PhD thesis

Progress in Programming Language Semantics Frameworks

Jan Tušil

Brno, January 2025

Masaryk University
Faculty of Informatics

Supervisor
Jan Obdržálek

Abstract

One can hardly imagine large-scale production of computer software without the use of high-level programming languages. These shield software engineers from the overwhelming amount of detail present in modern computer architectures: instruction sets, caches, out-of-order execution, and so on. But when these details are abstracted, what *does* remain of a programming language?

When the *remains* – a programming language's semantics – are precisely captured, important questions such as "Is this compiler correct?" or "Can my program go wrong?" become meaningful. Furthermore, with a *common notion* of language semantics shared between languages, the development of programming language tools becomes easier and more automated. For example, it is possible to write an interpreter or a deductive verifier such that it is parametric in the semantics of the target language, and then obtain the interpreter or verifier almost for free for any language expressed within the framework.

This thesis presents three contributions to the advancement of programming language semantics frameworks. The first contribution is Cartesian Reachability Logic (CRL) – a program logic for reasoning about a class of hyperproperties of programs written in any deterministic language with formal semantics of a certain (operational) kind. CRL extends previous work on language-parametric program logics, as well as on program logics for hyperproperties. The second contribution is a natural deduction proof system for matching logic – a logic used to justify operations of an existing, real-world programming language framework ($\mathbb{K}$-framework). This proof system is accompanied by a practical theorem-proving environment ("proof mode") embedded in the Coq proof assistant. The third contribution is the design, development, and formal verification of Minuska – a new, practical programming language framework capable of deriving interpreters from operational semantics. Interpreters generated by Minuska perform well and provide stronger guarantees than interpreters generated by $\mathbb{K}$-framework. Moreover, the design of Minuska avoids some soundness issues present in $\mathbb{K}$.

Thanks

I am thankful to my family – father Miloslav, mother Jana, sister Barbora, and the extended family – for their love and support. Beside them, I need to mention:

- my advisor Jan Obdržálek;
- my co-authors Traian, Peter, and Daniel;
- current and past members of the Formela FI MU laboratory, especially: Adam F., Adam Š., Adéla Š., Anička Ř., David D., David K., David S., Jindra S., Kuba Š., Martin J., Honza S., Paulína A., Terka K., Terka S., Vojta K., and Vítek J.;
- other personal friends, including Adam B., Filip M., Helena Z., Kuba B., Laco V., and Nishant R.

I also want to thank Ambrus Kaposi for his daring questions, Achim Blumensath for his comments on a draft of this thesis, Ada Nazarejová for her eye on practical matters, and Grigore Roşu for his vision and trust.

Contents

I Introduction ... 7
I.1 Author's publications ... 9
II On semantics engineering ... 11
II.1 Pens, papers, proof assistants, semantics frameworks ... 11
II.2 Reachability logics: partial correctness for all ... 13
II.2.1 Hoare logic ... 14
II.2.2 Operational semantics ... 15
II.2.3 Reachability logic ... 16
II.2.4 Implementations ... 32
II.2.5 Related work ... 34
II.3 Equivalence verification ... 34
II.4 Certification of a semantic framework's tasks ... 35
II.4.1 Overview ... 35
II.4.2 Strengths and weaknesses ... 36
III Beyond safety with CRL ... 41
III.1 Hyperproperties ... 41
III.2 Cartesian Hoare Logic ... 42
III.3 Cartesian reachability logic ... 44
III.3.1 CRL extends RL ... 46
III.3.2 A language-independent alternative to self-composition ... 47
III.3.3 Proof system for CRL ... 71
III.3.4 An example CRL proof ... 73
III.3.5 CRL soundness ... 77
III.3.6 CRL completeness ... 85
III.4 Concluding remarks ... 88
III.4.1 Nondeterminism ... 88
III.4.2 Similarities to CHL ... 96
III.4.3 Differences from CHL ... 96
III.4.4 Future work ... 96
IV An attempt to connect $\mathbb{K}$ to Coq ... 97
IV.1 Natural deduction without deduction theorem ... 99
IV.2 Details on applicative unsorted matching logic ... 105
IV.2.1 Semantics ... 106
IV.2.2 Proof system ... 108
IV.2.3 Equality ... 109
IV.3 A natural deduction system for matching logic ... 109
IV.3.1 Proof constraints ... 110
IV.3.2 Sequents ... 110
IV.4 Coq implementation of the Proof Mode ... 113
IV.4.1 Matching logic proof system in Coq ... 113
IV.4.2 MLPM in Action ... 114
IV.4.3 Proof constraints ... 115
IV.4.4 Architecture of MLPM ... 117
IV.5 Evaluation of MLPM ... 118

IV.6 Concluding remarks 121
V Minuska: A formal formal semantics framework 123
V.1 On semantics frameworks 124
V.2 Architecture of Minuska 124
V.2.1 MinusS 126
V.2.2 MinusLang 129
V.2.3 Sugared syntax for MinusLang 129
V.2.4 Algebra of primitive values 130
V.2.5 Interpreter 131
V.2.6 Trust base 134
V.3 Evaluation 134
V.3.1 Performance of Minuska vs $\mathbb{K}$ 136
V.3.2 IMP semantics in Minuska 137
V.3.3 Other benchmarks 139
V.4 Concluding remarks 140
VI Notes on future research directions 143
VI.1 On language definitions 143
VI.2 On separation logic 145
VII Final remarks 147
Bibliography 149

I Introduction

This thesis is all about *programming language semantics frameworks*. Such frameworks enable researchers to express the semantics of various programming languages *in a unified way*. But why would anyone want to do that?

Consider, for example, a software engineer writing a program in some programming language. If the software does not work as intended, the engineer may ask: "Is my program buggy, or is the compiler miscompiling my program?". This question implicitly assumes there is *something* that the compiler *should* do: that there is some *contract* between the engineer and the compiler. However, compilers are complex and often target multiple hardware architectures, each with its peculiarities: memory management, various forms of concurrency, instruction sets, cache invalidation, etc. Luckily, our engineer does not need to consider all these details every time when thinking about her program: the contract between her and the compiler is given in more abstract terms. The "something the compiler should do" is simply to preserve the meaning of the given program. That has a lot to do with *programming language semantics*.

Even if one wants to deal with programming language semantics, why do it in a unified way? With the unification that semantics frameworks bring into the game, one can manipulate language semantics as data with well-defined meaning. Engineers can then write software tools that analyze, query, and transform the semantics or generate other software artifacts. An incarnation of the same idea is being applied to syntactical analyzers, parsers, and grammars at least since the times of Algol60; applied to whole programming languages with their semantics, it enables the construction of concrete interpreters, symbolic execution engines, and program verifiers – parametric in a programming language. These language-parametric tools become useful when programming languages evolve or when one considers multiple programming languages and can enable faster iterations of the development of languages and their tools.

This thesis is meant to advance the world of such semantics frameworks. It is structured as follows.

- Chapter II of this thesis presents an overview of programming language semantics frameworks.
- Chapter III features Cartesian reachability logic (CRL) – a new program logic to enable formal verification of programs with respect to a larger class of properties than was supported before. In particular, CRL allows reasoning about a class of hyperproperties of programs written in any deterministic language with formal semantics of a certain (operational) kind. One can think of CRL as a conceptual merge of two extensions of Hoare logic – one extension that was previously done for that class of hyperproperties and another extension enabling language-parametric program verification.
- Chapter IV focuses on a natural deduction proof system for matching logic – a logic used to justify operations of $\mathbb{K}$-framework, a real-world semantics

framework. The proof system is accompanied by a practical theorem-proving environment ("proof mode") embedded in the Coq proof assistant.

- Chapter V describes the design, development, and formal verification of Minuska – a new, practical programming language framework capable of deriving interpreters from operational semantics. Interpreters generated by Minuska perform well and provide stronger guarantees than interpreters generated by $\mathbb{K}$. Moreover, the design of Minuska avoids some soundness issues present in $\mathbb{K}$-framework.
- Chapter VI sketches a few directions for future work.
- Chapter VII concludes.

I.1 Author's publications

Chapter III, Chapter IV and Chapter V are based on papers that were published in international conferences. The following table summarizes my share of the contributions with respect to each publication.

Chapter	Publication	My contribution
Chapter III	J. Tušil, T. Șerbănuță, and J. Obdržálek [98]	80%: I came up with the initial idea, designed the logic and the proof system, proved most of the theorems, and wrote the majority (>= 50%) of the text.
Chapter IV	J. Tušil, P. Bereczky, and D. Horpácsi [96]	70%: I came up with the idea of proof mode for matching logic, designed and implemented most of the Coq-based proof mode (except for first-order reasoning) and wrote the majority (>= 50%) of the text.
Chapter V	J. Tušil and J. Obdržálek [97]	90%: I came up with the idea, designed and implemented the whole framework, proved all the necessary theorems (in the Coq proof assistant), co-designed and executed the experiments, and prepared the artifact for evaluation; I also wrote the majority (>= 50%) of the text.

The CORE rank of the conferences at the time of acceptance was: A for [98] (LPAR'23), B for [96] (ICTAC'23), and B for [97] (SEFM'24).

II On semantics engineering

Computer systems are ubiquitous, and their failures often have severe consequences. To rigorously reason about software that controls computer systems, one needs to assign some mathematical meaning to it; the earliest works towards this goal were pioneered in the 1960s by R. W. Floyd [46] and C. A. R. Hoare [51]. Since then, various approaches to what is known as *formal semantics of programming languages* have emerged. In the 2000s, researchers started to develop computer programs that made working with programming language semantics easier and practically useful: M. Hills, T. Şerbănuţă, and G. Roşu [50] created $\mathbb{K}$, P. Sewell *et al.* [88] developed OTT, and M. Felleisen, R. B. Findler, and M. Flatt [42] proposed PLT-Redex.

This chapter is a guided tour of the related theoretical and practical developments until 2024. It may not be comprehensive or unbiased; however, I hope for it to be useful to those who want to familiarize themselves with the area of tool-supported semantics engineering and programming language semantics frameworks. The chapter starts with an introduction to the concept of a programming language framework, then discusses some logics that can be used in such a framework, and ends with some approaches to establishing the trustworthiness of a language framework.

II.1 Pens, papers, proof assistants, semantics frameworks

The first research on programming languages and their semantics in the 1960′s by R. W. Floyd [46] and C. A. R. Hoare [51], as well as that on compiler correctness by J. McCarthy and J. Painter [66] , was done in the traditional pen-and-paper style. In the early 1970s, Robin Milner developed a computer program for checking proofs – LCF (Logic for Computable Functions) – which was soon after used by R. Milner and R. Weyhrauch [69] to mechanize McCarthy's compiler correctness proof, which included mechanizing the semantics of the source and target language. More recently, in the 2000s, X. Leroy [61] implemented and formally verified in the Coq proof assistant [95] a compiler of a realistic subset of the C programming language. That work included formalizing a simplified version of the C language [13], the target assembly language [62], and some intermediate languages used during various stages in the compiler pipeline. However, despite this progress in applications of computer-checked theorem proving, some research in programming languages continues to be done using pen and paper only.

Pen-and-paper and mechanized theorem proving are not the only approaches to language semantics: one can also consider mechanization of the target language in specialized *semantics frameworks*. Using these, one can eliminate certain drawbacks of pen-and-paper while achieving a better effort/benefit ratio than with generic theorem provers. In the context of design of research languages, C. Klein *et al.* [56] argue that

- research done using the traditional pen-and-paper approach comes with flaws;
- these flaws can be eliminated by using a theorem prover, but

Side note

A separate line of related research was done in the context of development of practical programming languages and their toolings. For example, [58] opens with the following paragraph:

> Over the last decade, several research projects have focused on the automatic generation of programming environments given a formal specification of a desired language (for instance, [...]). A programming environment is here understood as a coherent set of interactive tools such as syntax-directed editors, debuggers, interpreters, code generators, and pretty printers to be used during the construction of texts in the desired language.
>
> — P. Klint [58]

Most of the research in this line was focused on parsing, editing, and static semantics (such as name resolution and type checking); only few frameworks allowed working with a language's semantics. A notable exception is the PSG system [7], in which one could define a denotational semantics[1] for a programming language using a lazy untyped lambda calculus extended with various builti-in types (like integers, strings, etc).

That line of research eventually lead to the development of modern environments such as Xtext [12] and MPS [99] for construction and use of domain specific languages; these are often called "language workbenches" and are applied mainly in the area of model-driven engineering. While there have been some recent attempts to integrate this user-facing research with the more formal work on language semantics and theorem proving [17], [18], until the present days (2024), the two research areas remain rather separated.

- theorem proving requires additional details and is thus rather heavy-weight; and
- lightweight mechanization of a programming language's semantics in a semantics framework (PLT-Redex) already pays off for a semantics engineer.

In other words, semantics frameworks can improve the effort/benefit ratio by lowering the numerator – the effort. The ratio could also be improved by enlarging the denominator, i.e., the benefit gained by using a semantics framework. The authors of another framework – $\mathbb{K}$ – state:

> It is hard to define a complete formal semantics of any real-world programming language, but if the objective is to tolerate no imprecision or ambiguity in the language, there is no way around it. But we can do it once and for all. Nothing else should be needed besides the canonical formal semantics, from

[1] Denotational semantics lie outside the scope of this thesis; an interested reader is referred to [49] for an introduction.

> which all implementations, tools, and documentations are auto-generated by the framework.
>
> — X. Chen and G. Roşu [23]

Therefore, in programming language semantics, tools like PLT-Redex and 𝕂 can be seen as providing – for a reasonable cost – benefits comparable with, or even exceeding, those provided by theorem provers.

My only objection to that perception is that it encompasses a *false dilemma* of choice between semantic frameworks and theorem provers. Theorem provers have some advantages – such as rigorous reasoning and provable correctness – over semantics frameworks like PLT-Redex and 𝕂, and the above perception raises the question: Can we not have the advantages of both semantics frameworks and theorem provers? We *can* have both, and Chapter V discusses some work in that direction.

II.2 Reachability logics: partial correctness for all

There are two approaches to semantics of programming language that were traditionally very far from each other: operational semantics and axiomatic semantics. In operational semantics, the meaning of a program is explained "in terms of a hypothetical computer which performs the set of actions that constitute the elaboration of that program" (A. van Wijngaarden *et al.* [100]). Defining operational semantics is often like writing an interpreter in the language of mathematics. In contrast, axiomatic semantics describe how program fragments change the properties that hold in program states.

Traditionally, program verifiers that aim to be provably sound (such as VST of A. W. Appel [3]) define two models of the target language: one operational, which can be relatively easily validated, and one axiomatic, which is then used for formal verification. Of course, the two models need to be equivalent; since proving this is much work, it is not surprising that many program verifiers (such as Frama-C [55]) do not employ any explicitly represented model of the target language and, therefore, do not give any formal guarantees on the behavior of verified software. This unfortunate situation motivated the work of G. Roşu and A. Stefanescu [80] on unifying operational and axiomatic approaches to formal semantics.

Some concrete outcome of that line of research was a collection of logics, today commonly referred to as *reachability logics* [81], [82], [92], as well as a language-independent program verification tool implemented as a part of 𝕂 by A. Stefănescu, D. Park, S. Yuwen, Y. Li, and G. Roşu [93]. This section serves as an introduction to reachability logics: I discuss the well-known basics of Hoare's logic [51] in Section II.2.1; discuss operational semantics in Section II.2.2; present reachability logic in its basic form [81] in Section II.2.3; and touch some extensions of reachability logic [82], [92] afterwards.

Side note

It is worth noting that some properties of a program cannot be described in the way Hoare suggests: examples include *temporal properties* and *hyperproperties*. Temporal properties play no part in this thesis; I discuss hyperproperties in Section III.1.

II.2.1 Hoare logic

One of the first works in program verification was done by C. A. R. Hoare [51]. In his paper, the author considers a simple imperative programming language and describes the behaviors of its programs using a *deductive system* today known as "Hoare logic". This deductive system has a dual role: it

1. serves as an exact specification of what programs should do and
2. provides means of proving a program's *functional properties*.

Regarding point 1, Hoare later wrote:

> The purpose of a formal language definition is to act as a "contract" between the implementor and user of the language. It is not intended to be read straight through, but rather to be used by both parties as a work of reference; it will also be used as the basis of more readable descriptions of the language.
>
> — C. A. R. Hoare [52]

Today, these deductive approaches to language definitions are known as "axiomatic semantics". Among the other approaches that exist, *operational semantics* are of importance for this thesis; these are briefly described in Section II.2.2.

About point 2, Hoare observes:

> One of the most important properties of a program is whether or not it carries out its intended function. The intended function of a program, or part of program, can be specified by making general assertions about the values which the relevant variables will take *after* execution of the program. These assertions will usually not ascribe practical values to each variable, but will rather specify certain general properties of the values and the relationships holding between them.
>
> — C. A. R. Hoare [51]

Thus, to specify and reason about functional-correctness properties of programs in the way Hoare suggests, a notion of *assertion* is needed; the assertions Hoare mentions above are commonly referred to as "postconditions".

The central concept of [51] is that of a (Hoare) *triple*. A Hoare triple has shape $\{P\}S\{Q\}$[2] where S is a program statement, P an assertion called the "precondi-

[2]The syntax commonly used today slightly differs from the original one.

$$\frac{}{\{\, P\left[\frac{e}{x}\right] \,\}\ \texttt{x := e;}\ \ \{\, P \,\}}\ \text{Assign}$$

$$\frac{\{\, P \,\}\, S \,\{\, Q \,\} \quad \vdash Q \to Q'}{\{\, P \,\}\, S \,\{\, Q' \,\}}\ \text{Impl1}$$

$$\frac{\vdash P \to P' \quad \{\, P' \,\}\, S \,\{\, Q \,\}}{\{\, P \,\}\, S \,\{\, Q \,\}}\ \text{Impl2}$$

$$\frac{\{\, P \,\}\, S_1 \,\{\, Q \,\} \quad \{\, Q \,\}\, S_2 \,\{\, R \,\}}{\{\, P \,\}\, S_1\ S_2 \,\{\, R \,\}}\ \text{Seq}$$

$$\frac{\{\, B \wedge P \,\}\, \text{S} \,\{\, P \,\}}{\{\, P \,\}\ \text{while B do S}\ \{\, \neg B \wedge P \,\}}\ \text{While}$$

Figure 3: Rules of Hoare's logic from [51].

tion", and Q an assertion called the "postcondition". The meaning of such a triple is the proposition (or *claim*)

> If the assertion P is true before initiation of the program S, then Q is true after its completetion (*if* it completes).

The actual deductive system ("Hoare logic") consists of axioms and inference rules for deriving valid claims of such form using the rules in Figure 3. Note that in [51], such derived claims are valid *by definition* – that is, Hoare considers the language to be defined by the deductive system. In later works, authors have considered the adequacy of deductive systems with respect to other definitions of a programming language. For example, [53] proves the soundness of a Hoare logic with respect to the operational semantics of the same language.

II.2.2 Operational semantics

One approach to operational semantics was developed by Gordon Plotkin as a part of his 1981 lecture notes, published only later in 2004 as [75]. In this approach, known as "structural operational semantics" (SOS) or "small-step semantics", operational semantics "was intended as being like an abstract machine but without all the complex machinery in the configurations," [76] which was to be avoided thanks to the use of rules. For example, when computing a value of a nested expression, an abstract-machine approach would employ an evaluation stack for remembering the context of the current subexpression, while the SOS approach would use a rule like

$$\frac{A \Rightarrow A'}{A + B \Rightarrow A' + B}$$

for the same purpose but without an explicit stack. This way, a *transition relation* consisting of relatively *small computational steps* is specified.

Another approach to operational semantics, commonly refered to as *big step semantics* today, was presented in 1987 by G. Kahn [54]. Also known as *natural semantics*, these consist of a deductive system that relates (in one big step) a program with its input to the desired result of computation. When the relation is deterministic, one can think of the deductive system as defining a single function; as T.-F. Șerbănuță, G. Roşu, and J. Meseguer [104] put it: "big-step semantics is "the most denotational" of the operational semantics". A typical big-step rule might look like:

$$\frac{\sigma \vdash B \Rightarrow \mathit{true} \qquad \sigma \vdash E_1 \Rightarrow v}{\sigma \vdash \texttt{if}\ B\ \texttt{then}\ E_1\ \texttt{else}\ E_2 \Rightarrow v}$$

The small-step approach can be extended with the notion of a *reduction context*, as shown in M. Felleisen and R. Hieb [43]. This style became popular C. Klein, J. A. McCarthy, S. Jaconette, and R. B. Findler [57], and allows concise specifications. [57] shows an example of a simple arithmetic language whose terms and contexts are specified by the grammar

$$\begin{array}{rcll} a & ::= & n & \textit{numbers} \\ & | & (\mathit{plus}\ \ a\ a) & \\ C & ::= & [] & \\ & | & (\mathit{plus}\ \ C\ a) & \\ & | & (\mathit{plus}\ \ a\ C) & \end{array}$$

and whose semantics is defined using a single reduction rule:

$$C[(\mathit{plus}\ \ n_1\ n_2)] \Rightarrow C[(n_1 + n_2)]$$

(where the operator $+$ represents the meta-level numerical addition).

In another style of operational semantics, promoted by $\mathbb{K}$, one organizes the control structure of a program as a structure sometimes called continuation (in [104]) and sometimes computation ([83]); see also T.-F. Șerbănuță and G. Roşu [103]. In this setting, a term (say `(0*1)+(2*(3+4))`) is decomposed into a sequence consisting of a redex and single-layer contexts (such as `3+4 ~> 2*[] ~> (0*1)+[]`), with special (and unconditional) "heating" rules taking responsibility for decomposition and "cooling" rules for recomposition of terms. The advantage of this approach with respect to context reduction semantics is the relative simplicity, as it does not rely on parsing for decomposing the redex from the context; also, this approach reduces the recompose-then-decompose-again overhead. See O. Danvy and L. R. Nielsen [37] for a related discussion.

II.2.3 Reachability logic

It has been observed in by C. A. R. Hoare and P. E. Lauer [53] that operational semantics and axiomatic semantics are each suitable for different purposes and that it can be worthwhile to have two (or more) definitions of the same language

in different styles, with a proof of their adequacy, serving as a reference. On the complexity of the task of developing such a pair, the authors note:

> The other problem, the complexity of formal methods when applied to existing programming languages, seems also impossible to solve. We believe that the problem is due to the genuine logical and semantical irregularities of current languages; and that in the long term, it will benefit both users and implementors of programming languages if their languages were more regular in this respect; and finally that a good intuitive criterion of "regularity" is that the language is easy to describe formally in more than one way.
>
> — C. A. R. Hoare and P. E. Lauer [53]

However, more than 30 years later, authors of $\mathbb{K}$ argue

> Despite impressive mechanical theorem proving advances in recent years, language designers still perceive the operational and axiomatic semantics as two distinct endeavors, and proving their formal relationship as a burden. With few notable exceptions, real languages are rarely given both semantics.
>
> — G. Roşu and A. Stefanescu [80]

and present the first version of *reachability logic*[3] – a logic for reasoning about partial correctness (in the spirit of Hoare logic) that is parametric in the definition of a programming language. The original version was sound but was lacking a completeness proof; later that year, the same authors came up with a reformulation of reachability logic that was shown to be complete in [81].

Reachability logic consists of two parts: a *static* part for reasoning about static program configurations, and a *dynamic* part for reasoning about the execution of programs with the flow of time.

II.2.3.1 Static reasoning

In the approach of [81], formulas for reasoning about static program configurations are called "matching logic patterns". These are constructed over symbols from a many-sorted algebraic signature. The semantics of patterns is given with respect to algebras.

Shortcut

Matching logic patterns are simply first-order logic (FOL) formulas with equality over a purely algebraic signature (that is, a signature without predicate symbols), where in addition, terms with free variables can be used as nullary predicates. Given a matching logic pattern φ, an algebra $\boldsymbol{T}$, a valuation (mapping from variables to elements of $\boldsymbol{T}$) ρ, and an element (aka "configuration") $\gamma \in \boldsymbol{T}$, we write $(\gamma, \rho) \vDash^{\mathrm{ML}} \varphi$ to say that "φ matches

[3] The actual name used in [80] for the logic is *matching logic*; however, the term *matching logic* later became used for the *static* part of the logic only.

γ in ρ". The configuration γ is used for evaluating terms that are used as nullary predicates: a term t matches γ in ρ iff the (unique) homomorphic extension $\overline{\rho}$ of ρ evaluates t to γ. Otherwise, matching logic semantics works as FOL semantics, where additionally γ propagates through all the connectives unchanged. A reader satisfied with this intuition may skip the technical definitions and jump straight to Section II.2.3.2.

II.2.3.1.1 Algebras

Standard definitions of an algebraic signature, a term, and an algebra follow.

Definition 1: Algebraic signature

A *function arity* over a set S (of *sorts*) is a pair (r, as) consisting of a return sort r and a list $as = [a_1, ..., a_n]$ of argument sorts. We let $Arity(S)$ denote the class of all function arities over S.

An algebraic signature (F, ar) is a set F (of *function symbols*) together with their signatures $ar : F \to Arity(S)$. We let $\mathbf{Sgn}_{alg}(S)$ denote the class of all algebraic signatures over S.

Intuitively, an algebraic signature defines the kinds of data and our operations in an algebra. In the context of reachability logic, these operations operate inside a particular program configuration (an element of an algebra); algebraic operations do *not* make the program evolve. They are meant only to construct (and observe) program configurations.

Example 2: Algebraic signature of IMP

For a simple imperative programming language with arithmetic and boolean expressions and basic control-flow commands, one would want to use some algebra over identifiers, integers, boolean values, program terms, and program configurations (that contain program terms together with maps from variable identifiers to their values).

$$\begin{aligned} S_{IMP0} = \{&SortIdent, SortBool, SortInt, \\ &SortMap, SortProg, SortCfg\} \end{aligned}$$

The signature of such an algebra would have function symbols for constructing program terms

$$\begin{aligned} F_{IMP0_syntax} = &\{pbval_b\}_{\{b \in \{0,1\}\}} \cup \{pzval_z\}_{\{z \in \mathbb{Z}\}} \cup \\ &\{pplus, pminus, peq, plt, \\ &pand, pneg, pite, pwhile, passign, pseq\} \cup \\ &\{pident\} \end{aligned}$$

as well as function symbols for manipulating the integer and boolean values; there would also be symbols for constructing and querying the store (a finite map).

$$
\begin{aligned}
F_{IMP0} = F_{IMP0_syntax} \cup & \\
& \{ident_s \mid s \text{ is a string}\} \cup \\
& \{bval_b \mid \{b \in \{0,1\}\}\} \cup \{band, bneg\} \cup \\
& \{zval_z \mid \{z \in \mathbb{Z}\}\} \cup \{zplus, ztimes, zlt, zeq\} \cup \\
& \{sempty, sinsert, sunion, shas, sget\} \cup \\
& \{cfg\}
\end{aligned}
$$

This set of symbols deserves a few comments.

1. The names of the symbols are chosen to reflect their interpretation in the intended algebra. The symbols whose names start with the letter "b" manipulate boolean values, those starting with "z" integers, those starting with "p" construct program terms, those with "s" manipulate stores, "i" manipulate identifiers and cfg constructs configurations.

2. It often happens that one wants to design the programming language such that programs can manipulate these values, too. This is the case in IMP: for example, we want to be able to write IMP programs that add numbers. Therefore, the algebraic signature of IMP has similarly-looking symbols, such as $pplus$ and $zplus$, whose arity nevertheless differs: $pplus$ is there for constructing program terms (so that the language can syntactically express the operations), while $zplus$ is there for performing operations on integers inside the algebra (so that the language can give meaning to some program terms). In a typical implementation of a semantic framework (such as $\mathbb{K}$ or Minuska), some of the symbols (like $zplus$) are provided by the framework independently of the programming language, while other are user-provided specifically for the programming language in question.

3. The intent behind the symbols is visible from their sorts, as we shall see now.

$$
\begin{aligned}
ar_{IMP0} = \{&(ident_s, (SortIdent, [])) \mid s \text{ is a string}\} \cup \\
&\{(bval_b, (SortBool, [])) \mid b \in \{0,1\}\} \cup \\
&\{(band, (SortBool, [SortBool, SortBool]))\} \cup \\
&\{(bneg, (SortBool, [SortBool]))\} \cup \\
&\{(zval_z, (SortInt, [])) \mid z \in \mathbb{Z}\} \cup \\
&\{(zplus, (SortInt, [SortInt, SortInt]))\} \cup \\
&\{(ztimes, (SortInt, [SortInt, SortIntl]))\} \cup \\
&\{(zlt, (SortBool, [SortInt, SortIntl]))\} \cup \\
&\{(zeq, (SortBool, [SortInt, SortIntl]))\} \cup \\
&\{(pident, (SortProg, [SortIdent]))\} \cup \\
&\{(pbval_b, (SortProg, [])) \mid b \in \{0,1\}\} \cup \\
&\{(pzval_z, (SortProg, [])) \mid z \in \mathbb{Z}\} \cup \\
&\{(pplus, (SortProg, [SortProg, SortProg]))\} \cup \\
&\{(pminus, (SortProg, [SortProg, SortProg]))\} \cup \\
&\{(peq, (SortProg, [SortProg, SortProg]))\} \cup \\
&\{(plt, (SortProg, [SortProg, SortProg]))\} \cup \\
&\{(pand, (SortProg, [SortProg, SortProg]))\} \cup \\
&\{(pneg, (SortProg, [SortProg]))\} \cup \\
&\{(pwhile, (SortProg, [SortProg, SortProg]))\} \cup \\
&\{(pite, (SortProg, [SortProg, SortProg, SortProg]))\} \cup \\
&\{(passign, (SortProg, [SortProg, SortProg]))\} \cup \\
&\{(pseq, (SortProg, [SortProg, SortProg]))\} \cup \\
&\{(sempty, (SortMap, []))\} \cup \\
&\{(sunion, (SortMap, [SortMap, SortMap]))\} \cup \\
&\{(sinsert, (SortMap, [SortMap, SortIdent, SortInt]))\} \cup \\
&\{(shas, (SortBool, [SortMap, SortIdent]))\} \cup \\
&\{(sget, (SortInt, [SortMap, SortIdent]))\} \cup \\
&\{(cfg, (SortCfg, [SortProg, SortMap]))\}
\end{aligned}
$$

Then, we can say that the language IMP has signature (F_{IMP0}, ar_{IMP0}) (over sorts S_{IMP0}).

In reachability logic, program configurations of a particular programming language are modeled as an algebra.

Definition 3: Algebra

Let $\Omega = (F, ar)$ be an S-sorted algebraic signature. An Ω-algebra A is a pair $\left((T_s)_{s \in S}, (f_A)_{f \in F}\right)$ consisting of a non-empty carrier set T_s (for every sort s) and a function interpretation f_A (for every function symbol f), where $f_A : T_{s_1} \times \ldots \times T_{s_n} \to T_s$, where $ar(f) = (s, [s_1, \ldots, s_n])$.

Often, the algebra is a term algebra of syntactic program fragments, extended with operations manipulating the program state. To show the algebra of IMP, we first need to define *concrete (ground) terms*.

Definition 4: Ground terms

For any S-sorted algebraic signature $\Omega = (F, ar)$, the set $\mathbb{T}_{\Omega,s}$ (of *s-sorted ground Ω-terms of sort s*) is defined inductively by

$$t_s \;::=\; f\left(t_{s_1}, \ldots, t_{s_k}\right) \quad \textit{symbol application}$$

where

- $s_1, \ldots, s_k, s \in S$, and
- $ar(f) = (s, [s_1, \ldots, s_k])$.

Example 5: Algebra of IMP

Algebra $A_{IMP0} = \left((T_s)_{s \in S_{IMP0}}, \left(f_{A_{IMP0}}\right)_{f \in F_{IMP0}} \right)$ consists of

- the set $S_{SortIdent} = \{x, y, z, ...\}$ (of *identifiers*),
- the set $S_{SortBool} = \{0, 1\}$ (of *boolean values*),
- the set $S_{SortInt} = \mathbb{Z}$ (of *integers*),
- the set $S_{SortMap} = S_{SortIdent} \underset{\text{fin}}{\rightharpoonup} S_{SortInt}$ (of *finite maps of identifiers to integers*),
- the set $S_{SortProg} = \mathbb{T}_{(\{SortProg\}, F_{IMP0_syntax}),\ SortProg}$,
- the set $S_{SortCfg} = S_{SortProg} \times S_{SortMAp}$

as carrier sets, and $f_{A_{IMP0}}$ defined as suggested above; for example,

- $(bval_b)_{A_{IMP0}} = \{([], b)\}$ for any $b \in \{0, 1\}$;
- $(bneg)_{A_{IMP0}} = \{([0], 1), ([1], 0)\}$;
- $(pbval_b)_{A_{IMP0}} = \{([], pbval_b())\}$ for any $b \in \{0, 1\}$;
- $(pneg)_{A_{IMP0}} = \left\{([t], pneg(t)) \mid t \in S_{SortProg}\right\}$;
- $(pplus)_{A_{IMP0}} = \left\{([t_1, t_2], pplus(t_1, t_2)) \mid t_1, t_2 \in S_{SortProg} h\right\}$;
- $(sempty)_{A_{IMP0}} = \{([], \emptyset)\}$;
- $(sunion)_{A_{IMP0}} = \left\{([m_1, m_2], m_1 \cup m_2) \mid m_1, m_2 \in S_{SortMap}\right\}$;
- $(shas)_{A_{IMP0}} = \{([m, k], 1) \mid \exists v \in \mathbb{Z}.(k, v) \in m\} \cup ([m, k], 0) \mid \neg\exists v \in \mathbb{Z}.(k, v) \in m\}$;
- $(sget)_{A_{IMP0}} = \{([m, k], v) \mid (k, v) \in m\} \cup \{([m, k], 0) \mid (k, v) \notin m\}$

and so on.

Static reasoning in reachability logic, similarly to algebraic reasoning, employs symbolic terms (that is, terms over *variables*).

Definition 6: Variables

For a set S, let $\mathbb{V}(S)$ denote the class of S-indexed families $(V_s)_{s \in S}$ of sets (of *variables of sort* s).

Definition 7: Symbolic terms

For any S-sorted algebraic signature $\Omega = (F, ar)$ and a class $V \in \mathbb{V}(S)$ of variables, the set $T_{\Omega,s}(V)$ (of *s-sorted symbolic Ω-terms over V*) is defined inductively by

$$\begin{array}{rcll} t_s & ::= & x & \textit{variable} \\ & | & f\left(t_{s_1}, ..., t_{s_k}\right) & \textit{symbol application} \end{array}$$

where
- $s_1, ..., s_k, s \in S$,
- $x \in \mathbf{Var}(s)$, and
- $ar(f) = (s, [s_1, ..., s_k])$.

Context 8: Context: Variables

From now on until specified, for every finite set S, assume an arbitrary but fixed class $(\mathbf{Var}_s(S))_{s \in S} = \mathbf{Var}(S) \in \mathbb{V}(S)$ of countably infinite sets of variables. (This is without loss of generality, since we could always choose $\mathbf{Var}_s = \{s\} \times \mathbb{N}$.) Let us take the freedom to write simply $\mathbf{Var}$ to mean $\mathbf{Var}(S)$ if S is known from the context.

Symbolic terms are related to ground terms using *valuations*: a valuation maps variables to ground terms, and a valuation extension maps terms with variables to ground terms.

Definition 9: Valuation

Assume $\Omega = (F, ar)$ is an S-sorted algebraic signature, and $A = \left((T_s)_{s \in S}, (f_A)_{f \in F}\right)$ is an Ω-algebra . An *algebraic A-valuation* (or simply a *valuation*) is an S-indexed family $\rho = (\rho_s)_{s \in S}$ where $\rho_s : \mathbf{Var}_s \to T_{\Omega,s}$.

Definition 10: Term extension of valuation

Assume $\Omega = (F, ar)$ is an S-sorted algebraic signature, and $A = \left((T_s)_{s \in S}, (f_A)_{f \in F}\right)$ is an Ω-algebra . Given an A-valuation ρ, the *(term-)extended valuation* $\overline{\rho} = (\overline{\rho}_s)_{s \in S}$, where $\overline{\rho}_s : T_{\Omega,s}(\mathbf{Var}) \to T_{\Omega,s}$, is defined recursively by
- $\overline{\rho}_s(x) = \rho(x)$ for any $x \in \mathbf{Var}(s)$, and
- $\overline{\rho}_s(f(t_1, ..., t_n)) = f_A\left(\overline{\rho}_{s_1}(t_1), ..., \overline{\rho}_{s_n}(t_n)\right)$ for any $f \in F$ and $ar(f) = (s, (s_1, ..., s_n))$.

A valuation can be seen as a witness relating a ground term to a term with variables in a particular algebra. To express more complex relations between ground

terms and terms with variables, and thus between program configurations and their static properties, [81] use *matching logic*.

II.2.3.1.2 Matching logic

Syntactically, matching logic formulas are first-order logic formulas extended with the notion of *terms-as-predicates*: a symbolic term can be used in the position of a predicate.

Definition 11: Matching logic formulas (patterns)

Assume $\Omega = (F, ar)$ is an S-sorted algebraic signature. The set $\mathbf{Pattern}(\Omega, s)$ (of *matching logic formulas of sort* s, aka *matching logic patterns of sort* s) is defined inductively by

$$\begin{array}{rcll} \varphi & ::= & t_s & \textit{term} \\ & | & t_{s'} = t_{s'} & \textit{equality} \\ & | & \varphi \wedge \varphi & \textit{conjunction} \\ & | & \bot & \textit{false} \\ & | & \exists x : s'.\varphi & \textit{existential quantifier} \end{array}$$

Let us use some standard abbreviations:

- $\neg\varphi \stackrel{\text{def}}{=} \varphi \Rightarrow \bot$;
- $\top \stackrel{\text{def}}{=} \neg \bot$;
- $\varphi_1 \vee \varphi_2 \stackrel{\text{def}}{=} \neg((\neg\varphi_1) \wedge (\neg\varphi_2))$; and
- $\forall x : s.\varphi \stackrel{\text{def}}{=} \neg(\exists x : s.(\neg\varphi))$.

