \mathcal{A}$ and $\mathcal{S} \models^{+}_{n-1} C$. We prove that $\mathcal{S} \models^{\Delta_C}_{n} \varphi \Rightarrow^{Q} \varphi'$.

   As $\varphi \Rightarrow^{Q} \varphi' \in \mathcal{A}$, it follows that $\mathcal{S} \models^{+}_{n} \varphi \Rightarrow^{Q} \varphi'$. But $\mathcal{S} \models^{+}_{n} \varphi \Rightarrow^{Q} \varphi'$ implies $\mathcal{S} \models^{\Delta_C}_{n} \varphi \Rightarrow^{Q} \varphi'$ independently of whether Δ_C is $+$ or $*$. Therefore $\mathcal{S} \models^{\Delta_C}_{n} \varphi \Rightarrow^{Q} \varphi'$, which is what we had to prove.

3. Reflexivity.

   Note that C is empty here. We trivially have that $\mathcal{S} \models^{*}_{n} \varphi \Rightarrow^{Q} \varphi$.

4. Transitivity.

   Let $n \in \mathbb{N} \setminus \{0\}$ be an arbitrary positive natural number such that $S \models^{+}_{n} \mathcal{A}$ and that $S \models^{+}_{n-1} C$. We prove that $S \models^{\Delta_C}_{n} \varphi_1 \Rightarrow^{Q} \varphi_3$. We distinguish on whether C is empty or not:

   - when C is empty, we have that $\Delta_C ='\ *'$.

     By the induction hypothesis, we have that $S \models^{*}_{n} \varphi_1 \Rightarrow^{Q} \varphi_2$ and that $S \models^{*}_{n} \varphi_2 \Rightarrow^{Q} \varphi_3$. This trivially implies that $S \models^{*}_{n} \varphi_1 \Rightarrow^{Q} \varphi_3$.

   - when C is not empty, we have that $\Delta_C ='\ +'$.

     By the induction hypothesis (for the first condition in the proof rule), we have that

     $$S \models^{+}_{n} \varphi_1 \Rightarrow^{Q} \varphi_2. \tag{3.1}$$

     By the hypothesis, we have that $S \models^{+}_{n-1} \mathcal{A} \cup C$. By the induction hypothesis (for the second condition in the proof rule) we have that

     $$S \models^{*}_{n-1} \varphi_2 \Rightarrow^{Q} \varphi_3. \tag{3.2}$$

     From Equation (3.1) and Equation (3.2), we immediately obtain that $S \models^{+}_{n} \varphi_1 \Rightarrow^{Q} \varphi_3$.

In either case, we have obtained that $S \models_n^{\Delta_C} \varphi_1 \Rightarrow^Q \varphi_3$, which is what we had to prove.

5. Consequence.

   Let $n \in \mathbb{N} \setminus \{0\}$ be an arbitrary configuration such that $S \models_n^+ \mathcal{A}$, that $S \models_{n-1}^+ C$, that $\models \varphi_1 \to \varphi_1'$, that $\models \varphi_2' \to \varphi_2$ and that $S \models_n^{\Delta_C} \varphi_1' \Rightarrow^Q \varphi_2'$. We prove that $S \models_n^{\Delta_C} \varphi_1 \Rightarrow^Q \varphi_2$.

   We distinguish on Q:

   (a) $Q = \forall$.

   Let $\tau = \gamma_1 ... \gamma_k$ be an arbitrary complete path of length $k \leq n$ and let $\rho : Var \to \mathcal{T}$ be an arbitrary valuation such that $(\gamma_1, \rho) \models \varphi_1$. We will show that there exists $i \in \{1, ..., k\}$ when $C = \emptyset$ (respectively $i \in \{2, ..., k\}$ when $C \neq \emptyset$) such that $(\gamma_i, \rho) \models \varphi_2$.

   As $(\gamma_1, \rho) \models \varphi_1$ and $\models \varphi_1 \to \varphi_1'$, it follows that $(\gamma_1, \rho) \models \varphi_1'$. But we have that $S \models_n^{\Delta_C} \varphi_1' \Rightarrow^\forall \varphi_2'$ and therefore, there exists $i \in \{1, ..., k\}$ when $C = \emptyset$ (respectively $i \in \{2, ..., k\}$ when $C \neq \emptyset$) such that $(\gamma_i, \rho) \models \varphi_2'$. But $\varphi_2' \to \varphi_2$ and therefore$(\gamma_i, \rho) \models \varphi_2$, which is exactly what we had to prove.

   (b) $Q = \exists$.

   We will show that $S \models_n^{\Delta_C} \varphi_1 \Rightarrow^\exists \varphi_2$. Let $\gamma \in \mathcal{T}_{Cfg}$ be an arbitrary configuration that terminates in strictly less than n steps and let $\rho : Var \to \mathcal{T}$ be a valuation such that $(\gamma, \rho) \models \varphi_1$. As $\models \varphi_1 \to \varphi_1'$, it follows that $(\gamma, \rho) \models \varphi_1'$. But $S \models_n^{\Delta_C} \varphi_1' \Rightarrow^\exists \varphi_2'$ and therefore there exists $\gamma' \in \mathcal{T}_{Cfg}$ such that $\gamma \Rightarrow_S^{\star\mathcal{T}} \gamma'$ when $C = \emptyset$ (respectively $\gamma \Rightarrow_S^{+\mathcal{T}} \gamma'$ when $C \neq emptyset$) and $(\gamma', \rho) \models \varphi_2'$. But $\models \varphi_2' \to \varphi_2$ and therefore $(\gamma', \rho) \models \varphi_2$. But this is exactly what we had to prove.

   We have shown in any case that $S \models_n^{\Delta_C} \varphi_1 \Rightarrow^Q \varphi_2$, which is what we had to prove.

6. Case Analysis.

   Let $n \in \mathbb{N} \setminus \{0\}$ be an arbitrary configuration such that $S \models_n^+ \mathcal{A}$, that $S \models_{n-1}^+ C$, that $S \models_n^{\Delta_C} \varphi_1 \Rightarrow^Q \varphi$ and that $S \models_n^{\Delta_C} \varphi_2 \Rightarrow^Q \varphi$. We show that $S \models_n^{\Delta_C} \varphi_1 \vee \varphi_2 \Rightarrow^Q \varphi$.

   We distinguish two cases:

(a) $Q = \forall$. Let $\tau = \gamma_1 ... \gamma_k$ be an arbitrary complete path of length $k \leq n$ and let $\rho : Var \to \mathcal{T}$ be an arbitrary valuation such that $(\gamma_1, \rho) \models \varphi_1 \vee \varphi_2$. We will show that there exists some $i \in \{1, ..., k\}$ when $C = \emptyset$ (respectively $i \in \{2, ..., k\}$ when $C \neq \emptyset$) such that $(\gamma_i, \rho) \models \varphi$.

As $(\gamma_1, \rho) \models \varphi_1 \vee \varphi_2$, there exists $j \in \{1, 2\}$ such that $(\gamma_1, \rho) \models \varphi_j$. But by hypothesis we have that $S \models_n^{\Delta_C} \varphi_j \Rightarrow^{\forall} \varphi$. Therefore there exists $i \in \{1, ..., k\}$ when $C = \emptyset$ (respectively $i \in \{2, ..., k\}$ when $C \neq \emptyset$) such that $(\gamma_2, \rho) \models \varphi$, which is what we had to prove.

(b) $Q = \exists$. Let $\gamma \in \mathcal{T}_{Cfg}$ be a configuration that terminates in strictly less than n steps and $\rho : Var \to \mathcal{T}$ a valuation such that $(\gamma, \rho) \models \varphi_1 \vee \varphi_2$. We will show that there exists some $\gamma' \in \mathcal{T}_{Cfg}$ such that $\gamma \Rightarrow_S^{\star\mathcal{T}} \gamma'$ when $C = \emptyset$ (respectively $\gamma \Rightarrow_S^{+\mathcal{T}} \gamma'$ when $C \neq \emptyset$) such that $(\gamma', \rho) \models \varphi$.

As $(\gamma, \rho) \models \varphi_1 \vee \varphi_2$, there exists $j \in \{1, 2\}$ such that $(\gamma, \rho) \models \varphi_j$. But $S \models_n^{\Delta_C} \varphi_j \Rightarrow^{\exists} \varphi$ and therefore there exists $\gamma' \in \mathcal{T}_{Cfg}$ such that $\gamma \Rightarrow_S^{\star\mathcal{T}} \gamma'$ when $C = \emptyset$ (respectively $\gamma \Rightarrow_S^{+\mathcal{T}} \gamma'$ when $C \neq \emptyset$) and $(\gamma', \rho) \models \varphi$. But this is what we had to prove.

In both cases, we have shown that $S \models_n^{\Delta_C} \varphi_1 \vee \varphi_2 \Rightarrow^{Q} \varphi$, which is what we had to prove.

7. Abstraction.

Let $n \in \mathbb{N}$ be an arbitrary positive natural number such that $S \models_n^{+} \mathcal{A}$, that $S \models_{n-1}^{*} C$, that $S \models_n^{\Delta_C} \varphi \Rightarrow^{Q} \varphi'$ and that X is a set of variables such that $X \cap FreeVars(\varphi') = \emptyset$. We show that $S \models_n^{\Delta_C} \exists X.\varphi \Rightarrow^{Q} \varphi'$.

We distinguish two cases:

(a) $Q = \forall$. Let $\tau = \gamma_1 ... \gamma_k$ be an arbitrary complete path of length $k \leq n$ and let $\rho : Var \to \mathcal{T}$ be an arbitrary valuation such that $(\gamma_1, \rho) \models \exists X.\varphi$. We will show that there exists $i \in \{1, ..., k\}$ when $C = \emptyset$ (respectively $i \in \{2, ..., k\}$ when $C \neq \emptyset$) such that $(\gamma_i, \rho) \models \varphi'$.

Because $(\gamma_1, \rho) \models \exists X.\varphi$, we have that there exists $\rho' : Var \to \mathcal{T}$ such that ρ and ρ' agree on $Var \setminus X$ and $(\gamma_1, \rho') \models \varphi$. But $S \models_n^{\Delta_C} \varphi \Rightarrow^{\forall} \varphi'$ and therefore there exists $i \in \{1, ..., k\}$ when $C = \emptyset$ (respectively $i \in \{2, ..., k\}$ when $C \neq \emptyset$) such that $(\gamma_i', \rho') \models \varphi'$. As ρ and ρ' agree on $Var \setminus X$ and $X \cap FreeVars(\varphi') = \emptyset$, we obtain $(\gamma_i, \rho) \models \varphi'$, which is what we had to prove.

(b) $Q = \exists$. Let $\gamma \in \mathcal{T}_{Cfg}$ be a configuration that terminates in strictly less than n steps and $\rho : Var \to \mathcal{T}$ a valuation such that $(\gamma, \rho) \models \exists X.\varphi$. We will show that there exists some $\gamma' \in \mathcal{T}_{Cfg}$ such that $\gamma \Rightarrow^{\star\mathcal{T}}_{\mathcal{S}} \gamma'$ when $C = \emptyset$ (respectively $\gamma \Rightarrow^{+\mathcal{T}}_{\mathcal{S}} \gamma'$ when $C \neq \emptyset$) such that $(\gamma', \rho) \models \varphi'$.

Because $(\gamma, \rho) \models \exists X.\varphi$, we have that there exists $\rho' : Var \to \mathcal{T}$ such that ρ and ρ' agree on $Var \setminus X$ and $(\gamma, \rho') \models \varphi$. But $S \models^{\Delta_C}_{n} \varphi \Rightarrow^{\forall} \varphi'$ and therefore there exists $\gamma' \in \mathcal{T}_{Cfg}$ such that $\gamma \Rightarrow^{\star\mathcal{T}}_{\mathcal{S}} \gamma'$ when $C = \emptyset$ (respectively $\gamma \Rightarrow^{+\mathcal{T}}_{\mathcal{S}} \gamma'$ when $C \neq \emptyset$) and $(\gamma', \rho') \models \varphi'$. As ρ and ρ' agree on $Var \setminus X$ and $X \cap FreeVars(\varphi') = \emptyset$, we obtain $(\gamma', \rho) \models \varphi'$, which is what we had to prove.

In both cases, we have shown that $S \models^{\Delta_C}_{n} \exists X.\varphi \Rightarrow^{Q} \varphi'$, which is what we had to prove.

8. Circularity.

By the induction hypothesis we know that for all positive naturals $m \in \mathbb{N} \setminus \{0\}$,

$$\text{if } S \models^{+}_{m} \mathcal{A} \text{ and } S \models^{+}_{m-1} C \cup \{\varphi \Rightarrow^{Q} \varphi'\} \text{ then } S \models^{+}_{m} \varphi \Rightarrow^{Q} \varphi'. \tag{3.3}$$

We prove that for all positive naturals $n \in \mathbb{N} \setminus \{0\}$, $S \models^{+}_{n} \mathcal{A}$ and $S \models^{+}_{n-1} C$ implies $S \models^{\Delta_C}_{n} \varphi \Rightarrow^{Q} \varphi'$, by induction on n.

(a) if $n = 1$, we trivially have that $S \models^{\Delta_C}_{n-1} \varphi \Rightarrow^{Q} \varphi'$. Therefore $S \models^{\Delta_C}_{n-1} C \cup \{\varphi \Rightarrow^{Q} \varphi'\}$. Applying Equation (3.3) with $m = n = 1$, we obtain that $S \models^{\Delta_C}_{n} \varphi \Rightarrow^{Q} \varphi'$, what we had to show.

(b) if $n > 1$, we have that $S \models^{\Delta_C}_{n-1} \varphi \Rightarrow^{Q} \varphi'$ by the inner induction hypothesis. We also have that $S \models^{\Delta_C}_{n-1} C$ and therefore $S \models^{\Delta_C}_{n-1} C \cup \{\varphi \Rightarrow^{Q} \varphi'\}$. Let $m = n$ in Equation (3.3). We obtain that $S \models^{\Delta_C}_{n} \varphi \Rightarrow^{Q} \varphi'$, what we had to show.

We have shown in any of the cases that for any natural number $n \in \mathbb{N} \setminus \{0\}$, if $\mathcal{S} \models^{+}_{n} \mathcal{A}$ and $\mathcal{S} \models^{+}_{n-1} C$, then $\mathcal{S} \models^{\Delta_C}_{n} \varphi \Rightarrow^{Q} \varphi'$, which concludes our proof.

□

3.6.2 Conditional One-Path Reachability Logic

Here we prove the soundness of the proof system in Figure 3.3.

Theorem 4 (**Soundness**). *If $\mathcal{S}$ is a weakly well-defined reachability system, then $\mathcal{S} \vdash \varphi \Rightarrow^{\exists} \varphi'$ implies $\mathcal{S} \models \varphi \Rightarrow^{\exists} \varphi'$.*

Proof. By Proposition 6 and Lemma 7. □

In order to prove soundness, we need the following helper definition, which will allow us to make the proof by induction on the termination proof of g.

Definition 18. *Let $g \in \mathcal{T}_{Cfg}$ be an arbitrary configuration. We say that the unconditional reachability rule $\varphi \Rightarrow^{\exists} \varphi'$ is $(g, \geq)$-**strongly-valid** (resp. $(g, \geq)$-**strictly-strongly-valid**) if for all γ such that $g \geq \gamma$ and for all valuations ρ such that $(\gamma, \rho) \models \varphi$, there exists γ' such that $\gamma \Rightarrow^{\star\mathcal{T}}_{\mathcal{S}} \gamma'$ (resp. $\gamma \Rightarrow^{+\mathcal{T}}_{\mathcal{S}} \gamma'$) and $(\gamma', \rho) \models \varphi'$.*

We write $\mathcal{S} \models^{*}_{g\geq} \varphi \Rightarrow^{\exists} \varphi'$ when $\varphi \Rightarrow^{\exists} \varphi'$ is $(g, \geq)$-strongly-valid and $\mathcal{S} \models^{+}_{g\geq} \varphi \Rightarrow^{\exists} \varphi'$ if $\varphi \Rightarrow^{\exists} \varphi'$ is $(g, \geq)$-strictly-strongly-valid.

Intuitively, "$(g, \geq)$-strongly-valid" is similar to "strongly valid", but only concerns configurations less than g, according to the termination dependence relation. If g terminates, then "$(g, \geq)$-strongly-valid" is similar to "valid". The following lemma captures the link between the two notions:

Proposition 6. *$\mathcal{S} \models \varphi \Rightarrow^{\exists} \varphi'$ if and only if, for all terminating configurations $g \in \mathcal{T}_{Cfg}$, $\mathcal{S} \models^{*}_{g\geq} \varphi \Rightarrow^{\exists} \varphi'$.*

Proof. We prove each implication separately.

"$\rightarrow$" Assume $\mathcal{S} \models \varphi \Rightarrow^{\exists} \varphi'$. We show that for all terminating $g \in \mathcal{T}_{Cfg}$, $\varphi \Rightarrow^{\exists} \varphi'$ is $(g, \geq)$-strongly-valid. Let g be an arbitrary terminating configuration, let γ be an arbitrary configuration smaller or equal according to $\geq$ than g (i.e. $g \geq \gamma$) and let ρ be a valuation such that $(\gamma, \rho) \models \varphi$. As g terminates, it follows that γ also terminates. As $\mathcal{S} \models \varphi \Rightarrow^{\exists} \varphi'$, we have that there exists γ' such that $\gamma \Rightarrow^{\star\mathcal{T}}_{\mathcal{S}} \gamma'$ and $(\gamma', \rho) \models \varphi'$. As γ was chosen arbitrarily such that $g \geq \gamma$, it follows that $\varphi \Rightarrow^{\exists} \varphi'$ is $(g, \geq)$-strongly-valid. As g was chosen arbitrarily such that it is terminating, it follows that for all terminating g, $\varphi \Rightarrow^{\exists} \varphi'$ is $(g, \geq)$-strongly-valid, which is what we had to show.

"$\leftarrow$" Assume that for all terminating $g \in \mathcal{T}_{Cfg}$, $\varphi \Rightarrow^{\exists} \varphi'$ is $(g, \geq)$-strongly-valid. We show that $\mathcal{S} \models \varphi \Rightarrow^{\exists} \varphi'$. Let γ be an arbitrary terminating configuration and let ρ be an arbitrary valuation such that $(\gamma, \rho) \models \varphi$. Let $g = \gamma$. We have that $g \geq \gamma$ and that g is terminating. Therefore, by the assumption that for all terminating g,

$\varphi \Rightarrow^{\exists} \varphi'$ is $(g, \succeq)$-strongly-valid, we obtain that there exists γ' such that $\gamma \Rightarrow^{\star\mathcal{T}}_{\mathcal{S}} \gamma'$ and $(\gamma', \rho) \models \varphi'$. As the terminating configuration γ and the valuation ρ were chosen arbitrarily, it follows that $\mathcal{S} \models \varphi \Rightarrow^{\exists} \varphi'$.

□

The following helper lemma is the core of the soundness proof. It shows that each proof in our proof system is $(g, \succeq)$-strongly-valid by induction on the proof tree and on g.

Lemma 7. *For any proof tree concluding $\mathcal{S}, C \vdash \mathcal{A} \Rightarrow^{\forall} \varphi\varphi'$, for all terminating configurations $g \in \mathcal{T}_{Cfg}$, if the conditional rules in $\mathcal{A}$ are weakly well-defined, if the unconditional rules in $\mathcal{A}$ are $(g, \succeq)$-strictly-strongly-valid and if C is $(g_0, \succeq)$-strictly-strongly-valid for all g_0 such that $g \succ g_0$, we have that:*

1. *if C is empty, then $\varphi \Rightarrow^{\exists} \varphi'$ is $(g, \succeq)$-strongly-valid and*
2. *if C is not empty, then $\varphi \Rightarrow^{\exists} \varphi'$ is $(g, \succeq)$-strictly-strongly-valid.*

Proof. By induction on the proof tree and case analysis on the last rule in the proof tree:

1. If the last rule is *Axiom*, let g be an arbitrary configuration and assume that the conditional rules in $\mathcal{A}$ are weakly well-defined, that the unconditial rules in $\mathcal{A}$ are $(g, \succeq)$-strictly-strongly-valid and that C is $(g_0, \succeq)$-strictly-strongly-valid for all configurations $g \succ g_0$. We show that $\varphi \wedge \psi \Rightarrow^{\exists} \varphi' \wedge \psi$ is $(g, \succeq)$-strictly-strongly-valid (this is the stronger conclusion of the two cases; $(g, \succeq)$-strictly-strongly-valid implies $(g, \succeq)$-strongly-valid for the case where C is empty). We distinguish two cases:

   (a) If $n > 0$, let γ be an arbitrary configuration such that $g \succeq \gamma$ and let ρ be an arbitrary valuation such that $(\gamma, \rho) \models \varphi \wedge \psi$. We show that there exists γ' such that $\gamma \Rightarrow^{+\mathcal{T}}_{\mathcal{S}} \gamma'$ and $(\gamma', \rho) \models \varphi' \wedge \psi$. We first show that $\varphi_i \Rightarrow^{\exists} \varphi'_i$ is ρ-strongly-valid for all $1 \leq i \leq n$.

   Let $\gamma_1, ..., \gamma_n$ be arbitrary configurations such that $(\gamma_i, \rho) \models \varphi_i$ for all $1 \leq i \leq n$. As the rule is weakly well-defined, $\gamma_1, ..., \gamma_n$ exist. As ψ is stateless, it follows that $(\gamma_i, \rho) \models \varphi_i \wedge \psi$ for all $1 \leq i \leq n$.

   By induction on $1 \leq i \leq n$, we show that $\gamma \succ \gamma_i$ and that $\varphi_j \Rightarrow^{\exists} \varphi'_j$ is ρ-strongly-valid for all $1 \leq j < i$.

   Let $1 \leq i \leq n$ be fixed. By choice of γ_i, we have that $(\gamma_i, \rho) \models \varphi_i \wedge \psi$.

By the induction hypothesis, we have that $\varphi_j \Rightarrow^\exists \varphi'_j$ is ρ-strongly-valid for all $1 \leq j < i$. Therefore, by the o.t. hypothesis, we have that $\gamma > \gamma_i$.

As the unconditional rules in $\mathcal{A}$ are $(g, \geq)$-strictly-strongly-valid and C is $(g_0, \geq)$-strictly-strongly-valid for any configuration $g > g_0$, it follows that the unconditional rules in $\mathcal{A} \cup C$ are $(g_0, \geq)$-strictly-strongly-valid for any configuration $\gamma > g_0$. In particular, the unconditional rules in $\mathcal{A} \cup C$ are $(\gamma_i, \geq)$-strictly-strongly-valid. By the (outer) induction hypothesis, we obtain that $\varphi_i \wedge \psi \Rightarrow^\exists \varphi'_i$ is $(\gamma_i, \geq)$-strongly-valid; therefore there exists γ'_i such that $\gamma_i \Rightarrow^{\star\mathcal{T}}_{\mathcal{S}} \gamma'_i$ and $(\gamma'_i, \rho) \models \varphi'_i$. As γ_i was chosen arbitrarily, it follows that $\varphi_i \Rightarrow^\exists \varphi'_i$ is ρ-strongly-valid, which is what we had to prove.

We have shown by induction on i that $\varphi_i \Rightarrow^\exists \varphi'_i$ is ρ-strongly-valid for all $1 \leq i \leq n$. Therefore, by the well-definedness of the conditional rule, we obtain that there exists γ' such that $\gamma \Rightarrow^{+\mathcal{T}}_{\mathcal{S}} \gamma'$ and $(\gamma', \rho) \models \varphi'$. As $(\gamma, \rho) \models \varphi \wedge \psi$, it follows that $(\gamma, \rho) \models \psi$; as ψ is stateless, it follows that $(\gamma', \rho) \models \psi$. As $(\gamma', \rho) \models \varphi'$ and $(\gamma', \rho) \models \psi$, it follows that $(\gamma', \rho) \models \varphi' \wedge \psi$. We have shown that there exists γ' such that $\gamma \Rightarrow^{+\mathcal{T}}_{\mathcal{S}} \gamma'$ and $(\gamma', \rho) \models \varphi' \wedge \psi$, which is what we had to show.

(b) If $n = 0$, let γ be an arbitrary configuration such that $g \geq \gamma$ and let ρ be an arbitrary valuation such that $(\gamma, \rho) \models \varphi \wedge \psi$. We show that there exists γ' such that $\gamma \Rightarrow^{+\mathcal{T}}_{\mathcal{S}} \gamma'$ and $(\gamma', \rho) \models \varphi' \wedge \psi$.

From $(\gamma, \rho) \models \varphi \wedge \psi$ we immediately obtain $(\gamma, \rho) \models \varphi$. As $\varphi \Rightarrow^\exists \varphi'$ is in $\mathcal{A}$, it must be, by hypothesis, $(g, \geq)$-strictly-strongly-valid. Therefore there exists γ' such that $\gamma \Rightarrow^{+\mathcal{T}}_{\mathcal{S}} \gamma'$ and $(\gamma', \rho) \models \varphi'$. As $(\gamma', \rho) \models \varphi' \wedge \psi$, we have $(\gamma, \rho) \models \psi$; as ψ is stateless, it follows that $(\gamma', \rho) \models \psi$. We already have that $(\gamma', \rho) \models \varphi'$ and therefore $(\gamma', \rho) \models \varphi' \wedge \psi$, which is what we had to show.

2. If the last rule is *Reflexivity*, let g be an arbitrary configuration. We show that $\varphi \Rightarrow^\exists \varphi$ is $(g, \geq)$-strongly-valid. Let $g \geq \gamma$ be an arbitrary configuration and let ρ be an arbitrary valuation such that $(\gamma, \rho) \models \varphi$. We show that there exists γ' such that $\gamma \Rightarrow^{\star\mathcal{T}}_{\mathcal{S}} \gamma'$ and $(\gamma', \rho) \models \varphi$. Indeed, it is sufficient to choose $\gamma' = \gamma$ and the conclusion trivially follows. As C is empty, this is the only case to consider.

3. If the last rule is *Transitivity*, we distinguish two cases:

   (a) If C is empty, let g be an arbitrary configuration. We show that $\varphi_1 \Rightarrow^\exists \varphi_3$ is $(g, \succeq)$-strongly-valid. By the induction hypothesis we have that $\varphi_1 \Rightarrow^\exists \varphi_2$ and $\varphi_2 \Rightarrow^\exists \varphi_3$ are $(g, \succeq)$-strongly-valid.

   Let $g \succeq \gamma_1$ be an arbitrary configuration and let ρ be an arbitrary valuation such that $(\gamma_1, \rho) \models \varphi_1$. We show that there exists γ_3 such that $\gamma_1 \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma_3$ and $(\gamma_3, \rho) \models \varphi_3$.

   As $\varphi_1 \Rightarrow^\exists \varphi_2$ is $(g, \succeq)$-strongly-valid, it follows that there exists γ_2 such that $\gamma_1 \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma_2$ and $(\gamma_2, \rho) \models \varphi_2$. As $\varphi_2 \Rightarrow^\exists \varphi_3$ is $(g, \succeq)$-strongly-valid, it follows that there exists γ_3 such that $\gamma_2 \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma_3$ and $(\gamma_3, \rho) \models \varphi_3$. In conclusion $\gamma_1 \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma_3$ and $(\gamma_3, \rho) \models \varphi_3$, which is what we had to show.

   (b) If C is not empty, let g be an arbitrary configuration. We show that $\varphi_1 \Rightarrow^\exists \varphi_3$ is $(g, \succeq)$-strictly-strongly-valid.

   Let $g \succeq \gamma_1$ be an arbitrary configuration and let ρ be an arbitrary valuation such that $(\gamma_1, \rho) \models \varphi_1$. We show that there exists γ_3 such that $\gamma_1 \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma_3$ and $(\gamma_3, \rho) \models \varphi_3$.

   By the induction hypothesis, we have that $\varphi_1 \Rightarrow^\exists \varphi_2$ is $(g, \succeq)$-strictly-strongly-valid. Therefore, there exists γ_2 such that $\gamma_1 \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma_2$ and $(\gamma_2, \rho) \models \varphi_2$.

   Also by the induction hypothesis, as the unconditional rules in $\mathcal{A} \cup C$ are $(g_0, \succeq)$-strictly-strongly-valid for any $g \succ g_0$, it follows that $\varphi_2 \Rightarrow^\exists \varphi_3$ is $(g_0, \succeq)$-strongly-valid for any $g \succ g_0$. In particular $\varphi_2 \Rightarrow^\exists \varphi_3$ is $(\gamma_2, \succeq)$-strongly-valid. Therefore, there exists γ_3 such that $\gamma_2 \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma_3$ and $(\gamma_3, \rho) \models \varphi_3$. In conclusion, $\gamma_1 \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma_2 \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma_3$ and $(\gamma_3, \rho) \models \varphi_3$, which is what we had to show.

4. If the last rule is *Consequence*, let g be an arbitrary configuration. We show that $\varphi_1 \Rightarrow^\exists \varphi_2$ is $(g, \succeq)$-strongly-valid (resp. $(g, \succeq)$-strictly-strongly-valid).

   Let $g \succeq \gamma_1$ be an arbitrary configuration and let ρ be an arbitrary valuation such that $(\gamma_1, \rho) \models \varphi_1$. We show that there exists γ_2 such that $\gamma_1 \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma_2$

(resp. $\gamma_1 \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma_2$) and $(\gamma_2, \rho) \models \varphi_2$.

As $\models \varphi_1 \to \varphi_2$, we have that $(\gamma_1, \rho) \models \varphi_1'$. By the induction hypothesis, $\varphi_1' \Rightarrow^{\exists} \varphi_2'$ is $(g, \geq)$-strongly-valid (resp. $(g, \geq)$-strictly-strongly-valid) and therefore there exists γ_2 such that $\gamma_1 \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma_2$ (resp. $\gamma_1 \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma_2$) and $(\gamma_2, \rho) \models \varphi_2'$. As $\models \varphi_2' \to \varphi_2$, it follows that $(\gamma_2, \rho) \models \varphi_2$, which is what we had to show.

5. If the last rule is *Case analysis*, let g be an arbitrary configuration. We show that $\varphi_1 \vee \varphi_2 \Rightarrow^{\exists} \varphi$ is $(g, \geq)$-strongly-valid (resp. $(g, \geq)$-strictly-strongly-valid).

   Let $g \geq \gamma_1$ be an arbitrary configuration and let ρ be an arbitrary valuation such that $(\gamma_1, \rho) \models \varphi_1 \vee \varphi_2$. We show that there exists γ_2 such that $\gamma_1 \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma_2$ (resp. $\gamma_1 \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma_2$) and $(\gamma_2, \rho) \models \varphi$.

   As $(\gamma_1, \rho) \models \varphi_1 \vee \varphi_2$ it follows that there exists $i \in \{1, 2\}$ such that $(\gamma_1, \rho) \models \varphi_i$. By the induction hypothesis, we have that $\varphi_i \Rightarrow^{\exists} \varphi$ is $(g, \geq)$-strongly-valid (resp. $(g, \geq)$-strictly-strongly-valid). Therefore, there exists γ_2 such that $\gamma_1 \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma_2$ (resp. $\gamma_1 \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma_2$) and $(\gamma_2, \rho) \models \varphi$. But this is exactly what we had to show.

6. If the last rule is *Abstraction*, let g be an arbitrary configuration. We show that $\exists X.\varphi \Rightarrow^{\exists} \varphi'$ is $(g, \geq)$-strongly-valid (resp. $(g, \geq)$-strictly-strongly-valid).

   Let $g \geq \gamma$ be an arbitrary configuration and let ρ be an arbitrary configuration such that $(\gamma, \rho) \models \exists X.\varphi$. We show that there exists γ' such that $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$ (resp. $\gamma \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma'$) and $(\gamma', \rho) \models \varphi'$.

   As $(\gamma, \rho) \models \exists X.\varphi$, it follows that there exists a valuation ρ' which differs from ρ only in X such that $(\gamma, \rho') \models \varphi$. By the induction hypothesis ($\varphi \Rightarrow^{\exists} \varphi'$ is $(g, \geq)$-strongly-valid (resp. $(g, \geq)$-strictly-strongly-valid)), it follows that there exists γ' such that $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$ (resp. $\gamma \Rightarrow_{\mathcal{S}}^{+\mathcal{T}} \gamma'$) and $(\gamma', \rho') \models \varphi'$. As X contains no free variable of φ' and ρ differs from ρ' only in X, it follows that $(\gamma', \rho) \models \varphi'$, which is what we had to show.

7. If the last rule is *Circularity*, we show by (an inner) induction on g that:

If the unconditional rules in $\mathcal{A}$ are $(g, \geq)$-strictly-strongly-valid and if C is strictly $(g_0, \geq)$-strictly-strongly-validfor all $g > g_0$, then $\varphi \Rightarrow^{\exists} \varphi'$ is $(g, \geq)$-strictly-strongly-valid.

Assume that the unconditional rules in $\mathcal{A}$ are $(g, \geq)$-strongly-valid and that C is $(g_0, \geq)$-strictly-strongly-validfor any $g > g_0$. By the (inner) induction hypothesis, we know that $\varphi \Rightarrow^{\exists} \varphi'$ is $(g_0, \geq)$-strictly-strongly-valid for any $g > g_0$. Therefore $C \cup \{\varphi \Rightarrow^{\exists} \varphi'\}$ is $(g_0, \geq)$-strictly-strongly-validfor any $g > g_0$. Therefore, by the (outer) induction hypothesis, we obtain that $\varphi \Rightarrow^{\exists} \varphi'$ is $(g, \geq)$-strictly-strongly-valid, which is what we had to show.

□

3.7 Relative Completeness

In this section we show that our language-independent proof systems in Figure 3.1 (for all-path reachability) and Figure 3.3 (for one-path reachability) are relatively complete, in the sense of Cook [23]. Note that the relative completeness of the one-path part of the proof system in Figure 3.1 follows from the relative completeness of the proof system in Figure 3.3, hence the proof system in Figure 3.1 is relatively complete for both all-path and one-path reachability when the operational semantics is defined with unconditional rules. This means that any valid reachability property of any programming language (or calculus, system, etc.) is formally derivable with our proof systems using the operational semantics rules of the language as axioms. Note that this is a stronger result than the relative completeness of Hoare logics, since the latter needs to be proved for each language separately, taking into account its particularities. We prove our relative completeness result once and for all languages, similarly to our soundness proof. Relativity here refers to the configuration model: since a matching logic includes a configuration model and since that model can comprise arbitrarily complex mathematical domains, we assume an oracle capable of answering FOL validity questions on that model.

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, 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 $\alpha : Cfg \to \mathbb{N}$ 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 $\mathcal{T}$.

More restrictions on $\mathcal{S}$ are needed for the case of one-path reachability. First, we assume that *calS* is well-defined. To see why this is necessary, consider a system consisting only of a rule $\pi \Rightarrow^\exists \pi_1 \vee \pi_2$, 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 $\mathcal{S} \models \pi \Rightarrow^\exists \pi_1$, since π_1 matches the pattern $\pi_1 \vee \pi_2$, but there is no way to derive $\mathcal{S} \vdash \pi \Rightarrow^\exists \pi_1$. However, the rule $\pi \Rightarrow^\exists \pi_1 \vee \pi_2$ is not well-defined. Next, the finite branching of the termination dependence relation $\succ$ is critical for proving the relative completeness of our proof system, since it allows 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 $\langle \mathsf{random} \rangle \Rightarrow^\exists \langle n \rangle$ (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. Finally, $\mathcal{S}$ is assumed ω-closed in order to derive the divergence of configurations with computations with infinite nesting of conditions (like in the case of a big-step semantics). As discussed in Section 3.3.4, it is easy to construct the ω-closure $\mathcal{S}^\omega$, which yields the same transition system as $\mathcal{S}$.

Table 3.1 summarizes all the discussion above, which we assume in the remainder of this section.

We would like to point our that our condition that $\mathcal{T}$ includes natural numbers with addition and multiplication in order to support Gödelization amounts to satisfying the expressivity hypothesis in Cook's completeness result [23]. This is a very strong condition, but one we believe all operational semantics of real-world

Assumptions for both All-Path and One-Path

The reachability system $\mathcal{S}$ is
— finite.
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 $\alpha : Cfg \to \mathbb{N}$.
The configuration model $\mathcal{T}$ interprets
— $\mathbb{N}$ as the natural numbers;
— operation symbols on $\mathbb{N}$ as corresponding operations;
— $\alpha : Cfg \to \mathbb{N}$ as an injective function.

Assumptions only for One-Path

The reachability system $\mathcal{S}$ is
— non-empty;
— well-defined;
— ω-closed; and
— its termination dependence relation $\succ$ is finitely branching:
for each $\gamma \in \mathcal{T}_{Cfg}$ there are finitely many $\gamma' \in \mathcal{T}_{Cfg}$ with $\gamma \succ \gamma'$.

Table 3.1: Relative completeness assumptions

programming languages satisfy.

We would also like to discuss our result in the context of Clark's incompleteness result [20]. It states that if a programming language includes five features (procedures as parameters in procedures calls, recursion, static scope, global variables in procedure bodies, and local procedure declarations), there is no sound and relatively complete Hoare logic proof system, regardless of what assertion logic is used. However, our result does not contradict it, because we have a slightly different definition of relative completeness. Hoare logic makes a fundamental distinction between programs and states. Intuitively, the Clark's result states that the complexity of the programming language is sufficient to force incompleteness, even if the state is simple (as simple as a mapping from variables to boolean values). Our setting is different in that we do not distinguish between programs and states, and instead we combine them in a configuration. Then, it does not matter where the complexity comes from (the program or the state): by assuming we have an oracle for the matching logic over the model of configuration, we effectively push any complexity inside the matching logic reasoning. Also, rather than assuming the expressivity hypothesis, we directly include natural numbers with addition and multiplication in $\mathcal{T}$ in order to guarantee it.

Formally, we have the following

Theorem 5. *If $\mathcal{S} \models \varphi \Rightarrow^{Q} \varphi'$ then $\mathcal{S} \vdash \varphi \Rightarrow^{Q} \varphi'$, for any semantics $\mathcal{S}$ satisfying the assumptions in Table 3.1.*

Our proof proceeds as follows: first we encode transition system operations in FOL, making use of Gödel's β predicate to eliminate quantifications over sequences of configurations (needed for expressing reachability); then we show that semantic validity of all-path reachability rules can also be expressed in FOL; finally, we prove our completeness result. Recall that, by Proposition 1, matching logic is 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. For the duration of the proof, we let $c, c', c_0, ..., c_n$ be distinct variables of sort *Cfg* which do not appear free in the rules in $\mathcal{S}$). We also let $\gamma, \gamma', \gamma_0, ..., \gamma_n$ range over (not necessarily distinct) configurations in the model $\mathcal{T}$, that is, over elements in $\mathcal{T}_{Cfg}$, and let ρ, ρ' range over valuations $Var \to \mathcal{T}$.

First we prove the relative completeness for all-path rules, and then we prove the relative completeness for conditional one-path rules.

3.7.1 All-Path Reachability Logic

Here we prove the relative completeness of the $\forall$ part of the proof system in Figure 3.1, which derives all-path rules.