Let $\mathrm{VT}(t)$ denote the set of variables occurring in the term t, and $\mathrm{FV}(\varphi)$ denote the set of free variables occurring in the pattern φ.

Definition 12: Matching logic semantics

Assume $\Omega = (F, ar)$ is an S-sorted algebraic signature with a distinguished sort $SortCfg \in S$, and $A = \left((T_s)_{s \in S}, (f_A)_{f \in F}\right)$ is an Ω-algebra. Matching logic semantics is given by means of the ternary *satisfaction relation*

$$_, _ \vDash_A^{\mathrm{ML}} _$$

relating
1. an element of A having sort $SortCfg$,
2. an A-valuation, and
3. a matching logic pattern of sort $SortCfg$.

The relation is defined inductively:
- $\gamma, \rho \vDash_A^{\mathrm{ML}} t$ iff $\gamma = \overline{\rho}_s(t)$ for any symbolic term t ;
- $_, \rho \vDash_A^{\mathrm{ML}} t_1 = t_2$ iff $\overline{\rho}_s(t_1) = \overline{\rho}_s(t_2)$ for any symolic terms t_1, t_2 of a common sort s.
- $\gamma, \rho \vDash_A^{\mathrm{ML}} \varphi_1 \wedge \varphi_2$ iff $\gamma, \rho \vDash_A^{\mathrm{ML}} \varphi_1$ and $\gamma, \rho \vDash_A^{\mathrm{ML}} \varphi_2$;
- $\gamma, \rho \vDash_A^{\mathrm{ML}} \exists x : s.\varphi'$ iff $\gamma, \rho' \vDash_A^{\mathrm{ML}} \varphi'$ for some ρ' such that $\rho'(y) = \rho'(x)$ for all $y \in \mathbf{Var}_s$ such that $y \neq x$;
- and $_, _ \vDash_A^{\mathrm{ML}} \bot$ never.

Example 13: Patterns related to IMP

1. The pattern $cfg(ite(b, c_1, c_2), s)$ (where b, c_1, c_2, and s are variables) would match a configuration consisting of an if-then-else command and some store.
2. The pattern

$$cfg\big(passign\big(pident(ident_x), padd\big(pident\big(ident_y\big), pzval_3\big)\big), s\big)$$
$$\wedge\, shas(s, y) = bval_1$$

with variables y and s matches configurations, intuitively, where a program fragment like `x := y + 3` is about to be executed, where
- `x` is the one particular program variable, spelled "x";
- there can be any program variable standing in place of `y` – so `y`, `z`, or even `x`;
- but this `y` has to be present in the store – as indicated by the use of the `shas` function.

The second example features a frequently-used idiom, where a pattern is composed of two patterns: *structural* and *structureless*.

Definition 14: Structural and structureless patterns

A matching logic pattern φ is *structural* iff it consists only of a term occurring as a predicate. A matching logic pattern φ is *structureless* iff it contains no term occuring as a predicate. That is, structureless patterns are as if defined by the following grammar:

$$\begin{array}{rcll} \varphi & ::= & t_{s'} = t_{s'} & \textit{equality} \\ & | & \varphi \wedge \varphi & \textit{conjunction} \\ & | & \bot & \textit{false} \\ & | & \exists x : s'.\varphi & \textit{existential quantifier} \end{array}$$

Unsurprisingly, structureless patterns behave like first-order logic formulas, in the sense that their semantics depends only on the valuation, as seen from the following lemma[4].

Lemma 15: On structureless patterns

For any algebra A, an A-valuation ρ, and $\gamma_1, \gamma_2 \in A$,

$$\gamma_1, \rho \vDash_A^{\mathrm{ML}} \varphi \Leftrightarrow \gamma_2, \rho \vDash_A^{\mathrm{ML}} \varphi$$

Proof: A straightforward induction on φ. □

Therefore, we can write $\rho \vDash_A^{\mathrm{ML}} \varphi$ to mean that $\gamma, \rho \vDash_A^{\mathrm{ML}} \varphi$ for some γ. Another important property is that, as in first-order logic, the meaning of a pattern is independent on the values of variables not present in the pattern.

Lemma 16: Independence of terms on 'other' variables

For all A-valuations ρ_1, ρ_2 satisfying $\rho_1(y) = \rho_2(y)$ for all $y \in \mathrm{VT}(t)$, we have that $\overline{\rho_1}(t) = \overline{\rho_2}(t)$.

Proof: By induction on t. □

Lemma 17: Independence of patterns

For all A-valuations ρ_1, ρ_2 satisfying $\rho_1(y) = \rho_2(y)$ for all y in $\mathrm{FV}(\varphi)$,

$$\gamma, \rho_1 \vDash_A^{\mathrm{ML}} \varphi \Leftrightarrow \gamma, \rho_2 \vDash_A^{\mathrm{ML}} \varphi$$

Proof: By induction on φ using Lemma 16. The only interesting case is the one of existential quantificaion. In that case, assume that $\rho_1(y) = \rho_2(y)$ for

[4]For ease of notations, I sometimes identify an algebra with its family of carrier sets, and omit sorts – as in this case.

all $y \in \mathrm{FV}(\exists x : s.\varphi)$; the inductive hypothesis says that for any two valuations ρ_1, ρ_2 satisfying $\rho_1(y) = \rho_2(y)$ for all $y \in \mathrm{FV}(\varphi)$, we have

$$\gamma, \rho_1 \vDash_A^{\mathrm{ML}} \varphi \Leftrightarrow \gamma, \rho_2 \vDash_A^{\mathrm{ML}} \varphi$$

for any γ; and have to prove that there exists some ρ_3 such that (1) $\rho_3(y) = \rho(y)$ for all $y \neq x$, and (2) $\gamma, \rho_3 \vDash_A^{\mathrm{ML}} \varphi$, if and only if there exists some ρ_4 such that (3) $\rho_4(y) = \rho_2(y)$ for all $y \neq x$, and (4) $\gamma, \rho_4 \vDash_A^{\mathrm{ML}} \varphi$. Let us show here only the proof of the "only if" implication, since the "if" one is similar. Assume there exists such ρ_3. Choose

$$\rho_4(y) = \begin{cases} \rho_4(y) \text{ if } y \neq x \\ \rho_3(y) \text{ if } y = x \end{cases}$$

from which (3) trivially follows. The point (4) follows from (2) and the inductive hypothesis after proving $\rho_3(y) = \rho_4(y)$ for all $y \in \mathrm{FV}(\varphi)$: either $y = x$, in which case $\rho_3(y) = \rho_4(y)$ holds by the definition of ρ_4 , or $y \neq x$, in which case we reduce the goal using the definition of ρ_4 to $\rho_3(y) = \rho_2(y)$. By (1), it is enough to show that $\rho_1(y) = \rho_2(y)$, which follows from the initial assumption. □

The following lemma is also similar to one in first-order logic.

Lemma 18: A semantic property of variable renaming

For any two variables x, y of the same sort, and for any two A-valuations ρ_1, ρ_2 which agree on all variables other than x and y, if

$$\rho_1(x) = \rho_2(y)$$

then for any matching logic formula φ in which y does not occur free,

$$\gamma, \rho_1 \vDash_A^{\mathrm{ML}} \varphi \Leftrightarrow \gamma, \rho_2 \vDash_A^{\mathrm{ML}} \varphi\left[\frac{y}{x}\right]$$

(where $\varphi\left[\frac{y}{x}\right]$ denotes the formula φ with occurrences of variable x changed to use y instead).

Proof: First prove a helper property that for all terms t, $\overline{\rho}_1(t) = \overline{\rho}_2\left(t\left[\frac{y}{x}\right]\right)$, by induction in t: if t is a variable, then perform case analysis on whether the variable is x; if t is a function application, then use the inductive hypothesis and the fact that function application preserves equality.

Now prove the lemma by induction on the size of φ, generalizing over ρ_1 and ρ_2. The case when φ is bottom is trivial. Then,

- if φ is a term t, apply the helper property;
- if φ is an equality $t_1 = t_2$, also use the above property;
- if φ is a conjunction, use the inductive hypothesis;

- if φ is an existentialy quantified formula $\exists z : s.\varphi'$, then
  - if $x = z$, what needs to be proven is that $\gamma, \rho' \vDash_A^{\mathrm{ML}} \varphi'$ for some ρ' satisfying $\rho'(y) = \rho_1(y)$ for all $y \neq z$, iff $\gamma, \rho'' \vDash_A^{\mathrm{ML}} \varphi'$ for some ρ'' satisfying $\rho''(y) = \rho_2(y)$ for all $y \neq z$ – which is easy, because one can simply choose ρ'' to give z the same value as ρ', and vice versa; and
  - if $x \neq z$, then we have to prove that $\gamma, \rho' \vDash_A^{\mathrm{ML}} \varphi'$ for some ρ' satisfying $\rho'(y) = \rho_1(y)$ for all $y \neq z$, iff $\gamma, \rho'' \vDash_A^{\mathrm{ML}} \varphi'\left[\frac{y}{x}\right]$ for some ρ'' satisfying $\rho''(y) = \rho_2(y)$ for all $y \neq z$, and since the size of φ' is smaller than the size of φ, and since $\rho'(x) = \rho_1(x) = \rho_2(y) = \rho''(y)$, we can conclude the proof by first-order reasoning using use the inductive hypothesis.

$\square$

One can easily convert a matching logic pattern to an "equivalent" (in some sense) FOL formula simply by introducing a new variable and replacing terms that occur as predicates in the pattern by equalities with respect to this new variable. The result of conversion is structureless; the conversion is no-op on already-structureless patterns; and the conversion preserves implication as well as equivalence.

Definition 19: FOL conversion [81]

Given a matching logic formula φ of sort s and a variable $\square$ of sort s, we let $\varphi^{\square}$ denote the FOL$_=$ formula formed from φ by replacing every term t that occurs at a predicate position by an equality $\square = t$. Given an algebra A, an A-valuation ρ, and an A-element γ of sort s, let the ρ^{γ} be the valuation defined by $\rho^{\gamma(\square)} = \gamma$ and $\rho^{\gamma(x)} = \rho(x)$ for $x \neq \square$.

Lemma 20: Result of FOL conversion is structureless

$\varphi^{\square}$ is structureless.

Lemma 21: Semantics of FOL conversion [81]

Whenever $\square$ is fresh in φ, we have

$$\gamma, \rho \vDash_A^{\mathrm{ML}} \varphi \Leftrightarrow \rho^{\gamma} \vDash_A^{\mathrm{ML}} \varphi^{\square}$$

Lemma 22: FOL conversion is identity on structureless patterns

$\varphi^{\square} = \varphi$ for any structureless pattern φ.

Lemma 23: FOL conversion and implication

If $\gamma, \rho \vDash_A^{\mathrm{ML}} \varphi_1 \to \varphi_2$, then $\gamma, \rho \vDash_A^{\mathrm{ML}} (\varphi_1^{\square})\left[\frac{x}{\square}\right] \to (\varphi_2^{\square})\left[\frac{x}{\square}\right]$

Proof: Let us have some γ,ρ,φ_1,φ_2 such that $\gamma, \rho \vDash_A^{\mathrm{ML}} \varphi_1 \to \varphi_2$, and let $\gamma, \rho \vDash_A^{\mathrm{ML}} (\varphi_1^{\square})\left[\frac{x}{\square}\right] \to (\varphi_2^{\square})\left[\frac{x}{\square}\right]$ be the goal. Change the goal to

$$\gamma, \rho \vDash_A^{\mathrm{ML}} \left((\varphi_1 \to \varphi_2)^{\square}\right)\left[\frac{x}{\square}\right]$$

by propagating implication through substitution and FOL conversion. Since $\rho(x) = \rho[\square := \rho(x)](\square)$, use Lemma 18 to change the goal to

$$\gamma, \rho[\square := \rho(x)] \vDash_A^{\mathrm{ML}} \left((\varphi_1 \to \varphi_2)^{\square}\right)\left[\frac{x}{\square}\right]$$

which is the same as

$$\gamma, \rho^{\rho(X)} \vDash_A^{\mathrm{ML}} \left((\varphi_1 \to \varphi_2)^{\square}\right)\left[\frac{x}{\square}\right]$$

because $\rho[\square := \rho(x)] = \rho^{\rho(X)}$. Now apply Lemma 21 to reduce the goal to

$$\gamma, \rho \vDash_A^{\mathrm{ML}} \varphi_1 \to \varphi_2$$

which was our assumption. □

Lemma 24: FOL conversion and equivalence

If $\gamma, \rho \vDash_A^{\mathrm{ML}} \varphi_1 \leftrightarrow \varphi_2$, then $\gamma, \rho \vDash_A^{\mathrm{ML}} (\varphi_1^{\square})\left[\frac{x}{\square}\right] \leftrightarrow (\varphi_2^{\square})\left[\frac{x}{\square}\right]$

Proof: Use Lemma 23 twice. □

Chapter III relies on these properties. After [81], matching logic was generalized in various ways. This thesis does not use these generalizations; the only exception is Chapter IV which describes a Coq mechanization designed to connect a practical tool to existing theory.

II.2.3.2 Dynamic reasoning

For dynamic reasoning – reasoning about program execution – reachability logic uses pairs of matching logic formulas. These pairs – *reachability logic rules* – are used both to describe a system as well as the system's *one-path partial correctness* properties. Even when reasoning about more complex properties – such as *all-path partial correctness* in all-Path reachability logic of A. Stefanescu, Ş. Ciobâcă, R. Mereuta, B. M. Moore, T.-F. Şerbănuţă, and G. Roşu [92] or *k-safety hyperproperties* of Cartesian reachability logic in Chapter III, one still uses reachability logic rules for system descriptions.

II.2.3.2.1 Reachability systems

A system of reachability logic rules assumes a common sort of all the rules. Thus, the sort is fixed already in the signature.

Definition 25: Reachability logic signature

A reachability logic signature $\mathbf{\Omega} = (\Omega, Cfg)$ over set S (of sorts) consists of an S-sorted algebraic signature $\Omega = (F, ar)$ and a distiguished sort $Cfg \in S$.

Definition 26: Reachability logic rule

Given a reachability logic signature $\mathbf{\Omega} = (\Omega, Cfg)$, a *reachability rule* $l \Rightarrow r$ is a pair of matching logic (over Ω) Cfg-formulas l, r.

Definition 27: Reachability system

Given a reachability logic signature $\mathbf{\Omega} = (\Omega, Cfg)$, a reachability system $\boldsymbol{R} = (A, R)$ consists of an Ω-algebra A and a set S of $\mathbf{\Omega}$-rules.

The semantics of a reachability system is given by means of transition systems.

Definition 28: Reachability system semantics

A reachability system $\boldsymbol{R} = (A, R)$ over an S-sorted signature $\Omega = (\Omega, cfg)$ (where $A = \left((T_s)_{s \in S}, f\right)$) induces the transition system

$$\left(T_{cfg}, \underset{\boldsymbol{R}}{\Rightarrow}\right)$$

where $\gamma \underset{\boldsymbol{R}}{\Rightarrow} \gamma'$ (for $\gamma, \gamma' \in T_{cfg}$) iff there is some rule $\varphi \Rightarrow \varphi' \in R$ and some A-valuation ρ such that

- $\gamma, \rho \vDash_A^{\mathrm{ML}} \varphi$; and
- $\gamma', \rho \vDash_A^{\mathrm{ML}} \varphi'$.

The intention is that the left-hand side of a rule describes when a rule can *fire*, and the right-hand side describes the *effect* of that rule. However, one can imagine some rules that do not conform to such intention – such as the rule

$$5 \Rightarrow \bot$$

(over the algebra of integers). Such rules are often prohibited in the literature [81] because they break the commonly relied-on operational intuitions.

Definition 29: Weakly well-defined systems

A reachability rule $\varphi \Rightarrow \varphi'$ is weakly well-defined with respect to an algebra A, iff for any γ and ρ satisfying $\gamma, \rho \vDash_A^{\mathrm{ML}} \varphi$ there exists some γ' such that $\gamma', \rho \vDash_A^{\mathrm{ML}} \varphi'$. A reachability system $\boldsymbol{R} = (A, R)$ is weakly well-defined iff each its rule $r \in R$ is weakly well-defined with respect to A.

Weakly well-definedness is a *semantic* property. This restriction is used in Chapter III. It occurs again in Chapter V, which presents an implementation of a new semantic framework named Minuska. (Minuska uses a purely syntactic check to filter out systems that are not weakly well-formed mechanically.)

Another problematic situation is when the system can perform steps that do not change the configuration – such as the rule

$$5 \Rightarrow 5$$

(again, over the algebra of integers). Let us call any system which can not perform such steps *irreflexive*.

Definition 30: Irreflexive systems

A reachability system $\boldsymbol{R}$ is *irreflexive* iff $\underset{\boldsymbol{R}}{\Rightarrow}$ is irreflexive as a relation.

II.2.3.2.2 Reachability claims

Given a programming language (represented as a reachability system $\boldsymbol{R}$) and a program fragment in a particular configuration (represented as a matching logic pattern φ), one would naturally want to ask: Is the program (partially) correct? This question can be formulated as: After a configuration γ satisfying φ is rewritten to a terminal configuration γ', does this γ' satisfy some other formula φ'?

There are multiple ways of formalizing this intuition. I would consider it natural to take all the possible executions into account; this leads to the notion of an *all-path reachability claim*.

Definition 31: All-path reachability claim [92]

An *all-path reachability claim* is a pair $\varphi \overset{\forall}{\Rightarrow} \varphi'$ of matching logic patterns φ and φ'. We write

$$\boldsymbol{R} \vDash \varphi \overset{\forall}{\Rightarrow} \varphi'$$

(and say that the claim $\varphi \overset{\forall}{\Rightarrow} \varphi'$ is satisfied in the reachability system $\boldsymbol{R}$) iff for every complete path π in the transition system induced by $\boldsymbol{R}$, if π starts with a configuration satisfying φ in some valuation ρ, then along the path π there is some configuration γ' satisfying φ' in ρ.

In a transition system, a "complete path" is such a path that cannot be further extended. The following points are worth noting:

- The definition does not talk about the last configuration in the path, which is what on would typically be interested in. This can be addressed by considering a pattern φ' which can be satisfied only by terminal configurations.

- The definition of the semantics is independent of the details of matching logic: only some relation between formulas and configurations is needed. Indeed, reachability logic has been generalized to other settings – e.g., [85] defines reachability logic directly on transition systems. This independence of reachability logic on the underlying "static" logic is also something that I would like to exploit in Minuska (Chapter V) in the future.

As an alternative to all-path reachability, one may consider one-path reachability, where an appropriate configuration is required only for one path. Someone might wonder why care about such a weaker notion of reachability; however, for deterministic systems, all-path and one-path reachability coincide, and the one-path variant is easier to reason about (and is also historically older).

Definition 32: One-path reachability claim [81],[92]

A *one-path reachability claim* is a pair $\varphi \overset{\exists}{\Rightarrow} \varphi'$ of matching logic patterns φ and φ'. We write

$$\boldsymbol{R} \vDash \varphi \overset{\exists}{\Rightarrow} \varphi'$$

(and say that the claim $\varphi \overset{\exists}{\Rightarrow} \varphi'$ is satisfied in the reachability system $\boldsymbol{R}$) iff for every configuration γ satisfying φ in some valuation ρ

1. in the transition system induced by $\boldsymbol{R}$ there exists an infinite path starting with γ; or
2. in the transition system induced by $\boldsymbol{R}$ there exists some (not necessarily complete) path π starting with γ ending in a state γ' that satisfies φ' in ρ.

Chapter III extends one-path reachability claims to express a class of *hyperproperties*; an extension of all-path reachability claims along the same lines is left as a future work.

II.2.3.2.3 One-path reachability proof system

Let us now review the sound and complete proof system of one-path reachability logic given by G. Roşu and A. Stefanescu [81]. (All-path reachability logic is equipped with a sound and complete proof system, too, but since it is not immediately relevant to this thesis, an interested reader is referred to [92]). The core of the proof system is defined as a ternary relation $_, _ \underset{\mathrm{RL}}{\vdash} _$ between a reachability system, a set of one-path reachability claims dubbed "circularities", and a one-path reachability claim; the relation is defined inductively by the rules in Figure 4[5].

II.2.4 Implementations

A. Stefănescu, D. Park, S. Yuwen, Y. Li, and G. Roşu [93] report on an automated

[5]The only difference between the version in Figure 4 and the version of [81] is, besides notation, that in [81] one can use the Abstraction rule to abstract multiple variables at once.

$$\frac{\varphi \Rightarrow \varphi' \in R}{R, C \underset{\mathrm{RL}}{\vdash} \varphi \overset{\exists}{\Rightarrow} \varphi'} \text{Axiom}$$

$$\frac{}{R, \emptyset \underset{\mathrm{RL}}{\vdash} \varphi \overset{\exists}{\Rightarrow} \varphi} \text{Reflexivity}$$

$$\frac{R, C \underset{\mathrm{RL}}{\vdash} \varphi_1 \overset{\exists+}{\Rightarrow} \varphi_2 \qquad R \cup \left\{\varphi \Rightarrow \varphi' \mid \varphi \overset{\exists}{\Rightarrow} \varphi' \in C\right\}, \emptyset \underset{\mathrm{RL}}{\vdash} \varphi_2 \overset{\exists}{\Rightarrow} \varphi_3}{R, C \underset{\mathrm{RL}}{\vdash} \varphi_1 \overset{\exists}{\Rightarrow} \varphi_3} \text{Transitivity}$$

$$\frac{R, C \underset{\mathrm{RL}}{\vdash} \varphi \overset{\exists}{\Rightarrow} \varphi' \qquad P \text{ is an FOL formula}}{R, C \underset{\mathrm{RL}}{\vdash} \varphi \wedge P \overset{\exists}{\Rightarrow} \varphi' \wedge P} \text{Logic Framing}$$

$$\frac{\vDash_A^{\mathrm{ML}} \varphi_1 \to \varphi_1' \qquad R, C \underset{\mathrm{RL}}{\vdash} \varphi_1' \overset{\exists}{\Rightarrow} \varphi_2' \qquad \vDash_A^{\mathrm{ML}} \varphi_2' \to \varphi_2}{R, C \underset{\mathrm{RL}}{\vdash} \varphi_1 \overset{\exists}{\Rightarrow} \varphi_2} \text{Consequence}$$

$$\frac{R, C \underset{\mathrm{RL}}{\vdash} \varphi_1 \overset{\exists}{\Rightarrow} \varphi \qquad R, C \underset{\mathrm{RL}}{\vdash} \varphi_2 \overset{\exists}{\Rightarrow} \varphi}{R, C \underset{\mathrm{RL}}{\vdash} \varphi_1 \vee \varphi_2 \overset{\exists}{\Rightarrow} \varphi} \text{Case analysis}$$

$$\frac{R, C \underset{\mathrm{RL}}{\vdash} \varphi \overset{\exists}{\Rightarrow} \varphi' \qquad x \notin \mathrm{FV}(\varphi')}{R, C \underset{\mathrm{RL}}{\vdash} \exists x.\varphi \overset{\exists}{\Rightarrow} \varphi'} \text{Abstraction}$$

$$\frac{R, C \cup \left\{\varphi \overset{\exists}{\Rightarrow} \varphi'\right\} \underset{\mathrm{RL}}{\vdash} \varphi \overset{\exists}{\Rightarrow} \varphi'}{R, C \underset{\mathrm{RL}}{\vdash} \varphi \overset{\exists}{\Rightarrow} \varphi'} \text{Circularity}$$

Figure 4: One-path reachability logic proof system [81]. The use of $\overset{\exists+}{\Rightarrow}$ in a sequent means it was derived without Reflexivity.

implementation of reachability logic reasoning in the $\mathbb{K}$-framework[6]. The paper features an algorithm (reproduced in Figure 5) for verifying all-path (or one-path) reachability claims. The algorithm assumes a particular form of reachability rules and claims: every matching logic pattern has to have the form of $\pi \wedge \psi$, where π is a structural pattern and ψ a structureless one, and the construction $frame(\varphi)$ in the algorithm is supposed to return the structural part. The algorithm's soundness is justified by the reachability logic proof systems, but I found the justification hard to understand.

A different implementation of reachability logic reasoning is reported in V. Rusu and A. Arusoaie [84], where the authors develop an incremental approach to proving reachability logic formulas. Yet another related work is A. Arusoaie, D. Nowak, V. Rusu, and D. Lucanu [6], reporting on a formally verified (in Coq) verification algorithm implementation for a reachability logic variant.

[6]https://kframework.org/

```
RLVERIFY(A, R, C):
foreach φ ⇒^∀ φ' ∈ C :
    let Q ← successors(φ)
    while Q is not empty:
        let φ_c, Q ← pop from Q
        if ⊨^ML_A φ_c → φ' then continue
        else if ∃σ such that ⊨^ML_A φ_c → σ(φ_l) for some φ_l ⇒^∀ φ_r ∈ C
        then
            let Q ← push σ(φ_r) ∧ frame(φ_c) to Q
        else
            let Q' ← successors(φ_c)
            if Q' is empty then fail
            push all of Q' to Q
```

Figure 5: An algorithm for verifying $(A, R) \vDash \varphi \overset{\forall}{\Rightarrow} \varphi'$ for every $\varphi \overset{\forall}{\Rightarrow} \varphi' \in C$.

II.2.5 Related work

To conclude the section on reachability logic, let me note that RL has been extended in [82] to work with systems defined using conditional rewrite rules, in the style of natural/big-step semantics or small-step semantics. Also, RL has been generalized to work with transition systems [85], and also to the settings of rewriting logic [64]. The intuition that reachability logic is based on coinduction is made more precise and turned into a coinduction-based verification system in B. M. Moore, L. Peña, and G. Roşu [71].

II.3 Equivalence verification

The theory of language-independent verification was also extended to *equivalence verification* of programs (in potentially different languages) [29], [30], [31]. From this line of work, Ş. Ciobâcă, D. Lucanu, V. Rusu, and G. Roşu [31] is of special interest, as it inspired me to develop the techniques powering Cartesian reachability logic (Chapter III). An interested reader may find an unofficial version of [31] from the webpage of Formal Systems Laboratory of Univeristy of Illinois[7], available at the time of writing, helpful, because [31] contains a small but severe typographic error making the presented proof system for full program equivalence illegible.

To use the proof system of [31], one has to consider two programs, P_1 and P_2, of two languages modeled as two reachability systems $\boldsymbol{R_1}, \boldsymbol{R_2}$ over two algebras $\mathcal{A}_1, \mathcal{A}_2$ that are extensions of a common sub-algebra $\mathcal{A}_c$. A part of the specification is a set E of matching logic formulas of a sort representing the product of

[7] https://fsl.cs.illinois.edu/publications/ciobaca-lucanu-rusu-rosu-2016-faoc.pdf

configurations of each language; each formula specifies a set of pairs of configurations that are considered to be equivalent. Another matching logic formula φ over the product of configuration specifies the "starting point" of the two programs, and one then wants to prove the (semantic) claim

$$\vDash \varphi \overset{\infty}{\Downarrow} E$$

(in words, φ *reaches* E) meaning that for all configurations γ_1, γ_2, if $< \gamma_1, \gamma_2 > \vDash^{\mathrm{ML}} \varphi$, then either

1. γ_1 and γ_2 diverge, or
2. $\gamma_1 \overset{*}{\underset{\boldsymbol{R_1}}{\Rightarrow}} \gamma_1'$ and $\gamma_2 \overset{*}{\underset{\boldsymbol{R_1}}{\Rightarrow}} \gamma_2'$ and $< \gamma_1, \gamma_2 > \vDash^{\mathrm{ML}} \psi$ for some formula $\psi \in E$ (and some configurations γ_1' and γ_2').

The proof system of [31] then uses a notion of *circularity* similar to that of reachability logic. Let us not discuss the details now; we return to this work only briefly in Chapter III.

At some point, an equivalence verifier based on the above theory was implemented in the $\mathbb{K}$-framework, but that implementation is no longer maintained.

II.4 Certification of a semantic framework's tasks

In 2024, $\mathbb{K}$-framework is over 14 years old, with G. Roşu and T.-F. Șerbănuță [83] mentioning it already in 2010. Recently, Z. Lin, X. Chen, M.-T. Trinh, J. Wang, and G. Roşu [63] from 2023 report $\mathbb{K}$-framework consisting of over half a million lines of unverified code written in Haskell, Java, and C++; moreover, $\mathbb{K}$-framework depends on existing SMT solvers such as Z3 [72]. With such a huge code base, the question of the trustworthiness of the framework becomes important.

A recent line of work [21], [63] proposes to address this question by *proof object generation*: tools derived from $\mathbb{K}$-framework should, besides performing the desired task, also emit a *proof object* (aka *proof certificate*) witnessing the correctness of the task's execution. So far, the proposed method has been implemented for two kinds of tools: concrete interpreters [21], and deductive verifiers [63]. This section gives a brief overview of the method and discusses its strong points as well as its weaknesses; I propose a direct way of addressing the weaknesses by designing and implementing a fully formally verified semantic framework in Chapter V.

II.4.1 Overview

Figure 6 shows a data-flow diagram describing the main processes and data elements in the $\mathbb{K}$-framework. The input to $\mathbb{K}$-framework is a language definition, written typically by a semantics researcher/engineer, stored in a `*.k` source file; in the following text, the `*.k` source file format is referred to as "$\mathbb{K}$ frontend language". The language definition gets processed by the $\mathbb{K}$-framework compiler (`kompile` command) into a different representation of the same programming language; this intermediate representation is known as "Kore". The Kore represen-

tation then gets translated by an encoder into a matching logic theory embedded in a formal language, Metamath.

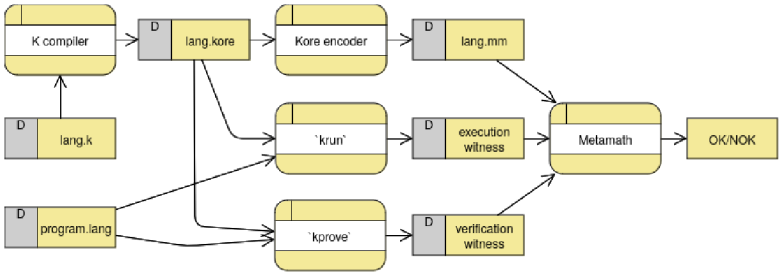

Figure 6: Flow of data in 𝕂-framework with proof object generation (based on [21], [63]).

Verification of concrete execution then works as follows. First, a component of the 𝕂-framework, Kore interpreter (wrapped as `krun` command), uses the compiled language definition in Kore format to execute a given program and emits a witness (in the Metamath format) that certifies that the execution was done in accord with the language semantics (`lang.kore`). Metamath is then used to check this witness. Program verification works similarly: another component of 𝕂-framework, a Kore-based verification tool (wrapped as `kprove` command), is given the Kore definition of the language and a program (annotated with the necessary invariants, etc.) and outputs a witness which is then checked with Metamath.

The actual approach proposed in [21], [63] differs from the one shown in Figure 6 in that `krun` and `kompile` themselves do not generate the proof objects – they only emit some information that is sufficient to construct the proof objects by an external tool (implemented in Python).

II.4.2 Strengths and weaknesses

The strength of the approach is that the practical tool, 𝕂-framework, can be kept almost as-is: the only modifications to the framework are those that emit some extra data in the form of proof objects. However, the approach also has weaknesses that motivate the proposal of a similar but fully formally verified framework in Chapter V. The weaknesses are as follows.

1. Only after-the-fact verification. One does not know whether the 𝕂-based tool will work correctly upfront.
2. For the interpreter (`krun`), the size of the proof object depends on the number of steps the interpreter takes. (This is less of a problem for the verification tool (`kprove`), which typically does not unroll loops).
3. Unnecessary performance cost for correctness. To trust the tools, one needs to store and verify large proof objects, which takes time and space.
4. Lack of direct formal semantics of the Kore intermediate language. One does not even know what the compiled definition (`lang.kore`) means by itself and, therefore, there are no applicable correctness criteria for the Kore encoder.

5. Possibility of logical inconsistencies in `lang.kore`. The 𝕂 tools do not check whether `lang.kore` (and the derived `lang.mm`) is logically consistent. Therefore, the proof objects may have no value, as anything can follow from an inconsistency.
6. No model theory in Metamath-encoding of matching logic. The authors encoded in Metamath only maching logic's syntax and proof system and not its semantics. Therefore, there is no guarantee that the encoding of the proof system is logically consistent, which creates a problem similar to the one in the previous point.

Let us now elaborate on some of these weaknesses more in-depth.

II.4.2.1 After-the-fact verification

Program verifiers are typically used in a controlled/development environment, where their malfunction cannot cause many problems. Therefore, having a guarantee that a particular verification succeeded *correctly* (that is, the result is not a *false negative* one) is enough for practical purposes, and there is little to gain from formally verifying the verifier. The situation is different for program interpreters: one may want to deploy such an interpreter to a production environment, and having no upfront guarantees about the interpreter's behavior is a major drawback of the proof-object generation approach.

II.4.2.2 Lack of direct formal semantics of Kore

In [63], the authors say the following:

> Kore behaves as the intermediate interface between the frontend and the backends. It is also the boundary between the informal and formal worlds. Since Kore is a formal specification language for writing matching logic theories, the formal semantics of a Kore definition is the matching logic theory that it defines.

However, the claim that "Kore is a formal specification language for writing matching logic theories" is doubtful, as demonstrated in this section. Specifically, someone may argue that

1. there is no formal semantics of Kore defined in (or referenced from) [21], [63], and
2. the gap between (informal) Kore and (formal) matching logic is wider than what one could reasonably expect from "formal specification language for writing matching logic theories."

First, we have a look at a fragment of a language definition in K and then in Kore; then, we discuss what it is supposed to mean in matching logic and what the gap between the middle and the latter is.

In [21], [63], language definitions are written by humans and are intended to be human-readable. Consider the following fragment of a language definition written in the frontend language of 𝕂 (adapted from [63]):

```
syntax Exp  ::= /* ... */
              | Int
syntax Stmt ::= /* ... */
              | "if" "(" Exp ")" Stmt "else" Stmt [strict(1)]
rule if (0) _ S => S
rule if (I) S _ => S requires I != 0
```

The `syntax` lines say that the language has, among others, sorts `Int` (which is built-in), `Exp`, `Stmt`, and a ternary `if` symbol; the `strict(1)` part instructs the framework to generate rules for evaluation of the first argument of the construct – that is, the condition. The first `rule` line says that if the condition is zero (it may have been zero all the time, or it could have been evaluated to zero from a more complex expression), the whole `if` statement is evaluated to its second argument – that is, the "else" branch is taken. The second `rule` line is trickier than the first one. Becuse the variable `I` is compared to zero (in the `requires` clause), 𝕂 infers that `I` has sort `Int`. Therefore, the rule says that whenever there is a non-zero integer in the expression of an `if` statement, the first branch is taken. (The sort inference made a real impact on the interpretation of the line, because if `I` was inferred to be `Exp`, a program fragment `if (0+0) S1 S2`) could[8] be evaluated to `S1` without evaluating the condition first. Unfortunately, this is not formally specified.

When translated using `kompile`[9] into Kore, the `syntax` lines get translated into something like

```
sort SortBExp{} []

symbol Lblif{}(SortBExp{}, SortBlock{}, SortBlock{}) : SortStmt{} [
    functional{}(),
    constructor{}(),
    strict{}("1"),
    right{}(),
    injective{}()
]
```

This should correspond to a declaration of a ternary symbol in an algebraic (or matching logic) signature. The comma-separated list of *attributes* in the square bracket is intended to express meaning beyond the signature – that the symbol is injective, behaves as a constructor ("constructs different object than all other constructors of the same sort"), etc. Some of these properties could be expressed in matching logic; however, the `constructor` property is not a "local" property

[8] due to nondeterminism

[9] Using the version of 𝕂 from LinCTWR23

but refers to other symbols, which makes the Kore "theory" more than just a list of axioms connected by an implicit connective (like conjunction in first-order logic).

Another problem is with Kore's handling of *subsorting*. There are two usual approaches one could take: either to make Kore *order-sorted*, with sort inclusions being explicitly handled by the logic, or it could be *many-sorted*, as is matching logic in Section II.2.3.1.2, and implement sort-inclusion by having injection symbols between sort. However, Kore is taking an approach somewhere in the middle: in a generated `*.kore` file, there is *one* symbol

```
symbol inj{From, To}(From) : To [sortInjection{}()]
```

representing all sort injections, the symbol is *parameterized* by source and target sorts, and the meaning of the `sortInjection` attribute is, intuitively, that only injections between the "right" sorts are "allowed". The "right" sorts are determined by another attribute, `subsort`, that is attached to some axioms, for example:

```
axiom{R} \exists{R} (Val:SortBExp{}, \equals{SortBExp{}, R} (
    Val:SortBExp{},
    inj{SortBool{}, SortBExp{}} (From:SortBool{}))
) [subsort{SortBool{}, SortBExp{}}()] // subsort
```

Thus, there is some implicit connection between the `subsort` attribute and the `sortInjection` attribute that is not straightforward to characterize precisely.

In summary, it is not easy to develop a model theory and a formal semantics for Kore, because the intentions behind some the attributes are unclear and rather convoluted. In the summer of 2023, I had several discussions (both verbal and written) with the main developers of the $\mathbb{K}$-framework, and as a result, they are aware of these issues.

II.4.2.3 No mechanized model theory

Weaknesses number 5 and 6 are related to the absence of a mechanized model theory of matching logic. These could be overcome by mechanizing matching logic, specifically the variant used by [21], [63], in a proof assistant, and using this mechanization as a "backend" (instead of using the limited Metamath mechanization) . To address 6, one would need at least to mechanize syntax, semantics, proof system, and proof of soundness of the proof system of matching logic; I lead an effort in this direction, resulting in the work presented in P. Bereczky, X. Chen, D. Horpácsi, T. B. Mizsei, L. Peña, and J. Tušil [9].

To address weakness number 5, we developed a Coq library for interactive matching logic reasoning presented in J. Tušil, P. Bereczky, and D. Horpácsi [96]. The motivation behind this work was that users of $\mathbb{K}$ often enhance their $\mathbb{K}$ development with custom lemmas that are justified only informally and added to their codebase as axioms, which can lead to inconsistency. One could formally

prove such lemmas with a matching logic reasoning infrastructure based on an interactive theorem-proving tool (such as Coq).

However, the mentioned work addressed only a part of weakness number 5. The second part is in that $\mathbb{K}$ users can arbitrarily axiomatize new symbols; as the symbols are new, they cannot be justified by appeal to other axioms, and they have to be justified model-theoretically. A trivial example is the K code

```
symbol Int ::= myFunction() [function]
rule myFunction() => 3
rule myFunction() => 4
```

which, when translated into Kore, results in two equalities from which a falsehood follows by transitivity. During 2023, I had several discussions with P. Bereczky and D. Horpácsi on addressing this issue. Ultimately, the efforts were fruitless mainly due to the lack of formal semantics of Kore (and in particular, the lack of model theory of Kore), which seemed hard to work around.

III Beyond safety with CRL

Most of the work known to me about deriving language tools from formal language semantics was done in 2010′s by Grigore Roşu's group. That work was mainly about interpreters, deductive verifiers of partial correctness, and deductive equivalence verifiers. The absence of similar work on other language-related topics like compilation, verification of richer classes of properties, and automatic analysis based on abstract interpretation [36], could make the more skeptical researchers suggest that in the end, semantics frameworks are not that useful as they can handle only certain tasks, and that for anything else we have to resort to traditional methods of tool construction.

To counter this view, I present a technique for language-parametric deductive verification of a class of *hyperproperties* beyond traditional *safety properties*.

III.1 Hyperproperties

Hyperproperties, introduced by M. R. Clarkson and F. B. Schneider [33], can relate multiple program executions. This is different to ("plain") properties, whose validity depends on a single execution trace. A particularly interesting class of hyperproperties are k-safety (hyper-)properties [1], [33], [44], [90]. A k-safety property is a hyperproperty whose violation can be witnessed by a k-tuple of execution traces. Many security policies – for example, noninterference (requiring that sensitive or privileged data do not influence insensitive or unprivileged computations) or observational determinism – are k-safety hyperproperties [32], [33]. Similarly, many functional-correctness properties are actually proper k-safety hyperproperties; for example, transitivity (which needs to be satisfied, e.g., by comparators when managing data in collections), associativity (important in the map/reduce paradigm), or monotonicity [90]. Techniques and tools for verifying hyperproperties of (finite-state) hardware [34], [45], as well as (infinite-state) software systems have been developed. For verification of software systems in particular, Cartesian Hoare logic (CHL), introduced in M. Sousa and I. Dillig [90], extends Hoare logic to allow reasoning about k-safety hyperproperties. To formalize Cartesian Hoare Logic, [90] uses a simple imperative language, whose looping constructs are while-loops with breaks. However, extending their approach to other constructs affecting control flow, or indeed other programming languages, can be both highly non-trivial and time-consuming.

This is where language-parametric approaches can be helpful. In J. Tušil, T. Şerbănuță, and J. Obdržálek [98], we have proposed a new logic, dubbed *Cartesian Reachability logic* (CRL), which properly extends reachability logic to allow reasoning about k-safety hyperproperties. Similarly to CHL, CRL has a sound and relatively complete proof system. A major advantage of CRL against CHL is that it works with any deterministic language for which reachability logic works. Technically, CRL does not extend CHL, because the two logics give different semantics to properties of nondeterministic programs; CHL supports non-determinism to simplify reasoning about loops. Despite this distinction, CRL extends CHL on

the deterministic fragment of the CHL-supported language (and in [90], CHL is neither applied nor advertised to be applicable to non-deterministic programs).

Personally, I found the intuitive knowledge of language-parametric equivalence verification, as discussed in Section II.3, very helpful when designing CRL. I do *not* elaborate on the intuitive connection between CRL and [29], [30], [31] in this text. However, I want to thank Nishant Rodrigues for inspirational discussions on the topic of equivalence verification and hyperproperties during Autumn 2022. The whole Chapter III is heavily based on [98].

III.2 Cartesian Hoare Logic

Introduced in [90], Cartesian Hoare Logic is a formalism for specifying and reasoning about k-safety properties in a way similar to how Hoare logic is used for reasoning about (safety) properties. In Hoare logic, properties are specified by means of *Hoare triples* (as discussed in Section II.2.1). Similarly, in Cartesian Hoare logic one specifies by means of a triple

$$\langle \Phi \rangle S_1 \circledast \ldots \circledast S_k \langle \Psi \rangle$$

with the following meaning: for any k-tuple $(\sigma_1, \ldots, \sigma_k)$ of states satisfying the formula Φ, if we execute each program S_i in the respective state σ_i and they all terminate, then the k-tuple $(\sigma'_1, \ldots, \sigma'_k)$ of the resulting program states satisfies Ψ. An important technical assumption here is that every program S_i operates on its own set of program variables, distinct from variables of other programs S_j (for $i \neq j$) - otherwise, the formulas Φ and Ψ would not be able to distinguish between program variables of different programs.

As an example, consider the program $P(x, y)$[10]

```
1  while(x > 0) { x--; y++; }
```

and the 2-safety property of monotonicity stating, intuitively: with growing inputs x and y, the resulting y also grows. In CHL, this can be written as:

$$\langle x_1 \leq x_2 \wedge y_1 \leq y_2 \rangle \;\; P(x_1, y_1) \circledast P(x_2, y_2) \;\; \langle y_1 \leq y_2 \rangle \qquad 1.$$

The main idea here is that the formulas in the precondition and postcondition can relate variables from different executions. Cartesian Hoare logic is equipped with a proof system that allows one to prove the validity of CHL triples. This proof system contains rules like

$$\frac{\langle \Phi \wedge c \rangle \;\; (B_1 \;\texttt{;}\; S) \circledast R \;\; \langle \Psi \rangle \qquad \langle \Psi \wedge \neg c \rangle \;\; (B_2 \;\texttt{;}\; S) \circledast R \;\; \langle \Psi \rangle}{\langle \Phi \rangle \;\; ((\texttt{if}\;(\;c\;)\; B_1 \;\texttt{else}\; B_2)\texttt{;} S) \circledast R \;\; \langle \Psi \rangle}$$

Figure 7: A CHL-rule for 'if' command. $\Phi, \Psi, c, B_1, B_2, S$ are meta-variables, with `()` being parantheses in the syntax of a program and () being meta-parentheses.

[10] with x, y being meta-variables ranging over program variables

the one in Figure 7 that replicate standard Hoare-logic reasoning, and is sound and complete. However, what is more interesting than completeness is that the proof system allows one to perform lockstep reasoning, even for loops. This is achieved by means of the rule from Figure 8.

$$\frac{\Phi \Rightarrow I \quad \langle I \wedge \neg c_2 \rangle \ \texttt{while(}\ c_1\ \texttt{)}\ \langle \Psi \rangle \quad \langle I \wedge \neg c_1 \rangle \ \texttt{while(}\ c_2\ \texttt{)}\ \langle \Psi \rangle \quad \langle I \wedge c_1 \wedge c_2 \rangle \ B_1 \circledast B_2 \ \langle I \rangle}{\langle \Phi \rangle \ (\ \texttt{while}\ (\ c_1\ )\ B_1 \circledast \texttt{while(}\ c_2\ )\ B_2) \ \langle \Psi \rangle}$$

Figure 8: A simplified CHL-rule for loops (version for two executions).

Note that the invariant I, assumed by this rule, can relate variables from both executions. The rule breaks reasoning about a pair of loops into three cases: one case where both loop conditions hold and two cases where one of the conditions does not hold. In the first case, both loops are executed "in lockstep", performing one iteration each, and their execution must preserve the invariant. In the remaining two cases, only one of the loops executes (in a state satisfying the invariant and negation of the other loop condition), resulting in a state satisfying the postcondition.

This lockstep reasoning is a powerful tool because the required invariants (relating different executions) are often very simple. For example, consider the program P and the formula in Equation 1 above. In this case it is enough to choose the invariant to be the same as the precondition (i.e. $x_1 \leq x_2 \wedge y_1 \leq y_2$). In contrast, to prove the above example without lockstep reasoning, one must find (non-relational) loop invariants strong enough to summarize the whole loop.

Unfortunately, lockstep reasoning rules become more complicated as one adds other features into the language – for example, the break statement (as done in [90]). It is not immediately obvious how to extend this approach to handle, e.g., recursion, continue, or goto. Also, observe that a single language feature (while loops) needed to be considered five times in order for CHL to support it soundly: the semantics of while is present in the operational semantics of the target language, in the Hoare logic for that language, in the Cartesian Hoare logic for that language, and in the proofs of soundness for both of the logics.

CRL is made to address these limitations. We based CRL on matching logic as discussed in Section II.2.3.1.2 and reachability logic as discussed in Section II.2, and follow simplifying assumptions as in the relevant literature.

Assumption 33: Restriction of reachability rules

Following the original reachability logic literature [81], [92], let us restrict the class of admissible reachability systems to those whose reachability rules have the shape

$$t \wedge P \Rightarrow t' \wedge P'$$

where t, t' are terms (that is, structural patterns) and P, P' FOL formulas (that is, structureless patterns). As argued in these papers, such rules can support several styles of operational semantics, including evaluation contexts, the chemical abstract machine, and continuation-based semantics.

Assumption 34: Restriction to irreflexive reachability systems

Following the literature, the CRL development assumes that the reachability system under consideration is irreflexive (Definition 30).

III.3 Cartesian reachability logic

Cartesian reachability logic is an extension of (one-path) reachability logic. To express k-safety properties, reachability claims $\varphi \overset{\exists}{\Rightarrow} \varphi'$ are extended to *cartesian reachability claims.*

Definition 35: CRL claim

Let $\mathbf{\Omega} = (\Omega, Cfg)$ be a reachability logic signature. A *cartesian reachability claim* has shape

$$[\varphi_1, ..., \varphi_k] \wedge P \overset{c\exists}{\Rightarrow} \exists \overline{Y}.\ [\varphi'_1, ..., \varphi'_k] \wedge P'$$

where $\varphi_1, ..., \varphi_k, \varphi'_1, ...\varphi'_k$ are structural matching logic patterns (that is, Ω -terms) of sort Cfg, P, P' structureless quantifier-free matching logic patterns (that is, quantifier-free FOL formulas), and $\overline{Y}$ is a (possibly empty) list $y_1, ..., y_n$ of variables.

The intuitive meaning of such a claim is as follows: there are k programs embedded into k source configurations, with i-th source configuration matching φ_i, and k target configurations where the i-th one matches φ'_i . Additionally, the FOL formula (or, structureless matching logic pattern) P can relate the source configurations, and the FOL formula P' can relate the target configurations. A formula of the shape

$$\overline{Y}.\ [\varphi'_1, ..., \varphi'_k] \wedge P'$$

is called an *existentially-quantified constrained list pattern (ECLP).*

Example 36: CRL claim

Consider an imperative programming language whose configurations have the shape

$$\ll C \mid \sigma \gg$$

where C is a *command* to be executed and σ a finite map from program variables to values. Let P denote the program

```
1 while(x > 0){ x--; y++; }
```

and let $\xi(C, X, Y)$ be the pattern

$$\ll C \mid \mathsf{x} \mapsto X, \mathsf{y} \mapsto Y \gg$$

matching those configurations of program C where the program variable x has value X and the program variable y has value Y.

Then, the claim Ω_{mono} defined as

$$[\xi(P, X_1, Y_1), \xi(P, X_2, Y_2)] \wedge X_1 \leq X_2 \wedge Y_1 \leq Y_2$$
$$\overset{c\exists}{\Rightarrow}$$
$$\exists X_1', Y_1', X_2', Y_2'.[\xi(\varepsilon, X_1', Y_1'), \xi(\varepsilon, X_2', Y_2')] \wedge Y_1' \leq Y_2'$$

(where ε denotes the empty program and $X_1, Y_1, X_1', Y_1', X_2', Y_2'$ are matching logic variables) expresses the property that the program P is monotone. That is, when we start an execution (using the semantics of the particular language) from some configuration γ_1 matching $\xi(P, X_1, Y_1)$ and a second execution from some configuration γ_2 matching $\xi(P, X_2, Y_2)$, if $X_1 \leq X_2$ and $Y_1 \leq Y_2$, after termination the configurations γ_1', γ_2' match $\xi(\epsilon, X_1', Y_1')$ and $\xi(\epsilon, X_2', Y_2')$ for some X_1', Y_1', X_2', Y_2' satisfying $Y_1' \leq Y_2'$. Note that we need the existential quantification on the right-hand side of the claims to be able to talk about the new values of program variables; this is because in CRL (and in RL) one distinguishes between program variables and logical variables.

For a formal definition of CRL semantics, we will need the notion of of a *terminating configuration*.

Definition 37: Terminating configuration

Given a transition system $(A, \Rightarrow)$ (where A is a set and $_ \Rightarrow _ \subseteq A \times A$ is a binary relation on A), a configuration $c \in A$ is said to be *terminating* iff there does not exist an infinite sequence $(c_i)_{i \in \mathbb{N}_0}$ starting in c (that is, such that $c_0 = c$) such that for every $i \in \mathbb{N}_0$, $c_i \Rightarrow c_{i+1}$.

The formal definition of CRL semantics follows.

Definition 38: CRL semantics

Let $\boldsymbol{\Omega} = (\Omega, Cfg)$ be a reachability logic signature and $\boldsymbol{R} = (A, R)$ a reachability system. A claim

$$[\varphi_1, ..., \varphi_k] \wedge P \overset{c\exists}{\Rightarrow} \exists \overline{Y} [\varphi'_1, ..., \varphi'_k] \wedge P'$$

is *valid* in a reachability system $\boldsymbol{R} = (A, R)$, written

$$(A, R) \vDash [\varphi_1, ..., \varphi_k] \wedge P \overset{c\exists}{\Rightarrow} \exists \overline{Y} [\varphi'_1, ..., \varphi'_k] \wedge P'$$

iff for all configurations $\gamma_1, ..., \gamma_k \in A_{Cfg}$ which terminate in A_{Cfg}, $\underset{\boldsymbol{R}}{\Rightarrow}$ and any A -valuation ρ, whenever $\gamma_i, \rho \vDash^{\mathrm{ML}}_A \varphi_i \wedge P$ for all $i \in \{1, ..., k\}$, then there exist configurations $\gamma'_1, ..., \gamma'_k \in A_{Cfg}$ such that $\gamma_i \overset{*}{\underset{\boldsymbol{R}}{\Rightarrow}} \gamma'_i$ for all $i \in \{1, ..., k\}$, and there also exists an A-valuation ρ' satisfying $\rho'(v) = \rho(v)$ for any $v \notin Y$, and $\gamma'_i, \rho' \vDash^{\mathrm{ML}}_A \varphi' \wedge P$ for all $i \in \{1, ..., k\}$.

Notation 39: Question mark notation

To make CRL formulas easier to read, we will use a notation adopted from the $\mathbb{K}$-framework: variables whose names start with a question mark are implicitly considered existentially quantified on the right-hand side. Also, an underscore is used to denote anonymous variables - we are not interested in their values. So, for example, the claim Ω_{mono} could be written as

$$[\xi(P, X_1, Y_1), \xi(P, X_2, Y_2)] \wedge X_1 \leq X_2 \wedge Y_1 \leq Y_2$$
$$\overset{c\exists}{\Rightarrow}$$
$$[\xi(\varepsilon, ?_, ?Y'_1), \xi(\varepsilon, ?_, ?Y'_2)] \wedge Y'_1 \leq Y'_2$$

III.3.1 CRL extends RL

It is worth pointing out that one cannot handle k-safety properties directly in RL, by simply replacing a CRL claim with k components by k RL claims. The reason is that in CRL formulas, one can relate variables from different components. First, note that in CRL, one can turn global constraints into local; for example, the claim Ω_{mono} is equivalent to

$$[\xi(P, X_1, Y_1), \xi(P, X_2, Y_2) \wedge X_1 \leq X_2 \wedge Y_1 \leq Y_2]$$
$$\overset{c\exists}{\Rightarrow}$$
$$[\xi(\varepsilon, ?_, ?Y'_1), \xi(\varepsilon, ?_, ?Y'_2) \wedge Y'_1 \leq Y'_2]$$

However, if one were to "split" this CRL claim into two, the resulting claims might express a different property than monotonicity. For example, the two claims

$$\xi(P, X_1, Y_1) \overset{\exists}{\Rightarrow} \xi(\varepsilon, ?_, ?\, Y_1')$$

$$\xi(P, X_2, Y_2) \wedge X_1 \leq X_2 \wedge Y_1 \leq Y_2 \overset{\exists}{\Rightarrow} \xi(\varepsilon, ?_, ?\, Y_2') \wedge Y_1' \leq Y_2'$$

hold for any reasonable program Q (reasonable here means that it either executes fully or diverges), because $?\, Y_1$ in the second claim is unrelated to $?\, Y_1$ in the first claim and thus one can set it to the value of $?\, Y_2$. On the other hand, if in the second claim we renamed $?\, Y_1$ to Y_1' (without a question mark), the claim would require that $?\, Y_2$ is greater than or equal to any integer (because Y_1' is not present on the left side), which clearly cannot hold in the model of integers.

CRL is an extension of RL in the following natural sense: If we restrict configuration lists in CRL claims to contain a single configuration each, validity of CRL and RL claims neatly coincide.

Lemma 40: CRL can express all RL claims

$$\Big((A, R) \vDash_\Omega [\varphi] \wedge \top \overset{c\exists}{\Rightarrow} [\varphi'] \wedge \top\Big) \Leftrightarrow \Big((A, R) \vDash_\Omega \varphi \overset{\exists}{\Rightarrow} \varphi'\Big)$$

Proof: Follows directly from the definitions of the semantics of CRL (Definition 38) and RL (Definition 28). □

III.3.2 A language-independent alternative to self-composition

Let us now discuss self-composition – a technique originally proposed by G. Barthe, P. R. D'Argenio, and T. Rezk [8] for reasoning about information-flow policies. The technique is used by M. Sousa and I. Dillig [90] for backing the soundness of their proof system of Cartesian Hoare Logic. With self-composition, a program P together with a k-safety hyperproperty is reduced to a sequential composition of P and copies of P (with renamed variables) with a safety property, in a semantics-preserving way. This technique allows one to use tools and techniques for verification of safety properties to perform verification of k-safety hyperproperties.

The technique crucially relies on a notion of *sequential composition* being present in the (fixed) programming language. Generalizing the technique to a language-parameteric settings is important for soundndess of the proof system of CRL, yet non-trivial. The challenge at hand is to generalize self-composition to work with any deterministic language, even if we do not know in advance how the language implements sequential composition, or if the language implements it at all.

To address this issue, we developed a novel, general technique, dubbed (for a lack of a better name) *star extension*. The main idea is to transform a CRL claim into an RL claim over an extended reachability system, whose configurations are lists of configurations of the original system. A part of the transformation is a function dubbed *flatten* that converts an ECLP (which contains a meta-level list of matching

logic patterns) into a matching logic formula (containing an object-level list) that matches lists inside the model. The transformation is quite straightforward but technical. Before presenting the technique in detail, let us have a look at the relevant theorem.

Theorem 41: Star extension is semantics-preserving

There exist a function $_^*$ on matching logic signatures, an equally-named function $_^*$ from reachability systems over the signature Ω to reachability systems over Ω^*, and a function $\mathit{flatten}$ from ECLPs over Ω to matching logic Ω^*-formulas, such that

$$\left(\boldsymbol{R} \vDash_{\Omega} \Psi \overset{c\exists}{\Rightarrow} \Psi'\right) \Leftrightarrow \left(\boldsymbol{R}^* \vDash_{\Omega^*} \mathit{flatten}(\Psi) \overset{\exists}{\Rightarrow} \mathit{flatten}(\Psi')\right)$$

for any deterministic system $\boldsymbol{R}$.

This technique is a contribution of its own, since it allows reducing k-safety hyperproperties of programs in arbitrary deterministic languages to safety properties. (The price one pays for such reduction is that the resulting system is no longer deterministic.)

The technique consists of two transformations that play nicely together: a transformation of models and a transformation of formulas. Let us first discuss the transformation of models and some of its properties.

III.3.2.1 Models transformation

Definition 42: Star extension

- Let $\boldsymbol{\Omega} = (\Omega, Cfg)$ be a reachability logic signature, where $\Omega = (F, ar)$ is an algebraic signature. Define

$$\boldsymbol{\Omega}^* = (\Omega^*, Cfg^*)$$

  where $\Omega^* = (F^*, ar^*)$, where

$$F^* = F \uplus \{cfgitem, cfgconcat, cfgheat, cfgnil\},$$

  where all the symbols on the right-hand side of $\uplus$ are fresh, and

$$\begin{aligned} ar^* = ar \cup \{&(cfgitem, (Cfg^*, [Cfg])) \\ &(cfgconcat, (Cfg^*, [Cfg^*, Cfg^*])) \\ &(cfgheat, (Cfg^*, [Cfg^*, Cfg, Cfg^*])) \\ &(cfgnil, (Cfg^*, [])) \} \end{aligned}$$

- Let R be a set of reachability rules over $\boldsymbol{\Omega}$. The set of reachability rules R^* over $\boldsymbol{\Omega}^*$ is generated by

- defining a function $heat : Var \times \mathbf{Pattern}(\Omega, Cfg) \times Var \to \mathbf{Pattern}(\Omega^*, Cfg^*)$ by

$$heat(L, t \wedge P, R) = cfgheat(L, t, R) \wedge P$$

(the function is total only assuming that the given patterns have the shape as required by Assumption 33);
- setting

$$R^* = \{heat(W, \varphi, W') \Rightarrow heat(W, \varphi', W') \mid (\varphi \Rightarrow \varphi') \in R\}$$

where W, W' are distinct, fixed, fresh variables not occuring in any rule in R.

- Let Ω by an algebraic signature, A an Ω-algebra. Define A^* to be the Ω^*-algebra which interprets all sorts and symbols of Ω as in A, and in addition interprets
  - the sort Cfg^* as the set $A^*_{Cfg^*}$ of all finite lists $[c_1; ...; c_n]$ for $n \in \mathbb{N}_0$, where c_i is an element of sort Cfg for any $1 \leq i \leq n$;
  - the symbol $cfgitem$ as the function $\{(c, [c]) \mid c \in A_{Cfg}\}$,
  - the symbol $cfgnil$ as the empty list [] (technically, as a function $\{(\langle\rangle, [])\}$ mapping a unit to the empty list) ;
  - the symbol $cfgconcat$ as the function $\{((l_1, l_2), l_1 \mathbin{++} l_2) \mid l_1, l_2 \in A_{Cfg^*}\}$;
  - the symbol $cfgheat$ as the function $\{((l_1, c, l_2), l_1 \mathbin{++} [c] \mathbin{++} l_2) \mid l_1, l_2 \in A_{Cfg^*} \wedge c \in A_{Cfg}\}$

  (where ++ represents list concatenation).
- Let Ω by an algebraic signature, A an Ω-algebra, and ρ an A-valuation. Define ρ^* to be the A^*-valuation that results by letting $\rho^*(v) = \rho(v)$ for any variable v of sort other than Cfg^*, and $\rho^*(w) = a$ (where $a \in A^*_{Cfg^*}$ is arbitrary) for any variable w of sort Cfg^*.
- Let $\boldsymbol{R} = (A, R)$ be a reachability system over signature $\boldsymbol{\Omega} = (\Omega, Cfg)$. Define the reachability system $\boldsymbol{R}*$ (over $(\Omega, Cfg)^*$) by $\boldsymbol{R}* = (A^*, R^*)$.

One would want this construction to satisfy some natural properties. Here is one.

Lemma 43: On star-extension and valuation extension

Let Ω be an algebraic signature signature, A an Ω-algebra, and ρ an A-valuation. Then

$$\overline{\rho^*}(t) = \overline{\rho}(t)$$

for all Ω-terms t.

Proof: By structural induction on the term t, the definition of a valuation extension (Definition 10), and the definition of a star-extension (Definition 42). □

Lemma 43 is used in the proof of the following conservativeness property of star-extension.

Lemma 44: Star-extension is conservative

Star-extension is conservative, in the sense that

$$(\gamma, \rho \vDash_A^{\mathrm{ML}} \varphi) \Leftrightarrow (\gamma, \rho^* \vDash_{A^*}^{\mathrm{ML}} \varphi)$$

for every Ω-algebra A, every A-valuation ρ, every Ω-sort s, every term $\gamma \in A_s$, and every matching logic pattern φ of sort s.

Proof: By induction on φ (generalizing over s, γ, and ρ).

- If φ is $t_1 = t_2$, the conclusions follows from Lemma 43.
- If φ is a term t of sort s, the conclusions follows from Lemma 43.
- If φ is $\varphi_1 \wedge \varphi_2$, the conclusion follows from the inductive hypothesis
- If φ is $\neg\varphi'$, the conclusion follows from the inductive hypothesis.
- If φ is $\exists x : s'.\varphi''$, the situation is more complex, and I want to introduce a new notation for writing proofs like this one. Let us manage the proof state as one would do in a proof assistant, displaying it each few steps, so that the reader does not get lost in the proof. Let us keep the notation simple and informal, omitting less interesting information etc. In the proof state that we have now, there is a single hypothesis – the inductive hypothesis – and a conclusion. It looks like this:

inductive hypothesis

For any s, γ, ρ, $(\gamma, \rho \vDash_A^{\mathrm{ML}} \varphi'') \Leftrightarrow (\gamma, \rho^* \vDash_{A^*}^{\mathrm{ML}} \varphi'')$.

Conclusion

For any s, γ, ρ, $(\gamma, \rho \vDash_A^{\mathrm{ML}} \exists x : s'.\varphi'') \Leftrightarrow (\gamma, \rho^* \vDash_{A^*}^{\mathrm{ML}} \exists x : s'.\varphi'')$

Let us prove each implication in the equivalence separately.

- To prove the first implication, consider the following proof state:

inductive hypothesis

For any s, γ, ρ, $\left(\gamma, \rho \vDash_A^{\mathrm{ML}} \varphi''\right) \Leftrightarrow \left(\gamma, \rho^* \vDash_{A^*}^{\mathrm{ML}} \varphi''\right)$.

left side

$\gamma, \rho \vDash_A^{\mathrm{ML}} \exists x : s'.\varphi''$

Conclusion

$\gamma, \rho^* \vDash_{A^*}^{\mathrm{ML}} \exists x : s'.\varphi''$

Using the semantics of existential quantifiers in matching logic (Definition 12), the proof state changes into

inductive hypothesis

For any s, γ, ρ,
$\left(\gamma, \rho \vDash_A^{\mathrm{ML}} \varphi''\right) \Leftrightarrow \left(\gamma, \rho^* \vDash_{A^*}^{\mathrm{ML}} \varphi''\right)$.

H1

For any variable y different from x, $\rho'(y) = \rho(y)$.

H2

$\gamma, \rho' \vDash_A^{\mathrm{ML}} \varphi''$

Conclusion

There exists some valuation ρ'' such that
- For any variable y different from x, $\rho''(y) = \rho(y)$; and
- $\gamma, \rho'' \vDash_{A^*}^{\mathrm{ML}} \varphi''$.

(for some valuation ρ'). The proof is concluded by choosing ρ^* for ρ'' and first-order reasoning.

- The second implication corresponds to the proof state

inductive hypothesis

For any s, γ, ρ, $(\gamma, \rho \vDash^{\mathrm{ML}}_{A} \varphi'') \Leftrightarrow (\gamma, \rho^* \vDash^{\mathrm{ML}}_{A^*} \varphi'')$.

right-hand side

$\gamma, \rho^* \vDash^{\mathrm{ML}}_{A^*} \exists x : s'.\varphi''$

Conclusion

$\gamma, \rho \vDash^{\mathrm{ML}}_{A} \exists x : s'.\varphi''$

which is concluded similarly to the first implication.

□

A step in a star-extended system work such that the variables W, W' together form something like a fixed frame, inside which lives one component that makes a step according to the original system.

Lemma 45: About step in a star-extended system

Given a reachability system $\boldsymbol{R} = (R, Cfg)$ over algebra A, for any two configurations $C, C' \in A^*_{Cfg^*}$, we have

$$C \underset{\boldsymbol{R}^*}{\Rightarrow} C'$$

if and only if there exists a rule $(t \wedge P \Rightarrow t' \wedge P') \in R$ and an A^*-valuation ρ such that

- $\rho \vDash^{\mathrm{ML}}_{A^*} P$;
- $\rho \vDash^{\mathrm{ML}}_{A^*} P'$;
- $C = \rho(W) \mathbin{++} \overline{\rho}(t) \mathbin{++} \rho(W')$; and
- $C' = \rho(W) \mathbin{++} \overline{\rho}(t') \mathbin{++} \rho(W')$

Proof: We have

$$C \underset{\boldsymbol{R}^*}{\Rightarrow} C'$$

iff (by RL semantics (Definition 28))

There exists a rule $\varphi \Rightarrow \varphi' \in R^*$ and an A^*-valuation ρ such that
- $C, \rho \vDash^{\mathrm{ML}}_{A^*} \varphi$, and
- $C', \rho \vDash^{\mathrm{ML}}_{A^*} \varphi'$.

iff (by the assumption on the shape of reachability rules (Assumption 33) and the definition of star-extension (Definition 42))

There exists a rule $(t \wedge P \Rightarrow t' \wedge P') \in R$ and an A^*-valuation ρ such that
- $C, \rho \vDash^{\mathrm{ML}}_{A^*} cfgheat(W, t, W') \wedge P$; and
- $C', \rho \vDash^{\mathrm{ML}}_{A^*} cfgheat(W, t', W') \wedge P'$.

iff (by matching logic semantics and properties of structureless patterns (Definition 12, Lemma 15))

There exists a rule $(t \wedge P \Rightarrow t' \wedge P') \in R$ and an A^*-valuation ρ such that
- $\rho \vDash^{\mathrm{ML}}_{A^*} P$
- $\rho \vDash^{\mathrm{ML}}_{A^*} P'$
- $C, \rho \vDash^{\mathrm{ML}}_{A^*} cfgheat(W, t, W')$; and
- $C', \rho \vDash^{\mathrm{ML}}_{A^*} cfgheat(W, t', W')$.

iff (by semantics of star extension; that is, by Definition 42 and Definition 12)

There exists a rule $(t \wedge P \Rightarrow t' \wedge P') \in R$ and an A^*-valuation ρ such that
- $\rho \vDash^{\mathrm{ML}}_{A^*} P$;
- $\rho \vDash^{\mathrm{ML}}_{A^*} P'$;
- $C = \rho(W) \mathbin{++} \overline{\rho}(t) \mathbin{++} \rho(W')$; and
- $C' = \rho(W) \mathbin{++} \overline{\rho}(t') \mathbin{++} \rho(W')$

which concludes the proof. □

Lemma 46: Steps in a star-extended system preserve configuration width

Let $\boldsymbol{R} = (A, R)$ be an (Ω, Cfg)-reachability system. Then for all $C, C' \in (A^*)_{Cfg^*}$, if

$$C \underset{\boldsymbol{R}^*}{\Rightarrow} C'$$

then the length of the list C (aka *width of* C) is the same as the length of the list C' (aka *width of* C').

Proof: By Lemma 45 applied twice and the fact that the length of a concatenation of two lists is the sum of the lenghts of the two lists. □

The following lemma presents the main property of star-extension.

Lemma 47: Characterization of a step in a star-extended system

Let $\boldsymbol{R} = (A, R)$ be a reachability system over the signature (Ω, Cfg). We have that

$$\left(c_i \underset{\boldsymbol{R}}{\Rightarrow} c'\right) \Leftrightarrow \left([c_1, ..., c_k] \underset{\boldsymbol{R}^*}{\Rightarrow} [c_1, ..., c_{i-1}, c', c_{i+1}, ..., c_k]\right)$$

for any $k \in \mathbb{N}_0$ such that $k \geq 1$, any $c_1, ..., c_k \in A_{Cfg}$, and any $i \in \{1, ..., k\}$.

Proof: By the previous lemma (Lemma 45), the right-hand side is equivalent to

> There exists a rule $(t \wedge P \Rightarrow t' \wedge P') \in R$ and an A^*-valuation ρ such that
> - $\rho \vDash^{\mathrm{ML}}_{A^*} P$; and
> - $\rho \vDash^{\mathrm{ML}}_{A^*} P'$; and
> - $[c_1, ..., c_k] = \rho(W) \mathbin{++} [\overline{\rho}(t)] \mathbin{++} \rho(W')$; and
> - $[c_1, ..., c_{i-1}, c', c_{i+1}, ..., c_k] = \rho(W) \mathbin{++} [\overline{\rho}(t')] \mathbin{++} \rho(W')$.

Suppose we have such a valuation ρ. We can construct a valuation ρ_0 by letting

$$\rho_0(v) = \begin{cases} \rho(v) \text{ if } \rho(v) \in A \\ a \text{ otherwise} \end{cases}$$

for some arbitrary element $a \in A$. Now, for any $\mathrm{VT}(t) \cup \mathrm{VT}(t') \cup \mathrm{FV}(P) \cup \mathrm{FV}(P')$, it holds that $((\rho_0)^*)(v) = \rho(v)$, because the sort of such a variable v is different from Cfg^* (and by the construction of a star-extended valuation). Therefore, the above is equivalent (one implication by independence of terms on 'other' variables (Lemma 16), the other by the choice of $\rho := (\rho_0)^*$) to

> There exists a rule $(t \wedge P \Rightarrow t' \wedge P') \in R$ and an A-valuation ρ_0 such that
> - $\rho_0 \vDash^{\mathrm{ML}}_{A} P$; and
> - $\rho_0 \vDash^{\mathrm{ML}}_{A} P'$; and
> - $[c_1, ..., c_k] = \rho_0(W) \mathbin{++} [\overline{\rho_0}(t)] \mathbin{++} \rho_0(W')$; and
> - $[c_1, ..., c_{i-1}, c', c_{i+1}, ..., c_k] = \rho_0(W) \mathbin{++} [\overline{\rho_0}(t')] \mathbin{++} \rho_0(W')$.

which is by properties of list concatenation equivalent to:

There exists a rule $(t \wedge P \Rightarrow t' \wedge P') \in R$ and an A-valuation ρ_0 such that there exists some $i' \in \{1, ..., k\}$ such that
- $\rho_0 \vDash_A^{\mathrm{ML}} P$; and
- $\rho_0 \vDash_A^{\mathrm{ML}} P'$; and
- $[c_1, ..., c_{i'-1}] = \rho_0(W)$
- $c_{i'} = \overline{\rho_0}(t)$
- $[c_{i'+1}, ..., c_k] = \rho_0(W')$; and
- $[c_1, ..., c_{i-1}, c', c_{i+1}, ..., c_k] = \rho_0(W) \mathbin{++} [\overline{\rho_0}(t')] \mathbin{++} \rho_0(W')$.

Now, let us define the family of c'_z (for $z \in \{1, ..., k\}$) by

$$c'_z = \begin{cases} c' \text{ if } \ z = i \\ c_z \text{ otherwise} \end{cases}$$

which makes the above equivalent to:

There exists a rule $(t \wedge P \Rightarrow t' \wedge P') \in R$ and an A-valuation ρ_0 such that there exists some $i' \in \{1, ..., k\}$ such that
- $\rho_0 \vDash_A^{\mathrm{ML}} P$; and
- $\rho_0 \vDash_A^{\mathrm{ML}} P'$; and
- $[c_1, ..., c_{i'-1}] = \rho_0(W)$
- $c_{i'} = \overline{\rho_0}(t)$
- $[c_{i'+1}, ..., c_k] = \rho_0(W')$; and
- $[c'_1, ..., c'_{i-1}] = \rho_0(W)$; and
- $c'_{i'} = \overline{\rho_0}(t')$; and
- $[c'_{i+1}, ..., c'_k] = \rho_0(W')$.

Now, since W, W' were fresh, they do not occur in t, t'. By independence of patterns (Lemma 17), we can equivalently say that

There exists a rule $(t \wedge P \Rightarrow t' \wedge P') \in R$ and an A-valuation ρ_0 such that there exists some $i' \in \{1, ..., k\}$ such that
- $\rho_0 \vDash_A^{\mathrm{ML}} P$; and
- $\rho_0 \vDash_A^{\mathrm{ML}} P'$; and
- $c_{i'} = \overline{\rho_0}(t)$
- $c'_{i'} = \overline{\rho_0}(t')$.

(the downward implication is trivial, as it is just removing some constraints on a valuation ρ_0; the upward implication follows from the fact that we can always choose ρ_0 such that it satisfies them). By the matching logic semantics of a conjunction and reachability system semantics (Definition 28), this is equivalent to

There exists some $i' \in \{1, ..., k\}$ such that $c_{i'} \underset{\boldsymbol{R}}{\Rightarrow} c'_{i'}$.

which is equivalent to the left-hand side of our lemma

$c_i \underset{\boldsymbol{R}}{\Rightarrow} c'_i$.

(one implication is by choice of $i' := i$, the other is as follows: if $i = i'$, we are done, but if $i \neq i'$, then by definition of $c'_{i'}$ we have $c_{i'} \underset{\boldsymbol{R}}{\Rightarrow} c_{i'}$, which contradicts irreflexivity of the reachability system (Assumption 34)). □

Lemma 48: At most one component changes in a star-extended system

Let $\boldsymbol{R} = (A, R)$ be an (Ω, Cfg)-reachability system. If

$$C \underset{\boldsymbol{R}^*}{\Rightarrow} C'$$

then there exists some $i \in \mathbb{N}_0$ such that for every $j \in \mathbb{N}_0$, if $j \neq i$ and j is smaller than the widths of C and C', then $C[j] = C'[j]$.