Encoding Transition System Operations in FOL

Figure 3.6 shows the definition of the one step transition relation ($\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$) and of the configurations that reach φ on all and complete paths. The former is a (proper) FOL formula, while the later is not, as it quantifies over a sequence of configuration. In Section 3.7.1 we use Gödel's β predicate to define $\overline{coreach}(\varphi)$, a FOL formula equivalent to $coreach(\varphi)$.

The following lemma states that $step(c,\ c')$ actually has the semantic properties its name suggests.

$$
\begin{aligned}
\mathit{step}(c,\ c') &\equiv \bigvee_{\mu \equiv \varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}} \exists \mathit{FreeVars}(\mu)\ (\varphi_l[c/\square] \wedge \varphi_r[c'/\square]) \\
\mathit{coreach}(\varphi) &\equiv \forall n \forall c_0 \ldots c_n \Big(\square = c_0 \to \bigwedge_{0 \le i < n} \mathit{step}(c_i,\ c_{i+1}) \to \neg \exists c_{n+1}\ \mathit{step}(c_n,\ c_{n+1}) \\
&\qquad\qquad \to \bigvee_{0 \le i \le n} \varphi[c_i/\square] \Big)
\end{aligned}
$$

Figure 3.6: FOL encoding of one step transition relation and all-path reachability.

Lemma 8. *$\rho \models \mathit{step}(c,\ c')$ iff $\rho(c) \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \rho(c')$.*

Proof. Assume $\rho \models \mathit{step}(c,\ c')$. Then, by the definition of $\mathit{step}(c,\ c')$, there exists some rule $\mu \equiv \varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}$ such that $\rho \models \exists \mathit{FreeVars}(\mu)\ (\varphi_l[c/\square] \wedge \varphi_r[c'/\square])$. Further, since c and c' do not occur in μ, there exists some ρ' which agrees with ρ on c and c' such that $\rho' \models \varphi_l[c/\square]$ and $\rho' \models \varphi_r[c'/\square]$. By Lemma 1, $\rho' \models \varphi_l[c/\square]$ iff $(\rho'(c), \rho') \models \varphi_l$ and $\rho' \models \varphi_r[c'/\square]$ iff $(\rho'(c'), \rho') \models \varphi_r$, so $(\rho'(c), \rho') \models \varphi_l$ and $(\rho'(c'), \rho') \models \varphi_r$. Since ρ and ρ' agree on c and c', it follows that $(\rho(c), \rho') \models \varphi_l$ and $(\rho(c'), \rho') \models \varphi_r$. By definition, we conclude $\rho(c) \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \rho(c')$.

Conversely, assume $\rho(c) \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \rho(c')$. Then, by definition, there exist some rule $\mu \equiv \varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}$ and some ρ' for which $(\rho(c), \rho') \models \varphi_l$ and $(\rho(c'), \rho') \models \varphi_r$. Further, since c and c' do not occur in μ, we can choose ρ' to agree with ρ on c and c'. Hence, $(\rho'(c), \rho') \models \varphi_l$ and $(\rho'(c'), \rho') \models \varphi_r$. By Lemma 1, $(\rho'(c), \rho') \models \varphi_l$ iff $\rho' \models \varphi_l[c/\square]$ and $(\rho'(c'), \rho') \models \varphi_r$ iff $\rho' \models \varphi_r[c'/\square]$, so $\rho' \models \varphi_l[c/\square]$ and $\rho' \models \varphi_r[c'/\square]$. Since the free variables occurring in $\varphi_l[c/\square] \wedge \varphi_r[c'/\square]$ are $\mathit{FreeVars}(\mu) \cup \{c, c'\}$ and ρ and ρ' agree on c and c', it follows that $\rho \models \exists \mathit{FreeVars}(\mu)\ (\varphi_l[c/\square] \wedge \varphi_r[c'/\square])$. By the definition of $\mathit{step}(c,\ c')$, we conclude $\rho \models \mathit{step}(c,\ c')$. □

The following lemma introduces a formula encoding a complete path of fixed length.

Lemma 9. *$\rho \models \bigwedge_{0 \le i < n} \mathit{step}(c_i,\ c_{i+1}) \wedge \nexists c_{n+1}\ \mathit{step}(c_n,\ c_{n+1})$ iff $\rho(c_0), \ldots, \rho(c_{n+1})$ is a complete $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$-path.*

Proof. By Lemma 8, we have that $\rho(c_i) \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \rho(c_{i+1})$ iff $\rho' \models \mathit{step}(c_i,\ c_{i+1})$, for each $0 \le i < n$. Further, $\rho(c_0), \ldots, \rho(c_{n+1})$ is complete, iff there does not exist γ such that $\rho(c_n) \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma$. Again, by Lemma 8, that is iff $\rho \models \nexists c_{n+1}\ \mathit{step}(c_n,\ c_{n+1})$. We conclude that $\rho \models \bigwedge_{0 \le i < n} \mathit{step}(c_i,\ c_{i+1}) \wedge \nexists c_{n+1}\ \mathit{step}(c_n,\ c_{n+1})$ iff $\rho(c_0), \ldots, \rho(c_{n+1})$ is a complete $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$-path, and we are done. □

The following lemma states that $\mathit{coreach}(\varphi)$ actually has the semantic properties its name suggests.

Lemma 10. *$(\gamma, \rho) \models \mathit{coreach}(\varphi)$ iff for all complete $\Rightarrow^{\mathcal{T}}_{S}$-paths τ starting with γ it is the case that $(\gamma', \rho) \models \varphi$ for some $\gamma' \in \tau$.*

Proof. First we prove the direct implication. Assume $(\gamma, \rho) \models \mathit{coreach}(\varphi)$, and let $\tau \equiv \gamma_0, ..., \gamma_n$ be a complete $\Rightarrow^{\mathcal{T}}_{S}$-path starting with γ. Then let ρ' agree with ρ on $\mathit{FreeVars}(\varphi)$ such that $\rho'(n) = n$ and $\rho'(c_i) = \gamma_i$ for each $0 \leq i \leq n$. According to the definition of $\mathit{coreach}(\varphi)$, we have that

$$(\gamma, \rho') \models \Box = c_0 \wedge \bigwedge_{0 \leq i < n} \mathit{step}(c_i,\ c_{i+1}) \wedge \nexists c_{n+1}\ \mathit{step}(c_n,\ c_{n+1}) \rightarrow \bigvee_{0 \leq i \leq n} \varphi[c_i/\Box]$$

Since, $\gamma = \gamma_0$ and $\rho'(c_0) = \gamma_0$, it follows that $\rho' \models \Box = c_0$. Further, by Lemma 9, since $\rho'(c_0), ..., \rho'(c_n)$ is a complete $\Rightarrow^{\mathcal{T}}_{S}$-path, it must be the case that

$$\rho' \models \bigwedge_{0 \leq i < n} \mathit{step}(c_i,\ c_{i+1}) \wedge \nexists c_{n+1}\ \mathit{step}(c_n,\ c_{n+1})$$

Thus, as $\Box$ does not occur in any $\varphi[c_i/\Box]$, we conclude that $\rho' \models \bigvee_{0 \leq i \leq n} \varphi[c_i/\Box]$, that is, $\rho' \models \varphi[c_i/\Box]$ for some $0 \leq i \leq n$. By Lemma 1, $\rho' \models \varphi[c_i/\Box]$ iff $(\gamma_i, \rho') \models \varphi$. Since ρ agrees with ρ' on $\mathit{FreeVars}(\varphi)$, we conclude that $(\gamma_i, \rho) \models \varphi$.

Conversely, assume that if τ is a finite and complete $\Rightarrow^{\mathcal{T}}_{S}$-path starting with γ. Then $(\gamma', \rho) \models \varphi$ for some $\gamma' \in \tau$. Let ρ' agree with ρ on $\mathit{FreeVars}(\varphi)$. Then we prove that

$$(\gamma, \rho') \models \Box = c_0 \wedge \bigwedge_{0 \leq i < n} \mathit{step}(c_i,\ c_{i+1}) \wedge \nexists c_{n+1}\ \mathit{step}(c_n,\ c_{n+1}) \rightarrow \bigvee_{0 \leq i \leq n} \varphi[c_i/\Box]$$

Specifically, assume $(\gamma, \rho') \models \Box = c_0 \wedge \bigwedge_{0 \leq i < n} \mathit{step}(c_i,\ c_{i+1}) \wedge \nexists c_{n+1}\ \mathit{step}(c_n,\ c_{n+1})$. As $\Box$ does not occur in any cc_ic_{i+1}, by Lemma 9, it follows that $\rho'(c_0), ..., \rho'(c_n)$ is a complete $\Rightarrow^{\mathcal{T}}_{S}$-path. Further, $(\gamma, \rho') \models \Box = c$, implies that $\rho'(c_0), ..., \rho'(c_n)$ starts with γ. Thus, there exists some $0 \leq i \leq n$ such that $(\rho'(c_i), \rho) \models \varphi$, or equivalently, since ρ and ρ' agree on $\mathit{FreeVars}(\varphi)$, such that $(\rho'(c_i), \rho') \models \varphi$. By Lemma 1, $(\rho'(c_i), \rho') \models \varphi$ iff $\rho' \models \varphi[c_i/\Box]$. Therefore, we have that $(\gamma, \rho') \models \bigvee_{0 \leq i \leq n} \varphi[c_i/\Box]$. Finally, since ρ' is an arbitrary valuation which agrees with ρ on $\mathit{FreeVars}(\varphi)$, by the definition of $\mathit{coreach}(\varphi)$ we can conclude that $(\gamma, \rho) \models \mathit{coreach}(\varphi)$, and we are done. $\Box$

$$\begin{aligned}\overline{coreach}(\varphi) \;\equiv\; & \forall n \forall a \forall b\, (\exists c\, (\beta(a,b,0,\alpha(c)) \wedge \square = c)\\ & \quad \wedge \forall i\, (0 \le i \wedge i < n \rightarrow \exists c \exists c'\, (\beta(a,b,i,\alpha(c))\\ & \qquad \wedge\, \beta(a,b,i+1,\alpha(c')) \wedge step(c,\ c')))\\ & \quad \wedge \exists c\, (\beta(a,b,n,\alpha(c)) \wedge \nexists c'\ step(c,\ c'))\\ & \quad \rightarrow \exists i\, (0 \le i \wedge i \le n \wedge \exists c\, (\beta(a,b,i,\alpha(c)) \wedge \varphi[c/\square])))\end{aligned}$$

Figure 3.7: FOL definition of *coreach*(φ)

The following lemma established a useful property of *coreach*(φ).

Lemma 11.

$$\models coreach(\varphi) \rightarrow \varphi \vee (\exists c'\ step(c,\ c') \wedge \forall c'\ (step(c,\ c') \rightarrow coreach(\varphi)[c'/\square]))$$

Proof. We prove that if $(\gamma, \rho) \models coreach(\varphi)$ then

$$(\gamma, \rho) \models \varphi \vee (\exists c'\ step(c,\ c') \wedge \forall c'\ (step(c,\ c') \rightarrow coreach(\varphi)[c'/\square]))$$

By Lemma 10, we have that for all complete $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$-paths τ starting with γ it is the case that $(\gamma'', \rho) \models \varphi$ for some $\gamma'' \in \tau$. We distinguish two cases

- $(\gamma, \rho) \models \varphi$. We are trivially done.
- $(\gamma, \rho) \not\models \varphi$. Then γ must have $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$-successors. Indeed, assume the contrary. Then $\tau \equiv \gamma$ is a complete $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$-path. It follows that $(\gamma, \rho) \models \varphi$, which is a contradiction. Thus, there exists some γ' such that $\gamma \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma'$. By Lemma 8, that is iff $\rho \models \exists c'\ step(c,\ c')$. Further, let γ' be a $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$-successor of γ and τ' a complete $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$-path starting with γ'. Then, $\gamma\tau$ is a complete $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$-path starting with γ. Thus, there exists some $\gamma'' \in \gamma\tau'$ such that $(\gamma'', \rho) \models \varphi$. Since $(\gamma, \rho) \not\models \varphi$, it follows that $\gamma'' \in \tau'$. Notice that γ' is an arbitrary configuration and τ' an arbitrary $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$-path, therefore by Lemma 10 and Lemma 1, we can conclude that $\rho \models \forall c'\ (step(c,\ c') \rightarrow coreach(\varphi)[c'/\square]))$.

□

Formula Gödelization

Figure 3.7 defines $\overline{coreach}(\varphi)$, the FOL equivalent of *coreach*(φ) using Gödel's β predicate. Formally, we have the following

Lemma 12. $\models coreach(\varphi) \leftrightarrow \overline{coreach}(\varphi)$.

Proof. Let us choose some arbitrary but fixed values for n, a and b, and let ρ' such that ρ and ρ' agree on $Var \setminus \{n, a, b\}$ and $\rho'(n) = n$ and $\rho'(a) = a$ and $\rho'(b) = b$. According to the definition of the β predicate, there exists a unique integer sequence $j_0, ..., j_n$ such that $\beta(a, b, i, j_i)$ holds for each $0 \leq i \leq n$. Since α is injective, we distinguish two cases

- there exists some $0 \leq i \leq n$ such that there is not any γ_i with $\alpha(\gamma_i) = j_i$
- there exists a unique sequence $\gamma_0, ..., \gamma_n$ such that $\alpha(\gamma_i) = j_i$ for each $0 \leq i \leq n$.

In the former case, if $i = n$ we get that $\rho' \not\models \exists c\ (\beta(a, b, n, \alpha(c)) \wedge \nexists c'\ step(c,\ c')$ while if $0 \leq i < n$ we get that $\rho' \not\models \exists c \exists c'\ (\beta(a, b, i, \alpha(c)) \wedge \beta(a, b, i+1, \alpha(c')) \wedge step(c,\ c'))$ as in both cases we can not pick a value for c. Thus, (γ, ρ') does not satisfy left-hand-side of the implication in $\overline{coreach}(\varphi)$, and we conclude that (γ, ρ') satisfies the implication.

In the later case, we have that there is a unique way of instantiating the existentially quantified variables c and c' in each sub-formula in which they appear, as they are always arguments of the β predicate. Thus,

$$(\gamma, \rho') \models \exists c\ (\beta(a, b, 0, \alpha(c)) \wedge \square = c)$$

iff $\gamma = \gamma_0$. By Lemma 9, we have that

$$\begin{aligned}\rho' \models\ & \forall i\,(0 \leq i \wedge i < n \rightarrow \exists c \exists c'\ (\beta(a, b, i, \alpha(c)) \wedge \beta(a, b, i+1, \alpha(c')) \wedge step(c,\ c'))) \\ & \wedge \exists c\ (\beta(a, b, n, \alpha(c)) \wedge \nexists c'\ step(c,\ c'))\end{aligned}$$

iff $\gamma_0 ... \gamma_n$ is a complete $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$-path. Finally, by Lemma 1

$$\rho' \models \exists i\ (0 \leq i \wedge i \leq n \wedge \exists c\ (\beta(a, b, i, \alpha(c)) \wedge \varphi[c/\square]))$$

iff $(\gamma_i, \rho') \models \varphi$ for some $0 \leq i \leq n$.

We conclude that (γ, ρ') satisfies the implication in $\overline{coreach}(\varphi)$ iff

- there is no sequence $\gamma_0, ..., \gamma_n$ such that $\alpha(\gamma_i) = j_i$ for each $0 \leq i \leq n$
- the unique sequence $\gamma_0, ..., \gamma_n$ such that $\alpha(\gamma_i) = j_i$ for each $0 \leq i \leq n$ is either not starting at γ, not a complete $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$-path or contains some γ' such that $(\gamma', \rho) \models \varphi$, as ρ and ρ' agree on $Var \setminus \{n, a, b\}$.

According to the property of β, for each sequence $j_0, ..., j_n$ there exist some values for a and b. Since n, a and b are chosen arbitrary, we conclude that $(\gamma, \rho) \models \overline{coreach}(\varphi)$ iff for all complete $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$-paths τ starting at γ, there exists some $\gamma' \in \tau$ such that $(\gamma', \rho) \models \varphi$. By Lemma 10, we have that the above iff $(\gamma, \rho) \models coreach(\varphi)$, and we are done. □

Encoding Semantic Validity in FOL

Lemma 13. *If $\mathcal{S} \models \varphi \Rightarrow^{\forall} \varphi'$ then $\models \varphi \rightarrow coreach(\varphi')$.*

Proof. Follows from the definition of semantic validity of $\varphi \Rightarrow^{\forall} \varphi'$ and Lemma 10. □

Relative Completeness

A matching logic formula ψ is *patternless* iff □ does not occur in ψ. Then we have the following lemma stating that we can derive on step on all paths

Lemma 14. *Sequent*

$$\mathcal{S}, \mathcal{A} \vdash_C \square = c \wedge \exists c' \; step(c,\; c') \wedge \psi \Rightarrow^{\forall} \exists c' \; (\square = c' \wedge step(c,\; c')) \wedge \psi$$

is derivable, where ψ is a patternless formula.

Proof. We derive the rule by applying the Step proof rule with the following prerequisites

$$\models \square = c \wedge \exists c' \; step(c,\; c') \wedge \psi \rightarrow \bigvee_{\varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}} \exists FreeVars(\varphi_l) \; \varphi_l$$

and for each $\varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}$ (since □ does not occur in ψ)

$$\models \exists c'' \; (c'' = c \wedge \exists c' \; step(c,\; c') \wedge \varphi_l[c''/\square]) \wedge \varphi_r \wedge \psi \rightarrow \exists c' \; (\square = c' \wedge step(c,\; c')) \wedge \psi$$

For the first prerequisite, we have the following (using the definition of *step*$(c,\ c')$)

$$\begin{aligned}
& \square = c \wedge \exists c'\ step(c,\ c') \wedge \psi \\
\rightarrow\ & \square = c \wedge \exists c'\ step(c,\ c') \\
\leftrightarrow\ & \square = c \wedge \exists c' \bigvee_{\mu \equiv \varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}} \exists FreeVars(\mu)\ (\varphi_l[c/\square] \wedge \varphi_r[c'/\square]) \\
\rightarrow\ & \square = c \wedge \exists c' \bigvee_{\mu \equiv \varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}} \exists FreeVars(\mu)\ \varphi_l[c/\square] \\
\rightarrow\ & \square = c \wedge \bigvee_{\mu \equiv \varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}} \exists FreeVars(\varphi_l)\ \varphi_l[c/\square] \\
\rightarrow\ & \bigvee_{\mu \equiv \varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}} \exists FreeVars(\varphi_l)\ \varphi_l
\end{aligned}$$

For the second prerequisite, let $\varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}$. Then we have that

$$\begin{aligned}
& \exists c''\ (c'' = c \wedge \exists c'\ step(c,\ c') \wedge \varphi_l[c''/\square]) \wedge \varphi_r \wedge \psi \\
\rightarrow\ & \varphi_l[c/\square] \wedge \varphi_r \wedge \psi \\
\rightarrow\ & \exists c'\ (\square = c' \wedge \varphi_l[c/\square] \wedge \varphi_r[c'/\square]) \wedge \psi \\
\rightarrow\ & \exists c'\ (\square = c' \wedge \bigvee_{\mu \equiv \varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}} (\varphi_l[c/\square] \wedge \varphi_r[c'/\square])) \wedge \psi \\
\rightarrow\ & \exists c'\ (\square = c' \wedge \bigvee_{\mu \equiv \varphi_l \Rightarrow^{\exists} \varphi_r \in \mathcal{S}} \exists FreeVars(\mu)\ (\varphi_l[c/\square] \wedge \varphi_r[c'/\square]) \wedge \psi \\
\rightarrow\ & \exists c'\ (\square = c' \wedge step(c,\ c')) \wedge \psi
\end{aligned}$$

and we are done. □

The following three lemmas show that we can derive a rule stating that all the configurations reaching φ in the transition system actually reach φ.

Lemma 15. *If*

$$\mathcal{S}, \mathcal{A} \vdash \square = c \wedge \exists c'\ step(c,\ c') \wedge \forall c'\ (step(c,\ c') \rightarrow coreach(\varphi)[c'/\square]) \Rightarrow^{\forall} \varphi$$

then $\mathcal{S}, \mathcal{A} \vdash coreach(\varphi) \Rightarrow^{\forall} \varphi$.

Proof. By Lemma 11

$$\models coreach(\varphi) \leftrightarrow \varphi \vee (\exists c'\ step(c,\ c') \wedge \forall c'\ (step(c,\ c') \rightarrow coreach(\varphi)[c'/\square]))$$

Thus, by Consequence and Case Analysis, it suffices to derive

$$\mathcal{S}, \mathcal{A} \vdash \varphi \Rightarrow^{\forall} \varphi$$
$$\mathcal{S}, \mathcal{A} \vdash \square = c \wedge \exists c' \, step(c, c') \wedge \forall c' \, (step(c, c') \rightarrow coreach(\varphi)[c'/\square]) \Rightarrow^{\forall} \varphi$$

The first sequent follows by Reflexivity. The second sequent is part of the hypothesis, and we are done. □

Lemma 16.

$$\mathcal{S}, \mathcal{A} \vdash \square = c \wedge \exists c' \, step(c, c') \wedge \forall c' \, (step(c, c') \rightarrow coreach(\varphi)[c'/\square]) \Rightarrow^{\forall} \varphi$$

Proof. Let μ be the rule we want to derive, namely

$$\square = c \wedge \exists c' \, step(c, c') \wedge \forall c' \, (step(c, c') \rightarrow coreach(\varphi)[c'/\square]) \Rightarrow^{\forall} \varphi$$

Then $\mathcal{S}, \mathcal{A} \vdash \mu$ follows by Circularity from $\mathcal{S}, \mathcal{A} \vdash_{\{\mu\}} \mu$. Hence, by Transitivity, it suffices to derive the two sequents below

$$\mathcal{S}, \mathcal{A} \vdash_{\{\mu\}} \square = c \wedge \exists c' \, step(c, c') \wedge \forall c' \, (step(c, c') \rightarrow coreach(\varphi)[c'/\square]) \Rightarrow^{\forall} \varphi'$$
$$\mathcal{S}, \mathcal{A} \cup \{\mu\} \vdash \varphi' \Rightarrow^{\forall} \varphi$$

where $\varphi' \equiv \exists c' \, (\square = c' \wedge step(c, c')) \wedge \forall c' \, (step(c, c') \rightarrow coreach(\varphi)[c'/\square])$. The first sequent follows by Lemma 14 with $\psi \equiv \forall c' \, (step(c, c') \rightarrow coreach(\varphi)[c'/\square])$. For the second sequent, by Abstraction with $\{c'\}$ and Consequence with

$$\models \square = c' \wedge step(c, c') \wedge \forall c' \, (step(c, c') \rightarrow coreach(\varphi)[c'/\square]) \rightarrow coreach(\varphi)$$

it suffices to derive $\mathcal{S}, \mathcal{A} \cup \{\mu\} \vdash coreach(\varphi) \Rightarrow^{\forall} \varphi$. Then, by Lemma 15, we are left to derive

$$\mathcal{S}, \mathcal{A} \cup \{\mu\} \vdash \square = c \wedge \exists c' step(c, c') \wedge \forall c' (step(c, c') \rightarrow coreach(\varphi)[c'/\square]) \Rightarrow^{\forall} \varphi$$

that is, $\mathcal{S}, \mathcal{A} \cup \{\mu\} \vdash \mu$, which trivially follows by Axiom and we are done. □

Lemma 17. $\mathcal{S}, \mathcal{A} \vdash coreach(\varphi) \Rightarrow^{\forall} \varphi$*.*

Proof. By Lemma 15, it suffices to derive

$$\mathcal{S}, \mathcal{A} \vdash \square = c \wedge \exists c' \, step(c, c') \wedge \forall c' \, (step(c, c') \rightarrow coreach(\varphi)) \Rightarrow^{\forall} \varphi$$

which follows by Lemma 16. □

Finally, we can prove our main theorem

Proof. By Lemma 13, we have that $\models \varphi \to \mathit{coreach}(\varphi')$. Further, by Lemma 17, we have that $\mathcal{S} \vdash \mathit{coreach}(\varphi') \Rightarrow^{\forall} \varphi'$. Then the theorem follows by CONSEQUENCE. □

3.7.2 Conditional One-Path Reachability Logic

Here we show that the conditional one-path reachability logic proof system in Figure 3.3 is relatively complete. This subsumes the relative completeness of the unconditional one-path reachability logic proof system in Figure 3.1.

Encoding Transition System Operations in FOL

Recall that one of the assumption of relative completeness is that $\mathcal{S}$ is well-defined (see Definition 14). For the purposes of this proof, we give an equivalent and finer-grained definition of the transition relation $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$ induced by a well-defined reachability system $\mathcal{S}$. Let $k, m \in \mathbb{N}$. Recall from Definition 10 that $\mathcal{R}_k$ is the transition relation obtained by applying at most $k-1$ "nested" conditional rules. We introduce $\mathcal{R}_{k,m}$ with $\mathcal{R}_{k,m} \subseteq \mathcal{R}_k \subseteq \mathcal{T}_{Cfg} \times \mathcal{T}_{Cfg}$ to denote the transition relation obtained by applying at most $k-1$ "nested" conditional rules, and by taking at most m steps in each condition. Formally,

- $\mathcal{R}_{0,m} = \emptyset$
- $\mathcal{R}_{k+1,m} = \{\, (\gamma,\ \gamma') \ \mid$ there exists some reachability rule

  $$\varphi \Rightarrow^{\exists} \varphi' \ \text{ if } \ \varphi_1 \Rightarrow^{\exists} \varphi_1' \wedge \ldots \wedge \varphi_n \Rightarrow^{\exists} \varphi_n'$$

  in $\mathcal{S}$ and some valuation $\rho\colon \mathit{Var} \to \mathcal{T}$ such that:

  1. $(\gamma, \rho) \models \varphi$ and $(\gamma', \rho) \models \varphi'$; and
  2. there exist $\gamma_1, \ldots \gamma_n, \gamma_1', \ldots, \gamma_n' \in \mathcal{T}_{Cfg}$ with $(\gamma_i\,, \rho) \models \varphi_i$ and $(\gamma_i'\,, \rho) \models \varphi_i'$ and such that $(\gamma_i,\ \gamma_i') \in \bigcup_{0 \le m' \le m} \mathcal{R}_{k,m}^{m'}$ for all $1 \le i \le n$, where $\mathcal{R}_{k,m}^{m'}$ is the transitive composition of $\mathcal{R}_{k,m}$ with itself m' times ($\mathcal{R}_{k,m}^{0}$ is the identity)$\}$

Notice that $\mathcal{R}_{k,m}$ existentially quantifies $\gamma_1, \ldots, \gamma_n$ ("there exist $\gamma_1', \ldots, \gamma_n'$"), unlike $\mathcal{R}_k$, which universally quantifies $\gamma_1, \ldots, \gamma_n$ ("for all $\gamma_1, \ldots, \gamma_n$"). However, the well-definedness of $\mathcal{S}$ implies that for a given ρ the said $\gamma_1, \ldots, \gamma_n$ always exist and are unique. Thus, we can replace universal with existential quantification. The following formally states that the relations $\mathcal{R}_{k,m}$ give an equivalent definition to the relations $\mathcal{R}_k$ and $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$:

Lemma 18. *$\mathcal{R}_k = \bigcup_{m\geq 0} \mathcal{R}_{k,m}$ and $\Rightarrow_{\mathcal{S}}^{\mathcal{T}} = \bigcup_{k\geq 0, m\geq 0} \mathcal{R}_{k,m}$.*

Proof. First, we prove $\mathcal{R}_k = \bigcup_{m\geq 0} \mathcal{R}_{k,m}$ by induction on k. The base case ($k = 0$) is trivial, as $\mathcal{R}_k = \emptyset$ and $\mathcal{R}_{k,m} = \emptyset$ for each $m \geq 0$. For the induction case, we assume the result for k and prove it for $k + 1$ by double inclusion.

To show $\mathcal{R}_{k+1} \subseteq \bigcup_{m\geq 0} \mathcal{R}_{k+1,m}$, we assume $(\gamma,\ \gamma') \in \mathcal{R}_{k+1}$ and we show $(\gamma,\ \gamma') \in \mathcal{R}_{k+1,m}$ for some m. By Definition 10, it follows that there exists some rule

$$\varphi \Rightarrow^{\exists} \varphi' \text{ if } \varphi_1 \Rightarrow^{\exists} \varphi'_1 \wedge \ldots \wedge \varphi_n \Rightarrow^{\exists} \varphi'_n$$

in $\mathcal{S}$ and some ρ such that

1. $(\gamma,\rho) \models \varphi$ and $(\gamma',\rho) \models \varphi'$; and
2. for all $\gamma_1, \ldots \gamma_n$ with $(\gamma_i\ ,\rho) \models \varphi_i$ for each $1 \leq i \leq n$ there exist $\gamma'_1, \ldots, \gamma'_n$ with $(\gamma'_i\ ,\rho) \models \varphi'_i$ such that $(\gamma_i,\ \gamma'_i) \in \mathcal{R}_k^*$ for each $1 \leq i \leq n$.

Since $\mathcal{S}$ is well-defined, it follows that there exist and are unique $\gamma_1, \ldots \gamma_n$ with $(\gamma_i\ ,\rho) \models \varphi_i$ for each $1 \leq i \leq n$, thus condition 2. above becomes: there exist $\gamma_1, \ldots \gamma_n,\ \gamma'_1, \ldots, \gamma'_n$ with $(\gamma_i\ ,\rho) \models \varphi_i$ and $(\gamma'_i\ ,\rho) \models \varphi'_i$ and such that $(\gamma_i,\ \gamma'_i) \in \mathcal{R}_k^*$ for all $1 \leq i \leq n$. Hence, there exist m_i and $\gamma_{i,0}, \ldots, \gamma_{i,m_i}$ with $\gamma_{i,0} = \gamma_i$ and $\gamma_{i,m_i} = \gamma'_i$ such that $(\gamma_{i,j}, \gamma_{i,j+1}) \in \mathcal{R}_k$ for each $1 \leq i \leq n$ and $0 \leq j < m_i$. By the induction hypothesis, $\mathcal{R}_k = \bigcup_{m\geq 0} \mathcal{R}_{k,m}$. Thus, there exist some $m_{i,0}, \ldots, m_{i,m_i-1}$ such that $(\gamma_{i,j}, \gamma_{i,j+1}) \in \mathcal{R}_{k,m_{i,j}}$ for each $1 \leq i \leq n$ and $0 \leq j < m_i$. It is easy to prove by induction on k that $\mathcal{R}_{k,m'} \subseteq \mathcal{R}_{k,m''}$ if $m' \leq m''$. Let m be the maximum of $\{m_i \mid 1 \leq i \leq n\} \cup \{m_{i,j} \mid 1 \leq i \leq n, 0 \leq j \leq m_i\}$. Then, it follows that $(\gamma_{i,j}, \gamma_{i,j+1}) \in \mathcal{R}_{k,m}$ for each $1 \leq i \leq n$ and $0 \leq j < m_i$, and consequently, that $(\gamma_i, \gamma'_i) \in \bigcup_{0\leq m'\leq m} \mathcal{R}_{k,m}^{m'}$. We can conclude that both conditions 1. and 2. in the definition of $\mathcal{R}_{k+1,m}$ are satisfied, that is, $(\gamma,\ \gamma') \in \mathcal{R}_{k+1,m}$ for some m. Therefore, $\mathcal{R}_{k+1} \subseteq \bigcup_{m\geq 0} \mathcal{R}_{k+1,m}$.

To show $\bigcup_{m\geq 0} \mathcal{R}_{k+1,m} \subseteq \mathcal{R}_{k+1}$, we assume $(\gamma,\ \gamma') \in \mathcal{R}_{k+1,m}$ for some m and we show $(\gamma,\ \gamma') \in \mathcal{R}_{k+1}$. By the definition of $\mathcal{R}_{k+1,m}$, it follows that there exists some rule

$$\varphi \Rightarrow^{\exists} \varphi' \text{ if } \varphi_1 \Rightarrow^{\exists} \varphi'_1 \wedge \ldots \wedge \varphi_n \Rightarrow^{\exists} \varphi'_n$$

in $\mathcal{S}$ and some ρ such that:

1. $(\gamma, \rho) \models \varphi$ and $(\gamma', \rho) \models \varphi'$; and
2. there exist $\gamma_1, ...\gamma_n, \gamma'_1, ..., \gamma'_n$ with $(\gamma_i, \rho) \models \varphi_i$ and $(\gamma'_i, \rho) \models \varphi'_i$ and such that $(\gamma_i,\ \gamma'_i) \in \bigcup_{0 \le m' \le m} \mathcal{R}^{m'}_{k,m}$ for all $1 \le i \le n$.

Since $\mathcal{S}$ is well-defined, it follows that there are unique $\gamma_1, ...\gamma_n$ with $(\gamma_i\ , \rho) \models \varphi_i$ for each $1 \le i \le n$, thus condition 2. above becomes: for all $\gamma_1, ...\gamma_n$ with $(\gamma_i\ , \rho) \models \varphi_i$ for each $1 \le i \le n$ there exist $\gamma'_1, ..., \gamma'_n$ with $(\gamma'_i\ , \rho) \models \varphi'_i$ such that $(\gamma_i,\ \gamma'_i) \in \bigcup_{0 \le m' \le m} \mathcal{R}^{m'}_{k,m}$ for each $1 \le i \le n$. By the induction hypothesis, $\mathcal{R}_k = \bigcup_{m \ge 0} \mathcal{R}_{k,m}$. Thus, $\mathcal{R}_{k,m} \subseteq \mathcal{R}_k$, and consequently, $\bigcup_{0 \le m' \le m} \mathcal{R}^{m'}_{k,m} \subseteq \mathcal{R}^*_k$. We can conclude that both conditions 1. and 2. in Definition 10 are satisfied, that is, $(\gamma,\ \gamma') \in \mathcal{R}_{k+1}$. Therefore, $\bigcup_{m \ge 0} \mathcal{R}_{k+1,m} \subseteq \mathcal{R}_{k+1}$.

The second part of the lemma follows from the first part, as $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$ is defined to be $\bigcup_{k \ge 0} \mathcal{R}_k$. □

Since our objective is to encode properties of the transition system $(\mathcal{T}_{Cfg}, \Rightarrow^{\mathcal{T}}_{\mathcal{S}})$ in FOL making use of the relations $\mathcal{R}_{k,m}$, we next discuss these relations in a bit more detail. Unless otherwise specified, we fix some arbitrary $k, m \in \mathbb{N}$ and assume that each rule in $\mathcal{S}$ has at most nc conditions. Then $(\gamma, \gamma') \in \bigcup_{0 \le m' \le m} \mathcal{R}^{m'}_{k,m}$ iff there exist some $\gamma_0, ..., \gamma_{m'} \in \mathcal{T}_{Cfg}$ with $0 \le m' \le m$, $\gamma_0 = \gamma$ and $\gamma_{m'} = \gamma'$ such that $(\gamma_{i_1}, \gamma_{i_1+1}) \in \mathcal{R}_{k,m}$ for each $0 \le i_1 < m'$. We can reformulate it as there exist some $\gamma_0, ..., \gamma_m \in \mathcal{T}_{Cfg}$ with $\gamma_0 = \gamma$ and $\gamma_m = \gamma'$ such that $(\gamma_{i_1}, \gamma_{i_1+1}) \in \mathcal{R}_{k,m}$ or $\gamma_{i_1} = \gamma_{i_1+1}$ for each $0 \le i_1 < m$. A pair $(\gamma_{i_1}, \gamma_{i_1+1})$ belongs to $\mathcal{R}_{k,m}$ iff there exist some rule $\mu \equiv (\varphi \Rightarrow^{\exists} \varphi'\ \ \mathsf{if}\ \ \varphi_1 \Rightarrow^{\exists} \varphi'_1 \bigwedge ... \bigwedge \varphi_n \Rightarrow^{\exists} \varphi'_n)$ in $\mathcal{S}$ and valuation ρ_{i_1} such that for each $1 \le i_2 \le n$ there exist some $\gamma_{i_1,i_2,0}$ and $\gamma_{i_1,i_2,m}$ with $(\gamma_{i_1,i_2,0}, \rho_{i_1}) \models \varphi_{i_2}$ and $(\gamma_{i_1,i_2,m}, \rho_{i_1}) \models \varphi'_{i_2}$ such that $(\gamma_{i_1,i_2,0}, \gamma_{i_1,i_2,m}) \in \bigcup_{0 \le m' \le m} \mathcal{R}^{m'}_{k-1,m}$. As before, that happens iff there exist some $\gamma_{i_1,i_2,1}, ..., \gamma_{i_1,i_2,m-1} \in \mathcal{T}_{Cfg}$ (we already introduced $\gamma_{i_1,i_2,0}$ and $\gamma_{i_1,i_2,m}$) such that $(\gamma_{i_1,i_2,i_3}, \gamma_{i_1,i_2,i_3+1}) \in \mathcal{R}_{k-1,m}$ or $\gamma_{i_1,i_2,i_3} = \gamma_{i_1,i_2,i_3+1}$ for each $0 \le i_3 < m$. This procedure continues until k reaches 0, when no rules can be used, and thus only identical configuration pairs belong to $\bigcup_{0 \le m' \le m} \mathcal{R}^{m'}_{0,m}$ (as $\mathcal{R}^0_{0,m}$ is the identity).

During this process, the occurring configurations have indexes of the form $\gamma_{i_1,i_2,...,i_{2j+1}}$ for some $0 \le j \le k$, with $i_1, i_3, ..., i_{2j-1}$ between 0 and $m - 1$, with $i_2, i_4, ..., i_{2j}$ between 1 and nc, and with $0 \le i_{2j+1} \le m$. The intuition for such a configuration $\gamma_{i_1,...,i_{2j+1}}$ is that it occurs when establishing a transition from position i_1 to $i_1 + 1$ on the path from γ to γ', and then when establishing a path for the i_2^{th} condition of the used rule for the said transition, and then when establishing a transition from position i_3 to position $i_3 + 1$ on the said path, and so on. Only the last index, i_{2j+1} can be m, as it can be either the source of a transition or the final configuration on a path. It is always the case that $(\gamma_{i_1,...,i_{2j},0}, \gamma_{i_1,...,i_{2j},m}) \in \bigcup_{0 \leq m' \leq m} \mathcal{R}^{m'}_{k-j,m}$.

With the above in mind, we introduce an indexing schema for configuration variables. For some $0 \leq j \leq k$, we use s to denote a sequence of indices $i_1, i_2, ..., i_{2j-1}, i_{2j}$ with each index i on an odd position in s ranging from 0 to $m - 1$ and each i on an even position ranging from 1 to nc. Let $0 \leq i \leq m$. Then we use $c_{s,i}$ to denote a (fresh) variable of sort *Cfg* indexed by the sequence s, i. Intuitively, $c_{s,i}$ is interpreted as the configuration $\gamma_{i_1,...,i_{2j},i_{2j+1}}$ mentioned above (with $i_{2j+1} = i$). Notice that c_0 and c_m (indexed by the sequences 0 and m, as s is empty) stand for γ and γ', the given configurations. Let I_j be the set of all such sequences s of length $2j$. Then we define the set of configuration variables associated with k and m as follows: $C_{k,m} = \{c_{s,i} \mid s \in I_j \text{ for some } 0 \leq j \leq k \text{ and } 0 \leq i \leq m\}$. Note that $C_{k,m}$ is finite. We quantify over finite sets of variables, like $\exists C_{k,m}$, as shorthand for quantifying over each variable in the set.