Proof: By Lemma 45, there exists a rule $(t \wedge P \Rightarrow t' \wedge P') \in R$ and an A^*-valuation ρ such that

- $\rho \vDash^{\mathrm{ML}}_{A^*} P$;
- $\rho \vDash^{\mathrm{ML}}_{A^*} P'$;
- $C = \rho(W) \mathbin{++} \overline{\rho}(t) \mathbin{++} \rho(W')$; and
- $C = \rho(W) \mathbin{++} \overline{\rho}(t') \mathbin{++} \rho(W')$.

Choose i to be the width of W and the rest follows easily. □

Before proceeding further with CRL, we need to prove some properties about a generic construction that is sometimes known as a *filtering subsequence*. The construction and its properties are based on the Coq formalization[11] of V. Zamfir *et al.* [102] (which is an otherwise unrelated paper on distributed systems), which itself is based on Y. Bertot [11].

[11] https://github.com/runtimeverification/vlsm/blob/d6c8cee56708c7be2431b9743fe80ca6a7a29a58/theories/VLSM/Lib/StreamFilters.v

Definition 49: Filtering subsequence

Given a set A, a subset $Q \subseteq A$, and a function $s : \mathbb{N}_0 \to A$, a function $f : \mathbb{N}_0 \to \mathbb{N}_0$ is said to be a *filtering subsequence for Q on s*, iff

- f is monotone;
- $s(x) \notin Q$ for any $x < f(0)$;
- $s(f(j)) \in Q$ for any $j \in \mathbb{N}_0$; and
- for every $j, x \in \mathbb{N}_0$ satisfying $f(j) < x < f(j+1)$, we have $s(x) \notin Q$.

(Intuitively, the last condition says that f does not skip any Q-element in s.)

Lemma 50: Existence of a filtering subsequence for infinite occurrences

Let A be a set, $Q \subseteq A$ a subset, and $s : \mathbb{N}_0 \to A$ a function whose output meets Q infinitely often: that is, for every $i \in \mathbb{N}_0$ there exists some $j \in \mathbb{N}_0$ such that $j \geq i$ and $s(j) \in Q$. Then there exists a filtering subsequence $f : \mathbb{N}_0 \to \mathbb{N}_0$ for Q on s.

Proof: In contrast to [102] and [11], who work in a constructive type theory, in our classical setting the proof is actually easy. Define

$$f(n) = \min\{m \in \mathbb{N}_0 \mid m \geq m, s(m) \in Q\}$$

and verify the four conditions of a filtering subsequence. The first three are easy; for the fourth one, assume $f(j) < x < f(j+1)$ and $f(x) \in Q$ and derive a contradiction as follows. Since $x < f(j+1)$, we have $x \notin \{m \in \mathbb{N}_0 \mid m \geq j+1, s(m) \in Q\}$. It follows that $x \geq j+1 \Rightarrow s(x) \notin Q$, and since we assumed $s(x) \in Q$, it follows that $x < j+1$, which is the same as $x \leq j$. Since $f(j) < x$, it follows that

$$\forall m \in \mathbb{N}_0 . m \geq j \wedge s(m) \in Q \Rightarrow m < x$$

which we can instantiate with $(m := x)$, resulting in

$$x \geq j \wedge s(x) \in Q \Rightarrow j < x$$

which is equivalent (we have $s(x) \in Q$ and use modus tollens) to

$$x \leq j \Rightarrow j < x$$

which is the desired contradiction, since we already know that $x \leq j$. $\square$

Lemma 51: Characterization of termination in a star-extended system

Let $\boldsymbol{R} = (A, R)$ be a reachability system. A configuration $[c_1, \ldots, c_k]$ is terminating in $\left(A^*, \underset{\boldsymbol{R}^*}{\Rightarrow}\right)$ iff every c_i (for $i \in \{1, \ldots, k\}$) is terminating in $\left(A, \underset{\boldsymbol{R}}{\Rightarrow}\right)$.

Proof: Let us prove each implication separately, by contraposition.

- Suppose some c_i is not terminating. In other words, there exists some infinite $\underset{\boldsymbol{R}}{\Rightarrow}$-sequence $\left(d_j\right)_{j\in\mathbb{N}_0}$. But then

$$\left(\left[c_1, ..., c_{i-1}, d_j, c_{i+1}, ..., c_k\right]\right)_{j\in\mathbb{N}_0}$$

  is an infinite sequence in $\left(A^*, \underset{\boldsymbol{R}^*}{\Rightarrow}\right)$ (by Lemma 47), witnessing the non-termination of $[c_1, ..., c_k]$.
- Suppose $[c_1, ..., c_k]$ is not terminating. In other words, there exists an infinite $\left(A^*, \underset{\boldsymbol{R}^*}{\Rightarrow}\right)$-sequence $(D_i)_{i\in\mathbb{N}_0}$ such that $D_0 = [c_1, ..., c_k]$. Then there exists some component $j \in \{1, ..., k\}$ that changes infinitely often along $(D_i)_{i\in\mathbb{N}_0}$. Consider now the function $s : \mathbb{N}_0 \to (A^*)_{Cfg^*} \times (A^*)_{Cfg^*}$ defined by

$$s(i) = (D_i, D_{i+1})$$

  and the relation $Q \subseteq (A^*)_{Cfg^*} \times (A^*)_{Cfg^*}$ defined by

$$(X, X') \in Q \Leftrightarrow ((X, X' \text{ have width at least } j) \text{ and } (X[j] \neq X'[j]))$$

  (where the square bracket operator is the list-indexing function). Now the output of s falls to Q infinitely often, since the component j infinitely often makes a step and this step is not along a self-loop (by Assumption 34). By Lemma 50 there exists a filtering subsequence f for Q on s, and then

$$D_{f(0)}[j] \underset{\boldsymbol{R}^*}{\Rightarrow} D_{f(1)}[j] \underset{\boldsymbol{R}^*}{\Rightarrow} D_{f(2)}[j] \underset{\boldsymbol{R}^*}{\Rightarrow} \cdots$$

  witnesses the nontermination of $c(j)$. Indeed, we have that
  - $D_{f(0)}[j] = D_0[j]$ by the second defining property of filtering subsequence (Definition 49), the definition of Q, and transitivity of equality; and
  - for any $i \in \mathbb{N}_0$, it holds that $D_{f(i)}[j] \underset{\boldsymbol{R}^*}{\Rightarrow} D_{f(i+1)}[j]$. This can be proved as follows. By the fourth defining property of filtering subsequence (Definition 49), and the definition of Q, we have $D_{f(i+1)}[j] = D_{f(i)+1}[j]$, so it is enough to show that

$$D_{f(i)}[j] \underset{\boldsymbol{R}^*}{\Rightarrow} D_{f(i)+1}[j] \ .$$

    That can be done using the lemma on the characterization of a step in a star-extended system (Lemma 47), by showing that there exists some $k \in \mathbb{N}_0, k \geq 1$, and $c_1, ..., c_k, c' \in A_{Cfg}$, and some $i \in \mathbb{N}_0, 1 \leq i \leq k$, such that $[c_1, ..., c_k] = D_{f(i)}$ and $[c_1, ..., c_{i-1}, c', c_{i+1}, ..., c_k] = D_{f(i)+1}$. But that follows from the fact that $D_{f(i)} \underset{\boldsymbol{R}^*}{\Rightarrow} D_{f(i)+1}$ and that $D_{f(i)}[j] \neq D_{f(i)+1}[j]$ because only one component can change per step in a star extended system (Lemma 48, using also Lemma 46).

□

Lemma 52: Characterization of the reflexive-transitive closure of a step in a star-extended system

For every reachability system $\boldsymbol{R} = (A, R)$, all $C, C' \in A^*_{Cfg^*}$, and all $c_1, ..., c_k, c'_1, ..., c'_k \in A_{Cfg}$ such that $C = [c_1, ..., c_k]$ and $C' = [c'_1, ..., c'_k]$, it holds that

$$C \overset{*}{\underset{\boldsymbol{R}^*}{\Rightarrow}} C'$$

iff for every $i \in \{1, ..., k\}$,

$$c_i \overset{*}{\underset{\boldsymbol{R}}{\Rightarrow}} c'_i \ .$$

Proof:

- For the "if" implication, assume that $c_i \overset{*}{\underset{\boldsymbol{R}}{\Rightarrow}} c'_i$ for every $i \in \{1, ..., k\}$; the goal is to prove that $[c_1, ..., c_k] \overset{*}{\underset{\boldsymbol{R}^*}{\Rightarrow}} [c'_1, ..., c'_k]$. By transitivity, it is enough to prove that for any $j \in \{1, ..., k\}$,

$$[c'_1, ..., c'_{j-1}, c_j, c_{j+1}, ..., c_k] \overset{*}{\underset{\boldsymbol{R}}{\Rightarrow}} [c'_1, ..., c'_{j-1}, c'_j, c_{j+1}, ..., c_k]$$

  (the difference between the two sides is in the prime over c_j) which follows from the assumption by Lemma 47.
- For the "only if" implication, assume $C \overset{*}{\underset{\boldsymbol{R}^*}{\Rightarrow}} C'$ and $i \in \{1, ..., k\}$; the goal is to prove that $c_i \overset{*}{\underset{\boldsymbol{R}}{\Rightarrow}} c'_i$. Let $C_1, ..., C_l \in A^*_{Cfg^*}$ be (for some $l \in \mathbb{N}_0$) a sequence witnessing $C \overset{*}{\underset{\boldsymbol{R}^*}{\Rightarrow}} C'$; that is, we have $C = C_1$, and $C' = C_l$, and $C_j \underset{\boldsymbol{R}^*}{\Rightarrow} C_{j+1}$ for any $j \in \{1, ..., l-1\}$. Now, consider some $i_1, ..., i_m \in \{1, ..., l-1\}$ such that $i_1 < ... < i_m$ and such that $C_{i_j}[i] \neq C_{i_j+1}[i]$ that maximize $m \in \mathbb{N}_0$ - which is, intuitively, the longest i-changing subsequence of the original sequence. By the maximality, we have $C_1[i] = C_{i_1}[i]$ and $C_{l[i]} = C_{i_m}[i]$ (otherwise we could make the subsequence longer). Let us now prove that

$$C_{i_1}[i] \underset{\boldsymbol{R}}{\Rightarrow} ... \underset{\boldsymbol{R}}{\Rightarrow} C_{i_m}[i]$$

  from which the conclusion follows. The goal is now to prove that for any $o \in \{1, ..., m\}$, $C_{i_o}[i] \underset{\boldsymbol{R}}{\Rightarrow} C_{i_{o+1}}[i]$. Let us set $d := i_{o+1} - i_o$. Now $d \geq 1$. Also, by the definition of d, we have $C_{i_{o+1}}[i] = C_{i_o+d}$. By the maximality of the subsequence, we have $C_{i_o+d}[i] = C_{i_o+1}[i]$ (since the component i can not change between the change at position i_o and the change at position i_{o+1}). So it is enough to show that $C_{i_o}[i] \Rightarrow C_{i_o+1}[i]$. By Lemma 47

(utilizing also Lemma 46 and Lemma 48), it is enough to show that $C_{i_o} \underset{\boldsymbol{R}^*}{\Rightarrow} C_{i_o+1}$, but that is trivial and we are done.

□

III.3.2.2 Formulas transformation

Now, let us have a look at the formula-transformation function.

Definition 53: Flatten

Define a function $mkList$ from non-empty lists of terms to terms by

$$\begin{aligned} mkList(t) &= cfgitem(t) \\ mkList(t_1, ..., t_k) &= cfgconcat(cfgitem(t), mkList(t_2, ..., t_k)) \text{ for } \ k \geq 2 \end{aligned}$$

and further, define a function $flatten$ from (potentially existentially quantified) constrained list patterns to matching logic patterns over a star-extended signature, by

$$\begin{aligned} flatten\Big(\exists \overline{X}.\ [\varphi_1, ..., \varphi_k] \wedge P\Big) = \exists \overline{X}.\ \ & mkList(Y_1, ..., Y_k) \\ & \wedge (\varphi_1^{\square})\left[\frac{Y_1}{\square}\right] \wedge ... \wedge (\varphi_k^{\square})\left[\frac{Y_k}{\square}\right] \wedge P \end{aligned}$$

where $Y_1, ..., Y_k, \square$ are fresh variables.

Intuitively, the function $flatten$ transforms an ECLP (a "formula" with meta-level lists) into an equivalent formula matching object-level lists.

Lemma 54: On mkList

For every A^*-valuation ρ and every $C, \varphi_1, ..., \varphi_k$,

$$C, \rho \vDash^{\mathrm{ML}}_{A^*} mkList(\varphi_1, ..., \varphi_k)$$

iff there exists $c_1, ..., c_k$ such that
- $C = [c_1, ..., c_k]$; and
- $c_j, \rho' \vDash^{\mathrm{ML}}_{A} \varphi_j$ for every $j \in \{1, ..., k\}$ and every A-valuation ρ' satisfying $\rho'(v) = \rho(v)$ for all $v \in \mathrm{FV}(mkList(\varphi_1, ..., \varphi_k))$.

Proof: By induction on k.
- If $k = 1$, we have to prove that

For every A^*-valuation ρ and every C, φ_1,

$$C, \rho \vDash^{\mathrm{ML}}_{A^*} cfgitem(\varphi_1)$$

iff there exists c_1 such that
- $C = [c_1]$; and
- $c_1, \rho' \vDash^{\mathrm{ML}}_{A} \varphi_j$ for every A-valuation ρ' satisfying $\rho'(v) = \rho(v)$ for any $v \in \mathrm{FV}(cfgitem(\varphi_1))$.

which is by matching logic semantics (Definition 12) equivalent to

For every A^*-valuation ρ and every C, φ_1,

$$C = \overline{\rho}(cfgitem(\varphi_1))$$

iff there exists c_1 such that
- $C = [c_1]$; and
- $c_1 = \overline{\rho'}(\varphi_j)$ for every A-valuation ρ' satisfying $\rho'(v) = \rho(v)$ for any $v \in \mathrm{FV}(cfgitem(\varphi_1))$.

which is by definition of star-extension (Definition 42) equivalent to

For every A^*-valuation ρ and every C, φ_1,

$$C = [cfgitem(\overline{\rho}(\varphi_1))]$$

iff there exists c_1 such that
- $C = [c_1]$; and
- $c_1 = \overline{\rho'}(\varphi_j)$ for every A-valuation ρ' satisfying $\rho'(v) = \rho(v)$ for any $v \in \mathrm{FV}(cfgitem(\varphi_1))$.

which holds because of the following argument. For the left-to-right implication, we let $c_1 := \overline{\rho}(\varphi_1)$ and have to prove that $\overline{\rho}(\varphi_1) = \overline{\rho'}(\varphi_1)$, which follows from Lemma 17. The right-to-left implication also follows Lemma 17.

- If $k = k' + 1$ for some $k' \in \mathbb{N}_0$ such that $k' \geq 1$, the goal looks like this:

inductive hypothesis

For every A^*-valuation ρ and every $C, \varphi_1, ..., \varphi_{k'}$,

$$C, \rho \vDash^{\mathrm{ML}}_{A^*} mkList(\varphi_1, ..., \varphi_{k'})$$

iff there exists $c_1, ..., c_{k'}$ such that
- $C = [c_1, ..., c_{k'}]$; and
- $c_j, \rho' \vDash^{\mathrm{ML}}_{A} \varphi_j$ for every $j \in \{1, ..., k'\}$ and every A-valuation ρ' satisfying $\rho'(v) = \rho(v)$ for any $v \in \mathrm{FV}(mkList(\varphi_1, ..., \varphi_{k'}))$.

Conclusion

For every A^*-valuation ρ and every $C, \varphi_1, ..., \varphi_{k'+1}$,

$$C, \rho \vDash^{\mathrm{ML}}_{A^*} mkList(\varphi_1, ..., \varphi_{k'+1})$$

iff there exists $c_1, ..., c_{k'+1}$ such that
- $C = [c_1, ..., c_{k'+1}]$; and
- $c_j, \rho' \vDash^{\mathrm{ML}}_{A} \varphi_j$ for every $j \in \{1, ..., (k'+1)\}$ and every A-valuation ρ' satisfying $\rho'(v) = \rho(v)$ for any $v \in \mathrm{FV}(mkList(\varphi_1, ..., \varphi_{k'+1}))$.

The conclusion is by the semantics of matching logic (Definition 12) equivalent to

For every A^*-valuation ρ and every $C, \varphi_1, ..., \varphi_{k'+1}$,

$$C = \overline{\rho}(mkList(\varphi_1, ..., \varphi_{k'+1}))$$

iff there exists $c_1, ..., c_{k'+1}$ such that
- $C = [c_1, ..., c_{k'+1}]$; and
- $c_j = \overline{\rho'}(\varphi_j)$ for every $j \in \{1, ..., (k'+1)\}$ and every A-valuation ρ' satisfying $\rho'(v) = \rho(v)$ for any $v \in \mathrm{FV}(mkList(\varphi_1, ..., \varphi_{k'+1}))$.

which is by the definition of a star-extension (Definition 42), after remembering C', equivalent to

For every A^*-valuation ρ and every $C, C', \varphi_1, ..., \varphi_{k'+1}$, $C = [\overline{\rho}(\varphi_1)] \mathbin{++} C'$ and

$$C' = \overline{\rho}(mkList(\varphi_2, ..., \varphi_{k'+1}))$$

iff there exists $c_1, ..., c_{k'+1}$ such that
- $C = [c_1, ..., c_{k'+1}]$; and
- $c_j = \overline{\rho'}(\varphi_j)$ for every $j \in \{1, ..., (k'+1)\}$ and every A -valuation ρ' satisfying $\rho'(v) = \rho(v)$ for any $v \in \mathrm{FV}(mkList(\varphi_1, ..., \varphi_{k'+1}))$.

which is by the inductive hypothesis specialized with $\varphi_1 := \varphi_2, ..., \varphi_k := \varphi_{k+1}$, matching logic semantics (Definition 12) and renaming of variables equivalent to:

For every A^*-valuation ρ and every $C, C', \varphi_1, ..., \varphi_{k'+1}$,

- $C = [\overline{\rho}(\varphi_1)] \mathbin{++} C'$ and
- there exists $c_2, ..., c_{k'+1}$ such that
  - $C' = [c_2, ..., c_{k'+1}]$
  - $c_j, \rho' \vDash_A^{\mathrm{ML}} \varphi_j$ for every $j \in \{1, ..., k'\}$ and every A -valuation ρ' satisfying $\rho'(v) = \rho(v)$ for any $v \in mkList(\varphi_2, ..., \varphi_{k'+1})$

iff

there exists $c_1, ..., c_{k'+1}$ such that
- $C = [c_1, ..., c_{k'+1}]$; and
- $c_j = \overline{\rho'}(\varphi_j)$ for every $j \in \{1, ..., (k'+1)\}$ and every A-valuation ρ' satisfying $\rho'(v) = \rho(v)$ for any $v \in \mathrm{FV}(mkList(\varphi_1, ..., \varphi_{k'+1}))$.

It is enough to prove

For every A^*-valuation ρ and every $C, C', \varphi_1, ..., \varphi_{k'+1}$ such that $C = [\overline{\rho}(\varphi_1)] \mathbin{++} C'$, and for every $c_2, ..., c_{k'+1}$,

- $C' = [c_2, ..., c_{k'+1}]$
- $c_j, \rho' \vDash_A^{\mathrm{ML}} \varphi_j$ for every $j \in \{1, ..., k'\}$ and every A-valuation ρ' satisfying $\rho'(v) = \rho(v)$ for any $v \in mkList(\varphi_2, ..., \varphi_{k'+1})$

iff

there exists c_1 such that
- $C = [c_1, ..., c_{k'+1}]$; and
- $c_j = \overline{\rho'}(\varphi_j)$ for every $j \in \{1, ..., (k'+1)\}$ and every A-valuation ρ' satisfying $\rho'(v) = \rho(v)$ for any $v \in \mathrm{FV}(mkList(\varphi_1, ..., \varphi_{k'+1}))$.

which can be proved as follows.
- For the top-down implication, choose $c_1 := \overline{\rho}(\varphi_1)$ and conclude using the semantics of matching logic (Definition 12).
- For the bottom-up implication, conclude using the semantics of matching logic (Definition 12).

□

Lemma 55: On flattening

For every Ω-algebra A, all $C \in A^*_{Cfg^*}$, and all A^*-valuation ρ, we have

$$C, \rho \vDash_{A^*}^{\mathrm{ML}} flatten\big(\exists \overline{X}.\ [\varphi_1, ..., \varphi_k] \wedge P\big)$$

iff there exist configurations $c_1, ..., c_k \in A_{Cfg}$ such that $C = [c_1, ..., c_k]$ and there also exists an A-valuation ρ_0 satisfying $\rho_0(v) = \rho(v)$ for every $v \notin \overline{X}$ such that for all $j \in \{1, ..., k\}$, $c_j, \rho_0 \vDash_A^{\mathrm{ML}} \varphi_j \wedge P$.

Proof: We have

$$C, \rho \vDash_{A^*}^{\mathrm{ML}} flatten\big(\exists \overline{X}.\ [\varphi_1, ..., \varphi_k] \wedge P\big)$$

iff (by definition of *flatten* (Definition 53) and matching logic semantics (Definition 12))

There exists an A^*-valuation ρ' satisfying $\rho'(v) = \rho(v)$ for all $v \notin \overline{X}$ such that
- $C, \rho' \vDash^{\mathrm{ML}}_{A^*} mkList(Y_1, ..., Y_k)$; and
- $C, \rho' \vDash^{\mathrm{ML}}_{A^*} \left(\varphi^{\square}_j\right)\left[\frac{Y_j}{\square}\right] \wedge P$ for all $j \in \{1, ..., k\}$.

iff (Lemma 54)

There exists an A^*-valuation ρ' satisfying $\rho'(v) = \rho(v)$ for all $v \notin \overline{X}$ such that
- there exists $c_1, ..., c_k$ such that
  - $C = [c_1, ..., c_k]$, and
  - $c_j, \rho'' \vDash^{\mathrm{ML}}_{A} Y_j$ for every $j \in \{1, ..., k\}$ and every A-valuation ρ'' satisfying $\rho''(v) = \rho'(v)$ for every $v \in \mathrm{FV}(mkList(Y_1, ..., Y_k))$;

  and
- $C, \rho' \vDash^{\mathrm{ML}}_{A^*} \left(\varphi^{\square}_j\right)\left[\frac{Y_j}{\square}\right] \wedge P$ for all $j \in \{1, ..., k\}$.

iff (by Definition 12)

There exists an A^*-valuation ρ' satisfying $\rho'(v) = \rho(v)$ for all $v \notin \overline{X}$ such that
- there exists $c_1, ..., c_k$ such that
  - $C = [c_1, ..., c_k]$, and
  - $\rho'(Y_j) = c_j$ for every $j \in \{1, ..., k\}$;

  and
- $C, \rho' \vDash^{\mathrm{ML}}_{A^*} \left(\varphi^{\square}_j\right)\left[\frac{Y_j}{\square}\right] \wedge P$ for all $j \in \{1, ..., k\}$.

iff (by Definition 12 and Lemma 15)

There exist configurations $c_1, ..., c_k \in A_{Cfg}$ such that $C = [c_1, ..., c_k]$, and there exists an A-valuation ρ' satisfying $\rho'(v) = \rho(v)$ for all $v \notin \overline{X}$ such that for all $j \in \{1, ..., k\}$,
- $\rho'(Y_j) = c_j$;
- $\rho' \vDash^{\mathrm{ML}}_{A^*} \left(\varphi^{\square}_j\right)\left[\frac{Y_j}{\square}\right]$
- $\rho' \vDash^{\mathrm{ML}}_{A^*} P$

iff (by Lemma 44)

There exist configurations $c_1, ..., c_k \in A_{Cfg}$ such that $C = [c_1, ..., c_k]$, and there exists an A-valuation ρ' satisfying $\rho'(v) = \rho(v)$ for all $v \notin \overline{X}$ such that for all $j \in \{1, ..., k\}$,
- $\rho'(Y_j) = c_j$;
- $\rho' \vDash_A^{\mathrm{ML}} \left(\varphi_j^{\square}\right)\left[\frac{Y_j}{\square}\right]$
- $\rho' \vDash_A^{\mathrm{ML}} P$

iff (by Lemma 18, since we have $\rho'(Y_j) = c_j$ in and we know that $\rho_0^{c_j}(\square) = c_j$ by definition (Definition 19); for the bottom-up implication we also need the assumption that Y_j was fresh and Lemma 17
- in order to choose the valuation ρ' satisfying $\rho'(Y_j) = c_j$)

There exist configurations $c_1, ..., c_k \in A_{Cfg}$ such that $C = [c_1, ..., c_k]$ and there exists an A-valuation ρ_0 satisfying $\rho_0(v) = \rho(v)$ for all $v \notin \overline{X}$ such that for all $j \in \{1, ..., k\}$,
- $\rho_0^{c_j} \vDash_A^{\mathrm{ML}} \varphi_j^{\square}$
- $\rho_0 \vDash_A^{\mathrm{ML}} P$

iff (Lemma 21, Definition 12, Lemma 15)

There exist configurations $c_1, ..., c_k \in A_{Cfg}$ such that $C = [c_1, ..., c_k]$ and there also exists an A-valuation ρ_0 satisfying $\rho_0(v) = \rho(v)$ for every $v \notin \overline{X}$ such that for all $j \in \{1, ..., k\}$, $c_j, \rho_0 \vDash_A^{\mathrm{ML}} \varphi_j \wedge P$.

which is what we wanted to prove. □

III.3.2.3 Star extension is semantics-preserving

The following theorem is a more concrete variant of Theorem 41, which then follows by instantiating the existential quantifiers appropriately.

Theorem 56: Reduction by flattening

$$\left(\boldsymbol{R} \vDash_{\Omega} \Psi \overset{c\exists}{\Rightarrow} \Psi'\right) \Leftrightarrow \left(\boldsymbol{R}^* \vDash_{\Omega^*} \mathit{flatten}(\Psi) \overset{\exists}{\Rightarrow} \mathit{flatten}(\Psi')\right)$$

for every deterministic system $\boldsymbol{R}$ over signature Ω.

Proof: Each implication is proved separately.
- For the left-to-right implication, let $\boldsymbol{R} = (A, R)$, let $\Psi = (\varphi_1, ..., \varphi_k) \wedge P$, and $\Psi' = \exists \overline{Y}.\ (\varphi'_1, ..., \varphi'_k) \wedge P'$, and assume that

  i $$\boldsymbol{R} \vDash_{\Omega} \Psi \overset{c\exists}{\Rightarrow} \Psi' \ ;$$

  that is,

i

for all configurations $c_1, ..., c_k \in A_{Cfg}$ which terminate in $\left(A_{Cfg}, \underset{\boldsymbol{R}}{\Rightarrow}\right)$ and every A-valuation ρ, whenever $c_i, \rho \vDash_A^{\mathrm{ML}} \varphi_i \wedge P$ for all $i \in \{1, ..., k\}$, then there exist configurations $c'_1, ..., c'_k \in A_{Cfg}$ such that $c_i \overset{*}{\underset{\boldsymbol{R}}{\Rightarrow}} c'_i$ for all $i \in \{1, ..., k\}$, and there also exists an A-valuation ρ' satisfying $\rho'(v) = \rho(v)$ for all $v \notin \overline{Y}$, and $c'_i, \rho' \vDash_A^{\mathrm{ML}} \varphi'_i \wedge P'$.

We have to prove that

for every $C \in A^*_{Cfg^*}$ such that C terminates in $\left(A^*_{Cfg^*}, \underset{\boldsymbol{R}^*}{\Rightarrow}\right)$ and for an A^*-valuation ρ such that $C, \rho \vDash_{A^*}^{\mathrm{ML}} \mathit{flatten}(\Psi)$, there exists some $C' \in A^*_{Cfg^*}$ such that $C \overset{*}{\underset{\boldsymbol{R}^*}{\Rightarrow}}$ and $C', \rho \vDash_{A^*}^{\mathrm{ML}} \mathit{flatten}(\Psi')$.

Assume such C and ρ, and let us proceed in five steps.

1. Decompose $C = [c_1, ..., c_k]$ for some $c_1, ..., c_k$.
2. Prove that c_i is terminating (for $i \in \{1, ..., k\}$).
3. Prove the premise of the assumption (i) for an appropriate valuation ρ.
4. From the assumption (i) get some appropriate $c'_1, ..., c'_k$ and a valuation ρ';
5. Let $C' := [c'_1, ..., c'_k]$ and prove that it is reachable from C and that it satisfies flattened Ψ' in ρ'.

We have

$$C, \rho \vDash_{A^*}^{\mathrm{ML}} \mathit{flatten}((\varphi_1, ..., \varphi_k) \wedge P)$$

which is by Lemma 55 equivalent to

ii

there exist configurations $c_1, ..., c_k \in A_{Cfg}$ such that $C = [c_1, ..., c_k]$, and there exists an A-valuation ρ_0 satisfying $\rho_0(v) = \rho(v)$ for every v, such that for every $j \in \{1, ..., k\}$, we have $c_j, \rho_0 \vDash_A^{\mathrm{ML}} \varphi_j \wedge P$.

Let ρ_0 be such valuation, and $c_1, ..., c_k$ such configurations. That completes the first step. For the second step, use Lemma 51. For the third step, let $\rho_1 := \rho_0$ and apply the assumption (ii). Now it follows that

iii

there exist configurations $c'_1, ..., c'_k \in A_{Cfg}$ such that $c_j \overset{*}{\underset{\boldsymbol{R}}{\Rightarrow}}$ for every $j \in \{1, ..., k\}$, and there also exists an A-valuation ρ'' satisfying $\rho''(v) = \rho_1(v)$ for all $v \notin \overline{Y}$, and $c_j, \rho'' \vDash_A^{\mathrm{ML}} \varphi'_j \wedge P$ for every $j \in \{1, ..., k\}$.

Let us now have such configurations $c'_1, ..., c'_k$ and such valuation ρ''. Choose $C' := [c'_1, ..., c'_k]$; it remains to be proven that

$$C \overset{*}{\underset{\boldsymbol{R}}{\Rightarrow}} C' \text{ and } C', \rho \vDash^{\mathrm{ML}}_{A^*} \mathit{flatten}(\Psi')$$

(where ρ is the original valuation). The first conjunct holds by Lemma 52. The second conjunct is by Lemma 55 equivalent to

there exist configurations $c'_1, ..., c'_k \in A_{Cfg}$ such that $C' = [c'_1, ..., c'_k]$ and there exists a valuation ρ'_0 satisfying $\rho'_0(v) = \rho(v)$ for every $v \notin \overline{Y}$, such that for every $j \in \{1, ..., k\}$, it holds that $c'_j, \rho'_0 \vDash^{\mathrm{ML}}_{A} \varphi'_j \wedge P$.

We choose exactly $c'_j := c'_j$ for each $j \in \{1, ..., k\}$, and choose ρ'_0 to be exactly ρ'' restricted to the variables of signature Ω; that is, $\rho'_0(v) = \rho''(v)$ for each v. The rest follows from (iii) by Lemma 17.

- For the opposite implication, let's assume that
  - $\Psi \equiv (\varphi_1, ..., \varphi_k) \wedge P$;
  - $\Psi' \equiv \exists \overline{Y}.\ (\varphi'_1, ..., \varphi'_k) \wedge P'$;
  - $\varphi_j \equiv t_j \wedge P_j$ for every $j \in \{1, ...k\}$;
  - $\varphi'_j \equiv t'_j \wedge P'_j$ for every $j \in \{1, ...k\}$;

  and assume that

  iv $$\boldsymbol{R}^* \vDash_{\Omega^*} \mathit{flatten}(\Psi) \overset{\exists}{\Rightarrow} \mathit{flatten}(\Psi')$$

  that is

  iv for every $C \in A^*_{Cfg^*}$ that terminates in $\left(A^*_{Cfg^*}, \underset{\boldsymbol{R}^*}{\Rightarrow}\right)$ and for every A^*-valuation ρ such that $C, \rho \vDash^{\mathrm{ML}}_{A^*} \mathit{flatten}(\Psi)$, there exists some $C' \in A^*_{Cfg^*}$ such that $C \overset{*}{\underset{\boldsymbol{R}^*}{\Rightarrow}} C'$ and $C', \rho \vDash^{\mathrm{ML}}_{A^*} \mathit{flatten}(\Psi')$.

  The goal to prove is

$$\boldsymbol{R} \vDash_{\Omega} \Psi \overset{c\exists}{\Rightarrow} \Psi' \ ;$$

that is,

for all configurations $c_1, ..., c_k \in A_{Cfg}$ which terminate in $\left(A_{Cfg,\underset{\boldsymbol{R}}{\Rightarrow}}\right)$ and every A-valuation ρ_1, whenever $c_j, \rho_1 \vDash_A^{\mathrm{ML}} \varphi_j \wedge P$ for all $j \in \{1, ..., k\}$, then there exist configurations $c'_1, ..., c'_k \in A_{Cfg}$ such that $c_j \overset{*}{\underset{\boldsymbol{R}}{\Rightarrow}} c'_j$ for all $j \in \{1, ..., k\}$, and there also exists an A-valuation ρ_2 satisfying $\rho_2(v) = \rho_1(v)$ for every $v \notin \overline{Y}$, and $c'_j, \rho_2 \vDash_A^{\mathrm{ML}} \varphi'_j \wedge P'$ for every $j \in \{1, ..., k\}$.

Let us have such configurations $c_1, ..., c_k \in A_{Cfg}$. The goal is to show that

there exist configurations $c'_1, ..., c'_k \in A_{Cfg}$ such that $c_j \overset{*}{\underset{\boldsymbol{R}}{\Rightarrow}} c'_j$ for all $j \in \{1, ..., k\}$, and there also exists an A-valuation ρ_2 satisfying $\rho_2(v) = \rho_1(v)$ for every $v \notin \overline{Y}$, and $c'_j, \rho_2 \vDash_A^{\mathrm{ML}} \varphi'_j \wedge P'$ for every $j \in \{1, ..., k\}$.

The proof will proceed as follows:
1. prove the premise of (iv) for $C := [c_1, ..., c_k]$;
2. decompose the obtained C' into $[c'_1, ..., c'_k]$; and
3. prove the desired properties of c'_j from properties of C'.

First, $C = [c_1, ..., c_k]$ is terminating by Lemma 51. Next, we have to show that

$$[c_1, ..., c_k], \rho \vDash_{A^*}^{\mathrm{ML}} \mathit{flatten}((\varphi_1, ..., \varphi_k) \wedge P)$$

where ρ is some arbitrary but fixed extension of ρ_1 to variables of Ω^*. By Lemma 55, this is equivalent to showing that

there exist configurations $\overline{c}_1, ..., \overline{c}_k \in A_{Cfg}$ such that $[c_1, ..., c_k] = [\overline{c}_1, ..., \overline{c}_k]$ and there exist an A-valuation ρ_0 satisfying $\rho_0(v) = \rho(v)$ for all v such that for any $j \in \{1, ..., k\}$ it holds that $c_j, \rho_0 \vDash_A^{\mathrm{ML}} \varphi_j \wedge P$.

Let us choose $\overline{c}_j := c_j$ and $\rho_0 := \rho_1$; then,

$$c_j, \rho_0 \vDash_A^{\mathrm{ML}} \varphi_j \wedge P$$

holds by assumption. We have obtained

there exists some $C' \in A^*_{Cfg^*}$ such that $[c_1, ..., c_k] \overset{*}{\underset{\boldsymbol{R}^*}{\Rightarrow}} C'$ and $C', \rho \vDash_{A^*}^{\mathrm{ML}} \mathit{flatten}(\Psi')$.

By Lemma 46 and induction on the length of the sequence witnessing the reachability, we get some $c'_1, ..., c'_k \in A_{Cfg}$ such that

$[c_1, ..., c_k] \overset{*}{\underset{\boldsymbol{R}^*}{\Rightarrow}} [c'_1, ..., c'_k]$ and $[c'_1, ..., c'_k], \rho \vDash^{\mathrm{ML}}_{A^*} \mathit{flatten}(\Psi')$.

The main goal is still to show that

there exist configurations $c'_1, ..., c'_k \in A_{Cfg}$ such that $c_j \overset{*}{\underset{\boldsymbol{R}}{\Rightarrow}} c'_j$ for all $j \in \{1, ..., k\}$, and there also exists an A-valuation ρ_2 satisfying $\rho_2(v) = \rho_1(v)$ for every $v \notin \overline{Y}$, and $c'_j, \rho_2 \vDash^{\mathrm{ML}}_{A} \varphi'_j \wedge P'$ for every $j \in \{1, ..., k\}$,

so let us choose $c'_j := c'_j$ for each $j \in \{1, ..., k\}$; we have to prove that

$c_j \overset{*}{\underset{\boldsymbol{R}}{\Rightarrow}} c'_j$ for all $j \in \{1, ..., k\}$, and there also exists an A-valuation ρ_2 satisfying $\rho_2(v) = \rho_1(v)$ for every $v \notin \overline{Y}$, and $c'_j, \rho_2 \vDash^{\mathrm{ML}}_{A} \varphi'_j \wedge P'$ for every $j \in \{1, ..., k\}$.

The first conjunct follows by Lemma 52. It remains to be proven that

there exists an A-valuation ρ_2 satisfying $\rho_2(v) = \rho_1(v)$ for every $v \notin \overline{Y}$, and $c'_j, \rho_2 \vDash^{\mathrm{ML}}_{A} \varphi'_j \wedge P'$ for every $j \in \{1, ..., k\}$.

and we already know that

$[c'_1, ..., c'_k], \rho \vDash^{\mathrm{ML}}_{A^*} \mathit{flatten}(\Psi')$.

By Lemma 55, we know that

there exists an A-valuation ρ_0 satisfying $\rho_0(v) = \rho(v)$ for all $v \notin \overline{Y}$ such that $c'_j, \rho_0 \vDash^{\mathrm{ML}}_{A} \varphi'$ for every $j \in \{1, ..., k\}$.

Let ρ'_0 be such valuation. We let $\rho_2(v) := \rho'_0(v)$ for every v, and obtain $\rho_2(v) = \rho'_0(v) = \rho(v) = \rho_1(v)$ for all $v \notin \overline{Y}$ by definitions. The rest of the goal follows from the assumption by Lemma 17. This concludes the proof.