Let $\bar{x} = x_1, ..., x_{nx}$ be the free variables occurring in the rules in $\mathcal{S}$. Let $\mu \equiv (\varphi \Rightarrow^{\exists} \varphi' \text{ if } \varphi_1 \Rightarrow^{\exists} \varphi'_1 \wedge ... \wedge \varphi_n \Rightarrow^{\exists} \varphi'_n)$ be a rule in $\mathcal{S}$ and let s be a sequence of indices (as above) and $0 \leq i \leq nc$ be an index. Then we define the following FOL formula:

$$rule^{\mu}_{s,i} \equiv \exists \bar{x}\, (\varphi[c_{s,i}/\square] \wedge \varphi'[c_{s,i+1}/\square] \wedge \bigwedge_{1 \leq i' \leq n} (\varphi_{i'}[c_{s,i,i',0}/\square] \wedge \varphi'_{i'}[c_{s,i,i',m}/\square])$$

Intuitively, $rule^{\mu}_{s,i}$ encodes the matching part of the definition of the transition relation $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$. It states that there exists some valuation of the free variables $x_1, ..., x_{nx}$ for which $c_{s,i}$ and $c_{s,i+1}$ match the LHS and RHS of μ, and $c_{s,i,i',0}$ and $c_{s,i,i',m}$ match the LHS and RHS of i'^{th} condition of μ. Configuration variables are indexed by sequence s, i to avoid name conflicts. If μ is unconditional, then the big conjunction is empty. Now we can encode the existence of a path

$$\begin{aligned} path_{k,m} &\equiv \bigwedge_{\substack{s \in I_k \\ 0 \leq i < m}} c_{s,i} = c_{s,i+1} \wedge \bigwedge_{\substack{0 \leq j < k \\ s \in I_j \\ 0 \leq i < m}} (\bigvee_{\mu \in \mathcal{S}} rule^{\mu}_{s,i} \vee c_{s,i} = c_{s,i+1}) \\ path(c, c') &\equiv \exists k \exists m \exists C_{k,m}\, (path_{k,m} \wedge c_0 = c \wedge c_m = c') \end{aligned}$$

The definition of $path(c, c')$ encodes $\Rightarrow^{\star\mathcal{T}}_{\mathcal{S}}$ by encoding $\bigcup_{0 \leq m' \leq m} \mathcal{R}^{m'}_{k,m}$, and thus resembles the informal process above of establishing that $(\gamma, \gamma') \in \bigcup_{0 \leq m' \leq m} \mathcal{R}^{m'}_{k,m}$.

The variables $c_{s,i}$ stand for the configurations $\gamma_{s,i}$. For each appropriate s and i, there is either a transition from $c_{s,i}$ to $c_{s,i+1}$ or $c_{s,i}$ is equal to $c_{s,i+1}$, unless s has length $2k$ when $c_{s,i}$ must be equal to $c_{s,i+1}$. Thus, there is some path from $c_{s,0}$ to $c_{s,m}$ of length at most m.

Using $path(c, c')$ we can encode the following: (1) the one step transition relation ($\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$), (2) the termination dependence relation ($\succ$), (3) the divergence predicate ($\uparrow$), and (4) the configurations reaching some formula φ. For the definitions below, a rule μ is assumed of the form $\varphi \Rightarrow^{\exists} \varphi'$ if $\varphi_1 \Rightarrow^{\exists} \varphi'_1 \bigwedge \dots \bigwedge \varphi_n \Rightarrow^{\exists} \varphi'_n$

$$
\begin{aligned}
step(c, c') &\equiv \exists c_1 ... c_{nc}\, \exists c'_1 ... c'_{nc} \exists \bar{x}\, (\bigvee_{\mu \in \mathcal{S}} (\varphi[c/\square] \wedge \varphi'[c'/\square] \\
&\quad \wedge \bigwedge_{1 \le i \le n} (\varphi_i[c_i/\square] \wedge \varphi'_i[c'_i/\square] \wedge path(c_i, c'_i)))) \\
succ(c, c') &\equiv step(c, c') \\
&\quad \vee\ \exists c_1 ... c_{nc}\, \exists c'_1 ... c'_{nc} \exists \bar{x}\, (\bigvee_{\mu \in \mathcal{S}} (\varphi[c/\square] \wedge \bigvee_{1 \le i \le n} (\varphi_i[c'/\square] \\
&\quad \wedge \bigwedge_{1 \le j < i} (\varphi_j[c_j/\square] \wedge \varphi'_j[c'_j/\square] \wedge path(c_j, c'_j))))) \\
diverge(c) &\equiv \forall m \exists c_0 ... c_m (\bigwedge_{0 \le i < m} succ(c_i, c_{i+1}) \wedge c_0 = c) \\
coreach(\varphi) &\equiv \exists c \exists c'\, (c = \square \wedge \varphi[c'/\square] \wedge path(c, c'))
\end{aligned}
$$

These definitions are not (yet) proper FOL formulae: they quantify over sets and sequences of variables. The definitions of $\overline{path}(c, c')$ and $\overline{diverge}(c)$ in Figure 3.8 are proper FOL formulae equivalent to $path(c, c)$ and $diverge(c)$ (Lemma 31). Then the remaining predicates can also be expressed in FOL.

The following lemmas state various properties of the transition system, leading to the conclusion that the above definitions have the semantic properties their names suggest. First, we establish a FOL relation between $\mathcal{R}_{k+1,m}$ and $\bigcup_{0 \le m' \le m} \mathcal{R}^{m'}_{k,m}$:

Lemma 19. *Let $k, m \in \mathbb{N}$ and $c, c', c_1, c'_1, ..., c_{nc}, c'_{nc} \in Var_{Cfg}$ and $\gamma, \gamma' \in \mathcal{T}_{Cfg}$. Then $(\gamma, \gamma') \in \mathcal{R}_{k+1,m}$ iff there exists some $\rho : Var \to \mathcal{T}$ such that ($\mu \equiv \varphi \Rightarrow^{\exists} \varphi'$ if $\varphi_1 \Rightarrow^{\exists} \varphi'_1 \bigwedge \dots \bigwedge \varphi_n \Rightarrow^{\exists} \varphi'_n$)*

$$
\rho \models \bigvee_{\mu \in \mathcal{S}} (\varphi[c/\square] \wedge \varphi'[c'/\square] \wedge \bigwedge_{1 \le i \le n} (\varphi_i[c_i/\square] \wedge \varphi'_i[c'_i/\square]))
$$

and $(\rho(c_i), \rho(c'_i)) \in \bigcup_{0 \le m' \le m} \mathcal{R}^{m'}_{k,m}$ for each $1 \le i \le nc$ and $\rho(c) = \gamma$ and $\rho(c') = \gamma'$. Moreover, $\gamma \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma'$ iff $\rho(c_i) \Rightarrow^{\star\mathcal{T}}_{\mathcal{S}} \rho(c'_i)$ for each i.

Proof. For the direct implication, assume that $(\gamma, \gamma') \in \mathcal{R}_{k+1,m}$. Then there must be some rule $\mu \in \mathcal{S}$ and some ρ such that $(\gamma,\rho) \models \varphi$ and $(\gamma',\rho) \models \varphi'$ and for each $1 \leq i \leq n$ there exist γ_i, γ'_i with $(\gamma_i,\rho) \models \varphi_i$ and $(\gamma'_i,\rho) \models \varphi'_i$ such that $(\gamma_i, \gamma'_i) \in \bigcup_{0 \leq m' \leq m} \mathcal{R}^{m'}_{k,m}$. Since $c, c', c_1, c'_1, ..., c_{nc}, c'_{nc}$ do not occur in μ, we can assume that ρ is such that: $\rho(c) = \gamma$ and $\rho(c') = \gamma'$; and $\rho(c_i) = \gamma_i$ and $\rho(c'_i) = \gamma'_i$ for each $1 \leq i \leq n$; and $\rho(c_i) = \rho(c'_i)$ for each $n+1 \leq i \leq nc$. Then we can conclude that $(\rho(c_i), \rho(c'_i)) \in \bigcup_{0 \leq m' \leq m} \mathcal{R}^{m'}_{k,m}$ for each $1 \leq i \leq nc$. By Lemma 1 we have that $(\rho(c),\rho) \models \varphi$ iff $\rho \models \varphi[c/\square]$ and $(\rho(c'),\rho) \models \varphi'$ iff $\rho \models \varphi'[c'/\square]$ and for each $1 \leq i \leq n$, $(\rho(c_i),\rho) \models \varphi_i$ iff $\rho \models \varphi_i[c_i/\square]$ and $(\rho(c'_i),\rho) \models \varphi'_i$ iff $\rho \models \varphi'_i[c'_i/\square]$. We can conclude that

$$\rho \models \varphi[c/\square] \wedge \varphi'[c'/\square] \wedge \bigwedge_{1 \leq i \leq n} (\varphi_i[c_i/\square] \wedge \varphi'_i[c'_i/\square])$$

and we are done.

For the reverse implication, we have that there must be some rule $\mu \in \mathcal{S}$ such that

$$\rho \models \varphi[c/\square] \wedge \varphi'[c'/\square] \wedge \bigwedge_{1 \leq i \leq n} (\varphi_i[c_i/\square] \wedge \varphi'_i[c'_i/\square])$$

Again, by Lemma 1 we have that $(\rho(c),\rho) \models \varphi$ iff $\rho \models \varphi[c/\square]$ and $(\rho(c'),\rho) \models \varphi'$ iff $\rho \models \varphi'[c'/\square]$ and for each $1 \leq i \leq n$, $(\rho(c_i),\rho) \models \varphi_i$ iff $\rho \models \varphi_i[c_i/\square]$ and $(\rho(c'_i),\rho) \models \varphi'_i$. Thus, it follows that $(\gamma,\rho) \models \varphi$ and $(\gamma',\rho) \models \varphi'$ and for each $1 \leq i \leq n$, $(\rho(c_i),\rho) \models \varphi_i$ and $(\rho(c_{i'}),\rho) \models \varphi'_i$. Therefore, $(\gamma,\gamma') \in \mathcal{R}_{k+1,m}$, and we are done.

We reduce the second part of the lemma to the first. For the direct implication, by Lemma 18, we have that $\gamma \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma'$ implies that there exist some k, m such that $(\gamma,\gamma') \in \mathcal{R}_{k+1,m}$. By the first part of the lemma, for each $1 \leq i \leq nc$, we have that $(\rho(c_i), \rho(c'_i)) \in \bigcup_{0 \leq m' \leq m} \mathcal{R}^{m'}_{k,m}$. Then, for each $1 \leq i \leq nc$, by Lemma 18, $(\rho(c_i), \rho(c'_i)) \in \bigcup_{0 \leq m' \leq m} \mathcal{R}^{m'}_{k,m}$ implies that $\rho(c_i) \Rightarrow^{\star\mathcal{T}}_{\mathcal{S}} \rho(c'_i)$. For the converse implication, for each $1 \leq i \leq nc$, by Lemma 18, $\rho(c_i) \Rightarrow^{\star\mathcal{T}}_{\mathcal{S}} \rho(c'_i)$ implies that there exist some k_i, m_i such that $(\rho(c_i), \rho(c'_i)) \in \bigcup_{0 \leq m_i' \leq m_i} \mathcal{R}^{m_i'}_{k_i,m_i}$. Let k be the maximum of k_i and m the maximum of m_i. By the first part of the lemma, we have that that $(\gamma,\gamma') \in \mathcal{R}_{k+1,m}$. By Lemma 18, we have that $(\gamma,\gamma') \in \mathcal{R}_{k+1,m}$ implies $\gamma \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma'$, and we are done. $\square$

The following formally states the property encoded by $path_{k,m}$:

Lemma 20. *Let $k, m \in \mathbb{N}$. Then $(\gamma,\gamma') \in \bigcup_{0 \leq m' \leq m} \mathcal{R}^{m'}_{k,m}$ iff there exists a $\rho : Var \to \mathcal{T}$ such that $\rho \models path_{k,m}$ and $\rho(c_0) = \gamma$, $\rho(c_m) = \gamma'$.*

Proof. We proceed by induction on k.

Base case If $k = 0$, then I_k only contains the empty sequence and $path_{0,m}$ becomes $\bigwedge_{0\leq i<m} c_i = c_{i+1}$. Thus, $\rho \models path_{0,m}$ iff $\rho(c_0) = ... = \rho(c_m)$. Such a ρ exists iff $\gamma = \gamma'$. On the other hand, since $\mathcal{R}_{0,m} = \emptyset$, it follows that $\bigcup_{0\leq m'\leq m} \mathcal{R}^{m'}_{0,m} = \mathcal{R}^{0}_{0,m}$, the reflexive closure of $\mathcal{R}_{0,m}$. Hence, we can conclude that $(\gamma, \gamma') \in \bigcup_{0\leq m'\leq m} \mathcal{R}^{m'}_{0,m}$ iff $\gamma = \gamma'$, and we are done.

Induction case We assume the lemma for k and we prove it for $k + 1$. For $0 \leq i_1 < m$ and $1 \leq i_2 \leq nc$ we define

$$\begin{aligned} path^{i_1,i_2}_{k,m} \quad \equiv \quad & \bigwedge_{\substack{s\in i_k \\ 0\leq i<m}} c_{i_1,i_2,s,i} = c_{i_1,i_2,s,i+1} \\ \wedge \quad & \bigwedge_{\substack{0\leq j<k \\ s\in I_j \\ 0\leq i<m}} (\bigvee_{\mu\in\mathcal{S}} rule^{\mu}_{i_1,i_2,s,i} \vee c_{i_1,i_2,s,i} = c_{i_1,i_2,s,i+1}) \end{aligned}$$

Intuitively, $path^{i_1,i_2}_{k,m}$ is $path_{k,m}$ with all the indexing sequences prefixed with i_1 and i_2. Then we can rearrange $path_{k+1,m}$ as follows

$$\begin{aligned} path_{k+1,m} \quad \leftrightarrow \quad & \bigwedge_{0\leq i<m} (\bigvee_{\mu\in\mathcal{S}} rule^{\mu}_{i} \vee c_i = c_{i+1}) \\ \wedge \quad & \bigwedge_{\substack{0\leq i_1<m \\ 1\leq i_2\leq nc}} (\bigwedge_{\substack{s\in i_k \\ 0\leq i<m}} c_{i_1,i_2,s,i} = c_{i_1,i_2,s,i+1} \\ & \qquad \wedge \bigwedge_{\substack{0\leq j<k \\ s\in I_j \\ 0\leq i<m}} (\bigvee_{\mu\in\mathcal{S}} rule^{\mu}_{i_1,i_2,s,i} \vee c_{i_1,i_2,s,i} = c_{i_1,i_2,s,i+1})) \end{aligned}$$

Essentially, we split the conjuncts over s from $path_{k+1,m}$ based on whether s is empty (first line) or the first two elements are some i_1 and i_2 (second and third lines). Notice that for some fixed i_1 and i_2, the last two lines are in fact $path^{i_1,i_2}_{k,m}$. Thus we can write $path_{k+1,m}$ as

$$path_{k+1,m} \leftrightarrow \bigwedge_{0\leq i<m} (\bigvee_{\mu\in\mathcal{S}} rule^{\mu}_{i} \vee c_i = c_{i+1}) \wedge \bigwedge_{\substack{0\leq i_1<m \\ 1\leq i_2\leq nc}} path^{i_1,i_2}_{k,m}$$

Notice that, for given i_1 and i_2, the only variables (possibly) shared by $path^{i_1,i_2}_{k,m}$ with the rest of the formula are $c_{i_1,i_2,0}$ and $c_{i_1,i_2,m}$. Thus, the existence of ρ with $\rho \models path_{k+1,m}$ becomes equivalent to the existence of ρ' and ρ_{i_1,i_2} for each $0 \leq i_1 < m$

and $1 \leq i_2 \leq nc$ with the following properties:

(1) $\rho' \models \bigwedge_{0 \leq i < m} (\bigvee_{\mu \in \mathcal{S}} rule_i^\mu \vee c_i = c_{i+1})$ and $\rho'(c_0) = \gamma$ and $\rho'(c_m) = \gamma'$; and

(2) $\rho_{i_1,i_2} \models path_{k,m}$ and $\rho_{i_1,i_2}(c_0) = \rho'(c_{i_1,i_2,0})$ and $\rho_{i_1,i_2}(c_m) = \rho'(c_{i_1,i_2,m})$ for each $0 \leq i_1 < m$ and $1 \leq i_2 \leq nc$.

By the induction hypothesis, the existence of ρ_{i_1,i_2} satisfying condition (2) is equivalent to $(\rho'(c_{i_1,i_2,0}), \rho'(c_{i_1,i_2,m})) \in \bigcup_{0 \leq m' \leq m} \mathcal{R}_{k,m}^{m'}$. Thus, it suffices to prove that $(\gamma, \gamma') \in \bigcup_{0 \leq m' \leq m} \mathcal{R}_{k+1,m}^{m'}$ iff there exists some ρ' with $\rho' \models \bigwedge_{0 \leq i < m} (\bigvee_{\mu \in \mathcal{S}} rule_i^\mu \vee c_i = c_{i+1})$ such that $(\rho'(c_{i_1,i_2,0}), \rho'(c_{i_1,i_2,m})) \in \bigcup_{0 \leq m' \leq m} \mathcal{R}_{k,m}^{m'}$ for each $0 \leq i_1 < m$ and $1 \leq i_2 \leq nc$.

Further, notice that for a given $0 \leq i < m$, the only variables shared by $\bigvee_{\mu \in \mathcal{S}} rule_i^\mu \vee c_i = c_{i+1}$ with the rest of the formula are c_i and c_{i+1}. Thus, there existence of ρ' with the above properties is equivalent to the existence of some $\gamma_0, ..., \gamma_m \in \mathcal{T}_{Cfg}$ and some $\rho_0, ..., \rho_{m-1}$ such that for each $0 \leq i < m$ it is the case that: $\rho_i(c_i) = \gamma_i$ and $\rho_i(c_{i+1}) = \gamma_{i+1}$; $(\rho_i(c_{i,i',0}), \rho_i(c_{i,i',m})) \in \bigcup_{0 \leq m' \leq m} \mathcal{R}_{k,m}^{m'}$, for each $1 \leq i' \leq nc$; and $\rho_i \models \bigvee_{\mu \in \mathcal{S}} rule_i^\mu \vee c_i = c_{i+1}$. By Lemma 19, we have that there exist some ρ_i with the first two properties such that $\rho_i \models \bigvee_{\mu \in \mathcal{S}} rule_i^\mu$ iff $(\gamma_i, \gamma_{i+1}) \in \mathcal{R}_{k+1,m}$. Thus, ρ_i exists iff $(\gamma_i, \gamma_{i+1}) \in \mathcal{R}_{k+1,m}$ or $\gamma_i = \gamma_{i+1}$. Therefore, it suffices to prove that $(\gamma, \gamma') \in \bigcup_{0 \leq m' \leq m} \mathcal{R}_{k+1,m}^{m'}$ iff there exist $\gamma_0, ..., \gamma_m$ such that for each $0 \leq i < m$, either $(\gamma_i, \gamma_{i+1}) \in \mathcal{R}_{k+1,m}$ or $\gamma_i = \gamma_{i+1}$, which is trivial, and we are done. □

The following states that $path(c, c')$ encodes $\Rightarrow_{\mathcal{S}}^{\star\mathcal{T}}$, the reflexive and transitive closure of the transition relation $\Rightarrow_{\mathcal{S}}^{\mathcal{T}}$:

Lemma 21. *Let $\rho : Var \to \mathcal{T}$. Then $\rho(c) \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \rho(c')$ iff $\rho \models path(c, c')$.*

Proof. We have that $\rho(c) \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \rho(c')$ iff there exist some $m' \in \mathbb{N}$ and some sequence $\gamma_0, ..., \gamma_{m'} \in \mathcal{T}_{Cfg}$ with $\gamma_0 = \rho(c)$ and $\gamma_{m'} = \rho(c')$ and $\gamma_i \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma_{i+1}$ for each $0 \leq i < m'$. By Definition 10, $\gamma_i \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \gamma_{i+1}$ iff for each $0 \leq i \leq m'$, there exists some k_i such that $(\gamma_i, \gamma_{i+1}) \in \mathcal{R}_{k_i}$. Further, by Lemma 18, that is iff for each $0 \leq i \leq m'$, there also exists some m_i such that $(\gamma_i, \gamma_{i+1}) \in \mathcal{R}_{k_i,m_i}$. Let k denote the maximum of $k_0, ..., k_{m-1}$ and m the maximum of $m', m_1, ..., m_{m'-1}$. Since $\mathcal{R}_{k_i,m_i} \subseteq \mathcal{R}_{k,m}$, we can conclude that $\rho(c) \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \rho(c')$ iff there exist some k and m such that $(\rho(c), \rho(c')) \in \bigcup_{0 \leq m' \leq m} \mathcal{R}_{k,m}^{m'}$. By Lemma 20, that is iff there exists some ρ' with $\rho'(c_0) = \rho(c)$ and $\rho'(c_m) = \rho(c')$ such that $\rho' \models path_{k,m}$. Since c, c' do not occur in $path_{k,m}$, that is iff there exists some ρ' with $\rho'(c) = \rho(c)$ and $\rho'(c') = \rho(c')$ such that $\rho' \models path_{k,m} \wedge c_0 = c \wedge c_m = c'$. But since c, c' are the only free variables in $path(c, c')$, that holds iff $\rho \models path(c, c')$, and we are done. □

The following states that $step(c, c')$ encodes the transition relation $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$:

Lemma 22. *Let $\rho : Var \to \mathcal{T}$. Then $\rho(c) \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \rho(c')$ iff $\rho \models step(c, c')$.*

Proof. By Lemma 19, second part, we have that $\rho(c) \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \rho(c')$ iff there exists some ρ' with $\rho'(c) = \rho(c)$ and $\rho'(c') = \rho(c')$ and $\rho'(c_i) \Rightarrow^{\star\mathcal{T}}_{\mathcal{S}} \rho'(c'_i)$ for each $1 \le i \le nc$ such that

$$\rho' \models \bigvee_{\mu \in \mathcal{S}} (\varphi[c/\square] \wedge \varphi'[c'/\square] \wedge \bigwedge_{1 \le i \le n} (\varphi_i[c_i/\square] \wedge \varphi'_i[c'_i/\square]))$$

By Lemma 21, we have that $\rho'(c_i) \Rightarrow^{\star\mathcal{T}}_{\mathcal{S}} \rho'(c'_i)$ iff $\rho' \models path(c_i, c'_i)$, for each $1 \le i \le nc$. Hence, we can combine the properties of ρ' into

$$\rho' \models \bigvee_{\mu \in \mathcal{S}} (\varphi[c/\square] \wedge \varphi'[c'/\square] \wedge \bigwedge_{1 \le i \le n} (\varphi_i[c_i/\square] \wedge \varphi'_i[c'_i/\square])) \\ \wedge \bigwedge_{1 \le i \le nc} path(c_i, c'_i)$$

Then, by rearranging the above, we can conclude that $\rho(c) \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \rho(c')$ iff there exist some ρ' with $\rho'(c) = \rho(c)$ and $\rho'(c') = \rho(c')$ such that

$$\rho' \models \bigvee_{\mu \in \mathcal{S}} (\varphi[c/\square] \wedge \varphi'[c'/\square] \wedge \bigwedge_{1 \le i \le n} (\varphi_i[c_i/\square] \wedge \varphi'_i[c'_i/\square] \wedge path(c_i, c'_i)) \\ \wedge \bigwedge_{n+1 \le i \le nc} path(c_i, c'_i)) \qquad (1)$$

Therefore, to prove the Lemma, it suffices to prove that there exists some ρ' with $\rho'(c) = \rho(c)$ and $\rho'(c') = \rho(c')$ satisfying (1) iff $\rho \models step(c, c')$.

For the direct implication, assume ρ' satisfy (1). Then, it follows that ρ' satisfies the first line in (1)

$$\rho' \models \bigvee_{\mu \in \mathcal{S}} (\varphi[c/\square] \wedge \varphi'[c'/\square] \wedge \bigwedge_{1 \le i \le n} (\varphi_i[c_i/\square] \wedge \varphi'_i[c'_i/\square] \wedge path(c_i, c'_i)))$$

Since the free variables occurring in the formula above are among c, c', c_1, c'_1, ..., c_{nc}, c'_{nc}, then the existence of such a ρ' implies that

$$\rho \models \exists c_1 ... c_{nc}\ \exists c'_1 ... c'_{nc} \exists \bar{x}\ (\bigvee_{\mu \in \mathcal{S}} (\varphi[c/\square] \wedge \varphi'[c'/\square] \\ \wedge \bigwedge_{1 \le i \le n} (\varphi_i[c_i/\square] \wedge \varphi'_i[c'_i/\square] \wedge path(c_i, c'_i))))$$

which is exactly the definition of $step(c, c')$. Thus $\rho \models step(c, c')$, and we are done.

For the reverse implication, assume $\rho \models step(c, c')$. Then, by the definition of $step(c, c')$ and since c, c' are the only free variables $step(c, c')$, we have that there exists some ρ' with $\rho'(c) = \rho(c)$ and $\rho'(c) = \rho(c)$ such that

$$\rho' \models \bigvee_{\mu \in \mathcal{S}} (\varphi[c/\square] \wedge \varphi'[c'/\square] \wedge \bigwedge_{1 \leq i \leq n} (\varphi_i[c_i/\square] \wedge \varphi'_i[c'_i/\square] \wedge path(c_i, c'_i)))$$

Then, there must be some $\mu \equiv \varphi \Rightarrow^{\exists} \varphi'$ if $\varphi_1 \Rightarrow^{\exists} \varphi'_1 \bigwedge ... \bigwedge \varphi_n \Rightarrow^{\exists} \varphi'_n$ such that

$$\rho' \models \varphi[c/\square] \wedge \varphi'[c'/\square] \wedge \bigwedge_{1 \leq i \leq n} (\varphi_i[c_i/\square] \wedge \varphi'_i[c'_i/\square] \wedge path(c_i, c'_i))$$

Since the variables $c_{n+1}, c'_{n+1}, ..., c_{nc}, c'_{nc}$ do not occur in the formula above, we can assume that $\rho'(c_{n+1}) = \rho'(c'_{n+1}), ..., \rho'(c_{nc}) = \rho'(c'_{nc})$. Trivially, $\rho'(c_{n+1}) \Rightarrow^{\star\mathcal{T}}_{\mathcal{S}} \rho'(c'_{n+1}), ..., \rho'(c_{nc}) \Rightarrow^{\star\mathcal{T}}_{\mathcal{S}} \rho'(c'_{nc})$. By Lemma 21, it follows that $\rho' \models path(c_{n+1}, c'_{n+1})$, ..., $\rho' \models path(c_{nc}, c'_{nc})$. Thus, we can conclude that

$$\begin{aligned}\rho' \ \models\ & \varphi[c/\square] \wedge \varphi'[c'/\square] \wedge \bigwedge_{1 \leq i \leq n} (\varphi_i[c_i/\square] \wedge \varphi'_i[c'_i/\square] \wedge path(c_i, c'_i)) \\ & \wedge \bigwedge_{n+1 \leq i \leq nc} path(c_i, c'_i)\end{aligned}$$

and therefore that ρ' satisfies (1), and we are done. □

The following establishes a relation between $path(c, c')$ and $step(c, c')$ which we use later on in Section 3.7.2:

Lemma 23. $\models path(c, c') \leftrightarrow c = c' \vee \exists c''\ (step(c, c'') \wedge path(c'', c'))$.

Proof. We prove that $\rho \models c = c' \vee \exists c''\ (step(c, c'') \wedge path(c'', c'))$ iff $\rho \models path(c, c')$. By Lemma 21, $\rho \models path(c, c')$ iff $\rho(c) \Rightarrow^{\star\mathcal{T}}_{\mathcal{S}} \rho(c')$, that is, iff there exist some $\gamma_0, ..., \gamma_n$ with $\rho(c) = \gamma_0$ and $\rho(c) = \gamma_n$ and $\gamma_i \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma_{i+1}$ for each $0 \leq i < n$. Equivalently, we can state it as either $\rho(c) = \rho(c')$ or there exist some $\gamma_0, \gamma_1, \gamma_n$ with $\rho(c) = \gamma_0$ and $\rho(c) = \gamma_n$ such that $\gamma_0 \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma_1$ and $\gamma_1 \Rightarrow^{\star\mathcal{T}}_{\mathcal{S}} \gamma_n$. Further, that is iff there exists some ρ' with $\rho'(c) = \rho(c)$ and $\rho'(c') = \rho(c')$ such that $\rho'(c) \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \rho'(c'')$ and $\rho'(c'') \Rightarrow^{\star\mathcal{T}}_{\mathcal{S}} \rho'(c')$. By Lemma 22 and Lemma 21, that is iff $\rho' \models step(c, c'') \wedge path(c'', c')$, and we are done. □

The following states that $succ(c, c')$ encodes the termination dependence relation $\succ$:

Lemma 24. *Let $\rho : Var \to \mathcal{T}$. Then $\rho(c') \prec \rho(c)$ iff $\rho \models succ(c, c')$.*

Proof. Recall from Definition 11 that $\gamma \succ \gamma'$ iff

- $\gamma \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \gamma'$; or
- there exists some rule

$$\varphi \Rightarrow^{\exists} \varphi' \ \text{ if } \ \varphi_1 \Rightarrow^{\exists} \varphi'_1 \wedge ... \wedge \varphi_n \Rightarrow^{\exists} \varphi'_n$$

  in $\mathcal{S}$, valuation $\rho' : Var \to \mathcal{T}$, and index $1 \leq i \leq n$ such that:

  (1) $(\gamma, \rho') \models \varphi$;

  (2) for each $1 \leq j < i$ and each $\gamma_j \in \mathcal{T}_{Cfg}$ with $(\gamma_j, \rho') \models \varphi_j$, there is some $\gamma'_j \in \mathcal{T}_{Cfg}$ such that $(\gamma'_j, \rho') \models \varphi'_j$ and $\gamma_j \Rightarrow^{\star\mathcal{T}}_{\mathcal{S}} \gamma'_j$; and

  (3) $(\gamma', \rho') \models \varphi_i$.

For each $1 \leq j < i$, since φ_j is well-defined (see Definition 14), there exists a unique γ_j with $(\gamma_j, \rho') \models \varphi_j$. Hence, condition (2) is equivalent to: "for each $1 \leq j < i$ there exist some $\gamma_j \in \mathcal{T}_{Cfg}$ with $(\gamma_j, \rho') \models \varphi_j$ and some $\gamma'_j \in \mathcal{T}_{Cfg}$ such that $(\gamma'_j, \rho') \models \varphi'_j$ and $\gamma_j \Rightarrow^{\star\mathcal{T}}_{\mathcal{S}} \gamma'_j$". By Lemma 22, $\rho(c) \Rightarrow^{\mathcal{T}}_{\mathcal{S}} \rho(c')$ iff $step(c, c')$. Thus, the first line in the definition of $succ(c, c')$ captures the first bullet in the definition of $\succ$. Therefore, it suffice to show that the second and third lines of the definition of $succ(c, c')$, namely

$$\rho \models \exists c_1 ... c_{nc} \, \exists c'_1 ... c'_{nc} \exists \bar{x} \, (\bigvee_{\mu \in \mathcal{S}} (\varphi[c/\square] \wedge \bigvee_{1 \leq i \leq n} (\varphi_i[c'/\square] \\ \wedge \bigwedge_{1 \leq j < i} (\varphi_j[c_j/\square] \wedge \varphi'_j[c'_j/\square] \wedge path(c_j, c'_j))))))$$

capture the second bullet. Since c, c' are the only free variables in the formula above, it follows that ρ satisfies it iff there exists some ρ' with $\rho'(c) = \rho(c)$ and $\rho'(c') = \rho(c')$ such that

$$\rho' \models \bigvee_{\mu \in \mathcal{S}} (\varphi[c/\square] \wedge \bigvee_{1 \leq i \leq n} (\varphi_i[c'/\square] \\ \wedge \bigwedge_{1 \leq j < i} (\varphi_j[c_j/\square] \wedge \varphi'_j[c'_j/\square] \wedge path(c_j, c'_j)))))$$

or, equivalently, iff there exist some rule μ and $1 \leq i \leq n$ such that

$$\rho' \models \varphi[c/\square] \land \varphi_i[c'/\square] \bigwedge_{1\leq j<i} (\varphi_j[c_j/\square] \land \varphi'_j[c'_j/\square] \land \mathit{path}(c_j, c'_j))$$

By Lemma 1, we have that $\rho' \models \varphi[c/\square]$ iff $(\rho'(c), \rho') \models \varphi$ and $\rho' \models \varphi_i[c'/\square]$ iff $(\rho'(c'), \rho') \models \varphi_i$, which are condition (1) and (3). Also by Lemma 1, for each $1 \leq j < i$, we have that $\rho' \models \varphi_j[c_j/\square]$ iff $(\rho'(c_j), \rho') \models \varphi_j$ and $\rho' \models \varphi'_j[c'_j/\square]$ iff $(\rho'(c'_j), \rho') \models \varphi'_j$, and, by Lemma 21, that $\rho' \models \mathit{path}(c_j, c'_j)$ iff $\rho'(c_j) \Rightarrow^{\star\mathcal{T}}_{\mathcal{S}} \rho'(c'_j)$, which is condition (2). We can conclude that the existence of some ρ' satisfying the formula above is equivalent to the existence of some ρ' satisfying the second bullet, and we are done. $\square$

The following states that $\mathit{diverge}(c)$ encodes the divergence predicate $\uparrow$:

Lemma 25. *Let $\rho : \mathit{Var} \to \mathcal{T}$. Then $\rho \models \mathit{diverge}(c)$ iff $\rho(c)$ does not terminate.*

Proof. For an arbitrary γ, we let $\mathit{Prop}(\gamma)$ be the property stating that there exists an infinite set P_γ of finite $\succ$-sequences starting at γ. First we prove that $\mathit{Prop}(\gamma)$ holds iff γ does not terminate. For the direct implication, we inductively construct an infinite $\succ$-sequence γ_0, ..., γ_n, ...such that $\gamma_0 = \gamma$ and $\mathit{Prop}(\gamma_n)$ holds for all n. $\mathit{Prop}(\gamma_0)$ holds because $\gamma_0 = \gamma$. Now, let us inductively assume $\mathit{Prop}(\gamma_n)$ holds and let $\succ(\gamma_n) = \{\gamma \mid \gamma_n \succ \gamma\}$ be the set of successors of γ_n. For each $\gamma \in \succ(\gamma_n)$, let P'_γ be the set $\{\tau \mid \tau \in P_{\gamma_n}$ and $\gamma_n\gamma$ is a prefix of $\tau\}$. Clearly, the sets P'_γ form a partition of P_{γ_n}. One of the assumption of relative completeness is that each configuration has a finite number of $\succ$-successors, that is, $\succ(\gamma_n)$ is finite. Since P_{γ_n} is infinite (because we assumed $\mathit{Prop}(\gamma_n)$), there is at least one $\gamma \in \succ(\gamma_n)$ with P'_γ infinite. Then we choose γ_{n+1} to be γ. Note that $\gamma_n \succ \gamma_{n+1}$ and $P_{\gamma_{n+1}} = \{\tau \mid \gamma_n\tau \in P'_{\gamma_{n+1}}\}$ is infinite, thus $\mathit{Prop}(\gamma_{n+1})$ holds. We can conclude that γ does not terminate. For the converse implication, it suffices to notice that if there is an infinite $\succ$-sequence starting at γ, then the set of finite prefixes of that sequence is infinite. A direct consequence of this result is that γ does not terminate iff for each n, there exists a $\succ$-sequence of length n starting at γ. Indeed, if γ does not terminate, then for each n we can take the prefix of length n of the infinite sequence starting at γ. Conversely, if there exists some $\succ$-sequence starting at γ for each n, then the set of such sequences if infinite, and thus γ does not terminate.

Using the result above, it suffices to prove that $\rho \models \mathit{diverge}(c)$ iff for each m there exists some $\succ$-sequence starting at $\rho(c)$. By Lemma 24, we have that for each

m there exists some ρ' with $\rho'(c) = \rho(c)$ such that $\rho' \models \bigwedge_{0 \le i < m} succ(c_i, c_{i+1}) \wedge c_0 = c$ iff $\rho'(c_0) = \rho'(c) = \rho(c)$ and $\rho'(c_0) \succ ... \succ \rho'(c_m)$, that is, iff there is some $\succ$-sequence of length m starting at $\rho(c)$, and we are done. □

The following establishes a relation between $diverge(c)$ and $succ(c, c')$ which we use later on in Section 3.7.2:

Lemma 26. $\models diverge(c) \leftrightarrow \exists c' \ (succ(c, c') \wedge diverge(c'))$.