□

The price paid for self-composition is that the property of the self-composed program is often hard to reason about. Therefore, M. Sousa and I. Dillig [90] does not apply self-composition directly, but only use its soundness to justify their technique – namely, the soundness of their proof system. Their proof system avoids any explicit construction of self-composed programs; the proof system

$$\frac{\boldsymbol{R} \vdash_{CRL} \Psi \Downarrow_{C \cup \{\Psi\}, E} \Psi'}{\boldsymbol{R} \vdash_{CRL} \Psi \Downarrow_{C,E} \Psi'} \text{Circularity}$$

$$\frac{\begin{array}{c}\varphi \Rightarrow \varphi' \in R \quad P' \text{ is an FOL formula} \quad \vDash_{A}^{\mathrm{ML}} \varphi_1 \leftrightarrow \varphi \wedge P' \\ A, R \vdash_{CRL} (\varphi_1, ..., \varphi_{i-1}, \varphi' \wedge P', \varphi_{i+1}, ..., \varphi_k) \wedge P \Downarrow_{\emptyset, C \cup E} \Psi'\end{array}}{A, R \vdash_{CRL} (\varphi_1, ..., \varphi_{i-1}, \varphi_i, \varphi_{i+1}, ..., \varphi_k) \wedge P \Downarrow_{C,E} \Psi'} \text{Step}$$

$$\frac{\Psi \in E}{\boldsymbol{R} \vdash_{CRL} \Psi \Downarrow_{C,E} \Psi'} \text{Axiom}$$

$$\frac{}{\boldsymbol{R} \vdash_{CRL} \Psi \Downarrow_{\emptyset,E} \Psi} \text{Reflexivity}$$

$$\frac{\begin{array}{c}\boldsymbol{R} \vdash_{CRL} (\varphi_1, ..., \varphi_{i-1}, \varphi_i, \varphi_{i+1}, ..., \varphi_k) \wedge P \Downarrow_{C,E} \Psi' \\ \boldsymbol{R} \vdash_{CRL} (\varphi_1, ..., \varphi_{i-1}, \psi_i, \varphi_{i+1}, ..., \varphi_k) \wedge P \Downarrow_{C,E} \Psi'\end{array}}{\boldsymbol{R} \vdash_{CRL} (\varphi_1, ..., \varphi_{i-1}, (\varphi_i \vee \psi_i), \varphi_{i+1}, ..., \varphi_k) \wedge P \Downarrow_{C,E} \Psi'} \text{Case}$$

$$\frac{\vDash_{A^*}^{\mathrm{ML}} \mathit{flatten}(\Phi) \rightarrow \mathit{flatten}(\Phi') \quad A, R \vdash_{CRL} \Phi' \Downarrow_{C,E} \Psi'}{A, R \vdash_{CRL} \Phi \Downarrow_{C,E} \Psi'} \text{Conseq}$$

$$\frac{X \notin \mathrm{FV}(\Psi') \quad \boldsymbol{R} \vdash_{CRL} \exists \overline{Y}.\ (\varphi_1, ..., \varphi_k) \wedge P \Downarrow_{C,E} \Psi'}{\boldsymbol{R} \vdash_{CRL} \exists X, \overline{Y}.\ (\varphi_1, ..., \varphi_k) \wedge P \Downarrow_{C,E} \Psi'} \text{Abstract}$$

$$\frac{(A^*, R^* \cup \{\mathit{flatten}(e) \Rightarrow \mathit{flatten}(\Psi') \mid e \in E\}) \underset{\mathrm{RL}}{\vdash} \mathit{flatten}(\Psi) \overset{\exists}{\Rightarrow} \mathit{flatten}(\Psi')}{(A, R) \vdash_{CRL} \Psi \Downarrow_{C,E} \Psi'} \text{Reduce}$$

Figure 9: The proof system of Cartesian Reachability Logic.

for Cartesian Reachability logic generalizes this to work for every deterministic language.

III.3.3 Proof system for CRL

While the intuition behind the semantics of CRL is similar to that of CHL, with only a few minor differences, coming up with a proof system for CRL is not straightforward. One could attempt to reuse the proof system of CHL and modify it somehow to be independent of the particular programming language. Since many CHL rules (e.g., its If rule shown in Figure 7) are simply Hoare logic rules acting on a particular component of a tuple of formulas (that is, symbolic states), one could, for example, lift rules of reachability logic (RL) to the tuple-context and be done. But even if it were sound and complete, it could not be competetive with Cartesian Hoare logic in terms of simplicity of reasoning: the distingushing feature of CHL is its ability to simplify reasoning by performing lockstep execution of loops, because that can, as [90] puts it, “greatly simplify the verification task (e.g., by requiring simpler invariants)”.

The proof system of CRL is displayed in Figure 9. Let us now discuss some of its features.

- The proof system derives sequents of the shape $\boldsymbol{R} \vdash_{CRL} \Phi \Downarrow_{C,E} \Psi$, where Φ, Ψ are of the shape $\exists \overline{X}.\ (\varphi_1, ..., \varphi_k) \wedge P$.
  - Φ represents a *premise*;
  - Ψ represents a *conclusion* and propagates through the proof system unchanged;
  - φ_i for $i \in \{1, ..., k\}$ is a matching logic pattern representing a particular component;
  - P is an FOL formula (*global constraint*) relating variables from different components;
  - C and E together implement the notion of *invariant*: C is a set of *synchronization points* (aka *circularities*), representing recently postulated invariants, and E is a set of *enabled synchronization points*, representing invariants that are ready to be used.

  A proof search starts with $E = C = \emptyset$, which will become clear when we take a look at the soundness theorem. Similarly to *circularities* of reachability logic found in G. Roşu and A. Stefanescu [81], synchronization points allow one to reason coinductively about (cartesian) reachability claims: one can introduce them at any point in a proof, but they can be used only after progress has been made by means of *symbolic execution* (the Step rule in CRL). The progress requirements resemble the concept of *productivity* of co-recursive definitions; a formal treatment of the relationship between coinduction and reachability logic is to be found in B. M. Moore, L. Peña, and G. Roşu [71].

- The proof system rules are *general* in the sense that they do not have the semantics of any particular language construct hard-wired into them. This is in contrast with the rules of Cartesian Hoare Logic of M. Sousa and I. Dillig [90], which uses five fairly complex rules only to support a single programming language construct that is essentially a `while` loop with `breaks` inside.

Before showing an example proof using the system, let us first discuss the general purpose of the individual rules.

- The Circularity rule is a key rule which allows lockstep reasoning about arbitrary program constructs. It allows the user to postulate the validity of the current claim by means of adding the current premise into the set of synchronization points, from which a k-tuple of program configurations satisfying the postcondition is claimed to be reachable. Once progress is made (by means of the Step rule), the added synchronization points are *enabled* and can be used to finish the proof using the Axiom rule.

- The Step rule performs symbolic execution on the selected i-th component (represented by the formula φ_i) using the semantic rule $\varphi \Rightarrow \varphi' \in R$. For this rule to apply, its left-hand side φ has to match all the program configurations matching φ_i. Therefore, the rule (semantically) decomposes φ_i into φ and an additional FOL formula P', which can be thought of as a part of *path condition* that is local to the i-th component; P' is then used to constrain the right-side of

φ' of the selected rule. This proof rule also enables the synchronization points from C by adding them to E.

- The Axiom rule uses an enabled synchronization point to finish the proof.
- The Reflexivity rule can be used to finish a proof when the premise corresponds to the conclusion.
- The Case rule implements case analysis on a selected component i.
- The Conseq rule is used to weaken (or generalize) the premise. It can also be used to propagate information between components and the global constraint.
- The Abstract rule can be used to remove existential quantifiers from the premise. Intuitively, this corresponds to a proof step in first-order logic that replaces an existential quantifier on the left-hand side of an implication with a universal quantifier over the implication, assuming that the variable bound by the existential quantifier does not occur free in the right-hand side. (In our setting, the typical way of *obtaining* a top-level existential quantifier in the premise is by means of the Conseq rule.)
- The Reduce rule is a way to get completeness into our proof system: it reduces the goal to reachability logic reasoning. This rule also provides a way to prove properties that do not benefit from lockstep reasoning.

III.3.4 An example CRL proof

An example proof sketch using the proof system above follows. The proof involves lockstep reasoning. To ease the notation, we write simply t_b instead of $t_b = true$ for boolean-sorted side conditions t_b. Consider again the claim Ω_{mono} as in Notation 39.

$$[\xi(P, X_1, Y_1), \xi(P, X_2, Y_2)] \wedge X_1 \leq X_2 \wedge Y_1 \leq Y_2$$
$$\overset{c\exists}{\Rightarrow}$$
$$[\xi(\varepsilon, ?_, ?\, Y_1'), \xi(\varepsilon, ?_, ?\, Y_2')] \wedge Y_1' \leq Y_2'$$

where P is the program

```
1  while(x > 0) { x--; y++; }
```

Let Ψ'_{mono} denote the right-hand side of the $\overset{c\exists}{\Rightarrow}$ above. Let us also recall the definition of ξ from Example 36; that is, $\xi(C, X, Y)$ is the pattern

$$\ll C \mid \mathsf{x} \mapsto X, \mathsf{y} \mapsto Y \gg$$

(representing configurations of the simple imperative programming language IMP where C is to be executed and the program variable x (y) has a value matching the pattern X (Y)).

Assume $\boldsymbol{R}_{imp} = (A_{imp}, R_{imp})$ is a reachability system modelling IMP; the precise details are not important here, because we will use the reachability system on a more abstract level to avoid cluttering the proof sketch with minor details of how *exactly* is the language modelled. We want to prove

$$\boldsymbol{R}_{imp} \vdash_{CRL} (\xi(P, X_1, Y_1), \xi(P, X_2, Y_2)) \wedge X_1 \underset{Int}{\leq} X_2 \wedge Y_1 \underset{Int}{\leq} Y_2 \Downarrow_{\emptyset,\emptyset} \Psi'_{mono} \ .$$

Intuitively, we would like to encode a relational invariant saying that the program variable `x` is in one execution smaller than in the other, and similarly for the program variable `y`. The invariant needs to be more general than the "current state", that is, the left-hand side of $\Downarrow_{\emptyset,\emptyset}$. Thus, we first apply the Conseq rule to change the goal to

$$\begin{aligned}\boldsymbol{R}_{imp} \vdash_{CRL} \exists X_1, Y_1, X_2, Y_2.\ (\xi(P, X_1, Y_1), \xi(P, X_2, Y_2)) \\ \wedge X_1 \underset{Int}{\leq} X_2 \wedge Y_1 \underset{Int}{\leq} Y_2 \Downarrow_{\emptyset,\emptyset} \Psi'_{mono}\ ,\end{aligned}$$

then apply the Circularity rule, followed by application of the Abstract rule, which basically changes the goal back, except that now we have a general synchronization point. The goal is now

$$\begin{aligned}\boldsymbol{R}_{imp} \vdash_{CRL} (\xi(P, X_1, Y_1), \xi(P, X_2, Y_2)) \\ \wedge X_1 \underset{Int}{\leq} X_2 \wedge Y_1 \underset{Int}{\leq} Y_2 \Downarrow_{\{\Phi_{sync}\},\emptyset} \Psi'_{mono},\end{aligned}$$

where

$$\Phi_{sync} \equiv \exists X_1, Y_1, X_2, Y_2.\ (\xi(P, X_1, Y_1), \xi(P, X_2, Y_2)) \wedge X_1 \underset{Int}{\leq} X_2 \wedge Y_1 \underset{Int}{\leq} Y_2.$$

This Conseq, Circularity, Abstract pattern is so frequent that we can have a standalone derived proof rule for it.

Lemma 57: Generalize with Circularity

The rule

$$\frac{\overline{X} \equiv \mathrm{FV}(\Phi) \setminus \mathrm{FV}(\Psi') \qquad \boldsymbol{R} \vdash_{CRL} \Phi \Downarrow_{C \cup \{\exists \overline{X}.\ \Phi\}, E} \Psi'}{\boldsymbol{R} \vdash_{CRL} \Phi \Downarrow_{C,E} \Psi'}\ \text{GenWithCirc}$$

follows from the CRL proof system.

Proof: Apply Conseq, then apply Circularity, then apply Abstract. □

Now, in our quest to prove the example claim, we perform symbolic execution on both components, using repeated applications of the Step rule, also enabling the synchronization point as we go. The exact details depend on the exact rules of $\boldsymbol{R}_{imp}$, but assuming that the semantics of the while statement is defined by unrolling into the `if` statement, we end up with a goal like

$$\begin{aligned}\boldsymbol{R}_{imp} \vdash_{CRL} \Big(&\xi\Big(\texttt{if(}X_1 \underset{Int}{>} 0\texttt{){y++;x--; }}P\texttt{ }\},X_1,Y_1\Big),\\ &\xi\Big(\texttt{if(}X_2 \underset{Int}{>} 0\texttt{){y++;x--; }}P\texttt{ }\},X_2,Y_2\Big)\Big)\\ &\wedge X_1 \underset{Int}{\leq} X_2 \wedge Y_1 \underset{Int}{\leq} Y_2 \Downarrow_{\emptyset,\{\Phi_{sync}\}} \Psi'_{mono}\end{aligned}$$

Now we want to perform a case analysis. To do so, we first use the Conseq rule to obtain disjunctions of patterns at the respective components; then we repeatedly apply the Case rule, leading to four goals; finally, we use the Conseq rule to propagate the constraints from the components to the global constraint. The Conseq rule is a bit unweildy, as it requires one to prove properties about flattened formulas in the star-extended algebra. However, we can abstract away this problem by means of the following lemma.

Lemma 58: Consequences of Conseq

The following four rules are consequences of the CRL proof system.

$$\frac{\begin{array}{c}\models^{\mathrm{ML}}_{A} \varphi_i \to \varphi'_i\\ (A,R) \vdash_{CRL} \exists\overline{X}.\ (\varphi_1,...,\varphi_{i-1},\varphi'_i,\varphi_{i+1},...,\varphi_k) \wedge P \Downarrow_{C,E} \Psi'\end{array}}{(A,R) \vdash_{CRL} \exists\overline{X}.\ (\varphi_1,...,\varphi_{i-1},\varphi_i,\varphi_{i+1},...,\varphi_k) \wedge P \Downarrow_{C,E} \Psi'}\ \text{ConseqL}$$

$$\frac{\models^{\mathrm{ML}}_{A} P \to P' \qquad (A,R) \vdash_{CRL} \exists\overline{X}.\ (\varphi_1,...,\varphi_k) \wedge P' \Downarrow_{C,E} \Psi'}{(A,R) \vdash_{CRL} \exists\overline{X}.\ (\varphi_1,...,\varphi_k) \wedge P \Downarrow_{C,E} \Psi'}\ \text{ConseqG}$$

$$\frac{\begin{array}{c}P \text{ is an FOL formula}\\ \boldsymbol{R} \vdash_{CRL} (\varphi_1,...,\varphi_{i-1},\varphi_i,\varphi_{i+1},...,\varphi_k) \wedge (P \wedge P') \Downarrow_{C,E} \Psi'\end{array}}{\boldsymbol{R} \vdash_{CRL} (\varphi_1,...,\varphi_{i-1},\varphi_i \wedge P',\varphi_{i+1},...,\varphi_k) \wedge P \Downarrow_{C,E} \Psi'}\ \text{PropOut}$$

$$\frac{\begin{array}{c}P \text{ is an FOL formula}\\ \boldsymbol{R} \vdash_{CRL} (\varphi_1,...,\varphi_{i-1},\varphi_i \wedge P',\varphi_{i+1},...,\varphi_k) \wedge P \Downarrow_{C,E} \Psi'\end{array}}{\boldsymbol{R} \vdash_{CRL} (\varphi_1,...,\varphi_{i-1},\varphi_i,\varphi_{i+1},...,\varphi_k) \wedge (P \wedge P') \Downarrow_{C,E} \Psi'}\ \text{PropIn}$$

Proof:

- ConseqL follows from Conseq. We only need to prove that

$$\models^{\mathrm{ML}}_{A} \varphi_i \to \varphi'_i$$

implies

$$\begin{aligned}\models^{\mathrm{ML}}_{A^*} \mathit{flatten}\Big(\exists\overline{X}.\ (\varphi_1,...,\varphi_{i-1},\varphi_i,\varphi_{i+1},...,\varphi_k) \wedge P\Big) \to\\ \mathit{flatten}\Big(\exists\overline{X}.\ (\varphi_1,...,\varphi_{i-1},\varphi'_i,\varphi_{i+1},...,\varphi_k) \wedge P\Big)\end{aligned}$$

which, after unfolding *flatten*, follows from Lemma 23.

- ConseqG follows from Conseq. We need to prove that

$$\models^{\mathrm{ML}}_{A} P \to P'$$

implies

$$\models^{\mathrm{ML}}_{A^*} \mathit{flatten}\Big(\exists \overline{X}.\ (\varphi_1, ..., \varphi_k) \wedge P\Big) \to \\ \mathit{flatten}\Big(\exists \overline{X}.\ (\varphi_1, ..., \varphi_k) \wedge P'\Big)$$

which follows immediatelly after unfolding *flatten*.

- PropIn, PropOut – follows from Conseq after unfolding *flatten* and using Lemma 22.

□

Now we can keep all the reasoning in the original algebra.

The four subgoals will be as follows (the differences are shown in blue):

$$\begin{aligned} \boldsymbol{R}_{imp} \vdash_{CRL} & (\xi(\texttt{if(}\mathit{true}\texttt{)\{y++;x--;}\ P\ \texttt{\}}, X_1, Y_1), \\ & \xi(\texttt{if(}\mathit{true}\texttt{)\{y++;x--;}\ P\ \texttt{\}}, X_2, Y_2)) \\ & \wedge X_1 \underset{Int}{\leq} X_2 \wedge Y_1 \underset{Int}{\leq} Y_2 \wedge X_1 \underset{Int}{>} 0 \wedge X_2 \underset{Int}{>} 0 \Downarrow_{\emptyset, \{\Phi_{sync}\}} \Psi'_{mono} \end{aligned} \tag{2}$$

$$\begin{aligned} \boldsymbol{R}_{imp} \vdash_{CRL} & (\xi(\texttt{if(}\mathit{true}\texttt{)\{y++;x--;}\ P\ \texttt{\}}, X_1, Y_1), \\ & \xi(\texttt{if(}\mathit{false}\texttt{)\{y++;x--;}\ P\ \texttt{\}}, X_2, Y_2)) \\ & \wedge X_1 \underset{Int}{\leq} X_2 \wedge Y_1 \underset{Int}{\leq} Y_2 \wedge X_1 \underset{Int}{>} 0 \wedge X_2 \underset{Int}{\leq} 0 \Downarrow_{\emptyset, \{\Phi_{sync}\}} \Psi'_{mono} \end{aligned} \tag{3}$$

$$\begin{aligned} \boldsymbol{R}_{imp} \vdash_{CRL} & (\xi(\texttt{if(}\mathit{false}\texttt{)\{y++;x--;}\ P\ \texttt{\}}, X_1, Y_1), \\ & \xi(\texttt{if(}\mathit{true}\texttt{)\{y++;x--;}\ P\ \texttt{\}}, X_2, Y_2)) \\ & \wedge X_1 \underset{Int}{\leq} X_2 \wedge Y_1 \underset{Int}{\leq} Y_2 \wedge X_1 \underset{Int}{\leq} 0 \wedge X_2 \underset{Int}{>} 0 \Downarrow_{\emptyset, \{\Phi_{sync}\}} \Psi'_{mono} \end{aligned} \tag{4}$$

$$\begin{aligned} \boldsymbol{R}_{imp} \vdash_{CRL} & (\xi(\texttt{if(}\mathit{false}\texttt{)\{y++;x--;}\ P\ \texttt{\}}, X_1, Y_1), \\ & \xi(\texttt{if(}\mathit{false}\texttt{)\{y++;x--;}\ P\ \texttt{\}}, X_2, Y_2)) \\ & \wedge X_1 \underset{Int}{\leq} X_2 \wedge Y_1 \underset{Int}{\leq} Y_2 \wedge X_1 \underset{Int}{\leq} 0 \wedge X_2 \underset{Int}{\leq} 0 \Downarrow_{\emptyset, \{\Phi_{sync}\}} \Psi'_{mono} \end{aligned} \tag{5}$$

- The case in Equation 5 represents the situation when both loops have finished their execution. We can solve this case by symbolically executing both programs (using the Step rule) to the end. It is easy to see that then the premise implies the conclusion; therefore, we finish this case by generalizing the premise (using the Conseq rule) to be exactly the conclusion, and then applying the Reflexivity rule.
- The case in Equation 3 represents the situation when the first loop continues execution and the second is finished. This can never happen – we see that the

side condition is contradictory. We finish this case using the Reduce rule. We can also prove the following lemma.

Lemma 59: Principle of explosion

The following rule is a consequence of the CRL proof system.

$$\frac{}{\boldsymbol{R} \vdash_{CRL} (\varphi_1, ..., \varphi_k) \wedge \mathit{false} \Downarrow_{C,E} \Psi'}\ \text{ExFalso}$$

Proof: Use the Reduce rule. □

- The case in Equation 4 represents the complementary situation - the first loop has finished its execution, while the second loop continues. This requires inventing an invariant capturing the idea that the execution of the second loop can only increase the difference between the values of y. We can Reduce this subgoal to simple RL reasoning.
- The case in Equation 2 is the one when we utilize the ability to perform lockstep reasoning. We symbolically execute both components until their program parts become the while loops again; that is, P. The goal is now

$$\begin{aligned}\boldsymbol{R}_{imp} \vdash_{CRL} &(\xi(P, X_1 -_{Int} 1, Y_1 +_{Int} 1), \xi(P, X_2 -_{Int} 1, Y_2 +_{Int} 1),\\ &\wedge X_1 \underset{Int}{\leq} X_2 \wedge Y_1 \underset{Int}{\leq} Y_2 \wedge X_1 \underset{Int}{>} 0 \wedge X_2 \underset{Int}{>} 0 \Downarrow_{\emptyset,\{\Phi_{sync}\}} \Psi'_{mono},\end{aligned}$$

  which implies the synchronization point Φ_{sync}, which is already enabled. We can therefore conclude the proof using Conseq and Axiom.

One can observe that the proof is rather low-level. On the other hand, the proof system is very simple, and as we have seen, it is easy to come up with derived rules that raise the level of abstraction.

III.3.5 CRL soundness

Now, I want to state and prove the soundness theorem of Cartesian reachability logic. The proof depends on three key ingrediences: (1) a translation of CRL proofs to RL proofs over star-extended system; (2) the soundness of reachability logic; and (3) Theorem 56. First, let us discuss the proof translation.

Lemma 60: CRL proof translation

We can translate CRL proofs to RL proofs over star-extended system.

$$\begin{aligned}&(A, R) \vdash_{CRL} \Psi \Downarrow_{C,E} \Psi' \Rightarrow\\ &(A^*, R^* \cup \{\mathit{flatten}(e) \Rightarrow \mathit{flatten}(\Psi') \mid e \in E\}),\\ &\{\mathit{flatten}(c) \Rightarrow \mathit{flatten}(\Psi') \mid c \in C\}\\ &\underset{\text{RL}}{\vdash} \mathit{flatten}(\Psi) \overset{\exists}{\Rightarrow} \mathit{flatten}(\Psi')\end{aligned}$$

Proof: We will use the RL proof system shown in Figure 4, and proceed by induction on the CRL proof.

- If the proof ends with Reduce, we are done.
- If the proof ends with Reflexivity, then we need to prove

$$R^*, \{flatten(e) \Rightarrow flatten(\Psi) \mid e \in E\}, \emptyset \underset{\text{RL}}{\vdash} flatten(\Psi) \overset{\exists}{\Rightarrow} flatten(\Psi)$$

  which we do by applying the Reflexivity rule of RL.
- If the proof ends with Axiom, then $\Psi \in E$, and we have to prove that

$$\begin{aligned} & R^*, \{flatten(e) \Rightarrow flatten(\Psi') \mid e \in E\}, \\ & \quad \{flatten(c) \Rightarrow flatten(\Psi') \mid c \in C\} \\ & \underset{\text{RL}}{\vdash} flatten(\Psi) \overset{\exists}{\Rightarrow} flatten(\Psi') \ . \end{aligned}$$

  By applying the proof rule Axiom of RL, it is enough to show that

$$flatten(\Psi) \Rightarrow flatten(\Psi') \in \{flatten(e) \Rightarrow flatten(\Psi') \mid e \in E\}$$

  which follows from $\Psi \in E$.
- If the proof ends with Case, then we have

$$R^* \cup E', C' \underset{\text{RL}}{\vdash} flatten((\varphi_1, ..., \varphi_{i-1}, \varphi_i, \varphi_{i+1}, ..., \varphi_k)) \overset{\exists}{\Rightarrow} flatten(\Psi')$$

  and

$$R^* \cup E', C' \underset{\text{RL}}{\vdash} flatten((\varphi_1, ..., \varphi_{i-1}, \psi, \varphi_{i+1}, ..., \varphi_k)) \overset{\exists}{\Rightarrow} flatten(\Psi')$$

  as hypotheses, and we have to prove

$$R^* \cup E', C' \underset{\text{RL}}{\vdash} flatten((\varphi_1, ..., \varphi_{i-1}, \varphi_i \vee \psi, \varphi_{i+1}, ..., \varphi_k)) \overset{\exists}{\Rightarrow} flatten(\Psi') \ ,$$

  where

$$E' = \{flatten(e) \Rightarrow flatten(\Psi') \mid e \in E\}$$

  and

$$C' = \{flatten(c) \Rightarrow flatten(\Psi') \mid c \in C\} \ .$$

  After simplifications, we get

$$\begin{aligned} R^* \cup E', C' \underset{\text{RL}}{\vdash} & mkList(X_1, ..., X_k) \wedge \left(\bigwedge_{j=1}^{k} \left(\varphi_j^{\square} \left[\frac{X_j}{\square} \right] \right) \right) \wedge P' \\ & \overset{\exists}{\Rightarrow} flatten(\Psi') \end{aligned}$$

and

$$R^* \cup E', C' \underset{\text{RL}}{\vdash} mkList(Y_1, ..., Y_k) \wedge \left(\bigwedge_{j=1}^{k} \left(\varphi_j^{\Box} \left[\frac{Y_j}{\Box} \right] \right) \right) \wedge \psi^{\Box} \left[\frac{Y_i}{\Box} \right] \wedge P' \overset{\exists}{\Rightarrow} flatten(\Psi')$$

as hypotheses, and have to prove that

$$R^* \cup E', C' \underset{\text{RL}}{\vdash} \left(mkList(Z_1, ..., Z_k) \wedge \left(\bigwedge_{j=1}^{k} \left(\varphi_j^{\Box} \left[\frac{Z_j}{\Box} \right] \right) \right) \wedge \varphi_i^{\Box} \left[\frac{Z_i}{\Box} \right] \wedge P' \right) \vee \left(mkList(Z_1, ..., Z_k) \wedge \left(\bigwedge_{j=1}^{k} \left(\varphi_j^{\Box} \left[\frac{Z_j}{\Box} \right] \right) \right) \wedge \psi^{\Box} \left[\frac{Z_i}{\Box} \right] \wedge P' \right) \overset{\exists}{\Rightarrow} flatten(\Psi')$$

(where $X_1, ..., X_k, Y_1, ..., Y_k, Z_1, ..., Z_k$ are fresh variables). Now we apply the Case Analysis rule. Then we transform the hypotheses to the respective goals by existentially quantifying the X_js (and Y_js, respectively) in the hypotheses using the Abstraction RL rule, alpha-renaming (using the Consequence rule) the X_js (and Y_js, respectively) into Z_js, and stripping the existential quantifiers (using the Consequence rule, again), and we are done.

- If the proof ends with Step, we can assume an FOL formula P', a rule $\varphi \Rightarrow \varphi' \in R$ such that $\models_A^{\text{ML}} \varphi_i \leftrightarrow \varphi \wedge P'$, and an inductive hypothesis

  $$R^* \cup \{flatten(e) \Rightarrow flatten(\Psi') \mid e \in C \cup E\}, \emptyset \underset{\text{RL}}{\vdash} flatten((\varphi_1, ..., \varphi_{i-1}, \varphi' \wedge P', \varphi_{i+1}, ..., \varphi_k) \wedge P) \overset{\exists}{\Rightarrow} flatten(\Psi')$$

  and have to construct a proof of

  $$R^* \cup \{flatten(e) \Rightarrow flatten(\Psi') \mid e \in E\}, \{flatten(c) \Rightarrow flatten(\Psi') \mid c \in C\} \underset{\text{RL}}{\vdash} flatten((\varphi_1, ..., \varphi_{i-1}, \varphi_i, \varphi_{i+1}, ..., \varphi_k) \wedge P) \overset{\exists}{\Rightarrow} flatten(\Psi') \ .$$

  By Definition 42, we also have

  $$(heat(L, \varphi, R) \Rightarrow heat(L, \varphi', R)) \in R^* \ .$$

  We apply the Transitivity rule with the second premise being our first inductive hypothesis, and it remains to prove the second premise, which is

$$
\begin{array}{l}
R^*, \{\mathit{flatten}(e) \Rightarrow \mathit{flatten}(\Psi') \mid e \in E\}, \\
\{\mathit{flatten}(c) \Rightarrow \mathit{flatten}(\Psi') \mid c \in C\} \\
\underset{\mathrm{RL}}{\vdash} \\
\mathit{flatten}((\varphi_1, ..., \varphi_{i-1}, \varphi_i, \varphi_{i+1}, ..., \varphi_k) \wedge P) \\
\overset{\exists}{\Rightarrow} \\
\mathit{flatten}((\varphi_1, ..., \varphi_{i-1}, \varphi' \wedge P', \varphi_{i+1}, ..., \varphi_k) \wedge P)
\end{array}
$$

that is (after simplification, assuming a reasonable choice of fresh variables)

$$
\begin{array}{l}
R^*, \{\mathit{flatten}(e) \Rightarrow \mathit{flatten}(\Psi') \mid e \in E\}, \\
\{\mathit{flatten}(c) \Rightarrow \mathit{flatten}(\Psi') \mid c \in C\} \\
\underset{\mathrm{RL}}{\vdash} \\
\mathit{mkList}(X_1, ..., X_k) \wedge \left(\bigwedge_{j=1, j \neq i}^{k} \left(\varphi_j^{\Box}\right) \left[\frac{X_j}{\Box}\right] \right) \wedge \left(\varphi_i^{\Box}\right) \left[\frac{X_i}{\Box}\right] \wedge P \\
\overset{\exists}{\Rightarrow} \\
\mathit{mkList}(X_1, ..., X_k) \wedge \left(\bigwedge_{j=1, j \neq i}^{k} \left(\varphi_j^{\Box}\right) \left[\frac{X_j}{\Box}\right] \right) \wedge \left((\varphi' \wedge P')^{\Box}\right) \left[\frac{X_i}{\Box}\right] \wedge P \ .
\end{array}
$$

By Lemma 24, our assumption of $\vDash_A^{\mathrm{ML}} \varphi_i \leftrightarrow \varphi \wedge P'$, we can apply the Consequence rule, and the goal changes to

$$
\begin{array}{l}
R^*, \{\mathit{flatten}(e) \Rightarrow \mathit{flatten}(\Psi') \mid e \in E\}, \\
\{\mathit{flatten}(c) \Rightarrow \mathit{flatten}(\Psi') \mid c \in C\} \\
\underset{\mathrm{RL}}{\vdash} \\
\mathit{mkList}(X_1, ..., X_k) \wedge \left(\bigwedge_{j=1, j \neq i}^{k} \left(\varphi_j^{\Box}\right) \left[\frac{X_j}{\Box}\right] \right) \wedge \left((\varphi \wedge P')^{\Box}\right) \left[\frac{X_i}{\Box}\right] \wedge P \\
\overset{\exists}{\Rightarrow} \\
\mathit{mkList}(X_1, ..., X_k) \wedge \left(\bigwedge_{j=1, j \neq i}^{k} \left(\varphi_j^{\Box}\right) \left[\frac{X_j}{\Box}\right] \right) \wedge \left((\varphi' \wedge P')^{\Box}\right) \left[\frac{X_i}{\Box}\right] \wedge P \ .
\end{array}
$$

Apply the Consequence rule again, changing the goal to

$$
\begin{array}{l}
R^*, \{flatten(e) \Rightarrow flatten(\Psi') \mid e \in E\}, \\
\{flatten(c) \Rightarrow flatten(\Psi') \mid c \in C\} \\
\underset{\mathrm{RL}}{\vdash} \\
mkList(X_1, ..., X_k) \wedge (\varphi)^{\Box}\left[\frac{X_i}{\Box}\right] \\
\wedge \left(\bigwedge_{j=1, j\neq i}^{k} \left(\varphi_j^{\Box}\right)\left[\frac{X_j}{\Box}\right]\right) \wedge \left((P')^{\Box}\right)\left[\frac{X_i}{\Box}\right] \wedge P \\
\overset{\exists}{\Rightarrow} \\
mkList(X_1, ..., X_k) \wedge (\varphi')^{\Box}\left[\frac{X_i}{\Box}\right] \\
\wedge \left(\bigwedge_{j=1, j\neq i}^{k} \left(\varphi_j^{\Box}\right)\left[\frac{X_j}{\Box}\right]\right) \wedge \left((P')^{\Box}\right)\left[\frac{X_i}{\Box}\right] \wedge P \ .
\end{array}
$$

Now we apply Logic Framing to remove the structureless parts that are the same in both the left and right-hand sides, resulting in the goal

$$
\begin{array}{l}
R^*, \{flatten(e) \Rightarrow flatten(\Psi') \mid e \in E\}, \\
\{flatten(c) \Rightarrow flatten(\Psi') \mid c \in C\} \\
\underset{\mathrm{RL}}{\vdash} \\
mkList(X_1, ..., X_k) \wedge (\varphi)^{\Box}\left[\frac{X_i}{\Box}\right] \\
\overset{\exists}{\Rightarrow} \\
mkList(X_1, ..., X_k) \wedge (\varphi')^{\Box}\left[\frac{X_i}{\Box}\right] \ .
\end{array}
$$

From

$$(heat(L, \varphi, R) \Rightarrow heat(L, \varphi', R)) \in R^*$$

it follows that $cfgheat(L, t, R) \wedge Q \Rightarrow cfgheat(L, t', R) \wedge Q' \in R^*$ where $\varphi = t \wedge Q$ and $\varphi' = t' \wedge Q'$. By semantic reasoning we can prove that

$$
\begin{aligned}
\models^{\mathrm{ML}}_{A^*} & \left((\varphi^{\Box}) \left[\frac{X_i}{\Box} \right] \wedge mkList(X_1, ..., X_k) \right) \\
& \leftrightarrow cfgheat(L, X_i, R) \\
& \wedge L = mkList(X_1, ..., X_{i-1}) \\
& \wedge R = mkList(X_{i+1}, ..., X_k) \\
& \wedge (t^{\Box}) \left[\frac{X_i}{\Box} \right] \wedge Q
\end{aligned}
$$

and that

$$
\begin{aligned}
\models^{\mathrm{ML}}_{A^*} & \left(\left((\varphi')^{\Box} \right) \left[\frac{X_i}{\Box} \right] \wedge mkList(X_1, ..., X_k) \right) \\
& \leftrightarrow cfgheat(L, X_i, R) \\
& \wedge L = mkList(X_1, ..., X_{i-1}) \\
& \wedge R = mkList(X_{i+1}, ..., X_k) \\
& \wedge \left((t')^{\Box} \right) \left[\frac{X_i}{\Box} \right] \wedge Q' \ .
\end{aligned}
$$

Now we apply Consequence and subsequently strip the L,R equalities using Logic Framing, thus getting

$$
\begin{aligned}
& R^*, \{flatten(e) \Rightarrow flatten(\Psi') \mid e \in E\}, \\
& \{flatten(c) \Rightarrow flatten(\Psi') \mid c \in C\} \\
& \underset{\mathrm{RL}}{\vdash} \\
& cfgheat(L, X_i, R) \wedge (t^{\Box}) \left[\frac{X_i}{\Box} \right] \wedge Q \\
& \overset{\exists}{\Rightarrow} \\
& cfgheat(L, X_i, R) \wedge \left((t')^{\Box} \right) \left[\frac{X_i}{\Box} \right] \wedge Q' \ .
\end{aligned}
$$

Now, use the Consequence rule to expand $t^{\Box}$ and $(t')^{\Box}$ into equalities, perform the substitution, and use the equalities to replace the X_i subterm of $cfgheat$ with t and t', respectively. The goal becomes

$$
\begin{array}{l}
R^*, \{flatten(e) \Rightarrow flatten(\Psi') \mid e \in E\}, \\
\{flatten(c) \Rightarrow flatten(\Psi') \mid c \in C\} \\
\underset{\text{RL}}{\vdash} \\
cfgheat(L, t, R) \wedge Q \\
\overset{\exists}{\Rightarrow} \\
cfgheat(L, t', R) \wedge Q'
\end{array}
$$

and we conclude using the Axiom rule.

- If the proof ends with Circularity, we can assume

$$
\begin{array}{l}
R^* \cup \{flatten(e) \Rightarrow flatten(\Psi') \mid e \in E\}, \\
\{flatten(c) \Rightarrow flatten(\Psi') \mid c \in C \cup \{\Psi\}\} \\
\underset{\text{RL}}{\vdash} \\
flatten(\Psi) \overset{\exists}{\Rightarrow} flatten(\Psi')
\end{array}
$$

and have to prove

$$
\begin{array}{l}
R^* \cup \{flatten(e) \Rightarrow flatten(\Psi') \mid e \in E\}, \\
\{flatten(c) \Rightarrow flatten(\Psi') \mid c \in C\} \\
\underset{\text{RL}}{\vdash} \\
flatten(\Psi) \overset{\exists}{\Rightarrow} flatten(\Psi')
\end{array}
$$

which follows from the assumption by Circularity.

- If the proof ends with Conseq, we can assume

$$
\models^{\text{ML}}_{A^*} flatten(\Psi) \rightarrow flatten(\Psi')
$$

and

$$
\begin{array}{l}
R^* \cup \{flatten(e) \Rightarrow flatten(\Psi') \mid e \in E\}, \\
\{flatten(c) \Rightarrow flatten(\Psi') \mid c \in C\} \\
\underset{\text{RL}}{\vdash} \\
flatten(\Phi') \overset{\exists}{\Rightarrow} flatten(\Psi')
\end{array}
$$

and have to prove that

$$
\begin{array}{l}
R^* \cup \{flatten(e) \Rightarrow flatten(\Psi') \mid e \in E\}, \\
\{flatten(c) \Rightarrow flatten(\Psi') \mid c \in C\} \\
\underset{\text{RL}}{\vdash} \\
flatten(\Phi) \overset{\exists}{\Rightarrow} flatten(\Psi')
\end{array}
$$

which follows from the Conseqence rule.

- If the proof ends with Abstract, we assume that

$$X \notin \mathrm{FV}(\Psi')$$

and

$$\begin{array}{l} R^* \cup \{\mathit{flatten}(e) \Rightarrow \mathit{flatten}(\Psi') \mid e \in E\}, \\ \{\mathit{flatten}(c) \Rightarrow \mathit{flatten}(\Psi') \mid c \in C\} \\ \underset{\mathrm{RL}}{\vdash} \\ \mathit{flatten}\Big(\exists \overline{Y}.\ (\varphi_1, ..., \varphi_k) \wedge P\Big) \\ \overset{\exists}{\Rightarrow} \\ \mathit{flatten}(\Psi') \end{array}$$

and have to prove that

$$\begin{array}{l} R^* \cup \{\mathit{flatten}(e) \Rightarrow \mathit{flatten}(\Psi') \mid e \in E\}, \\ \{\mathit{flatten}(c) \Rightarrow \mathit{flatten}(\Psi') \mid c \in C\} \\ \underset{\mathrm{RL}}{\vdash} \\ \mathit{flatten}\Big(\exists X, \overline{Y}.\ (\varphi_1, ..., \varphi_k) \wedge P\Big) \\ \overset{\exists}{\Rightarrow} \\ \mathit{flatten}(\Psi')\ . \end{array}$$

After simplifications, the second assumption becomes

$$\begin{array}{l} R^* \cup \{\mathit{flatten}(e) \Rightarrow \mathit{flatten}(\Psi') \mid e \in E\}, \\ \{\mathit{flatten}(c) \Rightarrow \mathit{flatten}(\Psi') \mid c \in C\} \\ \underset{\mathrm{RL}}{\vdash} \\ \exists \overline{Y}.\ \left(\mathit{mkList}(Z_1, ..., Z_k) \wedge \left(\bigwedge_{j=1}^{k} \Big((\varphi_j)^{\Box} \Big) \left[\frac{Z_j}{\Box} \right] \right) \wedge P \right) \\ \overset{\exists}{\Rightarrow} \\ \mathit{flatten}(\Psi') \end{array}$$

while the goal becomes

$$
\begin{aligned}
&R^* \cup \{\mathit{flatten}(e) \Rightarrow \mathit{flatten}(\Psi') \mid e \in E\},\\
&\{\mathit{flatten}(c) \Rightarrow \mathit{flatten}(\Psi') \mid c \in C\}\\
&\underset{\text{RL}}{\vdash}\\
&\exists X, \overline{Y}.\ \left(\mathit{mkList}(Z_1, ..., Z_k) \wedge \left(\bigwedge_{j=1}^{k} \left((\varphi_j)^{\square}\right)\left[\frac{Z_j}{\square}\right]\right) \wedge P\right)\\
&\overset{\exists}{\Rightarrow}\\
&\mathit{flatten}(\Psi')\ .
\end{aligned}
$$

We prove the goal using the Abstraction rule. Note that $X \notin \mathrm{FV}(\Psi')$ implies $X \notin \mathrm{FV}(\mathit{flatten}(\Psi'))$ because we are able to choose the fresh variables in $\mathit{flatten}$ as such.

This concludes the proof. □

Theorem 61: CRL proof system is sound

$$\left(\boldsymbol{R} \vdash_{CRL} \Psi \Downarrow_{\emptyset,\emptyset} \Psi'\right) \Rightarrow \left(\boldsymbol{R} \vDash \Psi \overset{c\exists}{\Rightarrow} \Psi'\right)$$

Proof: By Lemma 60, soundness of reachability logic, and Theorem 56. □

III.3.6 CRL completeness

Reachability logic is relatively complete with respect to an oracle for deciding validity of matching logic formulas in the particular algebra. One can lift this completeness result to CRL using the Reduce rule, if one can turn the oracle for validity in an algebra A into an oracle for validity in the star-extended algebra A^*. The construction to be presented assumes a framework similar to the one assumed in the completeness proof of A. Stefanescu, Ş. Ciobâcă, R. Mereuta, B. M. Moore, T.-F. Şerbănuţă, and G. Roşu [92]. Specifically, let us assume that the algebra A interprets

- the symbol α as an injective function from configurations to natural numbers (as a manifestation of countability of configurations); and
- the symbols $<, +, *$ as the usual ordering, addition, subtraction, and multiplication on natural numbers.

Then, Gödel's function $\beta : \mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$

$$\beta(x_1, x_2, x_3) = x_1 \bmod\ (1 + (x_3 + 1) \cdot x_2)$$

can be represented in FOL (and thus in matching logic) as a predicate

$$
\begin{aligned}
\beta_{t(x_1,x_2,x_3,y)} \equiv \exists w : \mathit{Nat}.\ x_1 &= (1 + (x_3 + 1) * x_2) * w + y\\
&\wedge\ y < (1 + (x_3 + 1) * x_2)
\end{aligned}
$$

(as in E. Mendelson [68], Lemma 3.22).

Lemma 62: Oracle lifting

For every Ω-model A interpreting natural numbers as above, there exists a function Θ from matching logic patterns over Ω^* to matching logic patterns over Ω such that

$$\left(\models_A^{\mathrm{ML}} \Theta(\varphi)\right) \Leftrightarrow \left(\models_{A^*}^{\mathrm{ML}} \varphi\right)$$

Let us only sketch the proof. Intuitively, we will perform Gödelization of the formula $\varphi^{\Box}$, using Gödel's β_t predicate. The idea is that the only construct appearing in FOL formulas over Ω^* and not in FOL formulas over Ω is comparison of lists of configurations for equality, and we will reduce this construct to equality of the corresponding elements of the list. For this purpose, we will introduce a meta-level function $lookup(M, l, n, y)$ representing a formula saying that the term l representing a list of configurations that has at position n the configuration y, where M is a mapping from list variables into the variable pairs that represents lists in the Gödel encoding. The predicate $length(M, l, n)$ holds if the length of l is exactly n.

Let us define two meta-level functions, $lookup$ and $length$, by mutual structural recursion on their second parameters (an algebraic term):

- $lookup(M, l, n, y) = \beta_t(a_l, b_l, n, \alpha(y))$ if l is a variable of sort Cfg^* and $(l, a_l, b_l) \in M$;
- and $lookup(_, cfgnil, _, _) = \bot$;
- and

$$lookup(M, cfgconcat(t_1, t_2), n, y) = \forall l : Nat.\ \ length(M, t_1, l) \to ite(n < l, lookup(M, t_1, n - l, y), lookup(M, t_2, n - l, y));$$

- and

$$lookup(M, cfgheat(l_1, c, l_2), n, y) = \forall l : Nat.\ \ length(M, t_1, l) \to ite(n < l, lookup(M, t_1, n, y), ite(n = l, y = c, lookup(M, t_2, n - l - 1, y)));$$

- and

$$length(M, l, n) = \forall i : Nat.\ (0 \leq i < n) \leftrightarrow (\exists v : Cfg.\ \ lookup(M, l, i, v)),$$

where ite is an if-then-else formula, implemented as

$$ite(B, \varphi_1, \varphi_2) = (B \wedge \varphi_1) \vee (\neg B \wedge \varphi_2).$$

The following property holds.

Lemma 63: Lookup, length correct

Let t be a term of sort Cfg^* and ρ a A-valuation. Let M be a relation such that for every free variable l of sort Cfg^* of t, there exist unique variables a_l , b_l of sort Nat such that $(l, a_l, b_l) \in M$ and $\rho(a_l), \rho(b_l)$ β-encodes $\rho(l)$ – that is, for every $i \in \mathbb{N}_0$ smaller than the length of the list $\rho(l)$, we have that $\beta(\rho(a_l), \rho(b_l), i) = A_\alpha(\rho(l)[i])$. Then for any natural numer n, a configuration γ, a variable c of sort Cfg, and a variable i of sort Nat, if $\rho(c) = \gamma$, then

$$\rho(t)[n] = \gamma \Leftrightarrow \rho \vDash_A^{\mathrm{ML}} lookup(M, t, i, c)$$

and also, for every natural numer n and a variable i of sort Nat, if $\rho(i) = n$, then we have that the length of $\rho(t)$ is exactly n iff $\rho \vDash_A^{\mathrm{ML}} length(M, t, i)$.

Proof: By mutual structural induction on t. □

Next, let us define a translation function tr which performs basic recursion on a given FOL Ω^* formula, except that quantification over lists and equality of lists is handled as follows:

- first,

$$tr(M, \forall(l : Cfg^*).\ \varphi) \equiv \forall(a_l, b_l : Nat).\ \ valid(a_l, b_l) \to tr(M \cup \{(l, a_l, b_l)\}, \varphi)$$

  for some fresh a_l, b_l;
- second,

$$tr(M, l_1 = l_2) \equiv \forall(c : Nat)(d : Cfg).\ (lookup(M, l_1, c, d) \leftrightarrow lookup(M, l_2, c, d))$$

where

$$valid(a, b) \equiv \exists(l : Nat).\ \forall(i : Nat).$$
$$i < l \leftrightarrow \exists(y : Nat.\ y < a \wedge \beta_t(a, b, i, y + 1))\ .$$

Intuitively, the *valid* formula used as a guard in the universal quantification case ensures that we consider only those pairs a_l,b_l which β-encode some sequence of natural numbers. Also, the definition of β guarantees that for fixed a, b, for large enough indices i, $\beta(a, b, i) = a$; the implementation of valid uses this property to guess the length l of the sequence being represented by a, b.

Let us consider the universal closure φ_c of $\varphi^\Box$ - that is, $\varphi_c = \forall x_1 : s_1 .... \forall x_n : s_n \varphi^\Box$ where $x_1, ..., x_n$ are all the variables occuring free in $\varphi^\Box$ and $s_1, ..., s_n$ their respective sorts. Finally, let us define define $\theta(\varphi) = tr(\varphi_c)$; the desired equivalence of oracle lifting (Lemma 62) holds by the properties of the Godel predicate β_t and properties of lists in the extended model T^*.

Now we can state and prove a relative completeness theorem.

Theorem 64: CRL Relative completeness

Let $\boldsymbol{R} = (A, R)$ be a reachability system, and assume an oracle for deciding $\vDash_A^{\mathrm{ML}}$. Then,

$$\left(\boldsymbol{R} \vDash \Psi \overset{c\exists}{\Rightarrow} \Psi'\right) \Rightarrow \left(\boldsymbol{R} \vdash_{CRL} \Psi \Downarrow_{\emptyset,\emptyset} \Psi'\right) .$$

Proof: Assume $\boldsymbol{R} \vDash \Psi \overset{c\exists}{\Rightarrow} \Psi'$. By Theorem 56, we obtaion $\boldsymbol{R}^* \vDash \mathit{flatten}(\Psi) \overset{\exists}{\Rightarrow} \mathit{flatten}(\Psi')$. By relative completeness of reachability logic, we obtain $\boldsymbol{R}^* \underset{\mathrm{RL}}{\vdash} \mathit{flatten}(\Psi) \overset{\exists}{\Rightarrow} \mathit{flatten}(\Psi')$ and we conclude the proof using the Inherit rule. Note that to apply relative completeness of RL, we need to have an oracle for deciding validity in the extended model ($\vDash_{A^*}^{\mathrm{ML}}$); fortunately, we can build one from the oracle for deciding $\vDash_A^{\mathrm{ML}}$ using Lemma 62. □

III.4 Concluding remarks

Let us now discuss a few high-level topics on Cartesian reachability logic, and in particular, on its relation to CHL.

III.4.1 Nondeterminism

Since CRL is based on the *one-path* variant of reachability logic, it inherits its known limitation: that the tight correspondence between one-path RL and Hoare logics is limited to deterministic languages. This is because in the CRL semantics we existentially quantify over reachable configurations, while in CHL, target states are quantified universally. But we can still prove that CRL extends CHL on the deterministic fragment of the supported language. To do so, we need to define the semantics of an all-path variant of CRL.

Definition 65: All-path CRL semantics

A claim

$$(\varphi_1, ..., \varphi_k) \wedge P \overset{c\forall}{\Rightarrow} \exists \overline{Y}.\ (\varphi'_1, ..., \varphi'_k) \wedge P'$$

is valid in a reachability system $\boldsymbol{R} = (A, R)$, written

$$(A, R) \vDash (\varphi_1, ..., \varphi_k) \wedge P \overset{c\forall}{\Rightarrow} \exists \overline{Y}.\ (\varphi'_1, ..., \varphi'_k) \wedge P'$$

iff for every k-tuple $(\gamma_1, ..., \gamma_k) \in A^k_{Cfg}$ of configurations and every A-valuation ρ such that $\gamma_i, \rho \vDash^{\mathrm{ML}}_A \varphi_1 \wedge P$ holds for all $i \in \{1, ..., k\}$, and for every k-tuple $(\pi_1, ..., \pi_k)$ of complete $\underset{\boldsymbol{R}}{\Rightarrow}$-paths (that is, finite paths that cannot be extended further) sayisfying $\pi_i[0] = \gamma_i$ for all $i \in \{1, ..., k\}$ there exists a k -tuple $(j_1, ..., j_k) \in \mathbb{N}^k_0$ of natural numbers such that $\gamma_i \overset{*}{\underset{\boldsymbol{R}}{\Rightarrow}} \pi_i[j_i]$ for all $i \in \{1, ..., k\}$ and $\pi_i, \rho'[j_i] \vDash^{\mathrm{ML}}_A \varphi'_i \wedge P'$ for all $i \in \{1, ..., k\}$, for some ρ' that agrees with ρ on all variables except $\overline{Y}$.

All-path CRL and one-path CRL have the same semantics on deterministic languages, assuming the RHS of the claim is terminal.

Lemma 66: All-path / One-path CRL correspondence

Let Φ be a CLP and Φ' an ECLP such that Φ' is terminal (that is, configurations matching Φ' cannot take a step). Then

$$\left(\boldsymbol{R} \vdash \Phi \overset{c\exists}{\Rightarrow} \Phi'\right) \Leftrightarrow \left(\boldsymbol{R} \vdash \Phi \overset{c\forall}{\Rightarrow} \Phi'\right).$$

Proof: Let $\boldsymbol{R} = (A, R)$, $\Phi \equiv (\varphi_1, ..., \varphi_k) \wedge P$, and $\Phi' \equiv (\varphi'_1, ..., \varphi'_k) \wedge P'$. We prove each implication separately.

- For the "if" part, assume $\boldsymbol{R} \vdash \Phi \overset{c\forall}{\Rightarrow} \Phi'$, and assume some terminating configurations $\gamma_1, ..., \gamma_k$, and some valuation ρ satisfying $\gamma_i, \rho \vDash^{\mathrm{ML}}_A \varphi_i \wedge P$ for all $i \in \{1, ..., k\}$. Let us have a k-tuple $(\pi_1, ..., \pi_k)$ of complete paths $\pi_i = \gamma^1_i, \gamma^2_i, ..., \gamma^{l_i}_i$ such that $\gamma^1_i = \gamma_i$ (for all $i \in \{1, ..., k\}$). Such paths exist, because γ_i (for $i \in \{1, ..., k\}$) are terminating – we can start with a path consisting of γ_i only and repeatedly extend the path until the last element has no successor. Now, since every φ'_i is terminal, all the j_is that exist by Definition 65 refer to the last configurations in the paths, because only those are terminal. That is, we have $j_i = l_i$ for all $i \in \{1, ..., k\}$. Therefore, we have some ρ' and $\gamma^{l_1}_1, ..., \gamma^{l_k}_k$ satisfying $\gamma^{l_i}_i, \rho' \vDash^{\mathrm{ML}}_A \varphi'_i \wedge P$ as required by Definition 38.
- For the "only if" part, assume $\boldsymbol{R} \vdash \Phi \overset{c\exists}{\Rightarrow} \Phi'$. Assume some configurations $\gamma_1, ..., \gamma_k$ and some valuation ρ satisfying $\gamma_i, \rho \vDash^{\mathrm{ML}}_A \varphi_i \wedge P$ for all $i \in \{1, ..., k\}$. Let $\pi_1, ..., \pi_k$ be complete paths $\pi_i = \gamma^1_i, \gamma^2_i, ..., \gamma^{l_i}_i$ such that

$\gamma_i^1 = \gamma_i$ (for all $i \in \{1, ..., k\}$). By Definition 38 there exist configurations $\gamma'_1, ..., \gamma'_k$ such that $\gamma_i \overset{*}{\underset{\boldsymbol{R}}{\Rightarrow}} \gamma'_i$ for all $i \in \{1, ..., k\}$, and there also exists an A-valuation ρ' that agrees with ρ on values of all variables except $\overline{Y}$ such that $\gamma'_i, \rho' \vDash_A^{\mathrm{ML}} \varphi'_i \wedge P'$. Now by determinism and the fact that γ'_i are terminating (because φ'_i are terminating), we have $\gamma_i^{l_i} = \gamma_i$. Thus, $\pi_i[j_i], \rho' \vDash_A^{\mathrm{ML}} \varphi'_i \wedge P$ for all $i \in \{1, ..., k\}$, which concludes the proof.

□

Definition 67: CHL's language's signature

Let L_{CHL} denote the CHL's imperative language and $\Omega_{L_{CHL}}$ denote the matching logic signature of a RL-based formalization of L_{CHL} that that has a distinct constant symbol $sym_{var}(x)$ and a distinct variable $var_{inj}(x)$ in the signature for ever program variable x of L_{CHL}, and that subsumes the syntax of the codomain of the CHL state. We define a function tr_{con} by

$$tr_{con}(P, \sigma) \equiv$$
$$\ll P \mid sym_{var}(x_1 \mapsto \sigma(x_1)), ..., sym_{var}(x_n \mapsto \sigma(x_n)) \gg$$

and a function end_{con} by

$$end_{con}(P, \sigma) \equiv$$
$$\ll \texttt{skip} \mid sym_{var}(x_1 \mapsto \sigma(x_1)), ..., sym_{var}(x_n \mapsto \sigma(x_n)) \gg$$

(where $x_1, ..., x_n$ are program variables occurring in P).

Assumption 68: Assume a formalization of L_{CHL}

Let us assume a sound formalization of L_{CHL} in the form of a reachability system $\boldsymbol{R}_{L_{CHL}} = \left(A_{L_{CHL}}, R_{L_{CHL}}\right)$ that for any P, σ and terminal γ' satisfies

$$\left(tr_{con}(P, \sigma) \overset{*}{\underset{\boldsymbol{R}_{L_{CHL}}}{\Rightarrow}} \gamma'\right) \Leftrightarrow ((\sigma, P \Downarrow \sigma') \wedge (\gamma' = end_{con}(P, \sigma')))$$

where $\Downarrow$ denotes the relation defined by the original big-step semantics of L_{CHL} in [90].

Definition 69: Symbolic translation functions

Let f be an injective function on program variables of L_{CHL}. Let us define a function tr_{sym} by

$$tr_{sym}(P, f) \equiv$$
$$\ll P \mid sym_{var}(x_1 \mapsto var_{inj}(f(x_1))), \ldots, sym_{var}(x_n \mapsto var_{inj}(f(x_n))) \gg$$

and a function end_{sym} by

$$end_{sym}(P, \sigma) \equiv$$
$$\ll \texttt{skip} \mid sym_{var}(x_1 \mapsto var_{inj}(f(x_1))), \ldots, sym_{var}(x_n \mapsto var_{inj}(f(x_n))) \gg$$

(where $x_1, \ldots, x_n$ are program variables occurring in P).

Lemma 70: Symbolic and concrete match

For every statement P, every injective function f on program variables of L_{CHL}, every finite map σ and every configuration γ, we have

$$\gamma, \rho \vDash^{\mathrm{ML}}_{A_{L_{CHL}}} tr_{sym}(P, f)$$

iff

$$(\gamma = tr_{con}(P, \sigma[f])) \wedge (\forall j \in \{1, \ldots, k\}.\ \rho(var_{inj}(f(x_j))) = (\sigma[f])(x_j))\ ;$$

and similarly, we have

$$\gamma, \rho \vDash^{\mathrm{ML}}_{A_{L_{CHL}}} end_{sym}(P, f)$$

iff

$$(\gamma = end_{con}(P, \sigma[f])) \wedge (\forall j \in \{1, \ldots, k\}.\ \rho(var_{inj}(f(x_j))) = (\sigma[f])(x_j))\ .$$

Proof: Follows from semantics of pattern matching in matching logic. □

Lemma 71: Static reasoning in CHL vs CRL

Let $\sigma_1, ..., \sigma_k$ be program states over variables $\overline{x}$. Let $r_i(x) = x_i$ for any $x \in \overline{x}$ and any $i \in \{1, ..., k\}$. Let P be a statement over variables $r_1(\overline{x}), ..., r_k(\overline{x})$. Let $\sigma = \biguplus_{1 \leq i \leq k} \sigma_i[r_i]$. Let ρ_σ be some matching logic valuation sending each matching logic variable $var_{inj}(x)$ to $\sigma(x)$ (and defined in an arbitrary way on the rest of its domain). Then it holds that

$$\sigma \underset{FOL}{\models} \varphi$$
$$\Leftrightarrow$$
$$\forall P.\ \forall i \in \{1, ..., k\}.\ tr_{con}(P, \sigma_i[r_i]), \rho_\sigma \models^{\mathrm{ML}}_{A_{L_{CHL}}} \big(tr_{sym}(P, r_i) \wedge \varphi\big)$$
$$\Leftrightarrow$$
$$\forall P.\ \forall i \in \{1, ..., k\}.\ end_{con}(P, \sigma_i[r_i]), \rho_\sigma \models^{\mathrm{ML}}_{A_{L_{CHL}}} \big(end_{sym}(P, r_i) \wedge \varphi\big)$$

Proof: Let us discuss only the first equivalence since the second one is analogous. By Lemma 70 and properties of conjunction in matching logic and structureless formulas, the right-hand side is equivalent to

- for any $j \in \{1, ..., k\}$, $\rho_\sigma\big(var_{inj}(r_i(x_i))\big) = (\sigma_i[r_i])(x_j)$; and
- $\rho_\sigma \models^{\mathrm{ML}}_{A_{L_{CHL}}} \varphi$.

The first item is always true: by the defining property of ρ_σ, it is enough to show that $\sigma\big(r_i(x_j)\big) = (\sigma_i[r_i])(x_j)$, which is trivially true. The second item is then equivalent to the left-hand side, since φ is structureless. □

Lemma 72: All-path CRL vs CHL

Let P be a command of L_{CHL} over variables $\overline{x} = x_1, ..., x_n$, and let φ, ψ be FOL formulas over the variables $\overline{x}_1, ..., \overline{x}_k$. Let $\overline{Y} = var_{inj}(\overline{x}_1, ..., \overline{x}_k)$. Then $\|\varphi\|\ P\ \|\psi\|$ if and only if

$$\models$$
$$\big(tr_{sym}(P, r_1), ..., tr_{sym}(P, r_k)\big) \wedge \varphi$$
$$\overset{c\forall}{\Rightarrow}$$
$$\exists \overline{Y}.\ \big(end_{sym}(P, r_1), ..., end_{sym}(P, r_k)\big) \wedge \psi\ .$$

Proof: The left side is (by definition) equivalent to

for every set of valuation pairs $\{(\sigma_1, \sigma'_1), ..., (\sigma_k, \sigma'_k)\}$ satisfying

$$\left(\left(\biguplus_{1 \leq i \leq k} \sigma_i[r_i]\right) \underset{FOL}{\models} \varphi\right)$$

and

$$\forall i \in \{1, ..., k\}.\ \sigma_i, P \Downarrow \sigma'_i$$

we also have

$$\left(\left(\biguplus_{1 \leq i \leq k} \sigma'_i[r_i]\right) \underset{FOL}{\models} \varphi\right)$$

which is (by Lemma 71 and Assumption 68) equivalent to

for every set of valuation pairs $\{(\sigma_1, \sigma'_1), ..., (\sigma_k, \sigma'_k)\}$ satisfying

$$\forall P.\ \forall i \in \{1, ..., k\}.\ tr_{con}(P, \sigma_i[r_i]), \rho_\sigma \models^{\mathrm{ML}}_{A_{L_{CHL}}} tr_{sym}(P, r_i) \wedge \varphi$$

and

$$\forall i \in \{1, ..., k\}.\ \underset{\boldsymbol{R}_{L_{CHL}}}{\overset{*}{\Rightarrow}} end_{sym}(P, r_i)$$

we also have

$$\forall P.\ \forall i \in \{1, ..., k\}.\ end_{con}(P, \sigma_i[r_i]), \rho_\sigma \models^{\mathrm{ML}}_{A_{L_{CHL}}} end_{sym}(P, r_i) \wedge \psi$$

(where $\sigma = \biguplus_{1 \leq i \leq k} \sigma_i[r_i]$ and $\sigma' = \biguplus_{1 \leq i \leq k} \sigma'_i[r_i]$).

We want this to be equivalent with

for all k-tuples $(\gamma_1, ..., \gamma_k) \in \left(\left(A_{L_{CHL}}\right)_{Cfg}\right)^k$ and any A-valuation ρ, whenever $\gamma_i, \rho \models^{\mathrm{ML}}_{A_{L_{CHL}}} tr_{sym}(P, r_k) \wedge \varphi$ for all $i \in \{1, ..., k\}$, then for all complete $\underset{\boldsymbol{R}_{L_{CHL}}}{\Rightarrow}$ -paths (that is, finite paths that cannot be extended be further) $\pi_1, ..., \pi_k$ satisfying $\pi_i[0] = \gamma_i$ for all $i \in \{1, ..., k\}$ there exist natural numbers $j_1, ..., j_k$ such that $\gamma_i \underset{\boldsymbol{R}_{L_{CHL}}}{\overset{*}{\Rightarrow}} \pi_i[j_i]$ for all $i \in \{1, ..., k\}$ and $\pi_i[j_i], \rho' \models^{\mathrm{ML}}_{A_{L_{CHL}}} end_{sym}(P, r_i)$ for all $i \in \{1, ..., k\}$, for some ρ' that agrees with ρ on all variables except $\overline{Y}$.

We prove the two implications one by one.

- For the top-down implication, assume such configurations and such complete paths. Let $l_1, ..., l_k$ denote the lengths of the paths $\pi_1, ..., \pi_k$. By Lemma 70, we have for every $j \in \{1, ..., k\}$ that

$$\gamma_j = tr_{con}(P, \sigma[r_j])$$

and therefore

$$tr_{con}(P, \sigma[r_j]) \underset{\boldsymbol{R}_{L_{CHL}}}{\overset{*}{\Rightarrow}} \pi_j[l_j] \ .$$

By Assumption 68, we have

$$\pi_j[l_j] = end_{con}(P, \sigma[r_j]) \ .$$

Because we have

$$\forall P.\ \forall i \in \{1, ..., k\}.\ end_{con}(P, \sigma_i[r_i]), \rho_\sigma \vDash^{\mathrm{ML}}_{A_{L_{CHL}}} end_{sym}(P, r_i) \wedge \psi$$

it follows that

$$\pi_j[l_j], \rho_{\sigma'} \vDash^{\mathrm{ML}}_{A_{L_{CHL}}} end_{sym}(P, r_i) \wedge \psi$$

which is what we needed to prove.
- For the bottom-up implication, assume a set of valuation pairs $\{(\sigma_1, \sigma'_1), ..., (\sigma_k, \sigma'_k)\}$ satisfying

$$\forall P.\ \forall i \in \{1, ..., k\}.\ (tr_{con}(P, \sigma_i[r_i]), \rho_\sigma) \vDash^{\mathrm{ML}}_{A_{L_{CHL}}} tr_{sym}(P, r_i) \wedge \varphi$$

and

$$\forall i \in \{1, ..., k\}.\ tr_{con}(P, \sigma_i) \underset{\boldsymbol{R}_{L_{CHL}}}{\overset{*}{\Rightarrow}} end_{con}(P, \sigma'_i) \ .$$

The second means we have for every $i \in \{1, ..., k\}$ some complete trace π_i of length l_i starting in $tr_{con}(P, \sigma_i)$ and ending in $end_{con}(P, \sigma'_i)$. We need to prove:

$$\forall P.\ \forall i \in \{1, ..., k\}. end_{con}(P, \sigma_i[r_i]), \rho_{\sigma'} \vDash^{\mathrm{ML}}_{A_{L_{CHL}}} end_{sym}(P, r_i) \wedge \psi$$

(where $\sigma = \biguplus_{1 \le i \le k} \sigma_i[r_i]$ and $\sigma' = \biguplus_{1 \le i \le k} \sigma'_i[r_i]$). Let P be some program and let i be in $\{1, ..., k\}$. We need to show that

$$end_{con}(P, \sigma_i[r_i]), \rho_{\sigma'} \vDash^{\mathrm{ML}}_{A_{L_{CHL}}} end_{sym}(P, r_i) \wedge \psi \ .$$

By Lemma 70 it is enough to show that

$$\forall j \in \{1, ..., k\}.\ \rho_{\sigma'}(var_{inj}(f(x_j))) = (\sigma'[f])(x_j)$$

(which holds by the definition of $\rho_{\sigma'}$) and that $\rho_{\sigma'} \vDash^{\mathrm{ML}}_{A_{L_{CHL}}} \psi$. But that follows from

$$\pi_i[l_i], \rho_{\sigma'} \vDash^{\mathrm{ML}}_{A_{L_{CHL}}} end_{sym}(P, r_i) \wedge \psi$$

which in turn follows from our assumptions by first-order reasoning.

□

Lemma 73: One-path CRL vs CHL

Let P be a deterministic statement of L_{CHL} over variables $\overline{x} = x_1, ..., x_n$, and let φ, ψ be FOL formulas over the variables $\overline{x}_1, .., \overline{x}_k$. Let $\overline{Y} = var_{inj}(\overline{x}_1, ..., \overline{x}_k)$. Then $\|\varphi\| \ P \ \|\psi\|$ if and only if

$$\begin{gathered} \boldsymbol{R}_{IMP} \underset{\Omega_{IMP}}{\vDash} \\ \big(tr_{sym}(P, r_1), ..., tr_{sym}(P, r_k)\big) \wedge \varphi \\ \overset{c\exists}{\Rightarrow} \\ \big(end_{sym}(P, r_1), ..., end_{sym}(P, r_k)\big) \wedge \psi \end{gathered}$$

Proof: Use Lemma 66 and Lemma 72. □

Theorem 74: CRL extends CHL

CRL extends CHL on the deterministic fragment of the CHL-supported language. That is, assume a sound reachability-logic formalization (that is, a reachability system) $\boldsymbol{R}_{IMP} = (A_{IMP}, R_{IMP})$ over signature Ω_{IMP} of the CHL's imperative language of [90]. (I am not aware of any such formalization of that particular language, but conjecture that it can be created.) Then there exist translation functions tr and end such that given any statement P in the deterministic fragment of the CHL's imperative language and any first order formulas φ, ψ, it holds that

$$\Big(\underset{CHL}{\vDash} \|\varphi\| \ P \ \|\psi\|\Big) \Leftrightarrow \Big(\boldsymbol{R}_{IMP} \underset{\Omega_{IMP}}{\vDash} tr(P, \varphi) \overset{c\exists}{\Rightarrow} end(P, \psi)\Big)$$

Proof: Let

$$\begin{aligned} tr(P, \varphi) &\equiv \big(tr_{sym}(P, r_1), ..., tr_{sym}(P, r_k)\big) \wedge \varphi \\ end(P, \varphi) &\equiv \exists \overline{Y}.\ \big(end_{sym}(P, r_1), ..., end_{sym}(P, r_k)\big) \wedge \varphi \end{aligned}$$

where P is a deterministic statement over the variables $x_1, ..., x_n$ and φ, ψ are FOL formulas over the variables $\overline{x}_1, ..., \overline{x}_k$, and $\overline{Y} = var_{inj}(\overline{x}_1, ..., \overline{x}_k)$. Then, apply Lemma 73. □

III.4.2 Similarities to CHL

My understanding of the inner workings of CHL is based on the extended, unpublished version ([91]) of [90]. There, the authors define a linearization operation on lists of programs, which roughly corresponds to the star extension of the language's semantics. Then, the authors prove lemmas saying that a CHL triple with a list of programs inside is derivable in the CHL proof system if and only if a Hoare triple having the same list of programs but linearized inside, is derivable; the "only if" implication corresponds to Lemma 60, where we construct an RL proof from a CRL proof, while the "if" implication corresponds to the Reduce rule of the proof system. Furthermore, in the proof of soundness of CHL, the authors assume soundness of the self-composition technique; self-composition corresponds to star extension of CRL and its soundness to our Theorem 56. Finally, [91] assumes soundness and relative completeness of the underling Hoare logic; similarly, we had to assume the soundness and relative completeness of reachability logic. However, for completeness, we had to prove that the star extension preserves decidability of validity.

III.4.3 Differences from CHL

There are also differences between the CHL and CRL techniques. First, CRL's proof system has only 8 rules, and they do not mention any programming language construct, while CHL has 17 rules (not counting the Expand rule), half of which are specific to the underlying language. Second, there is a redundancy between the language-specific CHL rules and the Hoare logic rules of the programming language: for example, the conditional statement ("if") has (1) a rule in the formal semantics of the language, (2) a rule in the Hoare logic (not shown in the paper), and (3) a rule in CHL. When considering that the three rules have to play nice together (that is, CHL and Hoare logic rules have to be sound with respect to the semantics), someone had to think about the conditional statement at least five times. I consider this to be highly uneconomical - and the situation is even worse for the looping construct ("while-with-breaks"), which is supported using additional CHL rules. In contrast, in the CRL/RL framework, it is enough to design each language construct once - when giving its operational semantics. Third, in CHL the support for lockstep reasoning is hard-wired into the rules for loops, while in the CRL framework, lockstep reasoning is not limited to loops, but can support arbitrary sources of circular behavior - including loops, recursion, gotos.

III.4.4 Future work

It would be nice to develop the all-path variant of CRL.

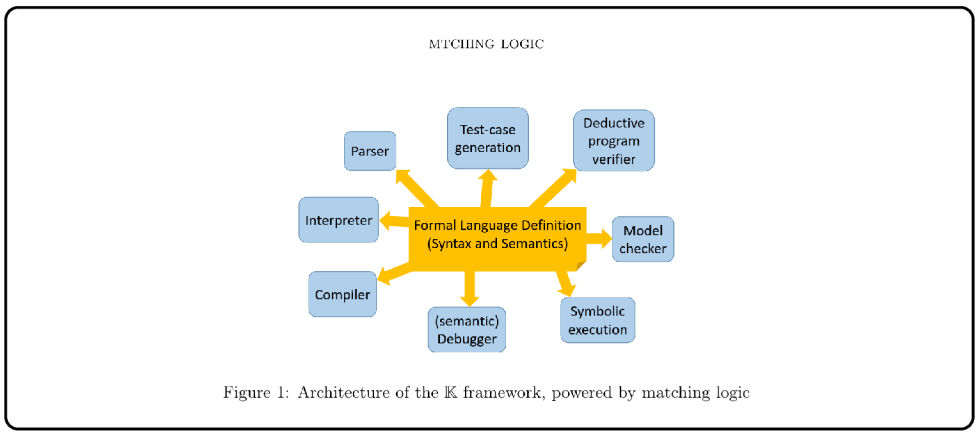

Figure 10: A screenshot of [79], page 3 (top), featuring an illustration of the architecture of and vision for the 𝕂-framework.

IV An attempt to connect 𝕂 to Coq

Reachability logic, as discussed in Section II.2, was for some time considered the logical foundation of the 𝕂-framework. The soundness of 𝕂-framework's formal verification algorithms was justified by appeal to reachability logic, and matching logic as an extension of first-order logic played only a minor role. In one line of research, represented by V. Rusu and D. Nowak [86] and V. Rusu, G. Grimaud, and M. Hauspie [85], reachability logic is even used without matching logic as the underlying formalism.

However, in the line of work by G. Roşu and his collaborators, matching logic was developed further and promoted to a more prominent role in the 𝕂-framework ecosystem. For example, G. Roşu [79], published in 2017, considers matching logic, not reachability logic, to be the logical power behind the 𝕂-framework (see also Figure 10). In the subsequent work of X. Chen and G. Roşu [24], matching logic was extended with a least-fixpoint operator and then simplified in X. Chen, D. Lucanu, and G. Roşu [22]. All these papers, and also X. Chen and G. Roşu [25], maintain the relationship of matching logic to programming language semantics, semantics frameworks, and ultimately, the 𝕂-framework.

Motivated by the relationship between these, we began the work with the co-authors of [9] on a formalization of matching logic in Coq. Eventually, the long-term goal of this project became to enable mechanical interactive reasoning in matching logic and to bring the power of proof assistants to the 𝕂-framework ecosystem – through matching logic. I consider the first part of the goal to be successful, partly due to [96] and partly because of the vast Coq-based infrastructure that we developed for matching logic mentioned neither in [9] nor in [96].

Regarding the second part of the goal – I consider it a failure. In hindsight, the project had no chance of succeeding for two reasons. First, because of the vast irregularities in the informal (and undocumented) semantics of Kore – the 𝕂-framework's intermediate language that we wanted to exploit for establishing

the connection between $\mathbb{K}$, matching logic, and Coq – a connection between Kore and matching logic would necessarily be "hacky" (see also Section II.4.2.2). Second, even if Kore had a reasonable, formalizable semantics, I would now still consider matching logic, and especially its latest version of [22], to be too low-level for purposes of a semantics framework. (As a side note – Chapter V presents a semantics framework designed entirely without matching logic).