Proof. We prove that $\rho \models diverge(c)$ iff $\rho \models \exists c' \ (succ(c, c') \wedge diverge(c'))$. By Lemma 25, $\rho \models diverge(c)$ iff $\rho(c)$ does not terminate, that is, iff there exist some $\gamma_0, ..., \gamma_n, ...$ with $\rho(c) = \gamma_0$ and $\gamma_i \succ \gamma_{i+1}$ for each $i \geq 0$. Equivalently, we can state it as there exist some γ_0, γ_1 with $\rho(c) = \gamma_0$ such that $\gamma_0 \succ \gamma_1$ and γ_1 does not terminate. Further, that is iff there exists some ρ' with $\rho'(c) = \rho(c)$ such that $\rho'(c) \succ \rho'(c')$ and $\rho'(c')$ does not terminate. By Lemma 25, that is iff $\rho' \models succ(c, c') \wedge diverge(c)$, and we are done. □

The following summarises the main properties of $step(c, c')$, of $path(c, c')$, of $succ(c, c')$, and of $diverge(c)$:

Lemma 27. *Let $\rho : Var \to \mathcal{T}$. Then $\rho(c) \Rightarrow_{\mathcal{S}}^{\mathcal{T}} \rho(c')$ iff $\rho \models step(c, c')$; $\rho(c) \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \rho(c')$ iff $\rho \models path(c, c')$; $\rho(c') \prec \rho(c)$ iff $\rho \models succ(c, c')$; and $\rho(c)$ does not terminate iff $\rho \models diverge(c)$.*

Proof. Follows by Lemmas 22 , 21 , 24 and 25. □

Finally, the following establishes the property of $coreach(\varphi)$:

Lemma 28. *Let $\gamma \in \mathcal{T}_{Cfg}$ and $\rho : Var \to \mathcal{T}$. Then $(\gamma, \rho) \models coreach(\varphi)$ iff there exists some $\gamma' \in \mathcal{T}_{Cfg}$ with $(\gamma', \rho) \models \varphi$ and $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$.*

Proof. We have that $(\gamma, \rho) \models coreach(\varphi)$ iff there exists some ρ' which agrees with ρ on $Var \setminus \{c, c'\}$ with $\rho'(c) = \gamma$ such that $\rho' \models \varphi[c'/\square] \wedge path(c, c')$. By Lemma 21, that is iff there exists some ρ' which agrees with ρ on $Var \setminus \{c, c'\}$ with $\rho'(c) = \gamma$ such that $\rho' \models \varphi[c'/\square]$ and $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \rho'(c')$. Let us denote $\rho(c')$ by γ'. Then $(\gamma, \rho) \models coreach(\varphi)$ iff there exist some γ' and some ρ' which agrees with ρ on $Var \setminus \{c, c'\}$ with $\rho'(c) = \gamma$ and $\rho'(c') = \gamma'$ such that $\rho' \models \varphi[c'/\square]$ and $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$. We have that $(\rho'(c'), \rho') \models \varphi$ iff $\rho' \models \varphi[c'/\square]$. Since c, c' do not occur in φ, and ρ and ρ' agree on $Var \setminus \{c, c'\}$, it follows that $\rho' \models \varphi[c'/\square]$ iff $(\gamma', \rho) \models \varphi$. Thus, we can conclude that $(\gamma, \rho) \models coreach(\varphi)$ iff there exists some γ' with $(\gamma', \rho) \models \varphi$ and $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$, and we are done. □

$$\begin{aligned}
\overline{rule}_\mu(\bar{s}, j, i) \equiv{}& \exists c \exists c' \exists c_1 ... c_n \exists c'_1 ... c'_n \; (\exists \bar{x} \; (\varphi[c/\Box] \wedge \varphi'[c'/\Box] \wedge \bigwedge_{1 \leq i' \leq n} \varphi_{i'}[c_{i'}/\Box] \wedge \varphi'_{i'}[c'_{i'}/\Box]) \\
& \wedge \beta(a, b, add(\bar{s}, j, i), \alpha(c)) \wedge \; \beta(a, b, add(\bar{s}, j, i+1), \alpha(c')) \\
& \wedge \bigwedge_{1 \leq i' \leq n} (\beta(a, b, add(\bar{s}, j, i, i', 0), \alpha(c_{i'})) \wedge \beta(a, b, add(\bar{s}, j, i, i', m), \alpha(c'_{i'})))) \\
\overline{id}(\bar{s}, j, i) \equiv{}& \exists c \exists c' \; (\beta(a, b, add(\bar{s}, j, i), \alpha(c)) \wedge \beta(a, b, add(\bar{s}, j, i+1), \alpha(c')) \wedge c = c') \\
\overline{path}(c, c') \equiv{}& \exists k \exists m \; (k \geq 0 \wedge m \geq 0 \wedge \exists a \exists b \; (\exists c_0 \; (\beta(a, b, 0, \alpha(c_0)) \wedge c = c_0) \\
& \wedge \exists c_m \; (\beta(a, b, m, \alpha(c_m)) \wedge c' = c_m) \\
& \wedge \forall \bar{s} \forall j \forall i \; (seq(\bar{s}, j) \wedge j = k \wedge 0 \leq i < m \rightarrow \overline{id}(\bar{s}, j, i)) \\
& \wedge \forall \bar{s} \forall j \forall i \; (seq(\bar{s}, j) \wedge j < k \wedge 0 \leq i < m \rightarrow \bigvee_{\mu \in \mathcal{S}} \overline{rule}_\mu(\bar{s}, j, i) \vee \overline{id}(\bar{s}, j, i)))) \\
\overline{diverge}(c) \equiv{}& \forall m \exists a \exists b \; (\exists c_0 \; (\beta(a, b, 0, \alpha(c_0)) \wedge c = c_0) \wedge \forall i \; (0 \leq i < m \\
& \rightarrow \exists c_i \exists c_{i+1} \; (\beta(a, b, i, \alpha(c_i)) \wedge \beta(a, b, i+1, \alpha(c_{i+1})) \wedge succ(c_i, c_{i+1}))))
\end{aligned}$$

Figure 3.8: FOL definitions of a finite $\Rightarrow^{\mathcal{T}}_{\mathcal{S}}$-sequence ($\overline{path}$) and an infinite $\succ$-sequence ($\overline{diverge}$)

Formulae Gödelization

We use Gödel's β predicate to encode quantification over sequences of configurations in FOL (see [104] for an accessible introduction to Gödelization and the β predicate). The predicate β relies on the reminder of a when divided by b, written a mod b and defined as

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

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

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

The assumptions on the model $\mathcal{T}$ allow us to express β. If $u_0, ..., u_n$ is a sequence of natural numbers, then there exist natural numbers a and b such that $\beta(a, b, i, x)$ holds iff $x = u_i$, for each $0 \leq i \leq n$ and x. Thus, for any given n, we can systematically translate sentences $\exists u_0, ..., u_n \; \varphi$ into equivalent sentences $\exists a, b \, \overline{\varphi}$. As part of this translation, each atomic formula *pred* of φ is translated into

$$\exists u_{i_1}, ..., u_{i_k} \; (\beta(a,\ b,\ i_1,\ u_{i_1}) \wedge \cdots \wedge \beta(a,\ b,\ i_k,\ u_{i_k}) \wedge pred)$$

where $u_{i_1}, ..., u_{i_k}$ are all the variables among $u_0, ..., u_n$ occurring in *pred*. Even if n itself is quantified, only a fixed (independent of n) subset of the variables $u_0, ..., u_n$ can occur in *pred*, making $\overline{\varphi}$ a proper FOL formula. Thus, β enables quantification over sequences.

Using the injective function $\alpha : \mathcal{T}_{Cfg} \to \mathbb{N}$ we can extend the result above to sequences of configurations. Sentences of the form $\exists c_0, ..., c_n\, \varphi$ 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 formula *pred* of φ containing variables $c_{i_1}, ..., c_{i_k}$ with

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

The injectivity of α guarantees that different free occurrences of the same variable c_i in φ are correctly related in $\overline{\varphi}$.

Thus far, we can encode quantification over finite sequences of configurations. A finite sequences is (syntactically) represented as a finite set of variables indexed by natural numbers. However, $path(c, c')$ quantifies over the set $C_{k,m}$, which contain variables indexed by sequences s, i. To encode a such a sequence s, i into a natural number, we need division ($a \div b$) and power (a^b), which can be defined as follows.

$$\begin{aligned} a \div b = d &\equiv b \times d \leq a \wedge b \times (d+1) > a \\ a^b = d &\equiv \exists x_0 ... x_b\, (x_0 = 1 \wedge x_b = d \wedge \forall i\, (1 \leq i \leq b \to x_i = x_{i-1} \times a)) \end{aligned}$$

Equivalently, we can define a^b using β and only four quantifiers.

$$\begin{aligned} a^b = d \ \equiv\ & b \geq 0 \wedge \exists a' \exists b'\, (\beta(a', b', 0, 1) \wedge \beta(a', b', b, d) \wedge \forall i\, (1 \leq i \leq b \\ & \to \exists x \exists x'\, (\beta(a', b', i, x)) \wedge \beta(a', b', i-1, x') \wedge x = x' \times a))) \end{aligned}$$

For notational simplicity, we write division, reminder, and power as functions rather than as relations. Also, to save space, we write $a \leq b \leq d$ instead of $a \leq b \wedge b \leq d$. Let p be the maximum of m and $nc + 1$, and let $s = i_1, ..., i_{2j}$ be a sequence of indices. Recall that $i_1, i_3, ...i_{2j-1}$ are between 0 and $m - 1$, and $i_2, i_4, ..., i_{2j}$ between 1 and nc. Then, we can view s as a number $\bar{s}$ in base p, namely $\bar{s} = \Sigma_{t=1}^{2j} i_t \times p^{t-1}$. Conversely, we encode the fact that $\bar{s}$ is indeed representing some sequence s as follows

$$\begin{aligned} seq(\bar{s}, j) \equiv\ & \bar{s} = 0 \wedge j = 0 \\ & \vee\ j > 0 \wedge p^{2\times j-1} \leq \bar{s} < p^{2\times j} \\ & \quad \wedge\ \forall t\ (1 \leq t \leq j \rightarrow \exists d\ (d = (\bar{s} \div p^{2\times t-2}) \bmod p \wedge 0 \leq d < m)) \\ & \quad \wedge\ \forall t\ (1 \leq t \leq j \rightarrow \exists d\ (d = (\bar{s} \div p^{2\times t-1}) \bmod p \wedge 1 \leq d \leq nc)) \end{aligned}$$

Intuitively, the first line covers the case of the empty sequence ($j = 0$). For the case of non-empty sequences, the second line states that $\bar{s}$ has exactly $2j$ digits, the third that the digits on even positions (corresponding to $i_1, ..., i_{2j-1}$) are between 0 and $m - 1$, and the fourth that the digits on odd positions (corresponding to $i_2, ..., i_{2j}$) are between 1 and nc. Similarly, to the sequence s, i we associate the number $\Sigma_{t=1}^{2j} i_t \times p^{t-1} + i \times p^{2\times j}$. For convenience, we define

$$\begin{aligned} add(\bar{s}, j, a) &\equiv \bar{s} + a \times p^{2\times j} \\ add(\bar{s}, j, a, b, d) &\equiv \bar{s} + a \times p^{2\times j} + b \times p^{2\times j+1} + d \times p^{2\times j+2} \end{aligned}$$

For some $\bar{s}$ associated to some s of length $2j$, these functions give the numbers associated to the sequences s, a and s, a, b, d.

Figure 3.8 presents the encoding of a finite $\Rightarrow_S^{\mathcal{T}}$-sequence ($\overline{path}(c, c')$) and an infinite $\succ$-sequence ($\overline{diverge}(c)$) using only a fixed number of quantifiers. Recall that $path(c, c')$ existentially quantifies over the finite set of variables $C_{k,m}$. Using the encoding of sequences s, i into numbers in base p with at most $2k + 1$ digits, we can instead quantify over a sequence of configurations of length at most p^{2k+1}. Further, using the β predicate and the injective function $\alpha : \mathcal{T}_{Cfg} \rightarrow \mathbb{N}$, we can instead quantify over two natural numbers, namely a, b. Then, we can replace the big conjunctions in $path(c, c')$ with universal quantification over $\bar{s}, j, i$ and the appropriate restrictions like $seq(\bar{s}, j)$, $0 \leq i < m$, etc. We introduce $\overline{rule_\mu}(\bar{s}, j, i)$ as the encoding of $rule_{s,i}^{\mu}$. It replaces the configuration variables indexed by sequences with locally quantified variables $c, c', c_1, c'_1, ..., c_n, c'_n$ (n is the number of conditions of μ), and it existentially quantifies $\bar{x}$, the variables occurring free in the rules. It also ensures that these local configuration variables are instantiated consistently across the formula. For example, for the variable $c_{s,i}$, the predicate $\beta(a,\ b,\ add(\bar{s}, j, i),\ \alpha(c))$ ensures that the variable c is instantiated with the value intended for $c_{s,i}$, that is, the configuration mapped by α into the number on the position $\bar{s} + i \times p^{2\times j}$ of the sequence of natural numbers Gödelized by a and b. Similarly, $\overline{id}(\bar{s}, j, i)$ encodes $c_{s,i} = c_{s,i+1}$. Formally, we have the following result:

Lemma 29. $\models path(c, c') \leftrightarrow \overline{path}(c, c')$.

Proof. Let $\rho, \bar{\rho} : Var \to \mathcal{T}$. We call $\rho, \bar{\rho}$ a *Gödel pair* iff

(1) ρ and $\bar{\rho}$ agree on c, c', k, m

(2) for each indexing sequence s of length $2j$ with $0 \leq j \leq \rho(k)$ and for each $0 \leq i \leq \rho(m)$ it is the case that

$$\beta(\bar{\rho}(a),\ \bar{\rho}(b),\ \bar{s} + i \times p^{2\times j},\ \alpha(\rho(c_{s,i})))$$

holds, where p is the maximum of $\rho(m)$ and nc. Intuitively, this states that the number associated to the configuration $\rho(c_{s,i})$ by the injective function α is on the position $\bar{s} + i \times p^{2\times j}$ (the position associated to the sequence s, i) in the sequence of natural numbers Gödelized by the pair $\bar{\rho}(a), \bar{\rho}(b)$.

We intend to use ρ to interpret the top-level existential quantifiers in $path(c, c')$ and $\bar{\rho}$ to interpret the top-level existential quantifiers in $\overline{path}(c, c')$. To simplify the notation, for variables a of sort $\mathbb{N}$, like k, m, a, b, i, j, etc, we use its syntactic name, e.g. k, to also refer to its interpretation, e.g. $\rho(k)$. Condition (1) above ensures that variables common to both $path(c, c')$ and $\overline{path}(c, c')$ are interpreted consistently by ρ and $\bar{\rho}$, and by valuations which agree with ρ or $\bar{\rho}$ on these common variables. With this convention, the instance of the β predicate above becomes

$$\beta(a,\ b,\ \bar{s} + i \times p^{2\times j},\ \alpha(\rho(c_{s,i})))$$

We prove that for each Gödel pair $\rho, \bar{\rho}$, the following two statements are equivalent

$$\rho \models c = c_0 \wedge c' = c_m \wedge path_{k,m} \tag{3}$$

$$\begin{aligned}
\bar{\rho} \models\ & \exists c_0\ (\beta(a, b, 0, \alpha(c_0)) \wedge c = c_0) \\
& \wedge \exists c_m\ (\beta(a, b, m, \alpha(c_m)) \wedge c = c_m) \\
& \wedge \forall \bar{s} \forall j \forall i\ (seq(\bar{s}, j) \wedge j = k \wedge 0 \leq i < m \to \overline{id}(\bar{s}, j, i)) \\
& \wedge \forall \bar{s} \forall j \forall i\ (seq(\bar{s}, j) \wedge j < k \wedge 0 \leq i < m \\
& \qquad \to \bigvee_{\mu \in \mathcal{S}} \overline{rule_\mu}(\bar{s}, j, i) \vee \overline{id}(\bar{s}, j, i))
\end{aligned} \tag{4}$$

First, we prove that $\rho \models rule^{\mu}_{s,i}$ iff $\bar{\rho} \models \overline{rule_\mu}(\bar{s}, j, i)$ for each indexing sequence s of length $2j$ with $0 \leq j < k$ and each $0 \leq i \leq m$ and each $\bar{s}$ such that $\bar{s}$ is the number associated to the sequence s, i. Recall that $\overline{rule}_\mu(\bar{s}, j, i)$ is obtained from $rule^\mu_{s,i}$ by

- substituting $c_{s,i}, c_{s,i+1}$ with c, c' and $c_{s,i,i',0}, c_{s,i,i',m}$ with c_j, c'_j for each $1 \leq i' \leq n$;
- by adding constrains on $c, c', c_1, c'_1, ..., c_n, c_n$ using the β predicate and the α function; and
- existentially quantifying $c, c', c_1, c'_1, ..., c_n, c'_n$.

Let $\bar{\rho}_\mu : Var \to \mathcal{T}$ which agrees with $\bar{\rho}$ on $Var \setminus \{c, c', c_1, c'_1, ..., c_n, c'_n\}$. Then we have that $\bar{\rho}_\mu$ satisfies the constrains on the configuration variables in the definition of $\overline{rule}_\mu(\bar{s}, j, i)$, namely

$$\begin{aligned}\rho_\mu \models\, & \beta(a, b, add(\bar{s}, j, i), \alpha(c)) \wedge\ \beta(a, b, add(\bar{s}, j, i + 1), \alpha(c')) \\ & \wedge \bigwedge_{1 \leq i' \leq n} (\beta(a, b, add(\bar{s}, j, i, i', 0), \alpha(c_{i'})) \\ & \qquad \wedge \beta(a, b, add(\bar{s}, j, i, i', m), \alpha(c'_{i'})))\end{aligned}$$

iff $\beta(a, b, add(\bar{s}, j, i), \alpha(\bar{\rho}_\mu(c)))$ and $\beta(a, b, add(\bar{s}, j, i + 1), \alpha(\bar{\rho}_\mu(c')))$ hold, and for each $1 \leq i' \leq n$, it is the case that both $\beta(a, b, add(\bar{s}, j, i, i', 0), \alpha(\bar{\rho}_\mu(c_{i'})))$ and $\beta(a, b, add(\bar{s}, j, i, i', m), \alpha(\bar{\rho}_\mu(c'_{i'})))$ hold. Since ρ and $\bar{\rho}$ are a Gödel pair, condition (2) implies that $\beta(a, b, add(\bar{s}, j, i), \alpha(\rho(c_{s,i})))$ and $\beta(a, b, add(\bar{s}, j, i + 1), \alpha(\rho(c_{s,i+1})))$ hold, and also that for each $1 \leq i' \leq n$, both $\beta(a, b, add(\bar{s}, j, i, i', 0), \alpha(\rho(c_{s,i,i',0})))$ and $\beta(a, b, add(\bar{s}, j, i, i', m), \alpha(\rho(c'_{s,i,i',m})))$ hold. Further, since α is injective, and the first three arguments of β uniquely determine the fourth argument, we can conclude there exists a unique $\bar{\rho}_\mu$ which satisfies the constrains above, namely the one with $\bar{\rho}_\mu(c) = \rho(c_{s,i})$ and $\bar{\rho}_\mu(c') = \rho(c_{s,i+1})$, and with $\bar{\rho}_\mu(c_{i'}) = \rho(c_{s,i,i',0})$ and $\bar{\rho}_\mu(c'_{i'}) = \rho(c_{s,i,i',m})$ for each $1 \leq i' \leq n$. Then, it follows that $\rho \models rule^\mu_{s,i}$ iff $\bar{\rho}_\mu$ satisfies $rule^\mu_{s,i}$ with the configuration variables substituted, that is, iff $\bar{\rho} \models \overline{rule}_\mu(\bar{s}, j, i)$.

Similarly, we prove that $\rho \models c_{s,i} = c_{s,i+1}$ iff $\bar{\rho} \models \overline{id}(s, j, i)$ for each indexing sequence s of length $2j$ with $0 \leq j \leq k$ and each $0 \leq i \leq m$ and each $\bar{s}$ such that $\bar{s}$ is the number associated to the sequence s, i. Let $\bar{\rho}_{\text{id}} : Var \to \mathcal{T}$ which agrees with $\bar{\rho}$ on $Var \setminus \{c, c'\}$. Then we have that

$$\bar{\rho}_{\text{id}} \models \beta(a, b, add(\bar{s}, j, i), \alpha(c)) \wedge\ \beta(a, b, add(\bar{s}, j, i + 1), \alpha(c'))$$

iff $\beta(a, b, add(\bar{s}, j, i), \alpha(\bar{\rho}_{\text{id}}(c)))$ and $\beta(a, b, add(\bar{s}, j, i+1), \alpha(\bar{\rho}_{\text{id}}(c')))$ hold. Since ρ and $\bar{\rho}$ are a Gödel pair, condition (2) implies that $\beta(a, b, add(\bar{s}, j, i), \alpha(\rho(c_{s,i})))$ and $\beta(a, b, add(\bar{s}, j, i+1), \alpha(\rho(c_{s,i+1})))$ hold. Then, we can conclude there exists a unique $\bar{\rho}_{\text{id}}$ which satisfies the constrains above, namely the one with $\bar{\rho}_{\text{id}}(c) = \rho(c_{s,i})$ and $\bar{\rho}_{\text{id}}(c') = \rho(c_{s,i+1})$. It follows trivially that $\rho \models c_{s,i} = c_{s,i+1}$ iff $\bar{\rho}_{\text{id}} \models c = c'$, that is, iff $\bar{\rho} \models \overline{id}(s, j, i)$.

Based on the above, and on the fact that $seq(\bar{s}, j)$ holds iff $\bar{s}$ is a number associated to a sequence s of length $2j$, we conclude that

$$\begin{aligned}
&\rho \models path_{k,m} \\
&\bar{\rho} \models \forall \bar{s} \forall j \forall i \, (seq(\bar{s}, j) \land j = k \land 0 \le i < m \to \overline{id}(\bar{s}, j, i)) \\
&\qquad \land \forall \bar{s} \forall j \forall i \, (seq(\bar{s}, j) \land j < k \land 0 \le i < m \\
&\qquad\qquad \to \bigvee_{\mu \in \mathcal{S}} \overline{rule_\mu}(\bar{s}, j, i) \lor \overline{id}(\bar{s}, j, i))
\end{aligned}$$

Finally, notice that by condition (2), $\beta(a, b, 0, \alpha(\rho(c_0)))$ and $\beta(a, b, m, \alpha(\rho(c_m)))$ hold. Thus, we have that

$$\begin{aligned}
&\bar{\rho} \models \exists c_0 \, (\beta(a, b, 0, \alpha(c_0)) \land c = c_0) \\
&\qquad \land \exists c_m \, (\beta(a, b, m, \alpha(c_m)) \land c' = c_m)
\end{aligned}$$

iff $\rho(c_0) = \bar{\rho}(c)$ and $\rho(c_m) = \bar{\rho}(c')$, that is, iff $\rho \models c = c_0 \land c' = c_m$. Therefore, we conclude that statements (3) and (4) are equivalent.

To complete the proof of the lemma, we show for each ρ' that $\rho' \models path(c, c')$ iff $\rho' \models \overline{path}(c, c')$. We have that $\rho' \models path(c, c')$ iff there exist some $k, m \in \mathbb{N}$ and some $\rho : Var \to \mathcal{T}$ which agrees with ρ' on c, c' such that statement (3) holds. We also have that $\rho' \models \overline{path}(c, c')$ iff there exist some $k, m, a, b \in \mathbb{N}$ and some $\bar{\rho} : Var \to \mathcal{T}$ which agrees with ρ' on c, c' such that statement (4) holds. We know that (3) and (4) are equivalent for Gödel pairs, hence it suffices to construct for each ρ satisfying (3) a $\bar{\rho}$ such that $\rho, \bar{\rho}$ are a Gödel pair, and conversely, to construct for each $\bar{\rho}$ satisfying (4) a ρ such that $\rho, \bar{\rho}$ are a Gödel pair. For both directions, we choose such that ρ and $\bar{\rho}$ agree on c, c', k, m, thus we satisfy (1). From ρ, we construct $\bar{\rho}$ by choosing a, b to be the Gödel numbers associated to the sequence $(\alpha(c_{s,i}))$, where each indexing sequence s is of length $2j$ with $0 \le j \le k$ and for each i is such that $0 \le i \le m$ (the numbers are ordered in a sequence according to the numbers associated to the sequences s, i). Conversely, from $\bar{\rho}$, (4) implies that

$\bar{\rho} \models \exists c'' \, (\beta(a,\ b,\ \bar{s} + i \times p^{2\times j},\ \alpha(c'')))$ for each number $\bar{s}$ associated to an indexing sequence s of length $2j$ with $0 \leq j \leq k$ and for each $0 \leq i \leq m$. Then we choose ρ such that $\rho(c_{s,i})$ is the unique configurations in which c'' is instantiated in the formula above. In both cases, the pair $\rho, \bar{\rho}$ satisfies (2), and we are done. $\square$

The encoding of *diverge*(c) follows the same pattern as that of *path*(c, c'): it replaces the quantification over a sequence of configurations with the quantification over a and b, it replaces the big conjunction with the universal quantification over i and the restriction $0 \leq i < m$, and uses β and α to ensure the locally quantified variables c_i and c_{i+1} are instantiated consistently. Formally, we have the following result:

Lemma 30. $\models$ *diverge*$(c) \leftrightarrow \overline{\textit{diverge}}(c)$.

Proof. The proof takes a similar approach to the previous one. Let $\rho, \bar{\rho} : \mathit{Var} \to \mathcal{T}$. We call $\rho, \bar{\rho}$ a *Gödel pair* iff

(1) ρ and $\bar{\rho}$ agree on c, m

(2) for each $0 \leq i \leq \rho(m)$ it is the case that

$$\beta(\bar{\rho}(a),\ \bar{\rho}(b),\ i,\ \alpha(\rho(c_i)))$$

holds. Intuitively, this states that the number associated to the configuration $\rho(c_i)$ by the injective function α is on the position i in the sequence of natural numbers Gödelized by the pair $\bar{\rho}(a), \bar{\rho}(b)$.

We intend to use ρ to interpret the top-level quantifiers in *diverge*(c) and $\bar{\rho}$ to interpret the top-level quantifiers in $\overline{\textit{diverge}}(c)$. To simplify the notation, for variables a of sort $\mathbb{N}$, like, m, a, b, i, etc, we use its syntactic name, e.g. m, to also refer to its interpretation, e.g. $\rho(k)$. Condition (1) above ensures that variables common to both *diverge*(c) and $\overline{\textit{diverge}}(c)$ are interpreted consistently by ρ and $\bar{\rho}$, and by valuations which agree with ρ or $\bar{\rho}$ on these common variables. With this convention, the instance of the β predicate above becomes

$$\beta(a,\ b,\ i,\ \alpha(\rho(c_i)))$$

We prove that for each Gödel pair $\rho, \bar{\rho}$, the following two statements are equivalent

$$\rho \models c = c_0 \wedge \bigwedge_{0 \le i < m} succ(c_i, c_{i+1}) \tag{3}$$

$$\begin{aligned}\bar{\rho} \models\ & \exists c_0\ (\beta(a, b, 0, \alpha(c_0)) \wedge c = c_0) \\ & \wedge \forall i\ (0 \le i < m \to \exists c_i \exists c_{i+1}\ (\beta(a, b, i, \alpha(c_i)) \\ & \qquad \wedge \beta(a, b, i + 1, \alpha(c_{i+1})) \wedge succ(c_i, c_{i+1})))\end{aligned} \tag{4}$$

First, we prove that $\rho \models succ(c_i, c_{i+1})$ iff

$$\begin{aligned}\bar{\rho} \models\ & \exists c_i \exists c_{i+1}\ (\beta(a, b, i, \alpha(c_i)) \wedge \beta(a, b, i + 1, \alpha(c_{i+1})) \\ & \wedge succ(c_i, c_{i+1}))\end{aligned} \tag{5}$$

for each $0 \le i < m$. Let $\bar{\rho}' : Var \to \mathcal{T}$ which agrees with $\bar{\rho}$ on $Var \setminus \{c_i, c_{i+1}\}$. Then we have that $\bar{\rho}'$ satisfies the constrains on the configuration variables namely

$$\bar{\rho}' \models \beta(a, b, i, \alpha(c_i)) \wedge \beta(a, b, i + 1, \alpha(c_{i+1}))$$

iff $\beta(a, b, i, \alpha(\bar{\rho}'(c_i)))$ and $\beta(a, b, i + 1, \alpha(\bar{\rho}'(c_{i+1})))$ hold. Since ρ and $\bar{\rho}$ are a Gödel pair, condition (2) implies that $\beta(a, b, i, \alpha(\rho(c_i)))$ and $\beta(a, b, i + 1, \alpha(\rho(c_{i+1})))$ hold. Further, since α is injective, and the first three arguments of β uniquely determine the fourth argument, we can conclude there exists a unique $\bar{\rho}'$ which satisfies the constrains above, namely the one with $\bar{\rho}'(c_i) = \rho(c_i)$ and $\bar{\rho}'(c_{i+1}) = \rho(c_{i+1})$. Then, it follows that $\rho \models succ(c_i, c_{i+1})$ iff $\bar{\rho}'$ satisfies

$$\bar{\rho}' \models \beta(a, b, i, \alpha(c_i)) \wedge \beta(a, b, i + 1, \alpha(c_{i+1})) \wedge succ(c_i, c_{i+1})$$

that is, iff $\bar{\rho}$ satisfies (5).

Based on the above, we can conclude that

$$\rho \models \bigwedge_{0 \le i < m} succ(c_i, c_{i+1})$$

$$\begin{aligned}\bar{\rho} \models\ & \forall i\ (0 \le i < m \to \exists c_i \exists c_{i+1}\ (\beta(a, b, i, \alpha(c_i)) \\ & \qquad \wedge \beta(a, b, i + 1, \alpha(c_{i+1})) \wedge succ(c_i, c_{i+1})))\end{aligned}$$

Finally, notice that by condition (2), the predicates $\beta(a, b, 0, \alpha(\rho(c_0)))$ holds. Thus, we have that

$$\bar{\rho} \models \exists c_0 \, (\beta(a, b, 0, \alpha(c_0)) \wedge c = c_0)$$

iff $\rho(c_0) = \bar{\rho}(c)$, that is, iff $\rho \models c = c_0$. Therefore, we conclude that statements (3) and (4) are equivalent.

To complete the proof of the lemma, we show for each ρ' that $\rho' \models \mathit{diverge}(c)$ iff $\rho' \models \overline{\mathit{diverge}}(c)$. We have that $\rho' \models \mathit{diverge}(c)$ iff for each $m \in \mathbb{N}$ there exists some $\rho : \mathit{Var} \to \mathcal{T}$ which agrees with ρ' on c such that statement (3) holds. We also have that $\rho' \models \overline{\mathit{diverge}}(c)$ iff for each $m \in \mathbb{N}$ there exist some $a, b \in \mathbb{N}$ and some $\bar{\rho} : \mathit{Var} \to \mathcal{T}$ which agrees with ρ' on c such that statement (4) holds. We know that (3) and (4) are equivalent for Gödel pairs, hence it suffices to construct for each ρ satisfying (3) a $\bar{\rho}$ such that $\rho, \bar{\rho}$ are a Gödel pair and agree on m, and conversely, to construct for each $\bar{\rho}$ satisfying (4) a ρ such that $\rho, \bar{\rho}$ are a Gödel pair and agree m. For both directions, we choose such that ρ and $\bar{\rho}$ agree on c, m, thus we satisfy (1). From ρ, we construct $\bar{\rho}$ by choosing a, b to be the Gödel numbers associated to the sequence $\alpha(c_0), ..., \alpha(c_m)$. Conversely, from $\bar{\rho}$, (4) implies that $\bar{\rho} \models \exists c' \, (\beta(a,\ b,\ i,\ \alpha(c')))$ for each number $0 \leq i \leq m$. Then we choose ρ such that $\rho(c_i)$ is the unique configurations in which c'' is instantiated in the formula above. In both cases, the pair $\rho, \bar{\rho}$ satisfies (2), and we are done. □

The following summarises the two results above:

Lemma 31. *$\models \mathit{path}(c, c') \leftrightarrow \overline{\mathit{path}}(c, c')$ and $\models \mathit{diverge}(c) \leftrightarrow \overline{\mathit{diverge}}(c)$.*

Proof. Follows by Lemma 29 and Lemma 30. □

Consequently, we can use $\overline{\mathit{path}}(c, c')$, to express $\mathit{step}(c, c')$, $\mathit{succ}(c, c')$ and $\mathit{coreach}(\varphi)$ in FOL. Then, $\overline{\mathit{diverge}}(c)$, which uses $\mathit{succ}(c, c')$, is a FOL formula. Since our relative completeness proof only uses these predicates 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 continue working with *path* and *diverge* instead of $\overline{\mathit{path}}$ and $\overline{\mathit{diverge}}$.

Semantic Validity and Relative Completeness

We derive a proof for a valid rule $\varphi \Rightarrow^{\exists} \varphi'$ with the proof system in Figure 3.1, using the FOL predicates encoding the transition system to express intermediate formulae. First, we show that the semantic validity of reachability rules can be framed as FOL validity. This does not give us relative completeness directly, but it enables us to begin the derivation process.

Lemma 32. $\mathcal{S} \models \varphi \Rightarrow^{\exists} \varphi'$ *iff* $\models \varphi \rightarrow \exists c\, (\square = c \wedge diverge(c)) \vee coreach(\varphi')$.

Proof. Let $\gamma \in \mathcal{T}_{Cfg}$ and $\rho : Var \rightarrow \mathcal{T}$. We establish the necessary and sufficient conditions such that

$$(\gamma, \rho) \models \varphi \rightarrow \exists c\, (\square = c \wedge diverge(c)) \vee coreach(\varphi')$$

According to the semantics of implications, the above happens iff $(\gamma, \rho) \models \varphi$ implies either $(\gamma, \rho) \models \exists c\, (\square = c \wedge diverge(c))$ or $(\gamma, \rho) \models coreach(\varphi')$. By Lemma 25, the former holds iff γ does not terminate, while by Lemma 28 the latter holds iff there exists some $\gamma' \in \mathcal{T}_{Cfg}$ such that $(\gamma', \rho) \models \varphi'$ and $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$. Thus, we can conclude that $\models \varphi \rightarrow \exists c\, (\square = c \wedge diverge(c)) \vee coreach(\varphi')$ iff for each γ and ρ we have that $(\gamma, \rho) \models \varphi$ implies that either γ does not terminate or there exists some $\gamma' \in \mathcal{T}_{Cfg}$ such that $(\gamma', \rho) \models \varphi'$ and $\gamma \Rightarrow_{\mathcal{S}}^{\star\mathcal{T}} \gamma'$. According to the definition of semantic validity for reachability rules (see Definition 13), the above conditions hold iff $\mathcal{S} \models \varphi \Rightarrow^{\exists} \varphi'$, and we are done. □

The following three lemmas are useful in proving our relative completeness result.

Lemma 33. *If* $\mathcal{A} \vdash \square = c \wedge path(c, c') \Rightarrow \square = c'$ *is derivable and* φ *is well-defined, then* $\mathcal{A} \vdash \varphi \wedge \varphi[c/\square] \wedge \varphi'[c'/\square] \wedge path(c, c') \Rightarrow \varphi'$ *is derivable.*

Proof. Since φ is well-defined, we have that $\models \varphi \wedge \varphi[c/\square] \rightarrow \square = c$, by Lemma 2. Also, it is easy to see that $\models \square = c' \wedge \varphi'[c'/\square] \rightarrow \varphi'$. Thus, $\mathcal{A} \vdash \varphi \wedge \varphi[c/\square] \wedge \varphi'[c'/\square] \wedge path(c, c') \Rightarrow \varphi'$ is derivable by Consequence with the two implications above from the sequent

$$\mathcal{A} \vdash \square = c \wedge \varphi'[c'/\square] \wedge path(c, c') \Rightarrow \square = c' \wedge \varphi'[c'/\square]$$

which follows by Logical Framing with $\varphi'[c'/\square]$ from

$$\mathcal{A} \vdash \square = c \wedge path(c, c') \Rightarrow \square = c'$$

The last sequent is derivable according to the hypothesis, and we are done. □

Lemma 34. *If* $\mathcal{A} \vdash \square = c \wedge step(c, c'') \wedge path(c'', c') \Rightarrow \square = c'$ *is derivable, then* $\mathcal{A} \vdash \square = c \wedge path(c, c') \Rightarrow \square = c'$ *is derivable.*

Proof. By Lemma 23, we have that

$$\models path(c,c') \rightarrow c = c' \vee \exists c''\ (step(c,c'') \wedge path(c'',c'))$$

Then, $\mathcal{A} \vdash \square = c \wedge path(c,c') \Rightarrow \square = c'$ is derivable by Consequence with the implication above, Abstraction with c'', and Case Analysis from

$$\mathcal{A} \vdash \square = c \wedge c = c' \Rightarrow \square = c'$$
$$\mathcal{A} \vdash \square = c \wedge step(c,c'') \wedge path(c'',c') \Rightarrow \square = c'$$

The former follows by Consequence and Reflexivity, while the latter is derivable according to the hypothesis, and we are done. □

Lemma 35. *If $\mathcal{S} \cup C \vdash \square = c \wedge path(c,c') \Rightarrow \square = c'$ is derivable, then $\mathcal{S} \vdash_C \square = c \wedge step(c,c') \Rightarrow \square = c'$ is derivable.*

Proof. The sequent $\mathcal{S} \vdash_C \square = c \wedge step(c,c') \Rightarrow \square = c'$ follows by Abstraction with $c_1, c'_1, ..., c_{nc}, c'_{nc}, \bar{x}$ (the top existentially quantified variables in the definition of $step(c,c')$) from

$$\begin{aligned}\mathcal{S} \vdash_C & \bigvee_{\mu \in \mathcal{S}} (\varphi[c/\square] \wedge \varphi'[c'/\square] \wedge \bigwedge_{1 \le i \le n} (\varphi_i[c_i/\square] \wedge \varphi'_i[c'_i/\square] \wedge path(c_i,c'_i))) \\ & \wedge \square = c \\ & \Rightarrow^{\exists} \square = c\end{aligned}$$

Then, by Consequence, we can drop $\square = c$ and substitute c by $\square$, and it suffices to derive

$$\begin{aligned}\mathcal{S} \vdash_C & \bigvee_{\mu \in \mathcal{S}} (\varphi \wedge \varphi'[c'/\square] \wedge \bigwedge_{1 \le i \le n} (\varphi_i[c_i/\square] \wedge \varphi'_i[c'_i/\square] \wedge path(c_i,c'_i))) \\ & \Rightarrow^{\exists} \square = c\end{aligned}$$

Recall that one of the assumptions of relative completeness is that $\mathcal{S}$ is not empty. Then, by $|\mathcal{S}| - 1$ applications of Case Analysis, it suffices to derive for each $\mu \in \mathcal{S}$

$$\mathcal{S} \vdash_C \varphi \wedge \varphi'[c'/\square] \wedge \bigwedge_{1 \le i \le n} (\varphi_i[c_i/\square] \wedge \varphi'_i[c'_i/\square] \wedge path(c_i,c'_i)) \Rightarrow \square = c$$

Recall that one of the assumptions of relative completeness is that φ' is well-defined.

By Lemma 2, $\models \varphi' \wedge \varphi'[c'/\square] \to \square = c'$, thus by Consequence it suffices to derive

$$\begin{aligned}&\mathcal{S} \vdash_C \varphi \wedge \varphi'[c'/\square] \wedge \bigwedge_{1\leq i\leq n} (\varphi_i[c_i/\square] \wedge \varphi'_i[c'_i/\square] \wedge path(c_i, c'_i))\\ &\quad \Rightarrow^{\exists} \varphi' \wedge \varphi'[c'/\square]\end{aligned}$$

The last sequent follows by Logical Framing with $\varphi'[c'/\square]$ and Axiom with $\mu \in \mathcal{S}$ from the prerequisites

$$\mathcal{S} \cup C \ \vdash \varphi_j \wedge \bigwedge_{1\leq i\leq n} (\varphi_i[c_i/\square] \wedge \varphi'_i[c'_i/\square] \wedge path(c_i, c'_i)) \Rightarrow \varphi'_j$$

for each $1 \leq j \leq n$. By Consequence, it suffices to derive

$$\mathcal{S} \cup C \ \vdash \varphi_j \wedge \varphi_j[c_j/\square] \wedge \varphi'_j[c'_j/\square] \wedge path(c_j, c'_j) \Rightarrow \varphi'_j$$

According to the hypothesis, $\mathcal{S} \cup C \ \vdash \square = c \wedge path(c, c') \Rightarrow \square = c'$
is derivable. Recall that one of the assumptions of relative completeness is that φ_j is well-defined. Then the last sequent follows by Lemma 33 with α-renaming c, c' into c_j, c'_j, and we are done. □

The following lemma states that if there is a path in the transition system from c to c' (expressed by $path(c, c')$) then we can derive it.

Lemma 36. $\mathcal{S} \ \vdash \square = c \wedge path(c, c') \Rightarrow \square = c'$.