It is natural, therefore, to wonder what contributions to the broader community this line of work brings. Besides the infrastructure for matching logic reasoning, which is now interesting only to a relativel small audience, there are two additional contributions.

1. The paper [96] reproduces the idea of logic-specific *proof mode* of [60] in a different setting. The proof mode of [60] targets a *shallow embedding* of Iris – an intuitionistic separation logic – into Coq (which itself implements intuitionistic logic), thus enabling formally verified, mechanical, interactive reasoning for Iris; in contrast, our work targets a *deep embedding* of rather *classical* matching logic (admitting the law of excluded middle). (The deep embedding poses an extra challenge, as many facilities of the background system (Coq) are not accessible in such a setting.)
2. The work presented in [96] can serve as an example of how one can translate proofs of a *natural deduction*-style proof system (presented in the form of a single-conclusion sequent calculus) into Hilbert-style proofs for a logic that does not admit a *deduction theorem* – at least, not the typical one saying that $\Gamma \cup \{\varphi\} \vdash \psi$ implies $\Gamma \vdash \varphi \to \psi$. This is interesting because the typical way of translating natural proofs to Hilbert proofs relies on the deduction theorem (for an example, see J. von Plato [74], section 3.6.(c); another example is M. H. Sørensen and P. Urzyczyin [94], section 5.1 and theorem 5.1.8).

In [96] we decided to present the natural deduction calculus first, and then use it to ground our proof mode. The historical development was different, though:

1. First, we engineered most of the proof mode in Coq on a by-need but systematic basis, grounding new tactics in already-proven lemmas and proving new lemmas using already-developed tactics.
2. When we decided to write a paper about it, we felt it was not systematic enough – the tactics felt like a separate proof system.
3. We developed a natural deduction proof system but felt that something was missing – first-order reasoning. So we added it to both the calculus and the proof mode.

This chapter presents our work in yet another way.

1. In Section IV.1 I describe how to arrive at a natural deduction system *and* corresponding proof mode for an FOL-like logic equipped with a Hilbert-style proof system that has the usual propositional and first-order rules but may contain more rules. Note that it is not a lack of proof rules that would break a typical deduction theorem, but their abundance, as a deduction theorem is typically proved by induction on the structure of the given proof of $\Gamma \cup \{\varphi\} \vdash \psi$.

2. In Section IV.2 I discuss the specifics of the particular variant of matching logic and present the results of [96].

The work on matching logic formalization in Coq was partially supported by RuntimeVerification Inc, the company developing $\mathbb{K}$-framework. The version of our Coq development accompanying [96] is available at https://github.com/harp-project/AML-Formalization/releases/tag/v1.0.15.

IV.1 Natural deduction without deduction theorem

Consider the fragment of a first-order logical language defined by the following grammar:

$$\begin{array}{rcl} \varphi & ::= & \bot \\ & | & \varphi \ \to \ \varphi \\ & | & \exists x.\ \varphi \\ & | & \ldots \end{array}$$

There may be more connectives (as there are in the variant of matching logic I have in mind). And consider the remaining standard FOL connectives to be defined as abbreviation; that is,

1. $\neg\varphi \equiv \varphi \to \bot$;
2. $\varphi_1 \vee \varphi_2 \equiv (\neg\varphi_1) \to \varphi_2$;
3. $\varphi_1 \wedge \varphi_2 \equiv \neg((\neg\varphi_1) \vee (\neg\varphi_2))$;
4. $\varphi_1 \leftrightarrow \varphi_2 \equiv (\varphi_1 \to \varphi_2) \wedge (\varphi_2 \to \varphi_1)$; and
5. $\forall x.\ \varphi \equiv \neg\exists x.\ (\neg\varphi)$.

(It makes little sense to discuss the semantics of such a language since the entire language is not defined at this point.) And consider a Hilbert-style proof system that contains the rules in Figure 11, and perhaps some not shown yet.

$$\frac{\varphi \in \Gamma}{\Gamma \vdash_H \varphi}\,\text{Assumption}$$

$$\frac{}{\Gamma \vdash_H \varphi \to (\psi \to \varphi)}\,\text{P1}$$

$$\frac{}{\Gamma \vdash_H (\varphi \to (\psi \to \xi)) \to ((\varphi \to \psi) \to (\varphi \to \xi))}\,\text{P2}$$

$$\frac{}{\Gamma \vdash_H ((\varphi \to \bot) \to \bot) \to \varphi}\,\text{P3}$$

$$\frac{\Gamma \vdash_H \varphi \qquad \Gamma \vdash_H \varphi \to \psi}{\Gamma \vdash_H \psi}\,\text{MP}$$

$$\frac{}{\Gamma \vdash_H \varphi\left[\frac{y}{x}\right] \to \exists x.\ \varphi}\,\text{Ex-quan}$$

$$\frac{\Gamma \vdash_H \varphi \to \psi \qquad x \notin FV(\psi)}{\Gamma \vdash_H (\exists x.\ \varphi) \to \psi}\,\text{Ex-gen}$$

Figure 11: A proof system for a first-order fragment of a logic; $FV(\varphi)$ denotes the set of all variables occurring freely in φ, defined in the usual way.

With such a proof system, we can prove (for example) $\varphi \to \varphi$ as follows:

$$\frac{\dfrac{}{\emptyset \vdash_H \varphi \to (\varphi \to \varphi)}\,\text{P1} \qquad \dfrac{\dfrac{}{\emptyset \vdash_H (\varphi \to ((\varphi \to \varphi) \to \varphi)) \to ((\varphi \to (\varphi \to \varphi)) \to (\varphi \to \varphi))}\,\text{P2} \qquad \dfrac{}{\emptyset \vdash_H \varphi \to ((\varphi \to \varphi) \to \varphi)}\,\text{P1}}{\emptyset \vdash_H (\varphi \to (\varphi \to \varphi)) \to (\varphi \to \varphi)}\,\text{MP}}{\emptyset \vdash_H \varphi \to (\varphi \to \varphi)}\,\text{MP}$$

Typically, we would want to prove propositional tautologies in a simpler way. Like this:

$$\frac{\dfrac{}{\{\varphi\} \vdash_? \varphi}\,\text{Hyp}'}{\emptyset \vdash_? \varphi \to \varphi}\,\text{impl_intro}'$$

And we would also want to justify $\vdash_?$-proofs using $\vdash_H$, by having a (computable) function of type

$$(\Gamma \vdash_? \varphi) \Rightarrow (\Gamma \vdash_H \varphi)\ .$$

But how to do that if the deduction theorem does not hold for $\vdash_H$? In particular, how to support a general-enough implication-introduction rule if there are some formulas φ_{bad} and φ_{ugly} (which may use the other, unknown connectives of our logical language) and some set Γ_{weird} such that $\Gamma_{weird} \cup \{\varphi_{ugly}\} \vdash_H \varphi_{bad}$ holds but $\Gamma_{weird} \vdash_H \varphi_{ugly} \to \varphi_{bad}$ does not? (With matching logic in particular, the deduction rule does not hold semantically.)

So my suggestion is: we fake it. We will pretend that we have an implication-introduction rule, but in the implementation (both paper-based and Coq-based),

we will just remember how deep we are in an implication chain – how many times the user performed implication-introduction – and manipulate the formula accordingly. On paper, we will have sequents of the shape

$$\Gamma, \Delta \vdash_N \varphi$$

(where Γ ranges over sets of formulas, Δ over finite lists of formulas, and φ over formulas) which will correspond to Hilbert sequents of the shape

$$\Gamma \vdash_H \varphi_1 \to ... \to \varphi_k \to \varphi$$

by letting $\Delta = [\varphi_1, ..., \varphi_k]$. On paper, the correspondence is theoretically given as a translation function (which ensures relative soundness of our new proof system), but one does not write down such a translation function explicitly: it would be too large and nobody would want to read it anyway. Instead, the soundness is ensured in an incremental way, by relying on the soundness of Coq's core. As I have suggested in the introduction, we incrementally prove more lemmas about $\vdash_H$, use them to add more rules to $\vdash_N$ and build more Coq tactics, and use these to prove even more lemmas about $\vdash_H$. And in Coq, we do not represent $\vdash_N$ explicitly – that is, we will not deeply embed it into the language of Coq. Instead, we embed it shallowly on top of our deep embedding of $\vdash_H$.

First, let us assume that we have proofs of the following schemas about implication and negation:

name	**schema**
identity	$\Gamma \vdash_H \varphi \to \varphi$
P4m	$\Gamma \vdash_H (\varphi \to \psi) \to ((\varphi \to \neg\psi) \to \neg\varphi)$
P4i	$\Gamma \vdash_H (\varphi \to \neg\varphi) \to \neg\varphi$
reorder	$\Gamma \vdash_H (\varphi \to (\psi \to \xi)) \to (\psi \to (\varphi \to \xi))$
transitivity	$\Gamma \vdash_H (\varphi \to \psi) \to ((\psi \to \xi) \to (\varphi \to \xi))$

Proofs of these are standard. With these, we can prove

Lemma 75: Conclusion weakening

$$\Gamma \vdash_H (\psi \to \psi') \to ((\varphi \to \psi) \to (\varphi \to \psi'))$$

and

$$\frac{\Gamma \vdash_H \psi \to \psi'}{\Gamma \vdash_H (\varphi \to \psi) \to (\varphi \to \psi')}$$

Proof: Using reorder and MP. □

Now comes the important part. One can generalize Lemma 75 using meta-level induction on natural numbers into the following schema:

Lemma 76: Nested conclusion weakening

$$\Gamma \vdash_H (\psi \to \psi') \to (\varphi_1 \to \ldots \to \varphi_k \to \psi) \to (\varphi_1 \to \ldots \to \varphi_k \to \psi')$$

Proof: By meta-level induction on k, Lemma 75, transitivity of implication, and MP. □

This lemma gets lifted to have meta-level implications in the top-level using MP, and which justifies the following proof rule of $\vdash_N$, which then justifies a corresponding tactic in our Coq proof mode, similar to Coq's native `apply` tactic.

$$\frac{\Gamma \vdash_H \varphi \to \varphi' \qquad \Gamma, \Delta \vdash_N \varphi}{\Gamma, \Delta \vdash_N \varphi'}\,\text{applyMeta}$$

But we can go one step further.

Lemma 77: Nested conclusion weakening under nested implication

$$\Gamma \vdash_H (\varphi_1 \to \ldots \to \varphi_m \to (\xi \to \xi') \to \psi_1 \to \ldots \to \psi_n \to \xi) \to \\ (\varphi_1 \to \ldots \to \varphi_m \to (\xi \to \xi') \to \psi_1 \to \ldots \to \psi_n \to \xi')$$

Proof: By meta-level induction on m, Lemma 76, transitivity of implication, MP, and Lemma 75. □

This lemma justifies the following proof rule of $\vdash_N$.

$$\frac{\Gamma, \Delta_1 \mathbin{++} [\xi \to \xi'] \mathbin{++} \Delta_2 \vdash_N \xi'}{\Gamma, \Delta_1 \mathbin{++} [\xi \to \xi'] \mathbin{++} \Delta_2 \vdash_N \xi'}\,\text{apply}$$

A lemma on strenghtening of a hypothesis, corresponding somewhat to Coq's native `apply ... in` tactic, is proved in a similar way: the "transitivity" rule above (together with MP) essentially strenghtens the (only) hypothesis in a chain, so we can generalize it into

Lemma 78: Strenghten premise

$$\begin{aligned}\Gamma \vdash_H (\psi' \to \psi) \to \\ (\varphi_1 \to \ldots \to \varphi_k \to \psi \to \xi_1 \to \ldots \to \xi_n) \to \\ (\varphi_1 \to \ldots \to \varphi_k \to \psi' \to \xi_1 \to \ldots \to \xi_n)\end{aligned}$$

Proof: By induction on k, **conclusion weakening** (perhaps counter-intuitively), and transitivity. □

which justifies the following proof rule (where ++ is the list concatenation operator).

$$\frac{\Gamma \vdash_H \varphi \to \varphi' \qquad \Gamma, \Delta \mathbin{++} [\varphi] \mathbin{++} \Delta' \vdash_N \psi}{\Gamma, \Delta \mathbin{++} [\varphi'] \mathbin{++} \Delta' \vdash_N \psi}\ \text{applyInMeta}$$

One can also use the first hypothesis to conclude a proof.

Lemma 79: Exactly the first hypothesis

$$\Gamma \vdash_H \varphi \to (\psi_1 \to \ldots \to \psi_k \to \varphi)$$

Proof: Induction on k, identity, transitivity. □

In fact, we can conclude a proof using any hypothesis, which is the basis for a tactic similar to Coq's `exact` and yet another $\vdash_N$ rule.

Lemma 80: Exactly a hypothesis

$$\Gamma \vdash_H \varphi_1 \to \ldots \to \varphi_n \to \psi \to \varphi'_1 \to \ldots \to \varphi'_m \to \psi$$

Proof: Induction on n, exactly the first hypothesis, transitivity. □

$$\frac{}{\Gamma, \Delta \mathbin{++} [\varphi] \mathbin{++} \Delta' \vdash_N \varphi}\ \text{Hyp}$$

Let us now discuss how to support a cut-like rule and the corresponding `assert`-like tactic. First, we prove a rule for adding an extra assumption (using P1 and MP).

$$\frac{\Gamma \vdash_H \psi}{\Gamma \vdash_H \varphi \to \psi}\ \text{addAssumption}$$

Then, we prove the lemma

Lemma 81: addLemma

$$\begin{aligned}\Gamma \vdash_H\ & (\varphi_1 \to \ldots \to \varphi_n \to \psi) \to \\ & (\varphi_1 \to \ldots \to \varphi_n \to \psi \to \xi) \to \\ & (\varphi_1 \to \ldots \to \varphi_n \to \xi)\end{aligned}$$

Proof: By induction on n using addAssumption, P2, weaken conclusion. □

serving as the basis for the rule:

$$\frac{\Gamma, \Delta \vdash_N \psi \qquad \Gamma, \Delta \mathbin{++} [\psi] \vdash_N \xi}{\Gamma, \Delta \vdash_N \xi}\ \text{assert}$$

In fact, we are able to prove a slightly more generalized rule:

$$\frac{\Gamma, \Delta_1 \vdash_N \psi \qquad \Gamma, \Delta_1 \mathbin{++} [\psi] \mathbin{++} \Delta_2 \vdash_N \xi}{\Gamma, \Delta_1 \mathbin{++} \Delta_2 \vdash_N \xi}\ \text{assert}'$$

Let us now discuss how a simple lemma about disjunction (known as "constructive dilemma") can be proved. I use the fact that one can convert any $\vdash_H$-goal into an $\vdash_N$-goal.

$$\begin{array}{cl}
\cline{1-1} \Gamma, [(\varphi_1 \to \psi_1), (\varphi_2 \to \psi_2), ((\neg\varphi_1) \to \varphi_2), (\psi_1 \to \bot), \varphi_1] \vdash_N \varphi_1 & \text{exactly 5th} \\
\cline{1-1} \Gamma, [(\varphi_1 \to \psi_1), (\varphi_2 \to \psi_2), ((\neg\varphi_1) \to \varphi_2), (\psi_1 \to \bot), \varphi_1] \vdash_N \psi_1 & \text{apply 1st} \\
\cline{1-1} \Gamma, [(\varphi_1 \to \psi_1), (\varphi_2 \to \psi_2), ((\neg\varphi_1) \to \varphi_2), (\psi_1 \to \bot), \varphi_1] \vdash_N \bot & \text{apply 4th} \\
\cline{1-1} \Gamma, [(\varphi_1 \to \psi_1), (\varphi_2 \to \psi_2), ((\neg\varphi_1) \to \varphi_2), (\psi_1 \to \bot)] \vdash_N \varphi_1 \to \bot & \text{intro} \to \\
\cline{1-1} \Gamma, [(\varphi_1 \to \psi_1), (\varphi_2 \to \psi_2), ((\neg\varphi_1) \to \varphi_2), (\neg\psi_1)] \vdash_N \neg\varphi_1 & \text{unfold} \neg \\
\cline{1-1} \Gamma, [(\varphi_1 \to \psi_1), (\varphi_2 \to \psi_2), ((\neg\varphi_1) \to \varphi_2), (\neg\psi_1)] \vdash_N \varphi_2 & \text{apply 3rd} \\
\cline{1-1} \Gamma, [(\varphi_1 \to \psi_1), (\varphi_2 \to \psi_2), ((\neg\varphi_1) \to \varphi_2), (\neg\psi_1)] \vdash_N \psi_2 & \text{apply 2nd} \\
\cline{1-1} \Gamma, [] \vdash_N (\varphi_1 \to \psi_1) \to (\varphi_2 \to \psi_2) \to ((\neg\varphi_1) \to \varphi_2) \to ((\neg\psi_1) \to \psi_2) & \text{intro} \to \\
\cline{1-1} \Gamma \vdash_H (\varphi_1 \to \psi_1) \to (\varphi_2 \to \psi_2) \to ((\neg\varphi_1) \to \varphi_2) \to ((\neg\psi_1) \to \psi_2) & \vdash_H \text{to} \vdash_N \\
\cline{1-1} \Gamma \vdash_H (\varphi_1 \to \psi_1) \to (\varphi_2 \to \psi_2) \to (\varphi_1 \vee \varphi_2) \to (\psi_1 \vee \psi_2) & \text{unfold} \vee
\end{array}$$

The proof is easy to read from the bottom upwards. It is also very easy to write, especially in our Coq-based proof mode, since it consists of applying our *fake hypotheses* and reshapings of the goal, such as implication introduction and definition unfolding.

Let me now return back to the high-level picture of things. We can use the notion of a "fake" context (we usually call it *local context*) to achieve compositional, natural-deduction-style proofs on top of a Hilbert proof system. Implication introduction is basically a no-operation, since it does not change the underlying Hilbert goal; therefore, we do not need any meta-level *deduction theorem*, which would be unavailable in our setting anyway. We use $\vdash_N$-proofs to justify $\vdash_H$-lemmas, and

also the other way around. For example, constructive dilemma is the basis for our implementation of disjunction elimination:

$$\frac{\Gamma, \Delta_1 \mathbin{++} [\varphi] \mathbin{++} \Delta_2 \vdash_N \xi \qquad \Gamma, \Delta_1 \mathbin{++} [\psi] \mathbin{++} \Delta_2 \vdash_N \xi}{\Gamma, \Delta_1 \mathbin{++} [\varphi \vee \psi] \mathbin{++} \Delta_2 \vdash_N \xi} \text{disjunction elimination}$$

IV.2 Details on applicative unsorted matching logic

Let us now have a look at the full syntax, semantics, and Hilbert proof system of applicative unsorted matching logic (AML) that the work of [96] works upon. The original source of these concepts is [22].

Applicative Matching logic (AML) is a simple but expressive logic designed mainly for reasoning about programs and programming language semantics. It can capture reasoning in first-order logic [79], in modal μ-calculus and reachability logic [24], and its Hilbert-style proofs can be checked using a simple checker [21]. The logic is parameterized by a signature that distinguishes between *element variables* and *set variables*.

Definition 82: AML syntax

A signature is a triple (EV, SV, Σ), where
1. EV is a countably infinite set of element variables, denoted $x, y, \ldots$;
2. SV is a countably infinite set of set variables, denoted $X, Y, \ldots$;
3. Σ is a countable set of (constant) symbols, denoted $\sigma, f, g, \ldots$.

EV, SV, and Σ are pairwise disjoint. When EV and SV are understood from the context, Σ is used to denote the whole signature. Given a signature Σ, the set of applicative matching logic formulas (aka *patterns*) is inductively defined by the grammar

φ	$::=$	x	*element variable*
	$\mid$	X	*set variable*
	$\mid$	σ	*symbol*
	$\mid$	$\varphi_1 \varphi_2$	*application*
	$\mid$	$\bot$	*bottom*
	$\mid$	$\varphi_1 \to \varphi_2$	*implication*
	$\mid$	$\exists x.\varphi$	*existential quantifier*
	$\mid$	$\mu X.\varphi$	*least fixpoint binder* (X can have only non-negative occurrences in φ)

Other connectives of firstorder logic are defined as notations in the usual way. The notation $\varphi\left[\frac{\psi}{x}\right]$ is used for capture-avoiding substitution, and let $\mathrm{FV}(\varphi)$ denote the set of free (element or set) variables of φ.

Definition 83: AML contexts

The syntax of *AML contexts* is defined as the syntax of AML patterns, except that there is a distinguished meta-symbol $\square$ (meaning *the hole*) that occurs in any context exactly once in a leaf position. More precisely, the syntax is as if defined by the following grammar:

$$
\begin{array}{rcll}
C & ::= & \square & \textit{hole} \\
 & | & x & \\
 & | & X & \\
 & | & \sigma & \\
 & | & \bot & \\
 & | & C\ C & \\
 & | & C \to C & \\
 & | & \exists x.\ C & \\
 & | & \mu X.\ C & \text{(where } X \text{ does not occur negatively in } C\text{)}
\end{array}
$$

Furthermore, the set of *applicative contexts* is generated by the following grammar:

$$
\begin{array}{rcl}
C^{\$} & ::= & \square \\
 & | & C^{\$}\ \varphi \\
 & | & \varphi\ C^{\$}
\end{array}
$$

We write $FV(\varphi)$ to mean the set of free (element or set) variables of the pattern φ, and similarly $FV(C)$ (and $FV(C^{\$})$) for variables of (an applicative) context.

IV.2.1 Semantics

Intuitively, a pattern (formula) is interpreted as the (often singleton) set of model elements that match the pattern in the given valuation. For example, when using matching logic to reason about a small-step semantics of a programming language as done in the $\mathbb{K}$-framework, the pattern

$$\texttt{state}\ (\texttt{incr i})\ (\texttt{map i}\ x)$$

(where `state`, `incr`, `i`, `map` are symbols and x is a variable) would, in the right model and the valuation ρ, match exactly the program state consisting of the program `incr i` (that is to be executed) and a memory that maps the program variable `i` to the value $\rho(x)$.

Definition 84: AML model

Given an AML signature $\boldsymbol{\Sigma} = (EV, SV, \Sigma)$, a $\boldsymbol{\Sigma}$-model is a triple $\boldsymbol{M} = \left(M, \cdot_{\boldsymbol{M}}, \{\sigma_{\boldsymbol{M}}\}_{\sigma \in \Sigma}\right)$ where

- M is a non-empty *carrier set*;
- $_ \cdot_{\boldsymbol{M}} _ : M \times M \to \mathcal{P}(M)$ is an *application function*; and
- $\sigma_{\boldsymbol{M}} \subseteq M$ is an *intrepretation of the symbol* σ.

Application from the model is extended to any subsets of the model in the pointwise manner; that is, application of two sets is defined as the union of application of pairs from the Cartesian product of the two sets:

$$A \cdot_{\boldsymbol{M}} B = \bigcup_{(a,b) \in A \times B} a \cdot_{\boldsymbol{M}} b \ .$$

Definition 85: AML semantics

Given an AML signature

$$\boldsymbol{\Sigma} = (EV, SV, \Sigma)$$

and a $\boldsymbol{\Sigma}$-model

$$\boldsymbol{M} = \left(M, \cdot_{\boldsymbol{M}}, \{\sigma_{\boldsymbol{M}}\}_{\sigma \in \Sigma}\right),$$

an $\boldsymbol{M}$*-valuation* is a function mapping element variables to elements of M and set variables to sets of elements of M. The semantics of a formula in a valuation is the set defined by recursion on the structure of the formula, as follows:

- $|x|_\rho = \{\rho(x)\}$;
- $|X|_\rho = \rho(X)$;
- $|\sigma|_\rho = \sigma_{\boldsymbol{M}}$;
- $|\bot|_\rho = \emptyset$;
- $|\varphi\ \psi|_\rho = |\varphi|_\rho \cdot_{\boldsymbol{M}} |\psi|_\rho$;
- $|\varphi \to \psi|_\rho = M \setminus \left(|\varphi|_\rho \setminus |\psi|_\rho\right)$;
- $|\exists x.\ \varphi|_\rho = \bigcup_{m \in M} |\varphi|_{\rho[x := m]}$ where $\rho[x := m]$ denotes a valuation update; and
- $|\mu X.\ \varphi|_\rho = \bigcap \left\{Q \subseteq M \mid |\varphi|_{\rho[X := Q]} \subseteq Q\right\}$.

IV.2.2 Proof system

$$\frac{\varphi \in \Gamma}{\Gamma \vdash_H \varphi}\ \text{Assumption}$$

$$\frac{}{\Gamma \vdash_H \varphi \to (\psi \to \varphi)}\ \text{P1}$$

$$\frac{}{\Gamma \vdash_H (\varphi \to (\psi \to \xi)) \to ((\varphi \to \psi) \to (\varphi \to \xi))}\ \text{P2}$$

$$\frac{}{\Gamma \vdash_H ((\varphi \to \bot) \to \bot) \to \varphi}\ \text{P3}$$

$$\frac{\Gamma \vdash_H \varphi \qquad \Gamma \vdash_H \varphi \to \psi}{\Gamma \vdash_H \psi}\ \text{MP}$$

$$\frac{}{\Gamma \vdash_H \varphi\left[\frac{y}{x}\right] \to \exists x.\ \varphi}\ \text{Ex-quan}$$

$$\frac{\Gamma \vdash_H \varphi \to \psi \qquad x \notin FV(\psi)}{\Gamma \vdash_H (\exists x.\ \varphi) \to \psi}\ \text{Ex-gen}$$

Figure 12: Hilbert-style proof system of AML: first-order fragment

$$\frac{}{\Gamma \vdash_H C^{\$}[\bot] \to \bot}\ \text{Prop}\bot$$

$$\frac{}{\Gamma \vdash_H C^{\$}[\varphi \vee \psi] \to C^{\$}[\varphi] \vee C^{\$}[\psi]}\ \text{Prop}\vee$$

$$\frac{x \notin FV(C^{\$})}{\Gamma \vdash_H C^{\$}[\exists x.\ \varphi] \to \exists x.\ C^{\$}[\varphi]}\ \text{Prop}\exists$$

$$\frac{\Gamma \vdash_H \varphi \to \psi}{\Gamma \vdash_H C^{\$}[\varphi] \to C^{\$}[\psi]}\ \text{Framing}$$

$$\frac{\Gamma \vdash_H \varphi}{\Gamma \vdash_H \varphi\left[\frac{\psi}{X}\right]}\ \text{Subst}$$

$$\frac{}{\Gamma \vdash_H \varphi\left[\frac{\mu X.\ \varphi}{X}\right] \to \mu X.\ \varphi}\ \text{PreFixp}$$

$$\frac{\Gamma \vdash_H \varphi\left[\frac{\psi}{X}\right] \to \psi}{\Gamma \vdash_H (\mu X.\ \varphi) \to \psi}\ \text{KT}$$

$$\frac{}{\Gamma \vdash_H \exists x.\ x}\ \text{Existence}$$

$$\frac{}{\Gamma \vdash_H \neg\big(C^{\$}[x \wedge \varphi] \wedge C^{\$}[x \wedge \neg\varphi]\big)}\ \text{Singleton}$$

Figure 13: Hilbert-style proof system of AML: matching and fixpoint reasoning ($C^{\$}$ ranges over applicative contexts)

IV.2.3 Equality

There is no built-in notion of equality in this variant of matching logic. However, equality can be defined in a theory $\Gamma_{\text{DEF}} = \{\text{def } x\}$ over signature with symbols $\Sigma = \{\text{def}\}$, by letting $\varphi_1 = \varphi_2 \equiv \neg(\text{def } (\neg(\varphi_1 \leftrightarrow \varphi_2)))$. The symbol def is known as *definedness* and is axiomatized such that it

- evaluates to empty set whenever applied to an empty set, and
- evaluates to full set whenever applied to a non-empty set.

It can be proved that equality defined in this way admits the following elimination principle: $\vdash_H (\varphi_1 = \varphi_2) \to (C[\varphi_1] \leftrightarrow C[\varphi_2])$. Furthermore, on can define a complementary operator to definedness, usually called *totality*: $tot\ \varphi \equiv \neg\ \text{def } (\neg\varphi)$. Totality admits a kind of a deduction theorem; the intuition is that matching logic formulas wrapped in totality behave in a classical, two-valued way, and have the same semantics inside a theory as on a left-hand side of an implication.

The following variant the of deduction theorem holds in AML.

Theorem 86: AML Deduction theorem

Assume a theory Γ that extends Γ_{def}, and a proof of $\Gamma \cup \{\psi\} \vdash_H \varphi$. If the proof

- does not use the KT rule, and
- does not use the Ex-gen rule on free (element) variables of ψ, and
- does not use the Subst rule on free (set) variables of ψ,

then there exists a proof of $\Gamma \vdash_H (tot\ \psi) \to \varphi$.

Proof: Mechanized in the Coq development accompanying [96], file `Theories/Definedness_ProofSystem.v`, theorem `deduction_theorem_noKT`; the original (slightly more restricted) version is in the extended version [26] of [24] as Theorem 14. □

The deduction theorem is used in the proof of the equality elimination principle. More precisely: we have proved multiple variants of the deduction theorem and of the equality elimination principle, where the stronger versions depend on the weaker ones. Using these versions of the deduction theorem in a backward-reasoning style is complicated by the fact that one is prohibited to use certain proof rules after the deduction theorem is used. We have developed a mechanism to ease working with this limitation and incorporated it into the proof system presented in Section IV.3.

IV.3 A natural deduction system for matching logic

As noted already in [48], Hilbert-style proof systems are not particularly intuitive for constructing proofs. Therefore, we have defined an alternative proof system $\vdash_N$ for matching logic, in the spirit of natural deduction sequent calculi [16], [73] – that is, as a natural deduction system presented in the sequent calculus style. This calculus is shown to be equivalent to $\vdash_H$.

IV.3.1 Proof constraints

The deduction theorem (Theorem 86) has three essential side conditions that constrain the rules usable in the proof of the premise. Unlike the Hilbert-style proof system $\vdash_H$, the proposed sequent calculus allows these conditions to be checked without inspecting the proof. In particular, the proof systems traces the availability of proof rules using the notion of proof constraints. This allows rephrasing the deduction theorem such that it only checks the constraint instead of inspecting the full proof. Besides easing pen-and-paper reasoning, this also speeds up work in an interactive proof assistant, as it eliminates the need to traverse (partially) constructed proofs for the purpose of checking compliance with the restrictions.

The syntax and semantics of proof constraints is defined as follows.

Definition 87: Proof constraint syntax

The set $\mathcal{C}$ of *proof constraints* (or simply *constraints*) is defined by $\mathcal{C} = \mathcal{P}(EV) \times \mathcal{P}(SV) \times \{true, false\}$. The notations c^{eg}, c^{sub}, and c^{kt} are used to denote the first, second, and third component of a constraint $c \in \mathcal{C}$, respectively. Let $\top_{\mathcal{C}} = (EV, SV, true)$, and $\bot_{\mathcal{C}} = (\emptyset, \emptyset, false)$.

Definition 88: Proof constraint semantics

A proof p of $\Gamma \vdash_H \varphi$ *satisfies* a constraint $c \in \mathcal{C}$, written $\Gamma \vdash_H^c \varphi$, iff
1. every $x \in EV$ used in some Ex-Gen step in p satisfies $x \in c^{eg}$;
2. every $X \in SV$ used in some Subst step in p satisfies $X \in c^{sub}$; and
3. if KT was used in p, then c^{kt} is $true$.

Proof constraints can be ordered in the expected way, with $\top_{\mathcal{C}}$ and $\bot_{\mathcal{C}}$ being the greatest and least elements, respectively.

Definition 89: Proof constraint ordering

Given $c_1, c_2 \in \mathcal{C}$, we write $c_1 \sqsubseteq c_2$ to mean that
1. $c_1^{eg} \subseteq c_2^{eg}$, and
2. $c_1^{sub} \subseteq c_2^{sub}$, and
3. if c_1^{kt} is $true$, then also c_2^{kt} is $true$.

Lemma 90: On constraint ordering

If $c_1 \sqsubseteq c_2$, then any proof satisfying c_1 also satisfies c_2.

IV.3.2 Sequents

Sequents carry proof constraints and also implement the intuition of "fake" proof contexts.

Definition 91: Sequent

A sequent is a quadruple $\Gamma \blacktriangleright_c \Delta \vdash_N \psi$, where

1. Γ is a (possibly infinite) set, called *theory*, of matching logic formulas;
2. Δ is a finite list, called *antecedent* or *local context*, of matching logic formulas;
3. ψ is a matching logic formula, called *succedent* or *conclusion*; and
4. $c \in \mathcal{C}$.

(Ordered) lists are used instead of the more typical (unordered) sets for a deep technical reason: to make sequents *syntactically* correspond to implication chains, as suggested in Section IV.1. We adopt the standard notation of writing lists modulo associativity, and of using comma for both list concatenation and construction of a list from a head and a tail – as typically written on paper.

The natural deduction system is defined by rules in Table 1, Table 2, and Table 3.

$\dfrac{\Gamma \blacktriangleright_c \Delta_1, \Delta_2 \vdash_N \psi}{\Gamma \blacktriangleright_c \Delta_1, \varphi, \Delta_2 \vdash_N \psi}$	Weaken
$\dfrac{\Gamma \blacktriangleright_c \Delta_1 \vdash_N \varphi \quad \Gamma \blacktriangleright_c \Delta_1, \varphi, \Delta_2 \vdash_N \psi}{\Gamma \blacktriangleright_c \Delta_1, \Delta_2 \vdash_N \psi}$	Cut
$\dfrac{}{\Gamma \blacktriangleright_c \Delta_1, \varphi, \Delta_2 \vdash_N \varphi}$	Hypothesis
$\dfrac{}{\Gamma \blacktriangleright_c \Delta_1, \bot, \Delta_2 \vdash_N \varphi}$	Bottom Elimination
$\dfrac{\Gamma \blacktriangleright_c \Delta_1, \Delta_2 \vdash_N \varphi_1 \quad \Gamma \blacktriangleright_c \Delta_1, \varphi_2, \Delta_2 \vdash_N \psi}{\Gamma \blacktriangleright_c \Delta_1, \varphi_1 \to \varphi_2, \Delta_2 \vdash_N \psi}$	Implication Elimination
$\dfrac{\Gamma \blacktriangleright_c \Delta, \varphi \vdash_N \psi}{\Gamma \blacktriangleright_c \Delta \vdash_N \varphi \to \psi}$	Implication Introduction
$\dfrac{\Gamma \blacktriangleright_c \Delta_1, \varphi_1, \varphi_2, \Delta_2 \vdash_N \psi}{\Gamma \blacktriangleright_c \Delta_1, \varphi_1 \wedge \varphi_2, \Delta_2 \vdash_N \psi}$	Conjunction Elimination
$\dfrac{\Gamma \blacktriangleright_c \Delta \vdash_N \varphi_1 \quad \Gamma \blacktriangleright_c \Delta \vdash_N \varphi_2}{\Gamma \blacktriangleright_c \Delta \vdash_N \varphi_1 \wedge \varphi_2}$	Conjunction Introduction
$\dfrac{\Gamma \blacktriangleright_c \Delta_1, \varphi_1, \Delta_2 \vdash_N \psi \quad \Gamma \blacktriangleright_c \Delta_1, \varphi_2, \Delta_2 \vdash_N \psi}{\Gamma \blacktriangleright_c \Delta_1, \varphi_1 \vee \varphi_2, \Delta_2 \vdash_N \psi}$	Disjunction Elimination
$\dfrac{\Gamma \blacktriangleright_c \Delta \vdash_N \varphi_1}{\Gamma \blacktriangleright_c \Delta \vdash_N \varphi_1 \vee \varphi_2}$	Disjunction Introduction Left
$\dfrac{\Gamma \blacktriangleright_c \Delta \vdash_N \varphi_2}{\Gamma \blacktriangleright_c \Delta \vdash_N \varphi_1 \vee \varphi_2}$	Disjunction Introduction Right

Table 1: $\vdash_N$ rules for propositional reasoning

$\dfrac{\Gamma \blacktriangleright_c \Delta_1, \varphi\left[\frac{y}{x}\right], \Delta_2 \vdash_N \psi}{\Gamma \blacktriangleright_c \Delta_1, \forall x.\ \varphi, \Delta_2 \vdash_N \psi}$	Universal Quantification Elimination
$\dfrac{\Gamma \blacktriangleright_c \Delta \vdash_N \forall \psi\left[\frac{y}{x}\right] \quad y \in c^{eg} \quad y \notin FV(\Delta, \forall x.\ \psi)}{\Gamma \blacktriangleright_c \Delta \vdash_N \forall x.\ \psi}$	Universal Quantification Introduction
$\dfrac{\Gamma \blacktriangleright_c \Delta_1, \varphi\left[\frac{y}{x}\right], \Delta_2 \vdash_N \psi \quad y \in c^{eg} \quad y \notin FV(\Delta, \exists x.\ \varphi, \psi)}{\Gamma \blacktriangleright_c \Delta_1, \exists x.\ \varphi, \Delta_2 \vdash_N \psi}$	Existential Quantification Elimination
$\dfrac{\Gamma \blacktriangleright_c \Delta \vdash_N \psi\left[\frac{y}{x}\right]}{\Gamma \blacktriangleright_c \Delta \vdash_N \exists x.\ \psi}$	Existential Quantification Introduction

Table 2: $\vdash_N$ rules for first-order reasoning

$\dfrac{\Gamma \vdash_H^c \psi}{\Gamma \blacktriangleright_c [] \vdash_N \psi}$	Inherit
$\dfrac{\Gamma \blacktriangleright_{c'} \Delta \vdash_N \psi \quad c' \sqsubseteq c}{\Gamma \blacktriangleright_c \Delta \vdash_N \psi}$	Relax

Table 3: Technical $\vdash_N$ rules

$\dfrac{\Gamma \cup \{\varphi\} \blacktriangleright_c \Delta \vdash_N \psi \quad c^{eg} \#\# \ FV(\varphi) \quad c^{sub} \#\# \ FV(\varphi) \quad c^{kt} = false \quad \Gamma^{\text{def}} \subseteq \Gamma}{\Gamma \blacktriangleright_{\top_{\mathcal{C}}} \Delta \vdash_N (tot \ \ \varphi) \to \psi}$	Deduction
$\dfrac{\Gamma^{\text{def}} \subseteq \Gamma}{\Gamma \blacktriangleright_{\top_{\mathcal{C}}} \Delta \vdash_N \psi = \psi}$	Equality Introduction
$\dfrac{\Gamma \blacktriangleright_{\top_{\mathcal{C}}} \Delta_1, \varphi_1 = \varphi_2, \Delta_2 \vdash_N C[\varphi_2] \quad C \text{ does not contain } \mu \quad \Gamma^{\text{def}} \subseteq \Gamma}{\Gamma \blacktriangleright_{\top_{\mathcal{C}}} \Delta_1, \varphi_1 = \varphi_2, \Delta_2 \vdash_N C[\varphi_1]}$	Equality Elimination

Table 4: Definedness-related $\vdash_N$ rules

Lemma 92: Correspondence

$$(\Gamma \blacktriangleright_c \varphi_1, ..., \varphi_k \vdash_N \psi) \Leftrightarrow (\Gamma \vdash_H^c \varphi_1 \to ... \to \varphi_k \to \psi)$$

Proof: For the $\Rightarrow$ implication, see the Coq formalization available at https://github.com/harp-project/AML-Formalization/releases/tag/v1.0.15, subdirectory `ProofMode`, where the approach sketched in Section IV.1 is formalized. For the $\Leftarrow$ direction, use Inherit followed by k-times Implication Introduction. □

Thus, the natural system with $k = 0$ is equivalent to the Hilbert one. In the Coq implementation, this equivalence allows one to transition in and out of our proof mode safely.