Proof. By Lemma 34, it suffices to derive

$$\mathcal{S} \ \vdash \square = c \wedge step(c, c'') \wedge path(c'', c') \Rightarrow \square = c'$$

Let $C \equiv \{\square = c \wedge step(c, c'') \wedge path(c'', c') \Rightarrow^{\exists} \square = c'\}$. Then, by Circularity, it suffices to derive

$$\mathcal{S} \ \vdash_C \square = c \wedge step(c, c'') \wedge path(c'', c') \Rightarrow \square = c'$$

which in turn follows by Transitivity from

$$\mathcal{S} \ \vdash_C \square{=}c \wedge step(c, c'') \wedge path(c'', c') \Rightarrow \square{=}c'' \wedge path(c'', c') \quad (1)$$

$$\mathcal{S} \cup C \ \vdash \square{=}c'' \wedge path(c'', c') \Rightarrow \square{=}c' \quad (2)$$

Sequent (1) follows by Logic Framing with $path(c'', c')$ from

$$\mathcal{S} \vdash_C \square = c \wedge step(c, c'') \Rightarrow \square = c''$$

By Lemma 35 (with α-renaming c'' into c'), it suffices to derive

$$\mathcal{S} \cup C \vdash \square = c \wedge path(c, c') \Rightarrow \square = c'$$

Further, by Lemma 34, it suffices to derive

$$\mathcal{S} \cup C \vdash \square = c \wedge step(c, c'') \wedge path(c'', c') \Rightarrow \square = c'$$

which follows by Axiom with the rule in C. By Lemma 34 (with α-renaming c'' into c), sequent (2) follows from

$$\mathcal{S} \cup C \vdash \square = c \wedge step(c, c'') \wedge path(c'', c') \Rightarrow \square = c'$$

which follows by Axiom with the rule in C, and we are done. □

The result above has the following consequence:

Lemma 37. $\mathcal{S} \vdash_C \square = c \wedge step(c, c') \Rightarrow \square = c'$.

Proof. By Lemma 35, it suffices to derive

$$\mathcal{S} \cup C \vdash \square = c \wedge path(c, c') \Rightarrow \square = c'$$

which follows form Lemma 36 and weakening with $\mathcal{S} \subseteq \mathcal{S} \cup C$. □

The next lemma states that if c diverges in the transition system then we can derive it.

Lemma 38. $\mathcal{S} \vdash \square = c \wedge diverge(c) \Rightarrow \omega$.

Proof. Let $C \equiv \{\square = c \wedge diverge(c) \Rightarrow^{\exists} \omega\}$. Then the sequent

$$\mathcal{S} \vdash \square = c \wedge diverge(c) \Rightarrow \omega$$

follows by Circularity from

$$\mathcal{S} \vdash_C \square = c \wedge diverge(c) \Rightarrow \omega$$

By Lemma 26, we have that

$$\models \mathit{diverge}(c) \to \exists c' \ (\mathit{succ}(c, c') \land \mathit{diverge}(c'))$$

Thus, by Consequence, and Abstraction with c', it suffices to derive

$$\mathcal{S} \ \vdash_C \square = c \land \mathit{succ}(c, c') \land \mathit{diverge}(c') \Rightarrow \omega$$

Let ψ_{cond} be defined as

$$\psi_{cond} \equiv \bigvee_{\mu \in \mathcal{S}} (\varphi[c/\square] \land \bigvee_{1 \le i \le n} (\varphi_i[c'/\square] \\ \land \bigwedge_{1 \le j < i} (\varphi_j[c_j/\square] \land \varphi'_j[c'_j/\square] \land \mathit{path}(c_j, c'_j))))$$

Then, according to the definition of $\mathit{succ}(c, c')$, by Case Analysis and Abstraction with $c_1, c'_1, ..., c_{nc}, c'_{nc}, \bar{x}$ (the top existentially quantified variables), it suffice to derive

$$\mathcal{S} \ \vdash_C \square = c \land \mathit{step}(c, c') \land \mathit{diverge}(c') \Rightarrow \omega \tag{1}$$

$$\mathcal{S} \ \vdash_C \square = c \land \psi_{cond} \land \mathit{diverge}(c') \Rightarrow \omega \tag{2}$$

By Lemma 37 and Logic Framing with $\mathit{diverge}(c')$, we derive

$$\mathcal{S} \ \vdash_C \square = c \land \mathit{step}(c, c') \land \mathit{diverge}(c') \Rightarrow \square = c' \land \mathit{diverge}(c')$$

By Axiom with the rule in C and α-renaming of c' into c, we derive

$$\mathcal{S} \cup C \ \vdash \square = c' \land \mathit{diverge}(c') \Rightarrow \omega$$

Then, sequent (1) follows by Transitivity with the two sequents above. Therefore, we are left to derive sequent (2). Let φ_{cond} be defined as

$$\varphi_{cond} \equiv \bigvee_{\mu \in \mathcal{S}} (\varphi \land \bigvee_{1 \le i \le n} (\varphi_i[c'/\square] \\ \land \bigwedge_{1 \le j < i} (\varphi_j[c_j/\square] \land \varphi'_j[c'_j/\square] \land \mathit{path}(c_j, c'_j))))$$

Then, by Consequence, we can drop $\square = c$ and substitute c by $\square$ in sequent (2), and it suffices to derive

$$\mathcal{S} \vdash_C \varphi_{cond} \wedge diverge(c') \Rightarrow \omega$$

Recall that one of the assumptions of relative completeness is that $\mathcal{S}$ is not empty. Then, by $|\mathcal{S}| - 1$ applications of Case Analysis, each followed by n applications of Case Analysis, it suffices to derive for each $\mu \in \mathcal{S}$ and each $1 \leq i \leq n$ (n is the number of conditions of μ)

$$\begin{aligned}\mathcal{S} \vdash_C\ & \varphi \wedge \varphi_i[c'/\square] \wedge \bigwedge_{1\leq j<i}(\varphi_j[c_j/\square] \wedge \varphi'_j[c'_j/\square] \wedge path(c_j, c'_j)) \\ & \quad \wedge diverge(c') \\ & \Rightarrow^{\exists} \omega\end{aligned}$$

Recall that one of the assumption of relative completeness is that $\mathcal{S}$ is ω-closed (see Definition 15). Thus, for μ and i there must be some rule $\mu_\omega \in \mathcal{S}$ of the form

$$\varphi \Rightarrow^{\exists} \omega \ \text{ if } \ \varphi_1 \Rightarrow^{\exists} \varphi'_1 \bigwedge ... \bigwedge \varphi_{i-1} \Rightarrow^{\exists} \varphi'_{i-1} \bigwedge \varphi_i \Rightarrow^{\exists} \omega$$

Then the sequent above follows by Axiom with μ_ω and with the prerequisites

$$\begin{aligned}\mathcal{S} \cup C \vdash\ & \varphi_k \wedge \varphi_i[c'/\square] \wedge \bigwedge_{1\leq j<i}(\varphi_j[c_j/\square] \wedge \varphi'_j[c'_j/\square] \wedge path(c_j, c'_j)) \\ & \quad \wedge diverge(c') \\ & \Rightarrow^{\exists} \varphi'_k \end{aligned} \tag{3}$$

$$\begin{aligned}\mathcal{S} \cup C \vdash\ & \varphi_i \wedge \varphi_i[c'/\square] \wedge \bigwedge_{1\leq j<i}(\varphi_j[c_j/\square] \wedge \varphi'_j[c'_j/\square] \wedge path(c_j, c'_j)) \\ & \quad \wedge diverge(c') \\ & \Rightarrow^{\exists} \omega \end{aligned} \tag{4}$$

for each $1 \leq k < i$. By Consequence, to derive each sequent (3), it suffices to derive

$$\mathcal{S} \cup C \vdash \varphi_k \wedge \varphi_k[c_k/\square] \wedge \varphi'_k[c'_k/\square] \wedge path(c_k, c'_k) \Rightarrow \varphi'_k$$

for each $1 \leq k < i$. By Lemma 36, we have that

$$\mathcal{S} \vdash \square = c \wedge path(c, c') \Rightarrow \square = c'$$

is derivable. Recall that one of the assumption of relative completeness is that φ_k is well-defined. Then, the sequents above follow by Lemma 33 with α-renaming c, c into c_k, c'_k. By Consequence, to derive sequent (4), it suffices to derive

$$\mathcal{S} \cup C \ \vdash \varphi_i \wedge \varphi_i[c'/\square] \wedge \mathit{diverge}(c') \Rightarrow \omega$$

Recall that one of the assumption of relative completeness is that φ_i is well-defined. By Lemma 2, we have that $\models \varphi_i \wedge \varphi_i[c'/\square] \rightarrow \square = c'$. Thus, by Consequence the last sequent follows from

$$\mathcal{S} \cup C \ \vdash \square = c' \wedge \mathit{diverge}(c') \Rightarrow \omega$$

which follows by Axiom with the rule in C and α-renaming c into c', and we are done. $\square$

Lemma 39. *$\mathcal{S} \ \vdash \mathit{coreach}(\varphi) \Rightarrow \varphi$.*

Proof. According to the definition of $\mathit{coreach}(\varphi)$, by Abstraction with c, c', it suffices to derive

$$\mathcal{S} \ \vdash \square = c \wedge \varphi[c'/\square] \wedge \mathit{path}(c, c') \Rightarrow \varphi$$

We have that $\models \square = c' \wedge \varphi[c'/\square] \rightarrow \varphi'$, thus the last sequent follows by Consequence from

$$\mathcal{S} \ \vdash \square = c \wedge \varphi[c'/\square] \wedge \mathit{path}(c, c') \Rightarrow \square = c' \wedge \varphi[c'/\square]$$

which in turn follows by Logic Framing with $\varphi[c'/\square]$ from

$$\mathcal{S} \ \vdash \square = c \wedge \mathit{path}(c, c') \Rightarrow \square = c'$$

which is derivable by Lemma 36, and we are done. $\square$

Using the lemmas above, we derive the following rule between the formula specifying the configurations reaching φ and the divergent configurations, on one hand, and φ itself, on the other hand.

Lemma 40. *$\mathcal{S} \ \vdash \exists c\, (\square = c \wedge \mathit{diverge}(c)) \vee \mathit{coreach}(\varphi) \Rightarrow \varphi$.*

Proof. By Case Analysis and Abstraction with c, it suffices to derive

$$\mathcal{S} \vdash \square = c \wedge \mathit{diverge}(c)) \Rightarrow \varphi$$
$$\mathcal{S} \vdash \mathit{coreach}(\varphi) \Rightarrow \varphi$$

The latter follows by Lemma 39. For former follows by Transitivity from

$$\mathcal{S} \vdash \square = c \wedge \mathit{diverge}(c)) \Rightarrow \omega$$
$$\mathcal{S} \vdash \omega \Rightarrow \varphi$$

The first sequent is derivable by Lemma 38. To derive the second, Circularity and Transitivity, it suffices to derive

$$\mathcal{S} \vdash_{\{\omega \Rightarrow^{\exists} \varphi\}} \omega \Rightarrow \omega$$
$$\mathcal{S} \cup \{\omega \Rightarrow^{\exists} \varphi\} \vdash \omega \Rightarrow \varphi$$

Recall that one of the assumption of relative completeness is that $\mathcal{S}$ is ω-closed. Then the first sequent follows by Axiom with $\omega \Rightarrow^{\exists} \omega$, while the second sequent follows by Axiom with $\omega \Rightarrow^{\exists} \varphi$, and we are done. □

Finally, the relative completeness follows from all the lemmas above. Note how the configuration model is being used, via Lemma 32, as an oracle to answer the semantic reachability question formulated as a FOL sentence.

Proof. Suppose that $\mathcal{S} \models \varphi \Rightarrow^{\exists} \varphi'$. Then Lemma 32 implies that $\models \varphi \rightarrow (\exists c\, (\square = c \wedge \mathit{diverge}(c)) \vee \mathit{coreach}(\varphi'))$. By Lemma 40 it follows that $\mathcal{S} \vdash \exists c\, (\square = c \wedge \mathit{diverge}(c)) \vee \mathit{coreach}(\varphi') \Rightarrow^{\exists} \varphi'$ is derivable. Then the theorem follows by Consequence. □

A direct consequence of the theorem above is the following (recall that for relative completeness $\mathcal{S}$ is assumed ω-closed)

Corollary 2. *If $\mathcal{S} \models \varphi\!\uparrow$ then $\mathcal{S} \vdash \varphi \Rightarrow^{\exists} \omega$.*

Chapter 4

Implementation and Evaluation

In this chapter we discuss our implementation of reachability logic. We implemented the $\mathbb{K}$ verification infrastructure (KVI) based on the proof system in Figure 3.1 and $\mathbb{K}$ semantics (Section 2.2). The infrastructure takes a $\mathbb{K}$ semantics and turns it into a correct-by-construction program verifier. To evaluate out infrastructure, we instantiate it with the semantics of three real-world languages, C, JAVA, and JAVASCRIPT. Then we evaluate the generated verifiers on challenging heap-manipulating programs implementing complex data-structures. The same data-structures are implemented in the three languages, and verified by our infrastructure, based only on the $\mathbb{K}$ semantics of the languages, without any language-specific support. Thus, we argue that our approach is truly language-independent and separates the language specific features from the mathematical reasoning. We detail this work in Section 4.2.

Prior to the KVI, we build MATCHC, a program verification prototype for KERNELC, a deterministic fragment of C (Section 4.1). MATCHC is also based on the proof-system in Figure 3.1; it specializes the proof system for the $\mathbb{K}$ semantics of KERNELC, thus mixing the language-independent reasoning with the operational semantics. For example, it hardcodes when to perform CASE ANALYSIS (for constructs like `if`), and when to perform heap abstractions folding/unfolding. Since KERNELC is deterministic, MATCHC verifies program correctness properties given as one-path reachability rules $\varphi \Rightarrow^{\exists} \varphi'$. KVI subsumes MATCHC; nevertheless, MATCHC was a useful prototype for early experimentation.

Much of the work in this chapter comes from Roșu and Ștefănescu [90], Ștefănescu [98], Roșu and Ștefănescu [93], Ștefănescu et al. [99], Park et al. [75], and Ștefănescu et al. [100].

```
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 4.1: C program exhibiting undefined behaviour.

4.1 MatchC

In this section we discuss MatchC, our first prototype verifier based on reachability logic. First we show some motivating examples (Section 4.1.1), and then we present the implementation and evaluation of the tool (Section 4.1.2).

4.1.1 Examples using MatchC

Here we discuss a few C examples that illustrate the expressiveness and practicality of our approach. Figure 4.1 shows an undefined program; Figure 4.2 a function that reverses a singly linked list; Figure 4.3 a function that reads a sequence of integers from the standard input into a singly-linked list; Figure 4.4 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 4.5 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 4.1.2).

The unannotated/unspecified program in Figure 4.1 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

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

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

Figure 4.2: C function reversing a singly-linked list.

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 3.1. For the next specifications, we discuss some MatchC notations that help avoid verbosity. (1) While all specifications are reachability rules $\varphi \Rightarrow^{\exists} \varphi'$, often φ and φ' share configuration context; we only mention the context once and distribute "$\Rightarrow^{\exists}$" 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 x $\mapsto$?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 ?x. (4) MatchC desugars invariants `inv` φ `loop` into rules φ[`loop`...] $\Rightarrow^{\exists}$ φ[...] $\wedge$ $\neg$*cond*(`loop`), with φ[`code`] the pattern obtained from φ setting the contents of $\langle ... \rangle_{\mathsf{k}}$ to `code`.

Function `reverseList` in Figure 4.2 reverses a singly-linked list. The matching logic rule specifying its behavior says that it returns a pointer ?p (here and in the rest of the thesis, $ 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 2.3.2). 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 4.2 would be specified in separation logic as

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

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

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 3.5, 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 ver-

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

struct listNode *readList(int n)
```
rule $\langle \$ \Rightarrow^{\exists} \texttt{return}\ ?\mathsf{x};\ \cdots\rangle_{\mathsf{k}} \langle \mathsf{A} \Rightarrow^{\exists} \cdot\ \cdots\rangle_{\mathsf{in}} \langle \cdots\ \cdot \Rightarrow^{\exists} \mathsf{list}(?\mathsf{x})(\mathsf{A})\ \cdots\rangle_{\mathsf{heap}}$
 if $\mathtt{n} = \mathsf{len}(\mathsf{A})$
```
{
 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 $\langle ?\mathsf{C}\ \cdots\rangle_{\mathsf{in}} \langle \cdots\ \mathsf{lseg}(\mathtt{x},\mathtt{p})(?\mathsf{B}),\ \mathtt{p} \mapsto [?\mathsf{v},\ \mathtt{NULL}]\ \cdots\rangle_{\mathsf{heap}}$
 $\wedge\ \mathtt{i} \le \mathtt{n}\ \wedge\ \mathsf{len}(?\mathsf{C}) = \mathtt{n} - \mathtt{i}\ \wedge\ \mathsf{A} = ?\mathsf{B}@[?\mathsf{v}]@?\mathsf{C}$
```
 while (i < n) {
   p->next = (struct listNode*)
            malloc(sizeof(struct listNode));
   p = p->next;
   scanf("%d", &(p->val));
   p->next = NULL;
   i += 1;
 }
 return x;
}
```

Figure 4.3: C function reading a sequence of integers from the standard input into a singly-linked list.

ification we deliberately avoid adding any automatic framing rules, simply because those are not necessary. For example, the "..." symbols in the specifications in Figure 4.2 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 4.3 reads `n` integers from standard input and stores them in a singly-linked list. The specification says that the function: (1) returns a pointer $?\mathsf{x}$; (2) reads from the standard input a sequence of integers A of length `n` (matches A and replaces it by the empty sequence $\cdot$); (3) allocates a list starting at $?\mathsf{x}$ with contents A (replaces the empty heap $\cdot$). The rest of the input buffer, the heap, and the configuration stay unchanged. The loop invariant states that the sequence $?\mathsf{C}$ is yet to be read, `x` points to a list segment ending at `p` with contents $?\mathsf{B}$, `p` points to a `nodeList` structure with the value field $?\mathsf{v}$ and the `next`

```
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 4.4: C program respecting a stack inspection policy.

field `NULL`, the loop index `i` is not greater than `n`, the size of ?C is `n` − `i`, and the initial sequence A equals the concatenation of ?B, ?v, and ?C. The list segment lseg(`x`, `p`) includes `x` but excludes `p`. The notation `p` ↦ [?v, `NULL`] stands for the *term* (and *not* formula) "`p` ↦ ?v, `p` + 1 ↦ `NULL`".

Figure 4.4 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

```
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 $\langle \$ \Rightarrow^{\exists}$ **`return`** ?l; $\cdots\rangle_{\mathsf{k}}$ $\langle\cdots$ tree(`x`)(T) $\Rightarrow^{\exists}$ list(?l)(tree2list(T)) $\cdots\rangle_{\mathsf{heap}}$
$\langle\cdots\ \cdot \Rightarrow^{\exists}$ rev(tree2list(T))$\rangle_{\mathsf{out}}$
```
{
  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) $\cdots\rangle_{\mathsf{heap}}$ $\langle\cdots$ rev(?A)$\rangle_{\mathsf{out}}$
$\wedge$ 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 4.5: Iterative C program flattening a tree into a list and printing its values in the process.

`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 4.5 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 $?\mathsf{l}$. The rule matches in the heap a tree rooted at `t` with contents T and replaces it with a list starting at $?\mathsf{l}$ with contents $\mathsf{tree2list}(\mathsf{T})$ (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 $?\mathsf{TS}$ and a list with contents $?\mathsf{A}$, the loop has printed so far the sequence $\mathsf{rev}(?\mathsf{A})$, and that the infix traversal sequence of T, $\mathsf{tree2list}(\mathsf{T})$, 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.

4.1.2 Implementation and Evaluation

Here we discuss our MatchC implementation of the one-path proof system in Figure 3.1. While the proof system can be easily implemented in most theorem proving environments, we preferred an implementation that emphasizes automated reasoning. Our results demonstrate that matching logic 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 3.1, general matching logic specifications are reachability rules between formulae. As seen in Section 4.1.1, our tool handles specifications of the form:

$$\exists X(\pi \wedge \psi) \Rightarrow^{\exists} \exists X'(\pi' \wedge \psi')$$

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 "$\cdot$" 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_{X}, we associate with the loop the following rule:

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

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.

We define the operational semantics of the C fragment in the $\mathbb{K}$ framework [89] as a set of reachability rules $\mathcal{S}$ over the configuration in Figure 2.5.

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 3.1 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 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 [21]. 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 [10] and Z3 [26]. MatchC comes with a library of $\sim$ 100 mathematical domain operators (like rev, in) and pattern abstractions (like list), together with their axioms and useful lemmas (see Section 2.3.2). It currently provides support for reasoning about lists, trees, queues and graphs.

Table 4.1 and 4.2 summarise 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 4.2 , 4.3 and 4.5. 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

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

Table 4.1: Results of MatchC program verification (part 1)

Program	Cells	Time (s)	# paths	SMT?
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

Table 4.2: Results of MatchC program verification (part 2)

multiset of elements.

The Schorr-Waite graph marking algorithm [96] 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 [46, 58]. 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.

4.2 KVI

In this section we discuss our language independent verification infrastructure based the proof system in Figure 3.1 and operational semantics defined in the $\mathbb{K}$ framework. First we look at a few motivating examples (Section 4.2.1), and then we present the implementation (Section 4.2.2) and the evaluation of the infrastructure (Section 4.2.3).

4.2.1 Motivating Example

Here we illustrate our approach by checking the correctness of binary search tree (BST) insertion implemented in C, Java, and JavaScript. A BST is a tree where the value stored in each node is greater than any value in the left subtree and less than any value in the right subtree. Insert recursively traverses the tree and adds a new leaf with the value, if the value is not already in the tree. We use the operational semantics of these languages for symbolic execution, and delegate reasoning about trees in the heap and BST invariants to the verification infrastructure. Although

```
struct node {
   int value;
   struct node *left, *right;
};

struct node* new_node(int v) {
   struct node *node;
   node = (struct node *)
      malloc(sizeof(struct node));
   node->value = v;
   node->left = NULL;
   node->right = NULL;
   return node;
}

struct node* insert(int v, struct node *t) {
   if (t == NULL)
      return new_node(v);
   if (v < t->value)
      t->left = insert(v, t->left);
   else if (v > t->value)
      t->right = insert(v, t->right);
   return t;
}
```

Figure 4.6: Binary search tree code in C

```
class Node {
   int value;
   Node left, right;

   public Node(int value) {
      this.value = value;
      left = right = null;
   }

   public static Node insert(int v, Node t) {
      if (t == null)
         return new Node(v);
      if (v < t.value)
         t.left = insert(v, t.left);
      else if (v > t.value)
         t.right = insert(v, t.right);
      return t;
   }
}
```

Figure 4.7: Binary search tree code in JAVA

```
function make_node(v) {
   var node = {
      value : v,
      left : null,
      right : null
   };
   return node;
}

function insert(v, t) {
   if (t === null)
      return make_node(v);
   if (v < t.value)
      t.left = insert(v, t.left);
   else if (v > t.value)
      t.right = insert(v, t.right);
   return t;
}
```

Figure 4.8: Binary search tree code in JAVASCRIPT

the three definitions feature different language constructs and memory models, the operational semantics successfully abstracts these details.

Figure 4.6 shows the implementation in C, Figure 4.7 in JAVA, and Figure 4.8 in JAVASCRIPT. C uses "`struct node`" to represent a tree node, while JAVA uses "`class Node`". JAVASCRIPT is a class-free, prototypal language, where objects dynamically inherit from other objects. In C, dynamically allocated memory (the "heap") is untyped; `malloc` allocates a block of bytes, which is then associated the effective type `struct node`. In JAVA all memory is typed; `new` creates an instance of `class Node`. In JAVASCRIPT, objects are modeled in memory as maps from property names (strings) to values (of any type). Each language has different memory access mechanisms. The C and JAVA trees store integers, while the JAVASCRIPT tree stores floats Other language-specific aspects are automatic type conversions and function/method calls.

Before we discuss the correctness specifications, we recall some useful $\mathbb{K}$ conventions. Specifications are reachability rules $\varphi \Rightarrow^{\forall} \varphi'$, with φ and φ' matching logic patterns (i.e. (symbolic) program configurations with constraints). If φ and φ' share program configuration context, we only mention the context once and distribute "$\Rightarrow^{\forall}$" through the context where the changes take place. Logical variables starting with "?" are existentially quantified. Rules only mention the parts of the configuration they read or write; the rest stays unchanged. The "requires" clause is implicitly conjuncted with the left-hand-side configuration, and "ensures" with the right-hand-side. It is common for operational semantics to have a preprocessing/initializing phase. C computes structure and function tables, JAVA a class table, while JAVASCRIPT creates objects and environments for all functions. A variable with the same name as a cell but with capital letters is a placeholder for the initial value of that cell after the preprocessing phase, which we statically compute using the semantics.

Figure 4.9 shows the correctness specifications. We discuss the C one first. The rule states that the call to `insert` with value V and pointer L_1 returns pointer $?L_2$. Since C is typed, each value is tagged with its type, in this case **`int`** or pointer to **`struct`** `node`. When the function is called, the memory contains a binary tree with root L_1 storing the algebraic tree T_1. When the function returns, the initial tree is replaced by another tree with root $?L_2$ storing $?T_2$. The requires clause states that T_1 is a BST and V is in the appropriate range for signed 32-bit integers. The ensures clause states that T_2 is also a BST, and the value set of $?T_2$ is the value set of T_1 union with V. The "$\cdots$" in the mem cell stands for a variable matching the rest of the memory (the *heap frame*), which stays unchanged. The threads cell contains only one thread and no "$\cdots$", which means this program is verified in a single-threaded environment (the program is not thread-safe). Variables FUNCTIONS, STRUCTS, and MEM are placeholders for the tables of function declarations and structure declarations, and the initial memory layout.

The JAVA specification is in many ways similar to the C one, reflecting the similarities between C and JAVA. The call to `insert` uses the fully qualified method name, which includes the class name `Node`. The type of R_1 and $?R_2$ mentioned in the rule is the static type of these references, `class Node`. The dynamic type can be any sublcass of `class Node`. Variable CLASSES : Bag stands for the statically computed class table.

Now we discuss the JAVASCRIPT specification. Since JAVASCRIPT is untyped, its values do not carry a type. V is not `NaN`, since `NaN` does not respect the order relation on non-`NaN` floats, and the code is incorrect if V or the values in T_1 were `NaN`. The JAVASCRIPT semantics creates new environments and objects at function call, which it does not garbage-collect at return. The ".Bag $\Rightarrow^{\forall}$?_ : Bag" in both the envs and objs cells states that there may be garbage left after the function returns ("." is the unit, while "_" is an anonymous variable, here existentially quantified). JAVASCRIPT does not have threads.

The tree heap abstraction is defined in matching logic, and is different for each

C

rule
⟨functions⟩ FUNCTIONS : Map ⟨/functions⟩
⟨structs⟩ STRUCTS : Map ⟨/structs⟩
⟨mem⟩...
    MEM : Map (tree(L_1, T_1 : Tree) $\Rightarrow^{\forall}$ tree(?L_2, ?T_2 : Tree))
...⟨/mem⟩
⟨threads⟩ ⟨thread⟩... ⟨k⟩
    `insert`(tv(V : Int, `int`), tv(L_1 : Loc, `struct node`))
    $\Rightarrow^{\forall}$ tv(?L_2 : Loc, `struct node`)
...⟨/k⟩ ...⟨/thread⟩ ⟨/threads⟩
requires bst(T_1) $\wedge$ $-2147483648 \leq V \wedge V \leq 2147483647$
ensures bst(?T_2) $\wedge$ tree_keys(?T_2) = {V} $\cup$ tree_keys(T_1)

JAVA

rule
⟨classes⟩ CLASSES : Bag ⟨/classes⟩
⟨objectStore⟩...
    tree(R_1, T_1 : Tree) $\Rightarrow^{\forall}$ tree(?R_2, ?T_2 : Tree)
...⟨/objectStore⟩
⟨threads⟩ ⟨thread⟩... ⟨k⟩
    `(class Node).insert(`
        V : Int :: `int`, R_1 : Ref :: `class Node`)
    $\Rightarrow^{\forall}$?R_2 : Ref :: `class Node`
...⟨/k⟩ ...⟨/thread⟩ ⟨/threads⟩
requires bst(T_1) $\wedge$ $-2147483648 \leq V \wedge V \leq 2147483647$
ensures bst(?T_2) $\wedge$ tree_keys(?T_2) = {V} $\cup$ tree_keys(T_1)

JAVASCRIPT

rule
⟨envs⟩... ENVS : Bag (.Bag $\Rightarrow^{\forall}$?_ : Bag) ...⟨/envs⟩
⟨objs⟩...
    OBJS : Bag (.Bag $\Rightarrow^{\forall}$?_ : Bag)
    (tree(L_1 : Loc, T_1 : Tree) $\Rightarrow^{\forall}$ tree(?L_2 : Loc, ?T_2 : Tree))
...⟨/objs⟩
⟨k⟩ `insert`(V : Float, O_1 : Object) $\Rightarrow^{\forall}$?O_2 : Object ...⟨/k⟩
requires bst(T_1) $\wedge$ $\neg$isNaN(V)
ensures bst(?T_2) $\wedge$ tree_keys(?T_2) = {V} $\cup$ tree_keys(T_1)

Figure 4.9: Binary search tree correctness specifications for C, JAVA, and JAVASCRIPT

language, taking into account the specifics of the memory model of each language. Also bst, tree_keys, etc., are domain operation symbols in the signature.

At a high level, the three specifications are very similar. The differences are down to language-specific and semantics-specific details: type systems, name resolution, garbage collection, or the statically computed information by each semantics. The tree heap abstraction hides the differences in memory models. Our generic verification infrastructure reasons about the tree abstraction and the mathematical properties of BST while deferring the symbolic execution to the semantics. The verification is fully automated and takes a few seconds (see Table 4.3).

It is possible to generate the specification rules automatically from classic verification annotations (pre/post conditions, loop invariants, class invariants, etc). This has been done previously by MatchC (Section 4.1). We have not implemented this feature, using instead a general-purpose notation which is faithful to both reachability logic and our implementation.

4.2.2 Implementation

We discuss our novel implementation of the $\mathbb{K}$ verification infrastructure, depicted in Figure 1.1, based on the language-independent proof system in Figure 3.1. Our framework takes an operational semantics defined in $\mathbb{K}$ [89] as a parameter and uses it to automatically derive program correctness properties. In other words, our verification infrastructure *automatically* generates a program verifier from the semantics, which is *correct-by-construction* w.r.t. the semantics. As discussed in Section 3.1, we view a semantics as a set of reachability rules $l \wedge b \Rightarrow^{\exists} r$. A major difficulty in a language-independent setting is that standard language features relevant to verification, like control flow or memory access, are not explicit, but rather implicit (defined through the semantics).

The generated program verifier proves a set of user provided reachability rules, representing the program correctness specifications of the code being verified, typically one for each recursive function and loop. For the sake of automation, the rules have the more restrictive form $\pi \wedge \psi \Rightarrow^{\forall} \pi' \wedge \psi'$, with $\pi \wedge \psi$ and $\pi' \wedge \psi'$ conjunctive patterns. A *conjunctive pattern* is a formula $\pi \wedge \psi$ with π a program configuration term with variables, and ψ a formula without any configuration terms. We use all-path rules for specifications to capture some of the local non-determinism (e.g. the non-deterministic C expression evaluation order). Section 4.1.1 shows examples of specifications. As discussed there, we use conventions already supported by $\mathbb{K}$ to have more compact specifications.

The generated program verifier is fully automated. The user only provides the program correctness specifications. Specifically, to prove a set C of rules between conjunctive patterns, it uses the following algorithm for each $\varphi \Rightarrow^{\forall} \varphi' \in C$:

```
 1  $Q$ := $successors(\varphi)$
 2  if $Q$ is empty and $\not\models \varphi \to \varphi'$ then fail
 3  while $Q$ not empty
 4    pop $\varphi_c$ from $Q$
 5    if $\models \varphi_c \to \varphi'$ continue
 6    else if $\exists\sigma$ with $\models \varphi_c \to \sigma(\varphi_l)$ for $\varphi_l \Rightarrow^{\forall} \varphi_r \in C$
 7      add $\sigma(\varphi_r) \land frame(\varphi_c)$ to $Q$
 8    else
 9      $Q'$ := $successors(\varphi_c)$
10      if $Q'$ is empty then fail
11      add all $Q'$ to $Q$
```

successors(φ) returns, as a set, the disjunction of conjunctive patterns representing the one-step successors of φ (see Section 4.2.2). σ is a substitution, and *frame*($\pi \land \psi$) returns ψ. The algorithm uses a queue Q of conjunctive patterns, which is initialized with the one-step successors of φ (lines 1-2). At each step the main loop (lines 3-11) processes a conjunctive pattern φ_c from Q. If φ_c implies φ' then verification succeeds on this execution path (line 5). If φ_c matches the left-hand-side of a specification rule in C then the respective rule is used to summarize its corresponding code (lines 6-7). Finally, if none of the cases above hold, add all one-step successors of φ_c to Q (lines 9-11). Using a specification is preferred over the operational semantics. If there are no *successors* (lines 2 and 10), the verification fails, as some concrete configurations satisfying the formula may not have a successor (e.g. a dereferenced pointer may be `NULL` in C). Our algorithm is incomplete, i.e., `fail` means that the specification cannot be verified successfully, not that it is violated by the code. Each pattern is simplified using function/abstraction definitions and lemmas before being added to Q.

The algorithm automates the proof system in Figure 3.1. Implementing the computation of multiple steps of symbolic execution across multiple paths with a queue corresponds to Transitivity and Reflexivity. Computing *successors* corresponds to Step, and splitting the subsequent disjunction to Case Analysis. Finishing an execution path (line 5) corresponds to Consequence. Using a specification rule (lines 6-7) corresponds to Consequence, Abstraction, and Axiom. Since Q is initialized with the successors of φ, a step of Transitivity already moved

$\mathcal{C}$ to $\mathcal{A}$. Consequence and Abstraction simplify a pattern before adding it to Q. Circularity allows for the rules in $\mathcal{C}$ to be used in their own proofs.

Our verification infrastructure is implemented in Java, and uses Z3 [26] for domain reasoning. It consists of approximately 30,000 non-blank lines of code, and it took about 2.5 man-years to complete. Next, we discuss in more details three aspects of our implementation: performing symbolic execution based on a $\mathbb{K}$ semantics, reasoning about the matching logic formulae (including using abstraction to express heaps properties), and integration with Z3.

Symbolic Execution

Language-independent symbolic execution is complicated by the absence of explicit control flow statements, which are language specific. We handle control flow statements by noticing they are generally unifiable with the left-hand-sides of several semantics rules. Consider the C code "`if (`*b*`) x = 1; else x = 0;`". It does not match the left-hand-side of any of the two semantics rules of `if` (they require the condition to be either the constant true or the constant false [30]), but it is unifiable with the left-hand-sides of both rules. We achieve symbolic execution by performing *narrowing* [2] (i.e., rewriting with unification instead of matching). When using the semantics rules, taking steps of rewriting on a ground configuration yields concrete execution, while taking steps of narrowing yields symbolic execution.

We compute *successors*$(\pi \wedge \psi)$ using *unification modulo theories*. We distinguish several theories (e.g. booleans, integers, sequences, sets, maps, etc) that the underlying SMT solver can reason about. Specifically, we unify $\pi \wedge \psi$ with the left-hand-side of a semantics rule $\pi_l \wedge \psi_l$. We begin with the syntactic unification of π and π_l. Upon encountering corresponding subterms (π' in π and π'_l in π_l) which are both terms of one of the theories above, we record an equality $\pi' = \pi'_l$ rather than decomposing the subterms further (if one is in a theory, and the other one is in a different theory or is not in any theory, unification fails). If this stage is successful, we end up with a conjunction ψ_u of equalities, some having a variable in one side and some with both sides in one of the theories. Then we check the satisfiability of $\psi \wedge \psi_u \wedge \psi_l$ using the SMT solver. If it is satisfiable, then $\pi_r \wedge \psi \wedge \psi_u \wedge \psi_l \wedge \psi_r$ is a successor of $\pi \wedge \psi$, where $\pi_r \wedge \psi_r$ is the right-hand-side of the semantics rule. Then *successors* is the disjunction of $\varphi_r \wedge \psi_u \wedge \psi \wedge \psi_l$ over all rules in $\mathcal{S}$ and all unification solutions ψ_u. While in general this disjunction may not be finite (see

Section 3.2), in practice it is finite for the examples we considered. Intuitively, "collecting" the constraints $\psi_u \wedge \psi_l \wedge \psi_r$ is similar to collecting the path constraint in traditional symbolic execution (but is done in a language-generic manner). For instance, the `if` case above, results in collecting the constraints $b = \texttt{true}$ and $b = \texttt{false}$. Notice that $\models \exists c\ (\varphi[c/\square] \wedge \varphi_l[c/\square])$ is satisfiable iff φ and φ_l are unifiable. Thus, we are sound by Step.

Several optimizations improve performance; we mention two. First, as the semantics of a real-world language consists of thousands of rules, the verifier uses an indexing algorithm to determine which rules may apply. Second, the verifier caches partial unification results, e.g., for each semantics rule, the verifier caches pairs of terms (t_1, t_2) that fail to unify with t_2 a subterm of the left-hand-side of the rule.

Matching Logic Prover

Matching logic reasoning is used in three cases in our algorithm: (1) to finish the proof (line 5), (2) to use a specification rule to summarize a code fragment (line 6), and (3) to simplify a pattern (before adding it to Q).

As discussed in Section 2.3, we use recursively-defined heap abstractions to specify the correctness of programs manipulating lists and trees in the heap. Such definitions exploit the recursive nature of the data-structures , e.g.,

$$\begin{aligned} &\text{tree}(x, \text{node}(n, t_l, t_r)) = \exists yz.x \mapsto [n, y, z],\ \text{tree}(y, t_l),\ \text{tree}(z, t_r) \\ &\text{tree}(0, \text{leaf}) = \text{emp} \end{aligned}$$

There is an extensive literature on such recursive definitions, especially in the context of separation logic (for example [68]).

We employ two heuristics. The first is natural proofs (see Chapter 5). We unfold a recursive definition during symbolic execution when we add conjunctive pattern $\pi \wedge \psi$ to Q if unfolding does not introduce a disjunction (i.e., ψ guarantee that only one of the cases in the definition holds). For example, in C, if ψ implies the head pointer p of a tree is `NULL`, then we conclude the tree is empty. If ψ implies p is not `NULL`, then we conclude p points to an object containing pointers to the left and right subtrees. Successful unfolding occurs at the start of symbolic execution, after a split (e.g. caused by **`if`**), or after using a specification rule (line 7). Unfolding makes a pattern more concrete, thus enabling operational semantics rules to apply. We similarly unfold recursive definitions on the right-hand-side of an implication. Unfolding is language-independent, as it is not triggered my memory accesses or other language-specific features.

While the above heuristic works on tree manipulating programs, it fails on list segment manipulating programs, as a list segment can be unfolded at both ends. We solve this by adapting the folding axioms proposed in [78] to work with data, and using them as additional lemmas for list segments on the left-hand-side of an implication, e.g.,

$$\text{lseg}(x, y, \alpha),\ \text{lseg}(y, 0, \beta) = \text{lseg}(x, 0, \alpha \cdot \beta)$$

Folding and unfolding are implemented by rewriting using the same infrastructure used for symbolic execution. The recursive definitions and the lemmas are all given as $\mathbb{K}$ rules.

As shown in Section 4.1.1, we use equationally constrained function and predicate symbols (like bst and tree_keys); e.g.,

$$\begin{aligned}
&\text{height}(\text{node}(\_, t_l, t_r)) = 1 + \max(\text{height}(t_l), \text{height}(t_r))\\
&\text{height}(\text{leaf}) = 0\\
&\text{height}(\_) \geq 0 = \text{true}
\end{aligned}$$

The first two define the height of a tree, while the third is a lemma. These equations are given as $\mathbb{K}$ rules, and are used in two ways: to simplify a formula by rewriting (oriented from left to right), and to be added in Z3 (see Section 4.2.2).

Integration with Z3

We use Z3 [26] to discharge the formulae that arise during matching logic reasoning (required by CONSEQUENCE and STEP). These formulae involve the following theories: integer, bitvector, set, sequence, and floating-point. We chose Z3 because of its very good performance, and because it offers features that are not part of the SMT-LIB standard, including variables instantiation patterns for universally quantified axioms, and mapping functions over arrays. While some of the formulae are not in decidable theories, in practice Z3 successfully checks them.

As discussed in Section 4.2.2, the formulae contain equationally constrained symbols. We encode these in Z3 as uninterpreted functions combined with assertions of the form "$\forall X.\ t = t'$". Z3 handles such assertions efficiently using E-matching [25]. By default, we specify the left-hand-side of these equations as the variables instantiation pattern, which in effect makes the equations only apply from left to right. This heuristic is effective in keeping the number of terms small. For a select few equations, like the ones for the sorted predicate for sequences, we wrote the patterns by hand.

Sets are one of the most important theories that we offer in our verifiers. We handle the set theory as proposed in [27]. We encode the sets themselves as arrays from the elements to true or false. Then, we encode the set operations as mapping of boolean functions over the arrays, and set membership as array lookup. The array map feature is only available in Z3, and is not part of the SMT-LIB standard. This results in a decidable theory for sets.

Unfortunately, this set encoding does not work well with the encoding of sequence theory symbols as equationally constrained uninterpreted functions. This case arises during the verification of the sorting examples. For this reason, we developed an encoding of sets using uninterpreted functions and universally quantified assertions. This encoding does not handle the set theory in a decidable way, but in practice it works with the sequence theory.

JavaScript verification generates floating-point constraints. Z3 has basic support for floating-point, but it does not integrate well with other theories. For this reason, we abstracted floating-point values to values in a partial-order relation, when the values only occur in comparisons and equality/inequality checks. This abstraction is used on the keys of the search trees or the values in the sorted lists.

For these reasons, we have different SMT encodings for the different programs we are verifying. We delegate to the user to choose which encodings are best suited for a given program.

4.2.3 Evaluation

We evaluate the $\mathbb{K}$ verification infrastructure by instantiating it with four different semantics, thus obtaining program verifiers for four different languages: KernelC (a simple toy C-like language), C, Java, and JavaScript (complex real-world languages). Our goal is to validate our hypothesis that building program verifiers by using $\mathbb{K}$ operational semantics and its verification infrastructure is effective both in terms of verification capabilities and tool building effort. To evaluate this hypothesis, first we implemented all the features required to verify the programs in Table 4.3 with KernelC: symbolic execution, reasoning with heap abstractions, integration with Z3, etc. Then we instantiated our framework with the off-the-shelf semantics of C11 [30, 43], Java 1.4 [14], and JavaScript 5.1 [75] to obtain corresponding program verifiers. We evaluated these verifiers by proving the correctness of the same programs in Table 4.3, but written in C, Java, and JavaScript. The implementation and the experiments are available on `http://kframework.org`.

The semantics we use are the most complete to date for their languages (see Table 4.4 for their size). As we mentioned before, given the complexity of real-world languages, we would like to separate the tricky language-specific features that are orthogonal to the verification process from the language-independent issues that make program verification hard. We achieve this by deferring to the semantics to handle the language-specific features (automatic promotions of integers in C, type checking, function call resolution, etc.). The $\mathbb{K}$ verification infrastructure handles the language-independent reasoning (heap-allocated mutable data structures, integers/bit-vectors/floating-points, etc.).

Operational Semantics

KernelC KernelC is a C-like pedagogical language distributed with the $\mathbb{K}$ framework. It has the following features: arithmetic and boolean expressions, `while` and `if` statements, recursive functions, structures, memory allocation/deallocation. It is slightly more complex than the standard languages used in program verification papers that introduce new heap logics/verification approaches. The language semantics consists of 106 $\mathbb{K}$ rules. Since the language is free of tricky features, so verification-friendly, using it with the $\mathbb{K}$ verification infrastructure did not reveal any unexpected challenges.

C [30, 43] The C operational semantics is the most complete formal semantics of C11. It captures over 75 types of undefined behavior in the ISO C standard, including restrict qualifiers, null pointers, out-of-bound object access, un-sequenced side effects, effective types, and many more. The semantics handles various basic data types like struct, enum, float, and bitfield. It also handles the tricky issues regarding integer types in C, including promotions, conversions, defined and undefined overflow and underflow. It supports the complex C statements, including goto. The semantics captures the intricacies of the C types system. It has over 2,500 $\mathbb{K}$ rules, developed over the course of several years. One advantage of the semantics arises from the fact that it is executable. Thus, the semantics has been thoroughly tested against a number of benchmarks in order to provide evidence of its correctness.

One of the most complex aspects of C is the memory model. The semantics faithfully specifies the C memory model, as a map from base pointers to objects, where each object is a map from offsets to bytes. Reads and writes happen at byte level. For a read of a value of a given type, the semantics reads a number of bytes from the memory equal to the size of the given type, and then interprets them to obtain a value of the given type. Similarly, for a write of a value of a given type, the semantics splits the value on a number of bytes equal to the size of the given type, and then writes them.

In terms of aspects specific to C verification, the operational semantics handles most of the tricky language features. The heap abstractions are more complex than for any other language, due to the fact that C models the memory at byte level. Thus, the heap abstractions need to mention how the value of each field of the struct implementing the data structure is obtained from bytes. Another aspect specific to C verification is due to the fact that the operational semantics models integers by using arbitrary precision integers modulo arithmetic, rather than bitvectors. Thus, specifications must state that all integer values are in the appropriate ranges.

Java [14] The Java operational semantics is the first complete formal semantics of Java 1.4. It was extensively tested with a large test suite developed alongside the project. The complexity of the semantics comes from the fact that Java is a statically, strongly typed, object-oriented, multi-threaded language. It has over 1,580 $\mathbb{K}$ rules and took 20 man-months to define.

Java semantics consists of two parts: (1) the preprocessing semantics that transforms the source Java program into another Java program restricted to use only a subset of the Java features, and (2) the execution semantics that deals with these core features. As the preprocessing semantics does not change the behavior of the program, we verified the output of the preprocessing stage directly.

In terms of verification, the Java semantics handles most of the complex language features. The heap abstractions take into account the possibility of object inheritance. Also, like in the case of C, the semantics models integer values by using arbitrary precision integers modulo arithmetic, forcing program specifications to also state that all integer values are in the appropriate ranges.

JavaScript [75] KJS is an operational semantics of JavaScript based on ECMAScript 5.1, which is the most complete and thoroughly tested semantics to date. It faithfully defines all the details of the language standard such as strict/non-strict modes, accessor (getter/setter) properties, dynamic (implicit) casting, `eval` construct, and (non-deterministic) `for-in` loop. It mechanizes the language standard, establishing an one-to-one correspondence with the standard, which facilitates manual inspection. It also has been tested against ECMAScript conformance test suite, and passes all 2,782 test programs for the core language. Combined with manual inspection, the test results increase the trust in the semantics. It consists of over 1,370 $\mathbb{K}$ rules and took four man-months to define.

In terms of verification purpose, KJS provides language-specific reasoning. Specifically, it deals with complicated object inheritances via prototype chains. Unlike C++ or JAVA, JAVASCRIPT objects directly inherit another object via a hidden, so-called prototype link. This direct inheritance (called prototypical inheritance) allows updating an object to immediately affect other objects whose inherited properties are dynamically modified, added, or removed. Due to this dynamic nature, even a simple object property lookup/update semantics is complicated, employing deeply nested case analyses to cover every possible situation. Reasoning about the complicated language-specific behavior is totally separated from the verification framework, being delegated to the semantics. JAVASCRIPT also has various unusual behaviors such as function-scoped environments or `this` value resolutions, taken care of by the semantics as well.

Verification Experiments

Here we discuss how effective in terms of proving capabilities it is to build program verifiers using $\mathbb{K}$ operational semantics. To this end, we have verified using our approach a number of challenging heap manipulating programs implementing the same data structure operations in KERNELC, C, JAVA, and JAVASCRIPT. These programs have been used before to evaluate verification approaches, e.g., in [77, 65, 68]. Our goal here is not to improve on the state-of-the-art in terms of proving capabilities or speed, but rather to show that we can also verify such programs at comparable performance, but in a language-independent setting. We conducted the experiments on a machine with Intel Core i7-4960X CPU 3.60GHz and DDR3 RAM 64GB 1333MHz

Our examples fall in two categories. (1) Singly-linked list manipulating programs, including implementations of common sorting algorithms. For each sorting function, we prove that the returned sequence is indeed sorted and has exactly the same elements as the original sequence. (2) Implementations of binary search tree,

Programs	KernelC				C				Java				JavaScript			
	Execution		Reasoning		Execution		Reasoning		Execution		Reasoning		Execution		Reasoning	
	Time	#Step	Time	#Query	Time	#Step	Time	#Query	Time	#Step	Time	#Query	Time	#Step	Time	#Query
BST find	0.6	192	1.2	95	10.4	1,028	3.6	246	1.9	322	2.8	244	4.5	1,736	1.8	93
BST insert	0.8	336	2.9	160	23.0	2,481	7.2	414	4.1	691	4.5	342	5.4	3,394	2.8	158
BST delete	1.4	582	5.6	420	55.1	4,540	16.6	938	9.8	1,274	15.1	1,125	15.6	5,052	5.6	373
AVL find	0.6	192	1.2	95	9.9	1,028	3.1	214	2.2	322	2.7	244	4.5	1,736	1.9	93
AVL insert	6.2	1,980	42.1	1,133	210.7	12,616	70.6	1,865	42.4	3,753	62.8	2,146	102.5	26,977	32.5	1,221
AVL delete	9.5	2,933	45.4	1,758	514.8	26,003	118.9	3,883	122.2	8,144	149.4	4,866	184.3	38,591	55.3	2,233
RBT find	0.6	192	1.1	95	11.5	1,064	3.0	214	2.1	322	2.9	244	4.9	1,736	1.9	93
RBT insert	7.6	2,331	48.1	1,392	722.0	30,924	181.8	4,394	39.9	4,240	75.7	2,547	84.9	28,082	29.6	1,381
RBT delete	10.6	3,891	33.7	2,033	1593.8	50,389	308.3	15,429	95.8	8,312	75.4	4,460	144.2	51,356	39.4	2,009
Treap find	0.6	200	1.4	118	11.2	1,064	3.2	214	2.0	322	2.9	244	4.6	1,736	1.9	116
Treap insert	1.4	753	4.5	247	52.4	4,954	15.3	724	12.7	1,469	10.4	563	13.7	7,738	5.2	243
Treap delete	2.0	831	9.4	509	73.9	5,512	16.5	656	12.0	1,694	16.4	1,021	24.8	8,333	8.4	460
List reverse	0.4	142	0.3	21	6.6	815	4.8	76	1.5	222	2.6	46	5.0	1,162	0.5	20
List append	0.4	171	0.5	45	7.4	909	7.4	128	1.8	239	5.5	106	4.5	1,392	0.8	46
Bubble sort	0.9	391	26.8	190	28.4	2,401	38.0	357	3.4	589	35.4	345	5.6	2,688	25.7	145
Insertion sort	1.1	468	24.5	300	26.6	2,555	35.3	451	4.1	731	27.0	371	8.3	3,119	36.5	213
Quick sort	1.1	604	31.6	269	31.0	3,601	48.2	518	7.1	958	40.0	413	15.0	5,046	33.1	252
Merge sort	1.7	970	55.0	478	81.6	6,589	89.0	1,070	14.1	1,566	72.9	737	22.8	7,021	43.2	480
Total	47.7	17,159	335.2	9,358	3470.5	158,473	970.6	31,791	379.3	35,170	604.5	20,064	654.9	196,895	326.3	9,629

Table 4.3: Summary of verification experiments: 'Execution' shows time (seconds) and number of operational semantic steps for symbolic execution (Section 4.2.2); 'Reasoning' shows time (seconds) and number of Z3 queries for reasoning (Section 4.2.2 & 4.2.2).

AVL tree, red-black tree (RBT), and Treap data-structure operations. For each function, we prove that it maintains the data-structure invariants and that the set of elements is as expected.

Table 4.3 summarises our experiments. For KernelC, which is idealized for verification, proving the implications required by Consequence (shown in the Reasoning column) dominates the total verification time. C, Java, and JavaScript are complex languages, so the semantics-based symbolic execution (shown in the Execution column) dominates the verification time. Note that since the programs implement the same data structure operations in different languages, the complexity of implications required by Consequence tends to be similar. Thus, the complexity of the operational semantics is the most important factor contributing to the difference in the verification times reported. As expected, since C has the most complex operational semantics, the times for C are the largest. The number of queries of logical reasoning for C and Java is higher than for JavaScript because of 32-bit integer range constraints, while the time spent on each query is similar along the different languages, reflecting that the reasoning is language-independent. Furthermore, each step of symbolic execution for JavaScript is much smaller than for C and Java, because the JavaScript semantics is more fine-grained.

The AVL and RBT insert and delete programs take considerably longer than the other programs because some of the auxiliary functions (like balance, rotate, etc) are not given specifications and thus their bodies are being inlined, resulting in a larger number of paths to analyze. To put this in perspective, VCDryad [77], a state-of-the-art separation logic verifier for C build on top of VCC, takes 260s to verify only the balance function in AVL, while it takes our generic infrastructure instantiated with the C semantics 210s to verify AVL insert (including balance). In general, we believe Table 4.3 suggests that our approach is practical and competitive with the state-of-the-art on such data-structures.

Development Cost

We discuss how cost effective in terms of tool development it is to build program verifiers using $\mathbb{K}$ operational semantics and our verification infrastructure. Recall that the semantics of C, Java, and JavaScript were developed as separate projects, independently from the verification infrastructure.

Table 4.4 shows the development effort of our approach. The language-specific effort consists of familiarizing with the semantics in order to be able to write the

	C	JAVA	JAVASCRIPT
Semantics development (months)	40	20	4
Semantics size (#rules)	2,572	1,587	1,378
Semantics size (LOC)	17,791	13,417	6,821
Language-specific effort (days)	7	4	5
Semantics changes size (#rules)	63	38	12
Semantics changes size (LOC)	468	95	49
Specifications	36	31	31
Abstractions	6	6	6
Function definitions	14	14	14
Lemmas	7	7	7

Table 4.4: The development costs

correctness specifications as reachability rules (like the ones in Section 4.1.1), and of making changes to the semantics. Most of changes to the semantics are bug fixes (see Section 4.2.3), but some are performance improvements or simplifications. The development effort scales with the language complexity. The effort for C is considerably larger than for JAVA and JAVASCRIPT due to the low level complexity of C. Overall, the numbers in Table 4.4 validate our hypothesis that program verification based on operational semantics and the $\mathbb{K}$ verification infrastructure is cost effective in terms of development effort.

For comparison, the state-of-the-art is to define a translator to an intermediate verification language, like Boogie, or to define a verification condition (VC) generator. For example, the VCC translator from C to Boogie consists of approximately 5000 lines of F# [1]. We believe that writing such a translator takes considerably more effort than we reported for our approach in Table 4.4 (we do not include the time to define the semantics into this comparison, since we assume the semantics already exist, and they serve other purposes as well). Moreover, we believe that one would have more confidence in an operational semantics to handle the tricky details of complex languages than in a translation or a VC generator, for two reasons. First, an operational semantics is more amenable to visual inspection, as it is written in a domain-specific language for defining semantics. Second, an operational semantics is executable and can be thoroughly tested. While this does not guarantee the absence of bugs (see Section 4.2.3), it greatly reduces their occurrence.

Even if a semantics is not already available, we believe that developing an operational semantics has an important advantage over building a translator or a VC generator: the semantics is used not only for verification, but for other purposes as well, so overall the semantics development cost is amortized. For example, the JavaScript semantics was used for bug finding in browsers [75].

Regarding number of annotations, our approach is comparable to the state-of-the-art language-specific approaches that do not infer invariants (VCC, Frama-C). The user provides one specification for each recursive function and loop. The user also provides the definitions for heap abstractions and auxiliary functions used in specifications. The user does not provide anything similar to ghost code or hints for the verifier. The user may need to provide additional lemmas and those lemmas apply to a class of programs rather than one particular program (e.g., the lemmas for list segments in Section 4.2.2 are shared by all sorting-related programs in all languages).

Operational Semantics Bugs

We found bugs in all the three operational semantics used for verification, despite the fact that these semantics are thoroughly tested on thousands of programs [30, 43, 14, 75].

The main source of bugs is the unintended non-determinism in the semantics. A semantics models a non-deterministic feature by having multiple rules that can apply at the same time. Such a feature is the expression evaluation order in C: "`f() + g()`" may call `f()` first and `g()` second or `g()` first and `f()` second. As a result, only a fraction of the possible behaviors are observed under testing. During symbolic execution, the $\mathbb{K}$ verifier considers all the rules that can apply (according to Step in Figure 3.1). This revealed that each semantics contained unintended non-determinism: pairs of rules where the semantics developers intended for one rule to always apply before the other, but in fact both rules can apply simultaneously. Applying the rules in the other order causes an incorrect result. We also found other kinds of bugs, mostly caused by incorrect side conditions of the semantics rules, or incorrect assumptions about the configuration.

We proposed fixes for the bugs we found and the semantics' authors accepted them. This indicates the existing methodology to validate semantics needs improvement.

Chapter 5

DRYAD

In this chapter, we turn our attention to automating reasoning about state properties, with an emphasis on heap properties. As discussed in Section 4.2.2, we want to implement automated reasoning techniques as part of the $\mathbb{K}$ verification infrastructure. Much of the work in this chapter comes from Madhusudan et al. [62] and Qiu et al. [80].

In recent years, the *automated deductive verification paradigm* for software verification that combines user written modular contracts and loop invariants with automated theorem proving of the resulting verification conditions has become very powerful. The latter process is often executed by automated logical decision procedures supported by SMT solvers, which have emerged as robust and powerful engines to automatically find proofs. Several techniques and tools have been developed [22, 8, 48] and there have been several success stories of large software verification projects using this approach (the Verve OS project [106], the Microsoft hypervisor verification project using VCC [22], and a recent verified-for-security OS+browser for mobile applications [63], to name a few).

Verification conditions do not, however, always fall into decidable theories. In particular, the verification of properties of the *dynamically modified heap* is a big challenge for logical methods. The dynamically manipulated heap poses several challenges, as typical correctness properties of heaps require complex combinations of structure (e.g., a pointer *p* points to a tree structure, or to a doubly-linked list, or to an almost balanced tree, with respect to certain pointer-fields), data (the integers stored in data-fields of the tree respect the binary search tree property, or the data stored in a tree is a max-heap), and separation (the procedure modifies one list and not the other and leaves the two lists disjoint at exit, etc.).

The fact that the dynamic heap contains an unbounded number of locations means that expressing the above properties requires *quantification* in some form, which immediately precludes the use of most SMT decidable theories (there are only a few of them known that can handle quantification; e.g., the array property fragment [17] and the STRAND logic [61, 60]). Consequently, *expressing* such properties naturally and succinctly in a logical formalism has been challenging, and reasoning with them automatically even more so.

For instance, in the Boogie line of tools (including VCC) of writing specifications using first-order logic and employing SMT solvers to validate verification conditions, the specification of invariants of even simple methods like *singly-linked-list insert* is tedious. In such code [1], second-order properties (reachability, acyclicity, separation, etc.) are smuggled in using carefully chosen *ghost variables*; for example, acyclicity of a list is encoded by assigning a ghost number (idx) to each node in the list, with the property that the numbers associated with adjacent nodes strictly increase going down the list. These ghost variables require careful manipulation when the structures are updated; for example, inserting a node may require updating the ghost numbers for other nodes in the list, in order to maintain the acyclicity property. Once such a ghost-encoding of the specification is formulated, the validation of verification conditions, which typically have quantifiers, are dealt with using sound heuristics (a wide variety of them including e-matching, model-based quantifier instantiation, etc. are available), but are still often not enough and have to be augmented by *instantiation triggers* from the verification engineer to help the proof go through.

In recent years, *separation logic*, especially in combination with recursive definitions, has emerged as a much more succinct and natural logic to express properties about structure and separation [85, 73]. However, the validation of verification conditions resulting from separation logic invariants are also complex, and has eluded automatic reasoning and exploitation of SMT solvers (even more so than tools such as Boogie that use classical logic). Again, help from the user in proving the verification conditions are currently necessary— the tools VERIFAST [48] and BEDROCK [19], for instance, admit separation logic specifications but require the user to write low-level lemmas and proof tactics to guide the verification. For example, in verifying an in-place reversal of a linked list[2], BEDROCK would require several lemmas and a hint package be supplied at the level of the code in order for the proof to go through.

The work in this paper is motivated by the opinion that entirely decidable logics are too restrictive, in general, to support the verification of complex specifications

[1] `http://vcc.codeplex.com/SourceControl/changeset/view/dcaa4d0ee8c2#vcc/Docs/Tutorial/c/7.2.list.c`

[2] `http://plv.csail.mit.edu/bedrock/Tutorial.html`

of functional correctness for heap manipulating programs, and the other extreme of user-supplied proof tactics and lemmas is too tedious, requiring of the user too much knowledge of the underlying proof systems/decision procedures. Our aim is to build completely automatic, sound, but incomplete proof techniques that can solve a large class of properties involving complex data-structures.

The natural proof methodology:

The natural proofs methodology exploits a *fixed* set of proof tactics, keeping the expressiveness of powerful logics, retaining the automated nature of proving validity, but giving up on completeness (i.e., giving up decidability, retaining soundness). The idea of natural proofs is to identify a subclass of proofs $\mathcal{N}$ such that (a) a large class of valid verification conditions of real-world programs have a proof in $\mathcal{N}$, and (b) searching for a proof in $\mathcal{N}$ is *decidable*. In fact, we would even like the search for a proof in $\mathcal{N}$ to be efficiently decidable, possibly utilizing the automatic logic solvers (SMT solvers) that exist today. Natural proofs are hence a fixed set of proof tactics whose application is itself expressible in a decidable logic.

The results in this chapter provide natural proofs for general properties of structure, data, and separation. Our contributions are: (a) we propose Dryad, a dialect of separation logic for heaps, with no explicit (classical) quantification but with recursive definitions, to express second-order properties; (b) we show that Dryad is both powerful in terms of expressiveness, and that the strongest-post of Dryad specifications with respect to bounded code segments can be formulated in Dryad, (c) we show how Dryad has been designed so that it can be systematically converted to *classical* logic using the theory of sets, allowing us to connect the more natural and succinct specifications to more verbose but classical logic, and (d) we develop a natural proof mechanism for classical logics with recursion and sets that implement sound but incomplete reductions to decidable theories that can be handled by an SMT solver.

Dryad: A separation logic with determined heaplets

The primary design principle behind separation logic is the decision to express *strict specifications*— logical formulas must naturally refer to *heaplets* (subparts of the heap), and, by default, the smallest heaplets over which the formula needs to refer to. This is in contrast to classical logics (such as FOL) which implicitly refer to the entire heap globally. Strict specifications permit elegant ways to capture how a particular sub-part of the heap changes due to a procedure, implicitly leaving the rest of the heap and its properties unchanged across a call to this procedure. Separation logic is a particular framework for strict specifications, where formulas are implicitly defined on strictly defined heaplets, and where heaplets can be combined using a *spatial conjunction operator* denoted by $*$. The frame rule in separation logic captures the main advantage of strict specifications: if the Hoare-triple $\{P\}C\{Q\}$ holds for some program C, then $\{P * R\}C\{Q * R\}$ also holds (with side-conditions that the modified variables in C are disjoint from the free variables in R).

Consider, for example, expressing that the location x is the root of a tree. This is a *second-order* property and formulations of it in classical logic using set or path quantification are quite complex and not easily amenable to automated verification. We prefer *inductive* definitions of structural properties without any explicit quantification. The separation logic syntax with recursive definitions and heaplet semantics allows simple quantifier-free formulas to express structural restrictions; for example. tree-ness can be expressed simply as:

$$tree(x) :: (x = \texttt{nil} \wedge \texttt{emp}) \vee (x \longmapsto (l, r) * tree(l) * tree(r))$$

We first define a new logic, Dryad, that permits no explicit quantification, but permits powerful recursive definitions to define integers, sets/multisets of integers, and sets of locations, using least fixed-points. The logic Dryad furthermore has a heaplet semantics and allows the spatial conjunction operator $*$. However, a key design feature of Dryad is that the heaplet for recursive formulas is essentially *determined* by the syntax as opposed to the semantics. In classical separation logic, a formula of the form $\alpha * \beta$ says that the heaplet can be partitioned into *any* two disjoint heaplets, one satisfying α and the other β. In Dryad, the heaplet for a (complex) formula is *determined* and hence if there is a way to partition the heaplet, there is precisely *one* way to do so. We have found that most uses of separation logic to express properties can be written quite succinctly and easily using Dryad (in fact, it is *easier* to write such deterministic-heap specifications). The key advantage is that this eliminates implicit existential quantification the separation operator provides. In a verification condition that combines the pre-condition in the negative and the post-condition in the positive, the classical semantics for separation logic invariably introduces universal quantification in the satisfiability query for the negation of the verification condition, which in turn is extremely hard to handle.

In Dryad, the semantics of a recursive definition $r(x)$ (such as *tree* above), requires that the heaplet be determined and defined as the set of all locations reachable from the node x through a set of pointer-fields $f_1, ..., f_k$ without passing through a set of locations (given by a set of location terms $t_1, ...t_n$). While our logical mechanisms can be extended beyond this notion (in deterministic ways), we have found that this captures most common properties required in proving data-structure manipulating programs correct.

Translating Dryad to classical logic with recursion:
The second key step in our paradigm is a technique to bridge the gap from separation logic to classical logic in order to utilize efficient decision procedures supported by SMT solvers. We show that heaplet semantics and separation logic constructs of Dryad can be effectively translated to classical logic where heaplets are modeled as *sets of locations*. We show that Dryad formulas can be translated into classical logic with free set variables that capture the heaplets corresponding to the strict semantics. This translation does not, of course, yield a decidable theory yet, as recursive definitions are still present (the recursion-free formulas are in a decidable theory). The carefully designed Dryad logic with determined heaplet semantics ensures that there is no quantification in the resulting formula in classical logic. The heaplets of recursively defined properties, which are defined using the set of all *reachable* nodes, are translated to recursively defined sets of locations.

Natural proofs for Dryad:
Finally, we develop a natural proof methodology for Dryad by showing a natural proof mechanism for the equivalent formulas in classical logic. The basic proof tactic that we follow is not just dependent on the formula embodying the verification condition, but also on the precise footprint touched by the program segment being verified. We unfold recursive definitions precisely across footprints, translating them to the frontier of the footprint, and then use a form of *formula abstraction* that treats recursive formulas on frontier nodes as uninterpreted functions. The resulting formula falls in a logic over sets and integers, which is then decided using the theory of uninterpreted functions and arrays using SMT solvers. The key feature is that heaplets and separation logic constructs, which get translated to recursively defined sets of locations, are unfolded along with other user-defined recursive definitions and formula-abstracted using this uniform natural proof strategy,

While our proof strategy is roughly as above, there are many technical details that are complex. For example, the heaplets defined by pre/post conditions intrinsically specify the modified locations of the heap, which have to be considered when processing procedure calls in order to ensure which recursively defined metrics on locations continue to hold after a procedure call. Also, the final decidable theories that we compile our conditions down to does require a bit of quantification, but it turns out to be in the *array property fragment* which admits automatic decision procedures.

Implementation and Evaluation:
Our proof mechanisms are essentially a class of decidable proof tactics that result in sound but incomplete validation procedures. To show that this class of natural proofs is effective in practice, we provide a prototype implementation of our technique, which handles a low-level programming language with pre-conditions and post-conditions written in Dryad. We show, using a large class of correct programs manipulating lists, trees, cyclic lists, and doubly linked lists as well as *multiple* data-structures of these kinds, that the natural proof mechanism succeeds in proving the verifications conditions automatically. These programs are drawn from a range of sources, from textbook data-structure routines (binary search trees, red-black trees, etc.) to routines from Glib low-level C-routines used in GTK+/Gnome to implement file-systems, from the Schorr-Waite garbage collection algorithm, to several programs from a recent secure framework developed for mobile applications [63]. The work presented here is by far the only one that we know of that can handle such a large class of programs, completely automatically. Our experience has been that the user-provided contracts and invariants are easily expressible in Dryad, and the automatic natural proof mechanisms work extremely fast. In fact, contrary to our own expectations, we also found that the tool is useful in *debugging*: in several cases, when the annotations supplied were incorrect, the model provided by the SMT solver for the natural proof was useful in detecting errors and correcting the invariants/program.

5.1 Motivating Example

In this section we give intuition into our verification approach through a motivating example. Recall that a max-heap is a binary tree such that for each node *n* the key stored at *n* is greater than or equal to the keys stored at each of its children. Heaps

```
void heapify(loc x) {
 if (x.left = nil)
   s := x.right;
 else if (x.right = nil)
   s := x.left;
 else {
   lx := x.left;
   rx := x.right;
   if (lx.key < rx.key)
     s := x.right;
   else
     s := x.left;
 }
 if (s =/= nil)
   if (s.key > x.key) {
     t := s.key;
     s.key := x.key;
     x.key := t;
     heapify(s);
   }
}
```

$$\varphi_{pre} \equiv \Big(x \overset{key,left,right}{\longmapsto} (k,\ l,\ r) * mheap^{\Delta}_{\vec{pf}}(l) * mheap^{\Delta}_{\vec{pf}}(r)\Big) \land keys^{\Delta}_{\vec{pf}}(x) = K$$

$$\varphi_{post} \equiv mheap^{\Delta}_{\vec{pf}}(x) \land keys^{\Delta}_{\vec{pf}}(x) = K$$

```
assume $x.left_0 \neq nil$
assume $x.right_0 \neq nil$
$lx := x.left_0$
$rx := x.right_0$
assume $lx.key_0 < rx.key_0$
$s := x.right_0$
assume $s \neq nil$
assume $s.key_0 > x.key_0$
$t := s.key_0$
$s.key_1 := x.key_0$
$x.key_2 := t$
heapify($s$)
```

$$\begin{aligned}
&\boldsymbol{mheap}^{\Delta}_{\vec{pf}}(x) \overset{def}{=} \Big(\\
&\quad x = \texttt{nil} \land \texttt{emp}\\
&\lor \big(x \overset{key,left,right}{\longmapsto} (k,\ l,\ r)\\
&\quad * (mheap^{\Delta}_{\vec{pf}}(l) \land \{k\} \geq keys^{\Delta}_{\vec{pf}}(l))\\
&\quad * (mheap^{\Delta}_{\vec{pf}}(r) \land \{k\} \geq keys^{\Delta}_{\vec{pf}}(r))\big)\Big)
\end{aligned}$$

$$\begin{aligned}
&\boldsymbol{keys}^{\Delta}_{\vec{pf}}(x) \overset{def}{=} \Big(\\
&\quad x = \texttt{nil} \land \texttt{emp} : \emptyset\ ;\\
&\quad x \overset{key,left,right}{\longmapsto} (k,\ l,\ r) * \texttt{true} :\\
&\qquad keys^{\Delta}_{\vec{pf}}(l) \cup \{k\} \cup keys^{\Delta}_{\vec{pf}}(r)\ ;\\
&\quad \text{default} : \emptyset\Big)
\end{aligned}$$

Figure 5.1: Motivating example: Heapify

are often used to implement priority queues. In Figure 5.1, in the lower right corner, we express the property that a location x points to a max-heap using recursive definitions $keys^{\Delta}_{\vec{pf}}(x)$ and $mheap^{\Delta}_{\vec{pf}}(x)$, with $\vec{pf} \equiv \{left, right\}$. These recursive definitions are written in Dryad, which is formally introduced in Section 5.2. Intuitively, Dryad extends quantifier free separation logic [85, 73] with recursive predicates and functions. These recursive definitions allow us to express structural and data properties on the heap, like those of max-heap, without explicit quantification.

For a location x, the recursive definition $keys^{\Delta}_{\vec{pf}}(x)$ returns the set of keys at the nodes of the tree rooted at x: if x is `nil` and the heaplet is empty, then the empty-set; otherwise, the union of the key stored at x and the keys stored in the left and right subtrees of x. Similarly, the recursive definition $mheap^{\Delta}_{\vec{pf}}(x)$ states that x points to a max-heap if: x is `nil` and the heaplet is empty; or x and the heaplets of the left and right subtrees of x are mutually disjoint (x points to a tree) and the key at x is greater than or equal to the keys of the left and right subtrees of x.

The method `heapify` in Figure 5.1 is at the heart of the procedure for deleting the root from a max-heap (removing the node with the maximum priority). If the max-heap property is violated at a node x while satisfied by its descendants, then `heapify` restores the max-heap property at x. It does so by recursively descending into the tree, swapping the key of the root with the key at its left or right child, whichever is greater. The precondition φ_{pre} binds the free variable K to the set of keys of x. The postcondition states that after the procedure call, x satisfies the max-heap property and the set of keys of x is unchanged (same as K).

One of the main aspects of our approach is to reduce reasoning about heaplet semantics and separation logic constructs to reasoning about sets of locations. We use set operations like union, intersection and membership to describe separation constraints on a heaplet satisfying a formula. This translation from Dryad formulas, like those in Figure 5.1, to formulas in classical logic with recursive predicates and functions is formally presented in Section 5.3. Intuitively, we associate a set of locations to each (spatial) atomic formula, which is the domain of the heaplet satisfying that formula. Dryad requires that this heaplet is *syntactically* determined for each formula. For example, the heaplet associated to the formula $x \mapsto ...$ is the singleton $\{x\}$; for recursive definitions like $mheap^{\Delta}_{\vec{pf}}(x)$ and $keys^{\Delta}_{\vec{pf}}(x)$, the domain of the heaplet is $reach_{\{left,right\}}(x)$, which intuitively is the set of locations reachable from x using the pointer fields *left* and *right*, and can be defined recursively.

As shown in Figure 5.1, φ_{pre} is a conjunction of two formulas. If G_{pre} is the domain of the heaplet associated to φ_{pre}, then the first conjunct requires G_{pre} to be the disjoint union of the sets $\{x\}$, $reach_{\{left,right\}}(left(x))$ and $reach_{\{left,right\}}(right(x))$. The second conjunct requires $G_{pre} = reach_{\{left,right\}}(x)$. From these heaplet constraints, we can translate φ_{pre} to the following formula in classical logic *over the global heap*:

$$\begin{aligned}
& G_{pre} = \{x\} \cup reach_{\{left,right\}}(left(x)) \cup reach_{\{left,right\}}(right(x)) \\
\wedge\ & x \notin reach_{\{left,right\}}(left(x)) \wedge x \notin reach_{\{left,right\}}(right(x)) \\
\wedge\ & reach_{\{left,right\}}(left(x)) \cap reach_{\{left,right\}}(right(x)) = \emptyset \wedge x \neq nil \\
\wedge\ & mheap(left(x)) \wedge mheap(right(x)) \\
\wedge\ & G_{pre} = reach_{\{left,right\}}(x) \wedge keys(x) = K
\end{aligned}$$

Similarly, we translate φ_{post} to

$$G_{post} = reach_{\{left,right\}}(x) \wedge mheap(x) \wedge keys(x) = K$$

Note that the recursive definitions *mheap* and *keys* without the "Δ" superscript are in the classical logic (without the heaplet constraint). Hence the recursive predicate *mheap* satisfies

$$\begin{aligned}
mheap(x) \leftrightarrow\ & (x = nil \wedge reach_{\{left,right\}}(x) = \emptyset) \\
\vee\ & \Big(x \neq nil \wedge x \notin reach_{\{left,right\}}(left(x)) \wedge x \notin reach_{\{left,right\}}(right(x)) \\
& \wedge reach_{\{left,right\}}(left(x)) \cap reach_{\{left,right\}}(right(x)) = \emptyset \\
& \wedge (reach_{\{left,right\}}(x) = \{x\} \cup reach_{\{left,right\}}(left(x)) \\
& \qquad \cup reach_{\{left,right\}}(right(x))) \\
& \wedge mheap(left(x)) \wedge \{key(x)\} \geq keys(left(x)) \\
& \wedge mheap(right(x)) \wedge \{key(x)\} \geq keys(right(x))\Big)
\end{aligned}$$

The right side of Figure 5.1 presents a basic path from method `heapify`, corresponding to the case when both children of x are not *nil* and the key of the right child is greater than the keys of the left child and the root. The subscript of a pointer/data field denotes the timestamp. A key insight is that any basic path touches a finite number of locations and may call some recursive procedures. We refer to the touched locations as the *footprint*, and to the adjacent locations which are not part of the footprint as the *frontier*. For this example, the footprint is $\{\ x,\ lx,\ rx\ \}$ (s is known to be equal with rx) and the frontier is $\{\ left_0(lx),\ right_0(lx),\ left_0(rx),\ right_0(rx)\ \}$. We capture the effect of the path until the call to `heapify` by

$$\begin{aligned}
& left_0(x) \neq nil \wedge right_0(x) \neq nil \wedge lx = left_0(x) \wedge rx = right_0(x) \\
\wedge\ & key_0(lx) < key_0(rx) \wedge\ s = right_0(x) \wedge s \neq nil \\
\wedge\ & key_0(s) > key_0(x) \wedge t = key_0(s) \\
\wedge\ & key_1 = key_0\{s \leftarrow key_0(x)\} \wedge key_2 = key_1\{x \leftarrow t\}
\end{aligned}$$

Once we have expressed the verification condition in classical logic with recursive definitions over the global heap, we prove it using the *natural proof* methodology. We unfold the recursive definitions $mheap(x)$, $keys(x)$ and $reach_{\{left,right\}}(x)$ for x, lx and rx (the footprint), thus evaluating them in terms of their values on the frontier. The call to `heapify` preserves the recursive definitions on locations reachable from *lx*, and modifies those on *rx* according to the pre/post condition. Finally, we abstract the recursive definitions on the frontier with uninterpreted functions. We decide the resulted formula (which is in a decidable logic) using an SMT solver. Section 5.4 describes this process in detail.

5.2 The Logic Dryad

In this section we present our logic Dryad, which is defined using heaplet semantics and separation logic primitives. Hence, the logic is a quantifier-free heaplet logic augmented with recursively defined predicates/functions.

5.2.1 Syntax

Let us fix a finite set of *pointer-fields PF* and a finite set of *data-fields DF*. A record consists of a set of pointer-fields from *PF* and a set of data-fields from *DF*. Our logic also presumes that locations refer to entire records rather than particular fields, and that address arithmetic is precluded. We will use the term *locations* hence to refer to these records. We assume that every field is defined at every location, i.e., all memory records have the same layout (to simplify the presentation); our logic can easily be extended with record types.

Let $Bool = \{\texttt{true}, \texttt{false}\}$ stand for the set of Boolean values, *Int* stand for the set of integers and *Loc* stand for the universe of locations. For any set A, let $\mathcal{S}(A)$ denote the set of all finite subsets of A, and let $\mathcal{MS}(A)$ denote the set of all finite multisets with elements in A.

The Dryad logic allows expressing quantifier-free first-order properties over heaps/heaplets augmented with recursively defined notions for a location to express second-order properties, denoted as a function $r : Loc \to D$. The codomain D can be Int_L, $\mathcal{S}(Loc)$, $\mathcal{S}(Int)$, $\mathcal{MS}(Int)_L$ or $Bool$, where Int_L and $\mathcal{MS}(Int)_L$ extend Int and $\mathcal{MS}(Int)$ to lattice domains, respectively, in order to give least fixed-point semantics (explained later in this section). Typical examples of these recursive definitions include the definitions of the height of a tree or the height of black-nodes in the tree rooted at a node (recursively defined integers), the set of nodes reachable from a location following certain pointer fields (recursively defined sets of locations), the set/multiset of keys stored at a particular data-field under nodes reachable from a location (recursively defined set/multiset of integers), and the

$$\begin{array}{lllll}
i^{\Delta} : Loc \rightarrow Int_L & sl^{\Delta} : Loc \rightarrow \mathcal{S}(Loc) & si^{\Delta} : Loc \rightarrow \mathcal{S}(Int) & msi^{\Delta} : Loc \rightarrow \mathcal{MS}(Int)_L & p^{\Delta} : Loc \rightarrow Bool \\
j \in Int_L \text{ Variables} & L \in \mathcal{S}(Loc) \text{ Variables} & S \in \mathcal{S}(Int) \text{ Variables} & MS \in \mathcal{MS}(Int)_L \text{ Variables} & q \in Bool \text{ Variables} \\
x \in Loc \text{ Variables} & c : Int_L \text{ Constant} & pf \in PF & df \in DF &
\end{array}$$

$$\begin{array}{rlll}
Loc \text{ Term:} & lt & ::= & x \mid \mathtt{nil} \\
Int_L \text{ Term:} & it & ::= & c \mid j \mid i^{\Delta}_{\vec{pf},\vec{v}}(lt) \mid it + it \mid it - it \\
\mathcal{S}(Loc) \text{ Term:} & slt & ::= & \emptyset_l \mid L \mid \{lt\} \mid sl^{\Delta}_{\vec{pf},\vec{v}}(lt) \mid slt \cup slt \mid slt \cap slt \mid slt \setminus slt \\
\mathcal{S}(Int) \text{ Term:} & sit & ::= & \emptyset_s \mid S \mid \{it\} \mid si^{\Delta}_{\vec{pf},\vec{v}}(lt) \mid sit \cup sit \mid sit \cap sit \mid sit \setminus sit \\
\mathcal{MS}(Int)_L \text{ Term:} & msit & ::= & \emptyset_m \mid MS \mid \{it\}_m \mid msi^{\Delta}_{\vec{pf},\vec{v}}(lt) \mid msit \cup msit \mid msit \cap msit \mid msit \setminus msit
\end{array}$$

$$\begin{array}{rlll}
\text{Positive Formula:} & \varphi & ::= & \mathtt{true} \mid \mathtt{false} \mid q \mid p^{\Delta}_{\vec{pf},\vec{v}}(lt) \mid \mathtt{emp} \mid lt \overset{\vec{pf},\vec{df}}{\longmapsto} (\vec{lt}, \vec{it}) \mid lt = lt \mid lt \neq lt \mid it \leq it \mid it < it \mid sit \leq sit \mid sit < sit \\
& & & \mid msit \leq msit \mid msit < msit \mid slt \subseteq slt \mid slt \nsubseteq slt \mid sit \subseteq sit \mid sit \nsubseteq sit \mid msit \sqsubseteq msit \mid msit \not\sqsubseteq msit \\
& & & \mid lt \in slt \mid lt \notin slt \mid it \in sit \mid it \notin sit \mid it \in msit \mid it \notin msit \\
& & & \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi * \varphi \\
\text{Formula:} & \psi & ::= & \varphi \mid \psi \wedge \psi \mid \psi \vee \psi \mid \neg\psi
\end{array}$$

$$\begin{array}{rlll}
\text{Recursive function :} & f^{\Delta}_{\vec{pf},\vec{v}}(x) & \overset{def}{=} & (\varphi^f_1(x, \vec{v}, \vec{s}) : t^f_1(x, \vec{s})\ ;\ \ldots\ ;\ \varphi^f_k(x, \vec{v}, \vec{s}) : t^f_k(x, \vec{s})\ ;\ \text{default} : t^f_{k+1}(x, \vec{s})) \\
\text{Recursive predicate :} & p^{\Delta}_{\vec{pf},\vec{v}}(x) & \overset{def}{=} & \varphi^p(x, \vec{v}, \vec{s})
\end{array}$$

Figure 5.2: Syntax of Dryad

property that the tree rooted at a node is a binary search tree or a balanced tree or just a tree (recursively defined predicates).

A DRYAD formula φ is quantifier-free, but parameterized by a set of recursive definitions Def^{Δ}. The syntax of DRYAD logic is given in Figure 5.2, where the syntax of formulas is followed by the syntax for recursive definitions. Most symbols in DRYAD are common and self-explanatory. Note that the inequality ($<$ or $\leq$) between integer sets/multisets indicates that any integer in the left-hand side is less-than/not-greater-than any integer in the right-hand side. It is also noteworthy that the separating conjunction ($*$) from separation logic is also allowed, but only if it is not above any negation ($\neg$). We require that every recursive function/predicate used in the formula φ has a unique definition in Def^{Δ}. Each recursive function is parameterized by a set of pointer fields $\overrightarrow{pf}$ and a set of program variables $\vec{v}$, denoted as $f^{\Delta}_{\overrightarrow{pf},\vec{v}}$. The subscripts are used in defining the semantics of recursive functions in Section 5.2.2. We usually simply use f^{Δ} when the subscripts are not relevant in the context. Similarly, recursive predicates are denoted as $p^{\Delta}_{\overrightarrow{pf},\vec{v}}$ or simply p^{Δ}. The recursive functions are defined using the syntax:

$$(\varphi^f_1(x,\vec{v},\vec{s}) : t^f_1(x,\vec{s}) \; ; \; \ldots \; ; \; \varphi^f_k(x,\vec{v},\vec{s}) : t^f_k(x,\vec{s}) \; ; \; \text{default} : t^f_{k+1}(x,\vec{s}))$$

where $\varphi^f_u(x,\vec{v},\vec{s})/t^f_u(x,\vec{s})$ is a formula/term in our logic with $\vec{s}$ implicitly existentially quantified. The recursively defined predicates are defined using the syntax: $\varphi^p(x,\vec{v},\vec{s})$, which is a formula in our logic with $\vec{s}$ implicitly existentially quantified. The recursive function syntax above expresses a case-split, with the function evaluating to the first term whose guard evaluates to true. The restrictions on the recursive definitions are:

- Subtraction, set-difference, and negation are disallowed;
- Every variable in $\vec{s}$ should appear in the right hand side of a points-to relation binding it to x *exactly once*.

For examples of recursive functions and predicates, see the definitions $keys^{\Delta}_{\overrightarrow{pf}}(x)$ and $mheap^{\Delta}_{\overrightarrow{pf}}(x)$ in Figure 5.1, respectively. The set of program variables $\vec{v}$ parameterizing the definitions is empty in both these definitions and the set of implicitly existentially-quantified variables $\vec{s}$ is $\{k, l, r\}$.

5.2.2 Semantics

Our logic is interpreted on models that are *program states*:

Definition 19. *A program state is a tuple $C = (R, s, h)$ where*

- *$R \subseteq Loc \setminus \{nil\}$ is a finite set of locations;*
- *$s : Vars \rightarrow Int \cup Loc$ is a store mapping program variables to locations or integers (of appropriate type);*
- *$h : R \times (PF \cup DF) \rightarrow Int \cup Loc$ is a heaplet mapping non-nil locations and each pointer-field/data-field to values of the appropriate type.* □

Note that the set of locations is, in general, larger than the state R and hence R defines a subset of heap locations. The store maps variables to locations (not necessarily in R), but the heaplet h gives interpretations for pointer and data-fields only for elements in R.

Given a heaplet h, for every pointer field pf, we denote the projection of h on $R \times (PF \setminus \{pf\} \cup DF)$ as $h \nmid pf$; similarly, for every data-field df, we denote the projection of h on $R \times (PF \cup DF \setminus \{df\})$ as $h \nmid df$. Also, for every subset $S \subseteq R$, we denote the projection of h on $S \times (PF \cup DF)$ as $h \mid S$.

A term/formula with free variables F is interpreted by interpreting the free variables in F using the map s from variables to values. The semantics of Dryad is similar to that of classical Separation Logic (*SL*). In particular, a term/formula without recursive definitions is interpreted exactly in the same way in Dryad and *SL*. Hence we first give the semantics of the non-recursive part, followed by the semantics of recursive definitions.

Before defining the semantics of formulas, we define the *pure* property for terms/formulas. Intuitively, a term/formula is pure if it is independent of the heap. Syntactically, a term/formula is pure if it does not contain `emp`, $\longmapsto$ or any recursive definition. Note that in *SL* all terms are pure, but in Dryad, a term can be impure if it contains a recursive function f^{Δ}.

Semantics of terms

Each $\mathcal{T}$-term evaluates to either a normal value of type $\mathcal{T}$, or to `undef`, which is only used in interpreting recursive functions (will be explained later). As a special value, `undef` will be propagated throughout the formula: if a formula φ contains a sub-term that evaluates to `undef`, then φ will evaluate to `false` if it appears positively, and will evaluate to `true` otherwise. Intuitively, `undef` cannot help in making the formula true over a model.

The *Loc* terms are evaluated as follows:

$$\begin{array}{rcl} [\![x]\!]_C & = & s(x) \\ [\![\texttt{nil}]\!]_C & = & nil \end{array}$$

For any binary operator op, t op t' is evaluated as follows:

$$[\![t \text{ op } t']\!]_C = \begin{cases} [\![t]\!]_C \text{ op } [\![t']\!]_C & \text{if } t \text{ or } t' \text{ is pure} \\ [\![t]\!]_{C|R_1} \text{ op } [\![t']\!]_{C|R_2} & \text{else if there exist } R_1, R_2 \text{ such that} \\ & \quad R = R_1 \cup R_2, [\![t]\!]_{C|R_1} \neq \texttt{undef} \\ & \text{ and } [\![t']\!]_{C|R_2} \neq \texttt{undef} \\ \texttt{undef} & \text{otherwise} \end{cases}$$

where op is interpreted in the natural way.

For singletons, $\{it\}$ will evaluate to $\emptyset$ if it evaluates to $-\infty$ or ∞:

$$[\![\{it\}]\!]_C = \begin{cases} \texttt{undef} & \text{if } [\![it]\!]_C = \texttt{undef} \\ \emptyset & \text{if } [\![it]\!]_C = -\infty \text{ or } \infty \\ \{[\![it]\!]_C\} & \text{otherwise} \end{cases}$$

$\{it\}_m$ and $\{lt\}$ evaluate similarly.

Semantics of formulas

The formula `true` is always interpreted to be *true*:

$$(R, s, h) \models \texttt{true}$$

The formula `emp` asserts that the heap is empty:

$$(R, s, h) \models \texttt{emp} \quad \text{iff} \quad R = \emptyset$$

The formula $lt \overset{\vec{pf},\vec{df}}{\longmapsto} (\vec{lt}, \vec{it})$ asserts that the heap contains exactly one record consisting of fields $\overrightarrow{pf}$ and $\overrightarrow{df}$, at address lt, with values $\vec{lt}$ and $\vec{it}$, respectively. Formally, the semantics of this formula is given as:

$$\begin{aligned}(R,s,h) \models lt \overset{\vec{pf},\vec{df}}{\longmapsto} (\vec{lt},\vec{it}) \quad \text{iff} \quad R = \{[\![lt]\!]_{R,s,h}\} \ \text{and} \\ h([\![lt]\!]_{R,s,h}, pf_i) = [\![lt_i]\!]_{R,s,h} \ \ \text{for corresponding } pf_i \text{ and } lt_i, \\ h([\![lt]\!]_{R,s,h}, df_i) = [\![it_i]\!]_{R,s,h} \ \ \text{for corresponding } df_i \text{ and } it_i.\end{aligned}$$

Note that, as in separation logic, the above has a *strict semantics*— the heaplet must be a singleton set and cannot be a larger set.

For binary relations $t \sim t'$ between integers, sets, and multisets, including equality, the pure property plays an important role. Remember that in *SL* all terms are pure. To be consistent with *SL*, if both t and t' are pure, it is interpreted in the normal way. Otherwise, $t \sim t'$ is only defined on the minimum heaplet required by t and t', more concretely the union of the heaplet associated with t and t'.

$$\begin{aligned}(R,s,h) \models t \sim t' \qquad \text{iff} \qquad t \text{ or } t' \text{ is pure and } [\![t]\!]_C \sim [\![t']\!]_C \\ \text{or} \qquad t \text{ and } t' \text{ are impure and there exist } R_1, R_2 \text{ s.t. } R = R_1 \cup R_2 \\ \text{and } [\![t]\!]_{C|R_1} \neq \texttt{undef}, [\![t']\!]_{C|R_2} \neq \texttt{undef} \ \text{and } [\![t]\!]_{C|R_1} \sim [\![t']\!]_{C|R_2}\end{aligned}$$

where $\sim$ is interpreted in the natural way.

The semantics of the disjoint conjunction operator $*$ is defined as follows. The formula $\varphi_0 * \varphi_1$ asserts that the heap can be split into two disjoint parts in which φ_0 and φ_1 hold respectively:

$$\begin{aligned}(R,s,h) \models \varphi_0 * \varphi_1 \quad \text{iff} \quad \text{there exist } R_0, R_1 \text{ s.t. } R_0 \cap R_1 = \emptyset \text{ and} \\ R_0 \cup R_1 = R \ \text{and} \ (R_0, s, h \mid R_0) \models \varphi_0 \ \text{and} \ (R_1, s, h \mid R_1) \models \varphi_1\end{aligned}$$

Boolean combinations are defined in the standard way:

$$\begin{aligned}(R,s,h) \models \varphi_0 \wedge \varphi_1 \quad &\text{iff} \quad (R,s,h) \models \varphi_0 \ \text{and} \ (R,s,h) \models \varphi_1 \\ (R,s,h) \models \varphi_0 \vee \varphi_1 \quad &\text{iff} \quad (R,s,h) \models \varphi_0 \ \text{or} \ (R,s,h) \models \varphi_1 \\ (R,s,h) \models \neg\varphi \quad &\text{iff} \quad (R,s,h) \not\models \varphi\end{aligned}$$

Semantics of recursive definitions

The main semantical difference between DRYAD and *SL* is on recursive definitions. We would like to *deterministically delineate* the heap domain for any recursive definition, so that the heap domain required by any DRYAD formula can be *syntactically determined*. Given a recursive definition $rec^{\Delta}_{\vec{pf},\vec{v}}$, the subscripts $\vec{pf}$ and $\vec{v}$ play a role in delineating the heap domain. Intuitively, the heap domain for $rec^{\Delta}_{\vec{pf},\vec{v}}(l)$ is the set of locations reachable from l using pointer-fields in $\vec{pf}$, but *without* going through the locations $\vec{v}$. In other words, we want to take the set of locations that lie *in between* l and $\vec{v}$. Precisely, this set is determined by a location l and a program state (R, s, h). We denote it as $\mathrm{reachset}_{\vec{pf},\vec{v}}(l,(R,s,h))$. Formally it is the smallest set of locations L satisfying the following two conditions:

1. l is in the set L if l is not in $\vec{v}$ and $l \neq nil$;

2. for each c in L, with $c \in R$, and for each pointer pf, if $h(c, pf)$ is not in $\vec{v}$ and is not nil, then $h(c, pf)$ is also in L.

Note that even though the reach set is defined with respect to the edges in the heaplet, we can determine whether R includes all nodes reachable from l without going through $\vec{v}$ in the *global heap* by checking whether $R = \mathrm{reachset}_{\vec{pf},\vec{v}}(l,(R,s,h))$.

For each recursive definition $rec^{\Delta}_{\vec{pf},\vec{v}}$, we usually simply denote $\mathrm{reachset}_{\vec{pf},\vec{v}}$ as $\mathrm{reachset}^{rec}$, as the subscripts are implicitly known.

Now, given a program state $C = (R, s, h)$ and a recursive function/predicate rec^{Δ}, the semantics on a location l depends on whether the heap domain R is exactly the required reach set $\mathrm{reachset}^{rec}(l,(R,s,h))$. If this is not true, we simply interpret it as `undef` or `false`.

If the heap domain matches the reach set (i.e., $R = \mathrm{reachset}_{\vec{pf},\vec{v}}(l, (R, s, h))$), the semantics is defined in the natural way (using least fixed-points). The colon operator in the syntax of recursive function $f^{\Delta}_{\vec{pf},\vec{v}}$ translates into a nested if-then-else (ITE) operator. Formally,

$$[\![f^{\Delta}_{\vec{pf},\vec{v}}(l)]\!]_C = \mathrm{ITE}(\varphi^f_1, [\![t^f_1]\!]_{C|R_1},\ \mathrm{ITE}(\varphi^f_2, [\![t^f_2]\!]_{C|R_2}, \ldots [\![t^f_{k+1}]\!]_{C|R_{k+1}} \ldots))$$

where $R_1 \ldots R_{k+1} \subseteq R$ such that $[\![t^f_i]\!]_{C|R_i} \neq$ `undef`. In order to give least fixed-point semantics for recursive definitions in the logic, we extend the primitive data-types to lattice domains. *Bool* with the order `false` $\sqsubseteq$ `true` forms a complete lattice, and $\mathcal{S}(Loc)$ and $\mathcal{S}(Int)$ ordered by inclusion, with join as union and meet as intersection, form complete lattices. Integers and multisets are extended to lattices. Let $(Int_L, \leq)$ denote the complete lattice, where $Int_L = Int \cup \{-\infty, \infty\}$, and where the ordering is $\leq$, join is *max*, meet is *min*. Also, $\mathcal{MS}(Int)_L, \sqsubseteq$ denote the complete lattice constructed from $\mathcal{MS}(Int)$, where $\mathcal{MS}(Int)_L = \mathcal{MS}(Int) \cup \{\top\}$, and $\sqsubseteq$ extends the inclusion relation with $S \sqsubseteq \top$ for any $M \in \mathcal{MS}(Int)$. It is easy to see that $(Int_L, \sqsubseteq)$ and $(\mathcal{MS}(Int)_L, \sqsubseteq)$ are complete lattices.

Formally, let *Def* consists of definitions of integer functions *I*, set-of-locations functions *SL*, set-of-integers functions *SI*, multiset-of-integers functions *MSI* and predicates *P*. Since these definitions could rely on each other, we evaluate them altogether as a function vector

$$r^\Delta = (\overrightarrow{i^\Delta}, \overrightarrow{sl^\Delta}, \overrightarrow{si^\Delta}, \overrightarrow{msi^\Delta}, \overrightarrow{p^\Delta})$$

We take the cartesian product lattice of the individual lattices and take the least fixed-point of r^Δ to obtain the semantics for each definition. Let $\text{select}_{rec}(lfp(r^\Delta))$, for each recursive definition rec^Δ, denote the selection of the coordinate for rec^Δ in $lfp(r^\Delta)$.

Now we can formally define the semantics of recursive definitions. For any configuration C, the semantics of a recursive function f^Δ is defined as:

$$[\![f^\Delta(lt)]\!]_C = \begin{cases} \text{select}_f(lfp(r^\Delta))([\![lt]\!]_C) & \text{if } R = \text{reachset}^f([\![lt]\!]_C, C) \\ \texttt{undef} & \text{otherwise} \end{cases}$$

and the semantics of a recursive predicate p^Δ is defined as

$$[\![p^\Delta(lt)]\!]_C = \begin{cases} \text{select}_p(lfp(r^\Delta))([\![lt]\!]_C) & \text{if } R = \text{reachset}^p([\![lt]\!]_C, C) \\ \texttt{false} & \text{otherwise} \end{cases}$$

Remark: Note that we disallow negative operations (subtraction, set-difference and negation) in defining recursive definitions. This syntactical restriction guarantees that each iteration of r^Δ is monotonic. By Knaster-Tarski theorem, r^Δ admits a least fixed-point.

5.2.3 Examples

The Dryad logic was already used in Section 5.1 to define max-heaps. Note that the definition of a max-heap is precisely defined on the heaplet that includes the underlying tree nodes of the max-heap only, as the heaplet for a recursive definition is the set of all reachable nodes according to the two pointers.

To clarify the difference between Dryad and *SL*, consider now this recursive definition:

$$p^{\Delta}_{\{l,r\}}(x) \overset{def}{=} (x = \texttt{nil} \wedge \texttt{emp}) \;\vee\; \left[(x \overset{l,r}{\longmapsto} y, z) \;*\; \left(p^{\Delta}_{\{l,r\}}(y) \vee p^{\Delta}_{\{l,r\}}(z)\right)\right]$$

Now consider a global heap that has a tree rooted at x with pointer fields l and r. The above recursive formula, in separation logic, will be true on any heaplet that contains the nodes of a path in this tree from x to *nil*. However, in Dryad, we require that the heaplet must satisfy the heap constraints of the formula and also be the precise set of locations reachable from x using the pointer fields l and r. Consequently, if the tree pointed to by x has more than one path, the Dryad formula will be *false for any heaplet*.

The above example shows the advantage of Dryad; when heaplets are determined, we can avoid quantification. We have not found natural examples where an undetermined heaplet semantics helps in specifying properties of heaps.

Dryad can express structures beyond trees. The main restriction we do impose is that we allow only *unary* recursive definitions, as this allows us to find simpler natural proofs since there is only one way to unfold the definition across a footprint. However, Dryad can express structures like cyclic lists and doubly-linked lists.

A cyclic-list is captured as $(v \mapsto y) * lseg^{\Delta}_{next,v}(y)$. Here, v is a program variable which denotes the head of the cyclic-list and $lseg^{\Delta}_{next,v}(y)$ captures the list segment from y back to the head v, where the subscripts *next* and v indicate that the heaplet of the list segment is the locations that can be reached using the field *next*, but without going through v:

$$lseg^{\Delta}_{next,v}(y) \overset{def}{=} (y = v \wedge \texttt{emp}) \vee (\, (y \overset{next}{\mapsto} z) * lseg^{\Delta}_{next,v}(z)\,)$$

Another interesting example is a doubly-linked list. We define a doubly-linked list as the following unary predicate:

$$\begin{aligned} dll^{\Delta}_{next}(x) \;=\; & (x = \texttt{nil} \wedge \texttt{emp}) \;\vee\; (x \overset{next}{\mapsto} \texttt{nil}) \;\vee \\ & (\, x \overset{next}{\mapsto} y \;*\; ((y \overset{prev}{\mapsto} x * \texttt{true}) \;\wedge\; dll^{\Delta}_{next}(y))\,) \end{aligned}$$

The first two disjuncts in the definition cover the base case when x is *nil* or the location next to x is *nil*; otherwise, let y be the location next to x, then the *prev* pointer at y points to x and location y is recursively defined as a doubly-linked list.

5.3 Translation to a Logic over the Global Heap

We now show one of the main results of this chapter— a translation from Dryad logic to classical logic with recursive predicates and functions, but over the global heap. The formulation of separation logic primitives in the global heap allows us to express complex structural properties, like disjointness of heaplets and tree-ness, using recursive definitions over *sets of locations*, which are defined locally, and are amenable to unfolding across the footprint and hence amenable to natural proofs.

For example, consider the formula $mheap^{\Delta}(x) * mheap^{\Delta}(y)$, where $mheap^{\Delta}$ is defined in Section 5.1. Since the heaplets for $mheap^{\Delta}(x)$ and $mheap^{\Delta}(y)$ are precise, it can get translated to an equivalent formula with a *free* set variable G that denotes the global heap over which the formula is evaluated:

$$\begin{aligned} mheap(x) \wedge mheap(y) \wedge (reach^{mheap}(x) \cap reach^{mheap}(y) = \emptyset) \\ \wedge (reach^{mheap}(x) \cup reach^{mheap}(y) = G) \end{aligned}$$

where *mheap* and $reach^{mheap}$ are corresponding recursive definitions in classical logic, which will be defined later in this section. Note that we use italics and remove the $^{\Delta}$ superscript to show the difference from their counterpart in Dryad.

We assume the Dryad formula to be translated is in *disjunctive normal form*, i.e., $\vee$ operators should be above all $*$ and $\wedge$ operators. This is not a real restriction as one can always push the disjunction out. This normal form ensures that for all occurrences of the separation operator in a formula, there exists a unique way of splitting the heap so as to satisfy the $*$ separated sub-formulas. Also, it ensures that this unique heap-split can be determined syntactically from the structure of those sub-formulas.

In our translation, we model the heaplets associated with a formula or a term as a set of locations and all operations on these heaplets are modeled as set operations like set union, set intersections, etc. over set-of-location variables. For example the separating conjunction $P * Q$ is translated to the following set constraint: the intersection of the sets associated with the heaplets in the formulas P and Q is empty. Given a formula φ in Dryad and its associated heap domain modeled by a

Construct	Domain-exact	Scope
$var/const$	`false`	$\emptyset$
$\{t\}/\{t\}_m$	dom-ext(t)	scope(t)
t op t'	dom-ext(t) $\vee$ dom-ext(t')	scope(t) $\cup$ scope(t')
$f^{\Delta}(lt)$	`true`	reachset$^f(lt)$
`true/false`	`false`	$\emptyset$
`emp`	`true`	$\emptyset$
$lt \overset{\vec{pf},\vec{df}}{\longmapsto} (\vec{lt}, \vec{it})$	`true`	$\{lt\}$
$p^{\Delta}(lt)$	`true`	reachset$^p(lt)$
$t \sim t'$	dom-ext(t) $\vee$ dom-ext(t')	scope(t) $\cup$ scope(t')
$\varphi \wedge \varphi'$	dom-ext(φ) $\vee$ dom-ext(φ')	scope(φ) $\cup$ scope(φ')
$\varphi * \varphi'$	dom-ext(φ) $\wedge$ dom-ext(φ')	scope(φ) $\cup$ scope(φ')

Figure 5.3: Domain-exact property and Scope function. Both are defined only for terms and formulas without disjunction and negation. A formula is assumed in its disjunctive normal form.

set variable G, we define an inductive translation T into a classical logic formula $T(\varphi,\ G)$ in the quantifier-free theory of finite sets, integers and uninterpreted functions. The translated formula is not interpreted on a heaplet, but interpreted on a global heap (i.e., with the heap domain Loc).

The translation uses an auxiliary *domain-exact* property and an auxiliary *scope* function. The domain-exact property indicates whether a term evaluates to a well-defined value or a positive formula evaluates to true on a fixed heap domain or not. This is different from the property *pure*; a pure formula or term is not domain-exact but the reverse implication is not true, in general. For example, the formula $(lt \longmapsto it) * true$ is not domain-exact but is also not pure. The scope function maps a term to the minimum heap domain required to evaluate it to a normal value, and maps a positive formula to the minimum heap domain required to evaluate it to *true*. The domain-exact property and the scope function are defined inductively in Figure 5.3.

We describe the logic translation in detail in Figure 5.4. The ITE expression used in the translation is short for "if-then-else". It is just a conditional expression defined as follows: ITE(ϕ, t_1, t_2) evaluates to t_1 if ϕ is true, otherwise evaluates to t_2.

In general, our translation restricts an impure term/formula to be evaluated only on the syntactically determined heap domain according to the semantics of DRYAD. In particular, when evaluating a recursive formula or predicate p^{Δ}, we ensure that

$$
\begin{array}{rcl}
T(var \mid const,\ G) & \equiv & var \mid const \\
T(\{t\} \mid \{t\}_m,\ G) & \equiv & \{t\} \mid \{t\}_m \\
T(t \ \text{op}\ t',\ G) & \equiv & T(t,G)\ \text{op}\ T(t',G) \\
T(f^{\Delta}(lt),\ G) & \equiv & \text{ITE}\,(reach^{f}(lt) = G,\ f(lt),\ \texttt{undef}) \\
T(\texttt{true} \mid \texttt{false},\ G) & \equiv & \texttt{true} \mid \texttt{false} \\
T(\texttt{emp},\ G) & \equiv & G = \emptyset \\
T(lt \xmapsto{\vec{pf},\vec{df}} (\vec{lt}, \vec{it}),\ G) & \equiv & G = \{lt\} \wedge \bigwedge_{pf_i} pf_i(T(lt,G)) = T(lt_i,G) \\
 & & \qquad\qquad \wedge \bigwedge_{df_i} df_i(T(lt,G)) = T(it_i,G) \\
T(p^{\Delta}(lt),\ G) & \equiv & p(lt) \wedge G = reach^{p}(lt) \\
T(t \sim t',\ G) & \equiv & \begin{cases} t \sim t' & \text{if } t \sim t' \text{ is not domain-exact} \\ t \sim t' \wedge G = \text{scope}(t \sim t') & \text{otherwise} \end{cases} \\
T(\varphi \wedge \varphi',\ G) & \equiv & T(\varphi,G) \wedge T(\varphi',G) \\
T(\varphi \vee \varphi',\ G) & \equiv & T(\varphi,G) \vee T(\varphi',G) \\
T(\neg\varphi,\ G) & \equiv & \neg T(\varphi,G)
\end{array}
$$

$$
T(\varphi * \varphi',\ G) \equiv \begin{cases}
T(\varphi,\ \text{scope}(\varphi)) \ \wedge\ T(\varphi',\ \text{scope}(\varphi')) \\
\quad \wedge\ \text{scope}(\varphi) \cup \text{scope}(\varphi') = G \\
\quad \wedge\ \text{scope}(\varphi) \cap \text{scope}(\varphi') = \emptyset \\
\qquad\qquad\qquad \text{if both } \varphi \text{ and } \varphi' \text{ are domain-exact} \\
T(\varphi,\ \text{scope}(\varphi)) \ \wedge\ T(\varphi',\ G \setminus \text{scope}(\varphi)) \\
\quad \wedge\ \text{scope}(\varphi) \subseteq G \qquad \text{if only } \varphi \text{ is domain-exact} \\
T(\varphi',\ \text{scope}(\varphi')) \ \wedge\ T(\varphi,\ G \setminus \text{scope}(\varphi')) \\
\quad \wedge\ \text{scope}(\varphi') \subseteq G \qquad \text{if only } \varphi' \text{ is domain-exact} \\
T(\varphi,\ \text{scope}(\varphi)) \ \wedge\ T(\varphi',\ \text{scope}(\varphi')) \\
\quad \wedge\ \text{scope}(\varphi) \cup \text{scope}(\varphi') \subseteq G \\
\quad \wedge\ \text{scope}(\varphi) \cap \text{scope}(\varphi') = \emptyset \\
\qquad\qquad\qquad \text{if neither } \varphi \text{ nor } \varphi' \text{ is domain-exact}
\end{cases}
$$

Figure 5.4: Translating Dryad terms/formulas to classical logic

the heaplet is precisely the reach set $reach^p(lt)$. For a formula $\varphi * \varphi'$, translation to classical logic depends on whether the sub-formulas φ and φ' are domain-exact or not. If a sub-formula is domain-exact then it is evaluated on its scope. If it is not domain-exact, then it is evaluated on the rest of the heaplet.

Recursive definitions in DRYAD are also translated to recursive definitions in classical logic. Translating a recursive definition rec^Δ uses the corresponding definitions rec and $reach^{rec}$, both of which are defined recursively in classical logic. The set $reach^{rec}$ represents the domain of the required heaplet for evaluating rec^Δ, and the Δ-eliminated definition rec captures the value of rec^Δ when the heaplet is restricted to $reach^{rec}$. Formally, suppose rec^Δ is a recursive definition w.r.t. pointer fields $\vec{pf}$ and stopping locations $\vec{v}$, then $reach^{rec}$ is recursively defined as the least fixed-point of

$$reach^{rec}(x) \overset{def}{=} \mathrm{ITE}\Big(x = \mathtt{nil} \vee x \in \vec{v},\ \emptyset,\ \{x\} \cup \bigcup_{pf \in \vec{pf}} (reach^{rec}(pf(x))) \Big)$$

For each recursive predicate p^Δ defined as $p^\Delta(x) \overset{def}{=} \varphi^p(x, \vec{v}, \vec{s})$, we define

$$p(x) \overset{def}{=} T(\ \varphi^p(x,\ \vec{v}, \vec{s}),\ reach^p(x)\ )$$

Similarly, for each recursive function f^Δ defined as

$$f^\Delta(x) \overset{def}{=} (\varphi_1^f(x, \vec{v}, \vec{s}) : t_1^f(x, \vec{s})\ ;\ \ldots\ \varphi_k^f(x, \vec{v}, \vec{s}) : t_k^f(x, \vec{s})\ ;\ \text{default} : t_{k+1}^f(x, \vec{s}))$$

we define

$$\begin{aligned} f(x) \overset{def}{=} \mathrm{ITE}\Big(& T(\varphi_1^f(x, \vec{v}, \vec{s}), reach^f(x)),\ \ t_1^{f-\Delta}(x, \vec{s}) \\ & \mathrm{ITE}\Big(T(\varphi_2^f(x, \vec{v}, \vec{s}), reach^f(x)),\ \ t_2^{f-\Delta}(x, \vec{s}) \\ & \qquad \ldots, t_{k+1}^{f-\Delta}(x, \vec{s})\ \ \ldots\Big)\Big) \end{aligned}$$

where $t_i^{f-\Delta}(x, \vec{s})$ is just the classical logic counterpart of $t_i^f(x, \vec{s})$, when interpreted in a heap domain within $reach^f(x)$. Formally it is short for

$$\mathrm{ITE}\Big(\mathrm{scope}(t_i^f(x, \vec{s})) \subseteq reach^f(x),\ T(t_i^f(x, \vec{s}),\ \mathrm{scope}(t_i^f(x, \vec{s}))),\ \mathtt{undef}\Big)$$

Now for each set of recursive definitions Def^Δ in DRYAD, we can translate it to a set of recursive definitions Def in classical logic.

$$
\begin{array}{lcl}
P & :- & P\,;P \ \mid \ stmt \\
stmt & :- & u := v \mid u := \mathtt{nil} \mid u := v.pf \mid u.pf := v \\
 & & \mid \ j := u.df \ \mid \ u.df := j \ \mid \ j := aexpr \\
 & & \mid \ u := \mathtt{new} \ \mid \ \mathtt{free}\ u \ \mid \ \mathtt{assume}\ bexpr \\
 & & \mid \ u := f(\vec{v}, \vec{z}) \ \mid \ j := g(\vec{v}, \vec{z}) \\
aexpr & :- & int \ \mid \ j \ \mid \ aexpr + aexpr \ \mid \ aexpr - aexpr \\
bexpr & :- & u = v \mid u = \mathtt{nil} \mid aexpr \leq aexpr \\
 & & \mid \ \neg bexpr \mid bexpr \vee bexpr
\end{array}
$$

Figure 5.5: Syntax of programs

Theorem 6. *Let φ be a* Dryad *formula w.r.t. a set of recursive definitions Def^{Δ}. For every program state C with heap domain Loc, and for every interpretation of variables $\mathcal{I}$ including a valuation for set-variable G, $(C,\ \mathcal{I}) \models T(\varphi,\ G)$ w.r.t. Def if and only if $(C\ |_G,\ \mathcal{I} \setminus \{G\}) \models \varphi$ w.r.t. Def^{Δ}.* □

5.4 Natural Proofs for Dryad

In this section we show how Dryad can be used in reasoning about the correctness of imperative heap-manipulating programs, in terms of verifying Hoare-triples where the pre- and post-conditions are expressed in Dryad. We first introduce a simple programming language and the corresponding Hoare-triples, generate a classical logic formula as the verification condition, utilizing in part the translation defined in Section 5.3. Then we present the natural proof framework which consists of two steps. In the first step, we utilize the idea of unfolding across the footprint to strengthen the verification condition (VC). In the second step, we prove the validity of the VC soundly using the technique of formula abstraction.

5.4.1 Programs and Hoare-triples

We consider straight-line program segments that do destructive pointer-updates, data-updates and procedure calls. Parameterized by a set of pointer fields PF and a set of data-fields DF, the syntax of the programs is defined in Figure 5.5, where $pf \in PF$, $f \in DF$, u and v are program variables of type location, j and z are program variables of type integer, int is an integer constant. To simplify the presentation, we assume all program variables are local and are either pre-assigned or assigned once in the program.

We allow two kinds of recursive procedures, one returning a location $f(\vec{v}, \vec{z})$ and one returning an integer $g(\vec{v}, \vec{z})$. Each procedure/program is annotated with its pre- and post-conditions in DRYAD. The pre-condition is denoted as a formula $\psi_{pre}(\vec{v}, \vec{z}, \vec{c})$, where $\vec{v}$ and $\vec{z}$ are variables as the formal parameters/program variables, $\vec{c}$ is a set of implicitly existentially quantified complimentary variables (e.g., variable K in the pre-condition φ_{pre} in Figure 5.1). The post-condition is denoted as a formula $\psi_{post}(ret, \vec{v}, \vec{z}, \vec{c})$, where ret is the variable representing the returned value, of corresponding type, $\vec{v}$ and $\vec{z}$ are program variables, $\vec{c}$ is a set of complimentary variables that have appeared in the pre-condition ψ_{pre}.

Given a straight-line program with its pre- and post-conditions $\{\psi_{pre}\}\ P\ \{\psi_{post}\}$, we define its partial correctness without considering memory errors[3]: P is partially correct iff for every normal execution (memory-error free) of P, which transits state C to state C', if $C \models \psi_{pre}$, then $C' \models \psi_{post}$.

Given a Hoare-triple $\{\psi_{pre}\}\ P\ \{\psi_{post}\}$ as defined above, a set of recursive definitions and a set of annotated procedure declarations are presented here. Assume that P consists of n statements, then consider a normal execution $\mathcal{E}$, which can be represented as a sequence of program states $(C_0, \ldots, C_n)$, where each $C_i = (R_i, s_i, h_i)$ represents the program state after executing the first i statements. The verification condition is just a formula interpreted on a state sequence $(C_0, \ldots, C_n)$. Let $pf_i : Loc \to Loc$ be the function mapping every location l to its pf pointer, i.e., $pf_i(l) = h_i(l, pf)$ for every location l. Similarly, $df_i : Loc \to Int$ is defined such that $df_i(l) = h_i(l, df)$ for every l. Recall that every program variable is either pre-assigned or assigned once in the program, each s_i is an expansion of the previous one, and s_n is the store with all program variables defined. Hence we simply use v to denote $s_n(v)$. Moreover, every recursive predicate/function is also indexed by i. For example, p_i is the recursive predicate such that $p_i(l)$ is true iff $C_i \models T(p^{\Delta}(l), \text{reachset}^{p}(l))$. Now for every formula φ and every index i, we can give the index i to all the pointer fields, data fields and recursive definitions. We denote the indexed formula as $\varphi[i]$.

We algorithmically derive the verification condition ψ_{VC} corresponding to it in classical logic with recursive definitions on the global heap.

[3]We exclude memory errors in order to simplify the presentation. Memory errors can be handled using a similar VC generation for assertions that negate the conditions for memory errors to occur.

5.4.2 Unfolding Across the Footprint

The verification condition obtained above is a quantifier-free formula involving recursive definitions and the reachable sets of the form $reach^p(x)$, which are also defined recursively. While these recursive definitions can be unfolded *ad infinitum*, we exploit a proof tactic called *unfolding across the footprint*. Intuitively, the footprint is the set of locations explored by the program *explicitly* (not including procedure calls). More precisely, a location is in the footprint if it is dereferenced explicitly in the program. The idea is to unfold the recursive definitions over the footprint of the program, so that recursive definitions on the footprint nodes are related, as precisely as possible, to the recursive definitions on frontier nodes. This will enable effective use of the formula abstraction mechanism, as when recursive definitions on frontier nodes are made uninterpreted, the unfolding formulas ensure tight conditions that the frontier nodes have to satisfy.

Furthermore, to enable effective frame reasoning, it is also necessary to perform verification condition strengthening with a set of instances of the frame rule. More concretely, we need to capture the fact that a recursive definition (or a field) on a location is unchanged during a segment or procedure call of the program, if the reachable locations (or only the location itself) are not affected by the segment or procedure call.

We incorporate the above facts formally into the verification condition. Let us introduce a macro function *fp* that identifies the location variables that are in (or aliased to something in) the footprint. The footprint of P, FP, is the set of dereferenced variables in P (we call a location variable dereferenced if it appears on the left-hand side of a dereferencing operator "." in P). Then $fp(u) \equiv \bigvee_{v \in FP}(u = v)$.

Now we state the unfoldings and framings using a formula UnfoldAndFrame. Assume there are m procedure calls in P, then P can be divided into $m + 1$ basic segments (subprograms without procedure calls): $S_0 \;;\; g_1 \;;\; S_1 \;;\; \ldots \;;\; g_m \;;\; S_m$ where S_d is the $(d + 1)$-th basic segment and g_d is the d-th procedure call. Then

$$\textsc{UnfoldAndFrame} \equiv \bigwedge_{rec} \bigwedge_{0 \le d \le m} \bigwedge_{u \in LVars \cup \{nil\}} \Bigg[$$
$$\bigg(\Big(fp(u) \vee u = nil \Big) \Rightarrow \Big(\textsc{Unfold}^{rec}_{d}(u) \wedge \textsc{FieldUnchanged}_{d}(u) \Big) \bigg) \wedge$$
$$\bigg(\Big(\neg fp(u) \vee u = nil \Big) \Rightarrow \textsc{RecUnchanged}^{rec}_{d}(u) \bigg) \Bigg]$$

The formula enumerates every recursive definition *rec* and every index d, and for each location u that is either pointed to by a location variable or is *nil*, the formula checks if u is in the footprint, and then unfolds it or frames it accordingly. If u is in the footprint, then we unfold *rec* for the timestamps before and after S_d (represented by the formula $\textsc{Unfold}_d^{rec}(u)$); moreover, all fields of u are unchanged if it is not affected during calling g_d (represented by the formula $\textsc{FieldUnchanged}_d(u)$). If u is not in the footprint, i.e., in the frontier, then *rec* and its corresponding reach set $reach^{rec}$ are unchanged after executing S_d, if S_d does not modify any location in $reach^{rec}$; they are also unchanged if $reach^{rec}$ is not affected by calling g_d. These frame assertions are represented by the formula $\textsc{RecUnchanged}_d^{rec}(u)$.

Now we can strengthen the verification condition by incorporating the derived formula above:

$$\psi'_{\mathrm{VC}} \equiv \psi_{\mathrm{VC}} \wedge \textsc{UnfoldAndFrame}$$

Since the incorporated formula is implied by the verification condition, we can reduce the validity of ψ_{VC} to the validity of ψ'_{VC}.

Theorem 7. *Given a Hoare-triple $\{\psi_{pre}\}$ P $\{\psi_{post}\}$, its verification condition ψ_{VC} is valid if and only if ψ'_{VC} is valid.* □

5.4.3 Formula Abstraction

While checking the validity of the strengthened verification condition ψ'_{VC} is still undecidable, as we argued before, it is often sufficient to prove it by assuming that the recursive definitions are arbitrary, or uninterpreted. Moreover, the uninterpreted formula falls in the array property fragment [17], whose satisfiability is decidable and is supported by modern SMT solvers such as Z3 [26]. This tactic roughly corresponds to applying *unification* in proof systems.

To prove ψ'_{VC}, we first replace each recursive predicate rec_d with an uninterpreted predicate $\hat{rec}_d$, and replacing the corresponding reach-set function $reach_d^{rec}$ with an uninterpreted function $\widehat{reach}_d^{rec}$. Let the result formula be $\psi_{\mathrm{VC}}^{\mathrm{abs}}$. This conversion, called *formula abstraction*, is sound: if $\psi_{\mathrm{VC}}^{\mathrm{abs}}$ is valid, so is ψ'_{VC}. When a proof for $\psi_{\mathrm{VC}}^{\mathrm{abs}}$ is found, we call it a natural proof for ψ'_{VC} (and also for ψ_{VC}).

The formula abstraction step is the only step that introduces incompleteness in our framework, but helps us transform the verification condition to a decidable theory. Formula abstraction (combined with unfolding recursive definitions across the footprint) discovers *recursive proofs* where the recursion is structural recursion on the definitions of the data-structures. The use of these tactics comes from the observation that such programs often have such recursive proofs (see [101] also for use of formula abstractions).

Our goal now is to check the satisfiability of $\neg\psi_{\mathrm{VC}}^{\mathrm{abs}}$ in a decidable theory. The resulting formula can be expressed using the theory of maps (to model sets) and corresponding map operations to model set operations. Formulas of the kind $S_1 \leq S_2$, where S_1 and S_2 are sets/multi-sets of integers, are the only ones that introduce quantification, but they can be translated to formulas in the array property fragment, which is decidable [17]. We hence obtain a formula ψ^{APF} in the array property fragment combined with the theory of uninterpreted functions, maps, and arithmetic .

Theorem 8. *Given a Hoare-triple* $\{\psi_{pre}\}$ P $\{\psi_{post}\}$*, if the derived array formula* ψ^{APF} *is satisfiable, then the Hoare-triple is valid.* □

User-provided axioms:

While natural proofs are often effective in finding recursive proofs that unfold recursive definitions and do unification, they are not geared towards finding *relationships* between various recursive definitions themselves. We hence require the user to provide certain obvious relationships between the different recursive definitions as axioms. For example, $lseg(x, y) * list(y) \Rightarrow list(x)$ is such an axiom saying that a list segment concatenated with a list yields a list. Note that these axioms are *not* program-dependent, and hence are not program-specific tactics that the user provides. These axioms are necessary typically to relate partial data-structure properties (like list segments) to complete ones (like lists), especially in iterative programs (as opposed to recursive ones), and we can fix them for each class of data-structures. We also allow the use of the separating implication, $-*$, from separation logic while specifying these axioms. User-defined axioms are instantiated, using the natural proof philosophy, on precisely the footprint nodes uniformly, and get translated to quantifier-free formulas.

5.5 Experimental Evaluation

We have implemented a prototype of the natural proof methodology for DRYAD presented in this chapter. The prototype verifier takes as input a set of user-defined

Data-structure	Routine	Time (s) / Routine
Singly-Linked List	find_rec, insert_front, insert_back_rec, delete_all_rec, copy_rec, append_rec, reverse_iter	< 1s
Sorted List	find_rec, insert_rec, merge_rec, delete_all_rec, insert_sort_rec, reverse_iter, find_last_iter	< 1s
	insert_iter	1.4
	quick_sort_iter	64.8
Doubly-Linked List	insert_front, insert_back_rec, delete_all_rec, append_rec, mid_insert, mid_delete, meld	< 1s
Cyclic List	insert_front, insert_back_rec, delete_front, delete_back_rec	< 1s
Max-Heap	heapify_rec	8.8
BST	find_rec, find_iter, insert_rec, delete_rec, remove_root_rec	< 1s
	insert_iter	72.4
	find_leftmost_iter	4.7
	remove_root_iter	65.6
	delete_iter	225.2
Treap	find_rec, delete_rec	< 1s
	insert_rec	12.7
	remove_root_rec	9.5
AVL-Tree	balance, leftmost_rec	< 1s
	insert_rec	4.1
	delete_rec	13.9
RB-Tree	insert_rec	73.9
	insert_left_fix_rec	8.1
	insert_right_fix_rec	5.1
	delete_rec	12.1
	delete_left_fix_rec	7.6
	delete_right_fix_rec	5.5
	leftmost_rec	< 1s
Binomial Heap	find_min_rec	1.1
	merge_rec	152.7
Schorr-Waite (for trees)	marking_iter	< 1s
Tree Traversals	inorder_tree_to_list_rec	2.4
	inorder_tree_to_list_iter	42.7
	preorder_rec, postorder_rec	< 1s
	inorder_rec	3.76

Table 5.1: Results of verifying data-structure algorithms.

recursive definitions, a set of procedure declarations with contracts, and a set of straight-line programs (or *basic blocks*) annotated with a pre-condition and a post-condition specifying a set of partial correctness properties including structural, data and separation requirements. Both the contracts and pre-/post-conditions are written in DRYAD. For each basic block, the verifier automatically generates the abstracted formula ψ^{APF} as described in Section 5.4, and passes ψ^{APF} to Z3 [26], a state-of-the-art SMT solver, to check the satisfiability in the decidable theory of array property fragment. The front-end of our verifier is based on ANTLR and our tool is around 4000 lines of C# code. Using the verifier, we successfully proved the partial correctness of 59 routines over a large class of programs involving heap data structures like sorted lists, doubly-linked lists, cyclic lists and trees. Additionally, we pit our natural proofs methodology against real-world programs and successfully verified, in total, 47 routines from different projects including the list and queue implementations in the Glib open source library, the OpenBSD library, the Linux kernel and the memory regions and the page cache implementations from two different operating systems.

We conducted the experiments on a machine with a dual-core, 2.4GHz CPU and 6GB RAM. The first part of our experimental results is tabulated in Table 5.1. In general, for every routine, we checked the properties formalizing the complete verification of the routines— capturing the precise structure of the resulting heap-structure, the precise change to the data stored in the nodes and the precise heaplet modified by the respective routines.

For every routine, the suffix `rec` or `iter` indicates if the routine was implemented recursively or iteratively using while loops. The names for most of the routines are self-descriptive. Routines like `find`, `insert`, `delete`, `append`, etc. are the natural implementations of the corresponding data structure operations. The routine `delete_all` for singly-linked lists, sorted lists and doubly-linked lists recursively deletes all occurrences of a particular key in the input list. The max-heap routine `heapify` accepts an almost max-heap in which the heap property is violated only at the root, both of whose children are max-heaps, and recursively descends the tree to restore the max-heap property. The routine `remove_root` for binary search trees and treaps is an auxiliary routine which is called in `delete`. Similarly, the routines `leftmost` for AVL-trees and RB-trees and `delete_fix` and `insert_fix` for RB-trees are also auxiliary routines.

Schorr-Waite is a well-known graph marking algorithm which marks all the reachable nodes of the graph using very little additional space. The algorithm achieves this by manipulating the pointers in the graph such that the stack of nodes along the path from the root is encoded in the graph itself. The Schorr-Waite algorithm is used in garbage collectors and it is traditionally considered as a challenging problem for verification [46]. The routine `marking` is an implementation of Schorr-Waite for trees [58] and we check the property that the resulting output tree is well-marked.

The routines `inorder_tree_to_list` construct a list consisting of the keys of the input tree, which is traversed inorder. The iterative version of this algorithm achieves this by maintaining a worklist/stack of sub-trees which remain to be processed at any given time. The routines `inorder`, `preorder` and `postorder` number the nodes of an input tree according to the inorder, preorder and postorder traversal algorithm, respectively.

Table 5.2 shows the results of applying natural proofs to the verification of various other real world programs and libraries. Glib is the low-level C library that forms the basis of the GTK+ toolkit and the GNOME desktop environment, apart from other open source projects. Using our prototype verifier, we efficiently verified Glib implementation of various routines for manipulating singly-linked and doubly-linked lists. We also verified the queue library which forms part of the OpenBSD operating system.

ExpressOS is an operating-system/browser implementation which provides security guarantees to the user via formal verification [63]. The module `cachePage` maintains a cache of the recently used disc pages. The cache is implemented as a priority queue based on a sorted list. We prove that the methods `add_cachepage` and `lookup_prev` (both called whenever a disc page is accessed) maintain the sortedness property of the cache page.

In an OS kernel, a process address space consists of a set of intervals of linear addresses represented as a *memory region*. In the ExpressOS implementation, a memory region is implemented as a sorted doubly-linked list where each node of the list with a *start* and an *end* address represents an interval included in the address space. We also verified some key components of the Linux implementation of a memory region, present in the file *mmap.c*. In Linux, a memory region is represented as a red-black tree where each node, again, represents an address interval. We proved methods which find, remove and insert a *vma_struct* (*vma* is short for virtual memory address) into a memory region.

It also worth mentioning that in the process of experiments, we did make some unintentional mistakes, in writing both the basic blocks and the annotations.

Example	Routine	Time (s) / Routine
glib/gslist.c Singly Linked-List LOC: 1.1K	`free, prepend, concat, insert_before, remove_all, remove_link, delete_link, copy, reverse, nth, nth_data, find, position, index, last, length`	< 1s
	`append`	4.9
	`insert_at_pos`	11.4
	`remove`	3.1
	`insert_sorted_list`	16.6
	`merge_sorted_lists`	6.1
	`merge_sort`	3.0
glib/glist.c Doubly Linked-List LOC: 0.3K	`free, prepend, reverse, nth, nth_data, position, find, index, last, length`	< 1s
OpenBSD/queue.h Queue LOC: 0.1K	`simpleq_init, simpleq_remove_after`	< 1s
	`simpleq_insert_head`	1.6
	`simpleq_insert_tail`	3.6
	`simpleq_insert_after`	18.3
	`simpleq_remove_head`	2.1
ExpressOS/cachePage.c LOC: 0.1K	`lookup_prev`	2.4
	`add_cachepage`	6.4
ExpressOS/ memoryRegion.c LOC: 0.1K	`memory_region_init`	< 1s
	`create_user_space_region`	3.6
	`split_memory_region`	5.8
linux/mmap.c LOC: 0.1K	`find_vma, remove_vma, remove_vma_list`	< 1s
	`insert_vm_struct`	11.6

Table 5.2: Results of verifying open-source libraries.

For example, forgetting to free the deleted node, or using $\wedge$ instead of $*$ in the specification between two disjoint heaplets, were common mistakes. In these cases, Z3 provided counter-examples to the verification condition that captured the essence of the bugs, and turned out to be very helpful for us to debug the specification. These debugging hints are usually not available in other incomplete proof systems.

Our experiments show that the natural proof methodology set forth in this chapter is successful in efficiently proving full-functional correctness of a large variety of algorithms. Most of the VCs generated for the above examples were discharged by Z3 in a few seconds. To the best of our knowledge, this is the first automatic mechanism that can prove such a wide variety of algorithms correct, handling such complex properties of structure, data and separation.

Chapter 6

Related Work and Conclusion

The program verification literature is very rich, so we only discuss work close to ours.

The idea of regarding a program (fragment) as a specification transformer to analyze programs in a forwards-style goes back to Floyd in 1967 [36]. However, his rules are not concerned with structural configurations, are not meant to be operational, and introduce quantifiers.

We fully share the goal of the mechanical verification community to reduce the correctness of program verification to a trusted formal semantics of the target language [38, 69, 57, 47, 3], although our methods are different. Instead of a framework to ease the task of giving multiple semantics of the same language and proving systematic relationships between them, we advocate developing *only one* semantics, operational, and offering an underlying theory and framework with the necessary machinery to achieve the benefits of multiple semantics without the costs.

Dynamic logic [42] adds modal operators to FOL to embed program fragments within specifications. For example, $\psi \rightarrow [\mathtt{s}]\psi'$ means "after executing $\mathtt{s}$ in a state satisfying ψ, a state may be reached which satisfies ψ'". KeY [11] offers automatic verification for Java based on dynamic logic. Matching logic also combines programs and specifications for static properties, but dynamic properties are expressed in reachability logic which has a language-independent proof system that works with any operational semantics, while dynamic logic still requires language-specific proof rules (e.g., invariant rules).

Separation logic [72, 85] is an extension of Hoare logic. The major difference between separation and matching logic is that 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.

Shape analysis [95] allows to examine and verify properties of heap structures.

The ideas of shape analysis have also been combined with those of separation logic [29] to quickly infer invariants for programs operating on lists. They can likely be also combined with matching logic to infer patterns.

Using Hoare logic [45] to prove concurrent programs correct dates back to Owicki and Gries [74]. In the rely-guarantee method proposed by Jones [49] each thread relies on some properties being satisfied by the other threads, and in its turn, offers some guarantees on which the other threads can rely. O'Hearn [71] advances a Separation Hypothesis in the context of separation logic [85] to achieve compositionality: the state can be partitioned into separate portions for each process and relevant resources, respectively, satisfying certain invariants. More recent research focuses on improvements over both of the above methods and even combinations of them [33, 103, 84, 44].

The satisfaction of all-path-reachability rules can also be understood intuitively in the context of temporal logics. Matching logic formulae can be thought of as state formulae, and reachability rules as temporal formulae. Assuming CTL* on finite traces, the semantics rule $\varphi \Rightarrow^{\exists} \varphi'$ can be expressed as $\varphi \Rightarrow E \bigcirc \varphi'$, while an all-path reachability rule $\varphi \Rightarrow^{\forall} \varphi'$ can be expressed as $\varphi \Rightarrow A\Diamond\varphi'$. However, unlike in CTL*, the φ and φ' formulae of reachability rules $\varphi \Rightarrow^{\exists} \varphi'$ or $\varphi \Rightarrow^{\forall} \varphi'$ share their free variables. Thus, existing proof systems for temporal logics (e.g., the CTL* one by Pnueli and Kesten [79]) are not directly comparable with our approach, which considers only a specific type of temporal properties, as mentioned above.

Leroy and Grall [56] use the coinductive interpretation of a standard big-step semantics as a semantic foundation both for terminating and for non-terminating evaluation. Our approach is different in that: (1) our proof system can reason about reachability between arbitrary formulae, rather than just the evaluation of programs to values, and (2) although Circularity has a coinductive flavor, we take the inductive interpretation of our proof system (rather than the coinduction interpretation), and obtain soundness by appropriate guarding of the circular rules.

A popular approach to building program verifiers for real-world languages is to translate to an IVL and do verification at the IVL level. This results in some re-usability, as the VC generation and reasoning about state properties are implemented only once, at the IVL level. However, the development of translators is both time consuming and susceptible to bugs. Boogie [8] is a popular IVL integrated with Z3. There are several verifiers built on top of Boogie, including VCC [22] and HAVOC [52] for C, Spec# [9] for C #, and Dafny [54] and Chalice [55] for academic languages. VCDrayd [77] is a separation logic based verifier built on top of VCC. Why3 [35] is another IVL, also integrated with SMT solvers (and other provers). Tools built on top of Why3 include Frama-C [35] and Krakatoa [34], both using the Jessie plug-in of Why3. There are many other practical VC generation based tools (with or without an IVL), including Verifast [48] (an automatic verifier for programs written in both C and Java based on separation logic) and jStar [28]. In contrast, we use existing operational semantics directly for verification, without any translation to IVLs or language-specific VC generation.

Recent work proposes translating to a set of Horn clauses instead of an IVL, and performing the verification at the level of the Horn clauses [41]. A semantics based-approach to translation to Horn clauses for a fragment of C is presented in [24], but it is unclear if the approach is generic enough to scale to the entire C or to other real-world languages.

An approach for using the interpreter source code as a model of the language in for symbolic execution is proposed in [18], but it is used to generate tests, not verify programs.

Bedrock [19] is a Coq framework which uses computational higher-order separation logic and supports mostly-automated proofs about low-level programs. Bedrock requires the user to provide hints for lemma applications. Specifications use operators defined in a pure functional language, similarly to the operators defined algebraically in matching logic. It can serve as an IVL, and be the target of translations from other languages which can be certified in Coq based on their operational semantics. Our approach works with the operational semantics directly, and thus does not need any such proofs.

Bae et al [6], Rocha and Meseguer [86], and Rocha et al [87] use narrowing to perform symbolic reachability analysis in a transition system associated to a unconditional rewrite theory for the purposes of verification. There are two main differences between their work and ours. First, they express state predicates in equational theories. Matching logic is more general, being first-order logic over a model of configurations T. Consequently, the Step proof rule takes these issues into account when considering the successors of a state. Second, they use rewrite systems for symbolic model checking. Our work is complementary, in the sense that we use the operational semantics for program verification, and check properties more similar to those in Hoare logic.

Equational algebraic specifications have been used to express pre- and post-conditions and then verify programs forwards using term rewriting [40]. Evolving specifications [76] 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 Jahob system [107, 108] is one of the first attempts at full functional verification of linked data structures, which integrates a variety of theorem provers, including SMT solvers, and makes the process mostly automated. However, complex specifications combining structure, data and separation usually require more complex provers such as Mona [50], or even interactive theorem provers such as Isabelle [70] in the worst case. The success of the proof search also relies on users' manual guidance.

The idea of unfolding recursive definitions and formula abstraction also features in the work by Suter et al. [101, 102], where a procedure for algebraic data-types is presented. However, this work focuses on soundness and completeness, and is not terminating for several complex data structures like red-black trees. Moreover, the work limits itself to functional program correctness; in our opinion, functional programs are very similar to algebraic inductive specifications, leading to much simpler proof procedures.

There is also a rich literature on completely automatic decision procedures for restricted heap logics, some of which combine structure-logic and arbitrary data-logic. These logics are usually FOLs with restricted quantifiers, and usually are decided using SMT solvers. The logics Lisbq [51] and CSL [15, 16] offer such reasoning with restricted reachability predicates and quantification; see also the logics in [13, 81, 82, 67, 83, 7]. Strand is a relatively expressive logic that can handle some data-structure properties (like BSTs) and admits decidable fragments [61, 60], but is again not expressive enough for more complex properties of inductive data-structures. None of these logics can express the class of VCs for full functional verification explored in this dissertation.

The language-independent verification approach presented in this dissertation comes with several limitations. First, the performance of symbolic execution depends on the granularity of the semantics. In our evaluation, the more granular semantics of JavaScript is twice as slow as that of Java. Second, specifications are reachability rules between program configurations, which can be verbose. The reachability rules could be generated from in-code annotations (pre/post conditions, loop invariants, class invariants, etc), the same way that Hoare triples can be generated from in-code annotations. We have done this for our C specific prototype, MatchC, but our generic infrastructure, KVI does not support this yet. Next, non-determinism is handled by exhaustive interleaving. This works for the non-deterministic evaluation of C expressions, but is currently infeasible for threads. Finally, heap abstractions currently need to be defined for each language separately, leading to boilerplate code.

To conclude, this dissertation introduces a language-independent verification infrastructure that takes as input an operational semantics and automatically turns it into a correct-by-construction program verifier. The framework is instantiated with the semantics of C, Java, and JavaScript. The generated verifiers successfully check the functional correctness of challenging programs that implement the same algorithms in all three languages.

Several future directions look interesting. First, all the programs we verified are single-threaded. We would like to extend our framework to support modular reasoning about multi-threaded programs. Second, we would be interested to apply our semantics based verifiers to larger pieces of code. Finally, we would like to add invariant inference capabilities to our framework, to reduce the user annotation burden.

References

[1] VCC: A verifier for concurrent C. `http://vcc.codeplex.com`. Accessed: July 8, 2016. 3, 147

[2] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *J. ACM*, 47(4):776–822, 2000. 138

[3] A. W. Appel. Verified software toolchain. In *ESOP*, volume 6602 of *LNCS*, pages 1–17, 2011. 2, 181

[4] K. R. Apt. Ten years of Hoare's logic: A survey - part 1. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, 1981. doi: 10.1145/357146.357150. URL `http://doi.acm.org/10.1145/357146.357150`. 49

[5] K. R. Apt. Ten years of Hoare's logic: A survey part II: nondeterminism. *Theor. Comput. Sci.*, 28:83–109, 1984. doi: 10.1016/0304-3975(83) 90066-X. URL `http://dx.doi.org/10.1016/0304-3975(83)90066-X`. 49

[6] K. Bae, S. Escobar, and J. Meseguer. Abstract logical model checking of infinite-state systems using narrowing. In *RTA*, pages 81–96, 2013. 183

[7] I. Balaban, A. Pnueli, and L. D. Zuck. Shape analysis of single-parent heaps. In *VMCAI'07*, volume 4349 of *LNCS*, pages 91–105. Springer, 2007. 184

[8] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects (FMCO'05)*, volume 4111 of *LNCS*, pages 364–387, 2006. 3, 50, 149, 182

[9] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: the Spec# experience. *Commun. ACM*, 54(6):81–91, 2011. 182

[10] C. Barrett and C. Tinelli. CVC3. In *CAV*, pages 298–302, 2007. 128

[11] B. Beckert, R. Hähnle, and P. H. Schmitt. *Verification of Object-oriented Software: The KeY Approach.* Springer-Verlag, 2007. 181

[12] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, 1992. 10, 12, 17, 29

[13] N. Bjørner and J. Hendrix. Linear functional fixed-points. In *CAV'09*, volume 5643 of *LNCS*, pages 124–139. Springer, 2009. 184

[14] D. Bogdănaș and G. Roșu. K-Java: A complete semantics of Java. In *POPL*, pages 445–456. ACM, 2015. 6, 142, 143, 148

[15] A. Bouajjani, C. Drăgoi, C. Enea, and M. Sighireanu. A logic-based framework for reasoning about composite data structures. In *CONCUR'09*, volume 5710 of *LNCS*, pages 178–195. Springer, 2009. 184

[16] A. Bouajjani, C. Drăgoi, C. Enea, and M. Sighireanu. On inter-procedural analysis of programs with lists and data. In *PLDI'11*, pages 578–589. ACM, 2011. 184

[17] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *VMCAI'06*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006. 150, 174, 175

[18] S. Bucur, J. Kinder, and G. Candea. Prototyping symbolic execution engines for interpreted languages. In *ASPLOS*, pages 239–254. ACM, 2014. 183

[19] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *PLDI*, pages 234–245, 2011. 150, 183

[20] E. M. Clarke. Programming language constructs for which it is impossible to obtain good hoare axiom systems. *J. ACM*, 26(1):129–147, 1979. doi: 10.1145/322108.322121. URL `http://doi.acm.org/10.1145/322108.322121`. 80

[21] M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude*, volume 4350 of *LNCS*. Springer, 2007. 20, 128

[22] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*, volume 5674 of *LNCS*, pages 23–42, 2009. 2, 49, 149, 182

[23] S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM J. Comput.*, 7(1):70–90, 1978. 78, 79

[24] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Semantics-based generation of verification conditions by program specialization. In *PPDP*, pages 91–102. ACM, 2015. 183

[25] L. M. de Moura and N. Bjørner. Efficient E-matching for SMT solvers. In *CADE*, volume 4603 of *LNCS*, pages 183–198, 2007. 140

[26] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340, 2008. 6, 128, 138, 140, 174, 177

[27] L. M. de Moura and N. Bjørner. Generalized, efficient array decision procedures. In *FMCAD*, pages 45–52. IEEE, 2009. 141

[28] D. Distefano and M. J. Parkinson. jStar: Towards practical verification for Java. In *OOPSLA*, pages 213–226. ACM, 2008. 2, 183

[29] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, pages 287–302, 2006. 182

[30] C. Ellison and G. Roșu. An executable formal semantics of C with applications. In *POPL*, pages 533–544. ACM, 2012. 6, 10, 17, 19, 29, 138, 142, 148

[31] M. Felleisen and D. P. Friedman. Control operators, the SECD-machine, and the lambda-calculus. In *3rd Working Conference on the Formal Description of Programming Concepts*, pages 193–219. IFIP, Aug. 1986. 12

[32] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT, 2009. 10, 29

[33] X. Feng. Local rely-guarantee reasoning. In *POPL*, pages 315–327. ACM, 2009. 182

[34] J. Filliâtre and C. Marché. The why/krakatoa/caduceus platform for deductive program verification. In *CAV*, volume 4590 of *LNCS*, pages 173–177, 2007. 183

[35] J. Filliâtre and A. Paskevich. Why3 - where programs meet provers. In *ESOP*, volume 7792 of *LNCS*, pages 125–128, 2013. 2, 3, 183

[36] R. W. Floyd. Assigning meaning to programs. In *Symposium on Applied Mathematics*, volume 19, pages 19–32. A.M.S., 1967. 1, 181

[37] P. Gardner, S. Maffeis, and G. D. Smith. Towards a program logic for javascript. In *POPL*, pages 31–44. ACM, 2012. doi: 10.1145/2103656.2103663. URL `http://doi.acm.org/10.1145/2103656.2103663`. 2

[38] C. George, A. E. Haxthausen, S. Hughes, R. Milne, S. Prehn, and J. S. Pedersen. *The RAISE Development Method*. Prentice Hall, 1995. 181

[39] J. Giesl and T. Arts. Verification of Erlang processes by dependency pairs. *AAECC*, 12(1/2):39–72, 2001. 41

[40] J. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs*. MIT Press, 1996. 183

[41] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416. ACM, 2012. 183

[42] D. Harel, D. Kozen, and J. Tiuryn. Dynamic logic. In *Handbook of Philosophical Logic*, pages 497–604, 1984. 181

[43] C. Hathhorn, C. Ellison, and G. Roșu. Defining the undefinedness of C. In *PLDI*, pages 336–345. ACM, 2015. 6, 142, 148

[44] J. Hayman. Granularity and concurrent separation logic. In *CONCUR*, volume 6901 of *LNCS*, pages 219–234, 2011. 182

[45] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. 1, 182

[46] T. Hubert and C. Marché. A case study of C source code verification: the Schorr-Waite algorithm. In *SEFM*, pages 190–199, 2005. 131, 178

[47] B. Jacobs. Weakest pre-condition reasoning for Java programs with JML annotations. *J. Logic and Algebraic Programming*, 58(1-2):61–88, 2004. 2, 181

[48] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and Java. In *Proceedings of the Third International Conference on NASA Formal Methods (NFM'11)*, volume 6617 of *LNCS*, pages 41–55, 2011. 149, 150, 183

[49] C. B. Jones. Specification and design of (parallel) programs. In R. E. A. Mason, editor, *Information Processing 1983: World Congress Proceedings*, pages 321–332. Elsevier, 1984. ISBN 0444867295. 182

[50] N. Klarlund and A. Møller. *MONA*. BRICS, Department of Computer Science, Aarhus University, January 2001. Available from `http://www.brics.dk/mona/`. 184

[51] S. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In *POPL'08*, pages 171–182. ACM, 2008. 184

[52] S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *POPL*, pages 115–126, 2006. 182

[53] D. Leinenbach and T. Santen. Verifying the microsoft hyper-v hypervisor with VCC. In *FM*, volume 5850 of *LNCS*, pages 806–809, 2009. 4

[54] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, pages 348–370, 2010. 182

[55] K. R. M. Leino, P. Müller, and J. Smans. Deadlock-free channels and locks. In *ESOP*, volume 6012 of *LNCS*, pages 407–426, 2010. 182

[56] X. Leroy and H. Grall. Coinductive big-step operational semantics. *Inf. & Computation*, 207(2):284–304, 2009. 182

[57] H. Liu and J. S. Moore. Java program verification via a JVM deep embedding in ACL2. In *TPHOLs*, volume 3223 of *LNCS*, pages 184–200, 2004. 181

[58] A. Loginov, T. W. Reps, and M. Sagiv. Automated verification of the Deutsch-Schorr-Waite tree-traversal algorithm. In *SAS*, 2006. 131, 178

[59] S. Lucas, C. Marché, and J. Meseguer. Operational termination of conditional term rewriting systems. *Information Processing Letters*, 95(4): 446–453, 2005. 41

[60] P. Madhusudan and X. Qiu. Efficient decision procedures for heaps using STRAND. In *SAS'11*, volume 6887 of *LNCS*, pages 43–59. Springer, 2011. 150, 184

[61] P. Madhusudan, G. Parlato, and X. Qiu. Decidable logics combining heap structures and data. In *POPL'11*, pages 611–622. ACM, 2011. 150, 184

[62] P. Madhusudan, X. Qiu, and A. Ştefănescu. Recursive proofs for inductive tree data-structures. In *POPL*, pages 123–136. ACM, 2012. 149

[63] H. Mai, E. Pek, H. Xue, S. T. King, and P. Madhusudan. Verifying security invariants in ExpressOS. In *ASPLOS'13*, pages 293–304. ACM, 2013. 149, 154, 178

[64] J. Matthews and R. B. Findler. An operational semantics for Scheme. *JFP*, 18(1):47–86, 2008. 29

[65] A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In *PLDI*, pages 221–231, 2001. 144

[66] P. D. Mosses. *CASL Reference Manual*. Springer, 2004. 20

[67] G. Nelson. Verifying reachability invariants of linked structures. In *POPL'83*, pages 38–47. ACM, 1983. 184

[68] H. H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated verification of shape and size properties via separation logic. In *VMCAI*, volume 4349 of *LNCS*, pages 251–266, 2007. 139, 144

[69] T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998. 2, 181

[70] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. 184

[71] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, 2007. 182

[72] P. W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, 1999. 181

[73] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, volume 2142 of *LNCS*, pages 1–19, 2001. 62, 150, 155

[74] S. S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976. 182

[75] D. Park, A. Ștefănescu, and G. Roșu. KJS: A complete formal semantics of JavaScript. In *PLDI*, pages 346–356. ACM, 2015. 6, 119, 142, 143, 148

[76] D. Pavlovic and D. R. Smith. Composition and refinement of behavioral specifications. In *Automated Software Engineering*, pages 157–165, 2001. 183

[77] E. Pek, X. Qiu, and P. Madhusudan. Natural proofs for data structure manipulation in C using separation logic. In *PLDI*, pages 440–451. ACM, 2014. 144, 146, 182

[78] J. A. N. Pérez and A. Rybalchenko. Separation logic + superposition calculus = heap theorem prover. In *PLDI*, pages 556–566. ACM, 2011. 140

[79] A. Pnueli and Y. Kesten. A deductive proof system for CTL^*. In *CONCUR*, volume 2421 of *LNCS*, pages 24–40, 2002. 182

[80] X. Qiu, P. Garg, A. Ştefănescu, and P. Madhusudan. Natural proofs for structure, data, and separation. In *PLDI*, pages 231–242. ACM, 2013. 149

[81] Z. Rakamarić, J. D. Bingham, and A. J. Hu. An inference-rule-based decision procedure for verification of heap-manipulating programs with mutable data and cyclic data structures. In *VMCAI'07*, volume 4349 of *LNCS*, pages 106–121. Springer, 2007. 184

[82] Z. Rakamarić, R. Bruttomesso, A. J. Hu, and A. Cimatti. Verifying heap-manipulating programs in an SMT framework. In *ATVA'07*, volume 4762 of *LNCS*, pages 237–252. Springer, 2007. 184

[83] S. Ranise and C. Zarba. A theory of singly-linked lists and its extensible decision procedure. In *SEFM'06*, pages 206–215. IEEE-CS, 2006. 184

[84] U. S. Reddy and J. C. Reynolds. Syntactic control of interference for separation logic. In *POPL*, pages 323–336. ACM, 2012. 182

[85] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE, 2002. 2, 62, 150, 155, 181, 182

[86] C. Rocha and J. Meseguer. Proving safety properties of rewrite theories. In *CALCO*, volume 6859 of *LNCS*, pages 314–328, 2011. 183

[87] C. Rocha, J. Meseguer, and C. A. Muñoz. Rewriting modulo smt and open system analysis. In *WRLA*, LNCS, 2014. to appear. 183

[88] G. Roşu, A. Ştefănescu, S. Ciobâcă, and B. M. Moore. One-path reachability logic. In *LICS*, pages 358–367. IEEE, 2013. 28

[89] G. Roșu and T.-F. Șerbănuță. An overview of the K semantic framework. *J. Logic and Algebraic Programming*, 79(6):397–434, 2010. 5, 10, 29, 127, 136

[90] G. Roșu and A. Ștefănescu. Matching logic: a new program verification approach (NIER track). In *ICSE*, pages 868–871, 2011. 28, 119

[91] G. Roșu and A. Ștefănescu. From Hoare logic to matching logic reachability. In *FM*, volume 7436 of *LNCS*, pages 387–402. Springer, 2012. 28

[92] G. Roșu and A. Ștefănescu. Towards a unified theory of operational and axiomatic semantics. In *ICALP*, volume 7392 of *LNCS*, pages 351–363, 2012. 28

[93] G. Roșu and A. Ștefănescu. Checking reachability using matching logic. In *OOPSLA*, pages 555–574. ACM, 2012. 28, 119

[94] G. Roșu, C. Ellison, and W. Schulte. Matching logic: An alternative to Hoare/Floyd logic. In *AMAST*, volume 6486 of *LNCS*, pages 142–162, 2010. 5, 19

[95] 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. 181

[96] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Commun. ACM*, 10(8):501–506, 1967. 131

[97] T.-F. Șerbănuță, G. Roșu, and J. Meseguer. A rewriting logic approach to operational semantics. *Information and Computation*, 207(2):305–340, 2009. 12

[98] A. Ștefănescu. MatchC: A matching logic reachability verifier using the K framework. In *K Workshop*, 2011. Appeared in volume 304 of ENTCS, pages 183-198, 2014. 119

[99] A. Ștefănescu, S. Ciobâcă, R. Mereuță, B. M. Moore, T. F. Șerbănuță, and G. Roșu. All-path reachability logic. In *RTA*, volume 8560 of *LNCS*, pages 425–440, July 2014. 28, 119

[100] A. Ștefănescu, D. Park, S. Yuwen, Y. Li, and G. Roșu. Semantics-based program verifiers for all languages. In *OOPSLA*. ACM, November 2015. To appear. 119

[101] P. Suter, M. Dotta, and V. Kuncak. Decision procedures for algebraic data types with abstractions. In *POPL'10*, pages 199–210. ACM, 2010. 175, 184

[102] P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *SAS'11*, volume 6887 of *LNCS*, pages 298–315. Springer, 2011. 184

[103] V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, volume 4703 of *LNCS*, pages 256–271, 2007. 182

[104] G. Winskel. *The formal semantics of programming languages - an introduction*. Foundation of computing series. MIT Press, 1993. 102

[105] A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994. 12

[106] J. Yang and C. Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. In *PLDI'10*, pages 99–110. ACM, 2010. 149

[107] K. Zee, V. Kuncak, and M. C. Rinard. Full functional verification of linked data structures. In *PLDI'08*, pages 349–361. ACM, 2008. 184

[108] K. Zee, V. Kuncak, and M. C. Rinard. An integrated proof language for imperative programs. In *PLDI'09*, pages 338–351. ACM, 2009. 184