IV.4 Coq implementation of the Proof Mode

This section describes the *Coq proof mode for matching logic* (MLPM), available as [10]. It starts with a brief description of the embedding of $\vdash_H$ in Coq, as described in [9]; continues with an example of the use of MLPM follows; and ends with a description of the architecture of MLPM.

IV.4.1 Matching logic proof system in Coq

MLPM is based on the mechanization of matching logic in Coq reported in [9], where we developed a deep embedding of matching logic formulas, models, and proof system into Coq. We used the *locally nameless* encoding of binders, developed by A. Charguéraud [20], which means that bound variables are represented by indices (that is, natural numbers) and free variables by names (typically

strings); additionally, a well-formedness predicate is used to filter out representations that do not correspond to any formula. Reasoning about well-formedness of formulas presents a difficulty from the usability point of view, since most rules of the embedded proof system require the involved patterns to be well-formed. Two ways in which we overcome this difficulty are discussed in Section IV.4.4.

The price paid for using a deeply embedded proof system of [9] is that we could not easily reuse Coq's handling of propositional and first-order connectives. However, one can argue that a deep embedding of the matching logic from [9] is appropriate for our purposes. First, the calculus contains the Equality Elimination rule, which is based on its Hilbert-style counterpart from [27]. This rule relies on the deduction theorem of matching logic, which is proved by induction on the structure of the input proof—which is easier to do when one treats proofs as deeply-embedded, inductively-defined data. Another reason for using a deep embedding is that we planned to integrate our development with the work of X. Chen, Z. Lin, M.-T. Trinh, and G. Roşu [21] on proof checking, and do so by extracting the deeply-embedded proof objects from Coq to the external world using Coq's built-in extraction mechanism.

IV.4.2 MLPM in Action

Many features of MLPM can be demonstrated using Figure 14. Throughout the figure, the notation $\varphi\hat{}\{evar : n \mapsto \psi\}$ means φ with substituted pattern ψ for the de Bruijn index n representing a bound variable; $= ml$ denotes the matching logic equality (Section IV.2.3). The notations `all`, `$`, and `and` denote universal quantification, application, and conjunction, respectively.

The frames represent Coq's proof state; everything below the full line is Coq's goal; everything above the full line is Coq's context (which is omitted in all cases but the first one). When inside MLPM, the Coq's goal corresponds to a sequent (Definition 91). The goal is divided by the $\vdash$ symbol and a dashed line into three parts: the matching logic theory Γ, which is a (possibly infinite) set of patterns; the local context, which is a list of *named hypotheses*; and the conclusion. For example, the goal after proof step `(* 1 *)` corresponds to the sequent $\Gamma \blacktriangleright_{\top_{\mathcal{C}}} (\forall y.\ \varphi_2 = \varphi_1) \wedge \left(p\ \left(\varphi_1\left[\frac{x}{y}\right]\right)\right) \vdash_H \exists y.\ (p\ \varphi_2)$. In this example, the proof constraint is $\top_{\mathcal{C}}$ (that is, every proof rule is allowed) throughout the execution of the whole proof script, and is therefore automatically hidden. However, it is checked at step `(* 5 *)` when rewriting using equality (as in the Equality Elimination rule); the proof step would fail if the actual constraint were too strict.

We implemented automation for discharging freshness (non-occurrence of a variable in a pattern) proof obligations; similarly, we implemented the tactic `wf_auto2` for discharging well-formedness proof obligations generated when applying certain lemmas (such as `evar_open_closed`). However, most MLPM tactics do not require proving well-formedness, as tactics preserve well-formedness. All the names of MLPM tactics have the prefix `ml` and resemble names of Coq tactics with similar behavior. Tactics outside MLPM (not having the ml prefix) are used

to further simplify the goal and to discharge technical proof obligations. An interested reader is invited to see the proof mode tutorial (`examples/02_proofmode/theories/tutorial.v`) of the formalization.

```
Σ: Signature     defsyntax: definedness_syntax     Γ: Theory
HΓ: definedness_theory ⊆ Γ     x: evar     p, ϕ₁, ϕ₂: Pattern
mff: mu_free p     wff: well_formed p     wfϕ₁: well_formed (ex , ϕ₁)
wfϕ₂: well_formed (ex , ϕ₂)
____________________________________________________________________
Γ
⊢ (all , ϕ₂ =ml ϕ₁) and p $ ϕ₁^{evar:0↦x} --->
  (ex , p $ ϕ₂)

  (* 1 *) mlIntro "H".

____________________________________________________________________
Γ ⊢
"H" : (all , ϕ₂ =ml ϕ₁) and p $ ϕ₁^{evar:0↦x},
--------------------------------------
ex , p $ ϕ₂

  (* 2 *) mlDestructAnd "H" as "H1" "H2".
  (* 3 *) mlSpecialize "H1" with x.

____________________________________________________________________
Γ ⊢
"H1" : (ϕ₂ =ml ϕ₁)^{evar:0↦x},
"H2" : p $ ϕ₁^{evar:0↦x},
--------------------------------------
ex , p $ ϕ₂

  (* 4 *) mlExists x.
           mlSimpl. rewrite [p^{evar:0↦x}]evar_open_closed;[wf_auto2|].

____________________________________________________________________
Γ ⊢
"H1" : ϕ₂^{evar:0↦x} =ml ϕ₁^{evar:0↦x},
"H2" : p $ ϕ₁^{evar:0↦x},
--------------------------------------
p $ ϕ₂^{evar:0↦x}

  (* 5 *) mlRewriteBy "H1" at 1;[auto|(
                apply mu_free_in_path;cbn;by rewrite mff)|].

____________________________________________________________________
Γ ⊢
"H1" : ϕ₂^{evar:0↦x} =ml ϕ₁^{evar:0↦x},
"H2" : p $ ϕ₁^{evar:0↦x},
--------------------------------------
p $ ϕ₁^{evar:0↦x}

  (* 6 *) mlExact "H2".
```

Figure 14: An example interactive MLPM proof

IV.4.3 Proof constraints

When using Theorem 86 on paper, one has to be careful not to forget checking the theorem's side conditions. When using it in Coq, the proof assistant automatically

generates the corresponding proof obligations. But how to prove such obligations? Since we have formalized the concepts in the side-conditions of the deduction theorem as computable, bool-valued functions on the type of matching logic proofs, one can try to discharge the obligations by a simple computation.

However, after some time, we run into issues with this approach. First, for larger proofs, the computation can be slow. Second, this approach is not compositional: when a theorem T is applied after using the deduction theorem, the proof of T has to be examined despite the possibility that it has already been examined before – and with this approach, subproofs do not "remember" whether they have been examined or not. Third, we usually do not work with concrete theorems about concrete patterns, but with meta-theorems that are parameterized by patterns. This means that some function or theorem applications that occur in matching logic proofs cannot be fully reduced to constructor terms. Typically, the symbolic computation gets stuck on a dependent pattern matching.

Therefore, we resorted to a different approach. In our development, we annotate every theorem of $\vdash_H$, including its proofs rules, with a proof constraint that represents an overapproximation of the proof rules used in the proof of the theorem. Annotated theorems carry with them a Coq proof ensuring that the ML proof satisfies the constraint. The formalization of Theorem 86 then consumes such annotated proofs, and is then used to prove the soundness of the Equality Elimination principle. To ensure decidability, proof constraints do not use arbitrary sets but only co-finite sets in components of a constraint.

IV.4.4 Architecture of MLPM

```
Lemma prf_conj_split
{Σ : Signature} Γ a b l:
well_formed a ->
well_formed b ->
Pattern.wf l ->
Γ ⊢i (foldr patt_imp a l) --->
    (foldr patt_imp b l) --->
    (foldr patt_imp (a and b) l)
using BasicReasoning.

Lemma MLGoal_splitAnd
{Σ : Signature} Γ a b l i:
of_MLGoal (mkMLGoal Σ Γ l a i) ->
of_MLGoal (mkMLGoal Σ Γ l b i) ->
of_MLGoal (mkMLGoal Σ Γ l (a and b) i).
```

```
Ltac mlSplitAnd :=
_ensureProofMode;
apply MLGoal_splitAnd.
```

Listing 1: The architecture of conjuntion introduction

Conceptually, we recognize three layers in which MLPM is organized: (1) annotated admissible rules of $\vdash_H$, (2) proof-state transforming theorems, and (3) user-facing tactics. Typically, the tactics of layer (3) apply theorems of layer (2), which are based on layer (1) theorems, which are proved using already existing tactics of layer (3). For example, consider the `mlSplitAnd` tactic (implementing the Conjunction Introduction rule) whose architecture is shown in Listing 1.

In layer (1), we have the lemma `prf_conj_split`, which is proved by induction on `l` and repeated applications of Implication Introduction, Hypothesis, Weaken, and Cut (which are already defined at that point in the development). In layer (2), we have the lemma `MLGoal_splitAnd`, which is proved using `prf_conj_split`. Here, `mkMLGoal` is the constructor of the type `MLGoal` representing sequents, and is coerced by a function named `of_MLGoal` (Listing 2) to a corresponding (in the sense of Lemma 92) proof obligation about $\vdash_H$. The lemma `MLGoal_splitAnd` thus justifies the soundness of the Conjunction Introduction rule by transforming $\vdash_H$ proofs corresponding to premises of the rule to a $\vdash_H$ proof corresponding to its conclusion. This way, we do not have one global theorem of soundness of $\vdash_N$; rather, rules of $\vdash_N$ are justified by the corresponding lemmas from layer (2). This design allowed us to implement the rules incrementally.

```
Coercion of_MLGoal {Σ : Signature} (MG : MLGoal) : Type := Coq
let ψ := (mlConclusion MG) in let ϕs := (patterns_of (mlHypotheses MG)) in
well_formed ψ -> Pattern.wf ϕs ->
(mlTheory MG) ⊢i (fold_right patt_imp ψ ϕs) using (mlInfo MG).
```

Listing 2: A coercion from sequents to $\vdash_H$ (`ProofMode/Basics.v`). The `fold_right _ _ _` part represents the chain of implications of Lemma 92.

Listing 1 also illustrates another aspect of MLPM: its goals are shallowly embedded into Coq goals. The tactic `mlSplitAnd`, when invoked, generates two Coq goals, with a single MLPM goal (i.e., sequent) in each of them. This means that we reuse Coq's infrastructure for managing goals; that is, goal selectors, shelving, etc. Overall, our layered architecture with the coercion from the type of the sequent resembles that of Iris Proof Mode of R. Krebbers, A. Timany, and L. Birkedal [60].

IV.5 Evaluation of MLPM

In [96] we also evaluated the proof mode from the *succintness* point of view on a manually selected set of $\vdash_H$ goals; in that paper, an interested reader will also find hyperlinks to the actual evaluation artifacts.

Each experiment consits of one theorem of $\vdash_H$. We proved each theorem in multiple ways and observed various metrics, such as proof script size, for each way. The observed values have to be taken with a grain of salt, as there are often multiple ways of proving such theorems, and while we tried to come up with short proofs, we did not attempt to provide any formal guarantee of their minimality. One could also only informally argue that the proofs we used are "natural", in the sense that they were written by humans (us, the authors of the paper), and not generated by some automation.

Consider Table 5. The individual rows have the following meaning.

- *Proof size*: the number of $\vdash_H$-rules used in the resulting proof (after reduction to a constructor normal form).
- *Steps*: the number of high-level steps used in the proof script, which does not include solving technical side conditions (e.g., well-formedness, freshness). These steps include the use of MLPM tactics, application of existing theorems, or proof rules.
- *Length*: the total character count of the proof script (not considering leading white-spaces, comments).
- *Subproofs*: the number of subproofs that are explicitly written in the script to solve constraints about well-formedness, variable freshness, and replacing equal formulas (unfolding definitions, simplifying substitutions).
- *Formulas*: the number of matching logic formulas that have been manually specified as parameters during the proof.

The meaning of columns is as follows.

- $\vdash_H$: a manual proof using $\vdash_H$-based proof rules and lemmas (no proof mode tactics or lemmas proved with them were used).
- *MLPM1*: MLPM was used but the use of conjunction introduction (the `mlSplitAnd` tactic) was prohibited.
- *MLPM2*: MLPM without any restrictions
- *Ltac*: a proof of the same propositional statement expressed in Coq's logic, proved using Coq's own tactic language Ltac.

The use of MLPM results in smaller proof script (comparable to the equivalent Ltac proof) with fewer proof steps and fewer side conditions to be proved manually. The user only rarely needs to write matching logic patterns. The downside of using MLPM is that the generated proof terms are larger.

	$\vdash_H$	MLPM1	MLPM2	LTac
Proof size	103	2129	6279	-
Steps	34	10	4	4
Length	1212	224	65	31
Subproofs	43	1	0	0
Formulas	36	1	0	0

$$\Gamma \vdash_H^{\perp_e} \varphi \to \psi \to \varphi \wedge \psi$$

Table 5: A simple propositional tautology.

Table 6 requires reasoning about quantifiers. The rows have the same meaning as in Table 5; similarly, the meaning of the $\vdash_H$ and LTac columns is the same. Otherwise,

- *FOL MLPM* is MLPM without the use of propositional tactics (only tactics for quantifiers are allowed); and
- *MLPM* is MLPM without restrictions.

The situation is similar as in Table 5. The number of the proof steps is the same in all three proofs, since helper lemmas were used in the $\vdash_H$ proofs. Interestingly, the first-order tactics of MLPM increase the size of the generated proof term only slightly. This is because the first-order rules of $\vdash_N$ (on which the first-order MLPM tactics are based) are more direct consequences of the first-order $\vdash_H$ proof rules (Ex-Quan and Ex-Gen) than the propositional ones of the propositional rules.

	$\vdash_H$	FOL MLPM	MLPM	LTac
Proof size	488	574	1696	-
Steps	6	6	6	5
Length	810	244	138	75
Subproofs	18	3	0	0
Formulas	5	1	0	0

$$\Gamma \vdash_H^{\top e} (\exists x.\ \varphi_1 \wedge \varphi_2) \to \exists x.\ \varphi_1$$

Table 6: A first-order goal

Table 7 shows a goal that requires reasoning about equivalence of formulas. The meaning of the rows is as in the previous tables; the meaning of the columns is as follows.

- *Framing*: the Framing rule was used directly in the proof; no MLPM tactics were used.
- *Congruence*: a Congruence lemma $((\Gamma \vdash_H \varphi_1 \leftrightarrow \varphi_2) \Rightarrow (\Gamma \vdash_H C[\varphi_1] \leftrightarrow C[\varphi_2]))$ was used.
- *MLPM*: no restriction on MLPM; in particular, the `mlRewrite` tactics (behaving similarly to Coq's own `rewrite` tactic) based on Congruence lemma was used.
- *LTac*: the corresponding Coq proof, written in LTac.

Using the Congruence lemma directly results in more complex proof scripts and larger proof terms than using the Framing rule; on the other hand, Framing can be used only for applicative contexts without other connectives. In the Congruence case, the Congruence lemma was used manually with explicitly specified context `C := ϕ_1 x` (with a fresh variable name `x` used in place of $\square$), while in the MLPM case, the `mlRewrite` tactic applied the Congruence lemma with C automatically inferred as $(\varphi_1\ x) \to (\varphi_1\ \varphi_4)$. Again, the MLPM proof is shorter than the manual ones. Interestingly, it is even shorter than the Ltac proof, because the latter one has to explicitly use the axiom of propositional extensionality to turn the assumed equivalence into equality.

	Framing	Congruence	MLPM	LTac
Proof size	534	6005	42889	-
Steps	4	4	3	4
Length	132	418	59	81
Subproofs	3	10	0	0
Formulas	1	1	0	0

$$\left(\Gamma \vdash_H^{\top e} (\varphi_2\ \varphi_3) \leftrightarrow \varphi_4\right) \Rightarrow \left(\Gamma \vdash_H^{\top e} (\varphi_1(\ \varphi_2\ \varphi_3)) \to \varphi_1\ \varphi_4\right)$$

Table 7: A rewriting-based goal

Table 8 shows the results of experiments for the running example of Figure 14. With the hand-crafted $\vdash_H$ proof we followed the same proof strategy as in the MLPM case, using formalized Hilbert-style theorems. Therefore, the resulting proof terms have very similar size. However, when using MLPM, the proof script is smaller, with fewer subproofs.

	$\vdash_H$	MLPM
Proof size	10′650′557	10′653′299
Steps	10	7
Length	926	252
Subproofs	35	4
Formulas	1	1

$$\Gamma \vdash_H^{\top_e} (\forall x.\ \varphi_2(x) = \varphi_1(x)) \wedge (p\ \varphi_1(y)) \to (\exists x.\ p\ \varphi_2(x))$$

Table 8: Complex goal

We also proved a couple of less artificial theorems – that is, theorems that occur naturally in our development – using MLPM, such as

$$\Gamma \vdash_H^{\top_e} \varphi\left[\frac{\psi}{x}\right] \wedge (\exists y.\ \psi = y) \to \exists x.\ \varphi$$

and

$$\Gamma \vdash_H^{\top_e} (\forall x.\ \varphi) \wedge (\exists y.\ \psi = y) \to \varphi\left[\frac{\psi}{x}\right]$$

thus demonstrating the ability of MLPM to handle real-world examples. These theorems correspond to the FOL-style introduction of existential quantifier and elimination of universal quantifier, where the condition $\exists y.\ \psi = y$ ensures that ψ represents an FOL-style term.

IV.6 Concluding remarks

In general, reasoning with a Hilbert system differs from how proofs are explained in practice [48]. To bridge this gap between Hilbert proofs and mathematical deduction, Gentzen developed new formal calculi (natural deduction and sequent calculus) [48] that are equivalent to the Hilbert system. Afterward, many different versions of these calculi emerged; among these, one may want to highlight [16], [41] that contain single-conclusion sequent calculi similar to our own system for matching logic.

When implementing interactive reasoning support for a logic, one has basically two options: either to write a standalone tool for it (e.g., in [16] Bornat and Sufrin implement a proof calculator), or to embed their logic to an existing proof assistant and develop a “proof mode“ for the logic, as is the example of Iris Proof

Mode [60] for a variant of concurrent separation logic. Other works from this second category include, for example, [77], [101] for (focused) linear logic, [35] for linear temporal logic, [14] for differential dynamic logic, and [4], [59], [67] for various separation logic dialects. The main difference between [60] and our work is, besides the fact that the target logic is different, that our work is based on a deep embedding of the target logic. However, the work on Iris Proof Mode was still a major source of inspiration for me when starting the project.

Deep embedding of expressive domain logic in proof assistants for interactive proof construction is a challenging engineering problem; nevertheless, it comes with benefits: deep embedding allows one to do induction on the syntax and the proofs, supporting, amongst others, proof extraction and proving meta-theorems. On the other hand, proving object-level theorems becomes more tedious because the proof assistant's capabilities are not directly applicable to reasoning about object-level judgements.

Despite the progress in this line of work on logical foundations of 𝕂, I decided to not continue in this research direction. Instead, I attempted to address some of the issues in the design of 𝕂 mentioned in this section and in Section II.4.2 by designing and implementing new semantics framework: Minuska.

V Minuska: A formal formal semantics framework

$\mathbb{K}$ is a framework centered around formal semantics of programming languages - that is, it is a *formal semantics* framework. However, the lack of solid mathematical foundations for the languages used to represent programming language semantics in $\mathbb{K}$ makes me consider $\mathbb{K}$ not to be *truly formal*. The title of this chapter refers to a framework for formal semantics of programming languages, where the framework itself has formal semantics and formally proven properties.

A natural question is: Why build a new framework instead of simply fixing the existing one in a backwards-compatible manner? The answer is rooted in the weaknesses of $\mathbb{K}$ that I already discussed in Section II.4.2.3, the three most painful points being

1. the lack of a formal semantics of Kore and the complexity of the informal agreed-on semantics (that is not publicly documented),
2. the possibility of introducing a logical inconsistency when describing a language semantics in $\mathbb{K}$, and
3. the low-level nature of applicative matching logic (used to justify operations of $\mathbb{K}$).

Minuska is designed to avoid these problems. Minuska has simple formal semantics; the interpreter of programs in Minuska-defined languages is implemented using the Coq proof assistant and formally proven correct. Minuska's input language is strong enough to express the motivating example of X. Chen, Z. Lin, M.-T. Trinh, and G. Roşu [21] that the $\mathbb{K}$-based certifying interpreter of [21] could not handle, but Minuska's verified interpreter can. Moreover, on the benchmarks supported by both [21]'s $\mathbb{K}$ and Minuska, the verified interpreter of Minuska outperforms the certifying interpreter of [21], which was said to show "promising performance", by a factor of at least 20.

The main technical contributions of Minuska are:

1. *MinusLang* – a language embedded in Coq, with formal semantics, for specifying operational semantics of programming languages, with no room to introduce logical inconsistency to the language definition;
2. an *interpreter* for any programming language that has a well-formed MinusLang semantics;
3. a mechanized *proof of the soundness and completeness* of that interpreter.

On the conceptual level, the main takeaways of the Minuska experiment are the following:

1. One can have a programming language framework that is practical, formally verified, and inexpensive to develop. (It took about 4 person-months to develop the framework with the verified interpreter).
2. The use of foundational formal verification can result in a significantly more performant tool than retrofitting after-the-task certificate generation and checking to a professionally developed but unverified tool, even without clever optimizations on the side of the verified tool.

This chapter focuses on the foundations of Minuska and the formally verified interpreter, as presented in J. Tušil and J. Obdržálek [97]. More work in this area is in progress and some is left for future; I report on these in Chapter VI. Special thanks goes to Traian Șerbănuță for participating in fruitful conversations in the very beginnings of the project. The implementation of the Minuska framework is available at https://github.com/h0nzZik/minuska.

V.1 On semantics frameworks

A *programming language semantic framework* is a software tool in which formal semantics of various programming languages can be encoded and that derives other artifacts from the semantics. Among the most widely known semantic frameworks, there is Ott [89] that generates parsers, LaTeX code, and theorem prover (Coq, HOL, and Isabelle/HOL) definitions from a definition of a language's syntax and semantics; $\mathbb{K}$ [28] that generates parsers, interpreters, and deductive verifiers; and PLT-Redex [56] that generates parsers, interpreters, and LaTeX code. To the best of my knowledge, none of these tools has been formally verified. As mentioned earlier, Minuska is directly inspired by $\mathbb{K}$.

In the context of semantics frameworks, interpreters enable language designers to test that a formally defined language semantics behaves as intended. One has just to generate an interpreter and test, whether its exhibited behaviour corresponds to designers' intentions. This has already been observed in the literature around the $\mathbb{K}$, e.g. in [38]. However, for this testing to be effective, the generated interpreter needs to *faithfully capture* the language semantics[12]. Thus, an automatically generated interpreter formally verified with respect to the semantics would also increase the trustworthiness of the whole framework, including the languages modelled in it.

V.2 Architecture of Minuska

Minuska is implemented in the Coq proof assistant [95] and consists of three main parts:

1. the language *MinusS* for static reasoning about program configurations,
2. the language *MinusLang* for describing computational steps, and
3. a formally verified generator of interpreters.

To make developing a language definition in MinusLang easier to write, there is another component, the *frontend*, which takes as an input language definition in *sugared MinusLang*, which is an extension of MinusLang, and *desugars* the definition to MinusLang. Currently, the meaning of sugared MinusLang is given by its translation to MinusLang; a direct formal semantics for sugared MinusLang is left for future work. The translation is written in Coq. There is yet another component of Minuska, the *parser* (written in OCaml), which lifts a language definition from *concrete syntax* of sugared MinusLang written in a standalone file to Coq. Listing 3

[12] Automated testing of a verified compiler for a fixed language is discussed, e.g., in [70].

shows a full definition of a simple imperative programming language in concrete syntax of sugared MinusLang.

```
@frames: [simple(CODE): c[builtin.cseq [CODE,REST], STATE]];
@value(X): (bool.or(
    z.is(X), bool.or(
    bool.is(X),
    term.same_symbol(X, [unitValue[]])))) ;
@context(HOLE): c[HOLE, STATE];
@strictness: [
    neg of_arity 1 in [0],
    plus of_arity 2 in [0,1],
    minus of_arity 2 in [0,1],
    assign of_arity 2 in [1],
    seq of_arity 2 in [0],
    ite of_arity 3 in [0],
    eq of_arity 2 in [0,1],
    le of_arity 2 in [0,1],
    lt of_arity 2 in [0,1]
];
@rule [init]:
    builtin.init[X] => c[builtin.cseq[X, builtin.empty_cseq[]], map.empty()]
    where bool.true();
@rule/simple [aexpr.plus]:
    plus[X,Y] => z.plus(X, Y)
    where bool.and(z.is(X), z.is(Y)) ;
@rule/simple [aexpr.minus]:
    minus[X,Y] => z.minus(X, Y)
    where bool.and(z.is(X), z.is(Y)) ;
@rule [var.assign]:
    c[builtin.cseq[assign[X,V],REST], STATE]
 => c[builtin.cseq[unitValue[], REST], map.update(STATE, X, V)]
    where bool.and(term.same_symbol(X, [var[]]), z.is(V)) ;
@rule [var.lookup]:
    c[builtin.cseq[X, REST], STATE]
 => c[builtin.cseq[map.lookup(STATE, X), REST], STATE]
    where term.same_symbol(X, [var[]]) ;
@rule/simple [stmt.seq]: seq[unitValue[], X] => X where bool.true() ;
@rule/simple [bexpr.eq]: eq[X, Y] => z.eq(X, Y) where bool.and(z.is(X), z.is(Y)) ;
@rule/simple [bexpr.le]: le[X, Y] => z.le(X, Y) where bool.and(z.is(X), z.is(Y)) ;
@rule/simple [bexpr.lt]: lt[X, Y] => z.lt(X, Y) where bool.and(z.is(X), z.is(Y)) ;
@rule/simple [bexpr.neg]: not[X] => bool.neg(X) where bool.is(X) ;
@rule/simple [stmt.ite.true]:
    ite[B, X, Y] => X where bool.eq(B, bool.true()) ;
@rule/simple [stmt.ite.false]:
    ite[B, X, Y] => Y where bool.eq(B, bool.false()) ;
@rule/simple [while.unfold]:
    while[B, S] => ite[B, seq[S, while[B, S]], unitValue[]] where bool.true() ;
```

Listing 3: A language definition of IMP in concrete syntax of sugared MinusLang

While Minuska is directly inspired by $\mathbb{K}$ and uses similar concepts, it is based on a different foundational theory – *first-order term rewriting over primitive values*. This design decision has been crucial to formally verify Minuska. To formally capture

reasoning about program configurations and their relations with semantic rules, $\mathbb{K}$ uses matching logic, which is arguable more complex - for instance, matching logic has quantifiers. If one wanted to formally verify reasoning of $\mathbb{K}$ in a proof assistant such as Coq, one would have to formalize not only the matching logic itself but also many important idioms that $\mathbb{K}$ uses on top of matching logic.

Similarly, one would need to define a matching logic theory of built-in data types implemented in $\mathbb{K}$, such as integers, lists, and maps. Neither is considered in the existing work on the mechanization of matching logic in proof assistants, such as [96]. Also, note that the built-in data types of $\mathbb{K}$ are not implemented in the Metamath-based formalization of matching logic in [21]. In Minuska, the need to axiomatize behavior of primitive values is completely avoided – Minuska borrows primitive values, their behavior and properties directly from Coq and its libraries. The result is much simpler than what would be needed to formalize $\mathbb{K}$, and avoids the soundness holes mentioned in Section II.4.2.

V.2.1 MinusS

In Minuska, program configurations are represented as *ground terms* – first-order terms without variables but possibly with special constants: *primitive values*. Language definitions consists of *semantic rules*, which are composed of *symbolic terms* and *side conditions*, both of which use *expressions*. The (static) relations between ground terms, symbolic terms, expressions, and side conditions, are defined by what we can call the language MinusS.

To keep things simple, the design of Minuska is kept unsorted. From now on, let us fix a set S of *symbols* (or more precisely, *term-symbols*; one would usually use the set of character strings, or some other countably infinite set); a set X of *variables*; a (possibly infinite but countable) set V of *primitive values*; and a finite set F of *function-symbols*. In the subsequent text, the meta-variables s, s', s_1, s_2 and similar stand for symbols; x, x', x_1, x_2 and similar stand for variables; v, v', v_1, v_2; and similar stand for primitive values; and f, f', f_1, f_2 and similar stand for function-symbols.

For an example, a configuration of a simple imperative program may look like

```
c[ (y := x + y; z := y - 1), (x |-> 3, y |-> 4) ]
```

where `c` is a symbol, `y := x + y; z := y - 1;` is the program to be executed and `x |-> 3, y |-> 4` is a primitive value of a dictionary type (representing the execution environment) that maps the program variable `x` to value `3` and the program variable `y` to value `4`. (Both `3` and `4` are primitive values of the integer type). This configuration is matched by the symbolic term

```
c[ (V := E; Rest), Env]
```

where `V` is matched to `y`, `E` to `x + y`, `Rest` to `z := y - 1`, and `Env` to `(x |-> 3, y |-> 4)`. The formal definitions follow.

Definition 93: Ground Terms

The set T (of *ground terms*) is defined as the smallest set T such that
1. $v \in V$ implies $v \in T$ (a value is a term),
2. $s \in S$ and $t_1, ..., t_n \in T$ implies $s[t_1, ..., t_n] \in T$ (a term-symbol applied to terms is a term).

Definition 94: Symbolic Terms (Patterns)

The set P (of *symbolic terms* or *patterns*) is defined as the smallest set P such that
1. $x \in X$ implies $x \in T$ (a variable is a pattern),
2. $v \in V$ implies $v \in T$ (a value is a pattern),
3. $s \in S$ and $t_1, ..., t_n \in P$ implies $s[t_1, ..., t_n] \in P$ (a term-symbol applied to patterns is a pattern).

The notion of an *expression* is used to represent operations on primitive values and other ground terms.

Definition 95: Applicative algebraic signature

An *algebraic signature* is a finite set F (representing *function symbols*).

Definition 96: Applicative algebra

Given an applicative algebraic signature F, an *F-algebra* is a pair $\left(A, (r_f)_{\{f \in F\}}\right)$ of a set A and an F-indexed family of functions $r_f : A^\star \to A$.

Assume also that T forms an algebra with some $(r_f)_{\{f \in F\}}$; that is, $\left(T, (r_f)_{\{f \in F\}}\right)$ is a F-algebra.

Definition 97: Expressions

Given an algebraic signature F, the set E of *expressions* is defined as the smallest set E such that
1. $x \in E$ for $x \in X$;
2. $v \in E$ for $v \in V$; and
3. $f \in F$ and $e_1, ..., e_n \in E$ implies $f(e_1, ..., e_n) \in E$.

Definition 98: Expression-Terms

Given an algebraic signature F, the set ET of *expression-terms* is defined as the smallest set ET such that
1. $e \in ET$ for $e \in E$ (an expression is an expression-term) ;
2. $s \in S$ and $et_1, ..., et_n \in ET$ implies $s[et_1, ..., et_n] \in ET$ (a symbol applied to expression terms is an expression term).

To connect ground and symbolic terms, we use the notion of a *valuation*. A valuation δ is a finite partial function from variables to ground terms; let Δ denote the class of all valuations, and again use metavariables $\delta, \delta', \delta_1, \delta_2$ and similar to range over elements of Δ.

Definition 99: Expression evaluation

Given a valuation δ and an expression e such that all variables of e are in the domain of δ, the value of e in δ, written $\delta(e)$, is extended beyond variables by
1. $\delta(v) = v$; and
2. $\delta(f(e_1, ..., e_n)) = r_f(\delta(e_1), ..., \delta(e_n))$.

Definition 100: Side condition evaluation

A side condition (e_1, e_2) *holds* in a valuation ρ, written $\rho \vDash_c e_1 = e_2$, iff for any $t_1, t_2 \in T$, if $\rho, t_1 \vDash_e e_1$ and $\rho, t_2 \vDash_e e_2$ then $t_1 = t_2$. A list l of side conditions *holds* in a valuation ρ, written $\rho \vDash_{scs} l$, if $\rho \vDash_c c$ for every $c \in l$.

Definition 101: Pattern matching relation

The *pattern matching relation* $_, _ \vDash_p _ \subseteq T \times \Delta \times P$ is defined as the smallest relation such that
1. $v, \delta \vDash_p v$;
2. $t, (\{(x, t)\} \cup \delta) \vDash_p x$; and
3. $t_i, \delta \vDash_p p_i$ for all $i \in \{1, ..., n\}$ implies $s[t_1, ..., t_n], \vDash_p s[p_1, ..., p_n]$.

Definition 102: Expression-term matching relation

The *expression-term matching relation* $_, _ \vDash_{et} _ \subseteq T \times \Delta \times ET$ is defined as the smallest relation such that
1. $\delta(e), \delta \vDash_{et} e$;
2. $t_i, \delta \vDash_{et} et_i$ for all $i \in \{1, ..., n\}$ implies $s[t_1, ..., t_n], \delta \vDash_{et} s[et_1, ..., et_n]$

We also need to define the substitution function $subst : T \times X \times T \to T$ in the usual way - that is, inductively, by
1. $subst(x, x, t) = t$;

2. $subst(y, x, t) = y$ if $y \in X$ and $x \neq y$; and
3. $subst(s[t_1, ..., t_n], x, t) = s[subst(t_1, x, t), ..., subst(t_n, x, t)]$,

and use the notation $t\left[\frac{t'}{x}\right]$ for $subst(t, x, t')$. Let $FV(t)$ to denote the set of (free) variables of a symbolic term, or an expression, or a set or list thereof.

V.2.2 MinusLang

At runtime, programs evolve by applying rewriting rules of the form $l \Rightarrow r$ if $c_1, ..., c_k$ to configurations. Therefore, the language MinusLang builds on top of MinusS and adds *rewriting rules* to it. A finite set Γ of rewriting rules is called a *rewriting theory*. Such a theory encodes semantics of a given programming language. Both l and r are terms with variables, where r can also contain applications of function-symbols; hence l is a symbolic-term and r an expression-term. Rewriting rules are conditional.

For example, take the following rule `aexpr.plus`, for binary addition, from the IMP definition in fig:minuska-imp-full:

```
plus[X,Y] => z.plus(X, Y) where bool.and(z.is(X), z.is(Y))
```

Its left-hand side matches any term where the `plus` symbol is the top level operator, applied to two operands; the side condition introduced by the `where` keyword states that both operands have to be integers; and the right-hand side adds the two operands using the function for integer addition from the selected algebra.

Given a rewriting theory Γ, let $_ \rightsquigarrow_\Gamma _ \subseteq T \times T$ be the binary relation between ground terms defined by: $g_1 \rightsquigarrow_\Gamma g_2$ iff there exists some rule $(l \Rightarrow r \text{ if } c_1, ..., c_k) \in \Gamma$ and a valuation δ such that $\delta, g_1 \vDash_p l$, $\delta, g_2 \vDash_{et} r$, and $\delta \vDash_c c_i$ for all $i \in \{1, ..., k\}$.

V.2.3 Sugared syntax for MinusLang

However, the rule above (from Section V.2.2) is not directly applicable to configurations like the one in the Section V.2.1, which contains both a sequence of statements and an environment. Even for simpler configurations of the form

```
[expression, state]
```

c

we want to pattern-match only on the `expression` part of the configuration. This is achieved by the `@rule/simple [aexpr.plus]:` declaration, which states that the `aexpr.plus` rule uses the `simple(CODE): c[CODE, STATE] frame`. In such a frame, only the part corresponding to the meta-variable `CODE` is used for pattern matching and subsequent evaluation (cf. the `@frames:` declaration in the first line of Listing 3). Additionally, `@strictness` declarations like

```
plus of_arity 2 in [0,1]
```

of Listing 3 are used to generate rules that ensure that both arguments of `plus` are evaluated to values. A term is considered a value according to the predicate defined by the keyword `@value`. Finally, `@context` plays a role simillar to that of `@frames` for the strictness declarations.

This sugared syntax makes the MinusLang definitions more concise and readable.

V.2.4 Algebra of primitive values

In 𝕂, the user can describe the desired behavior of primitive values by using axioms in matching logic, which can lead to inconsistencies or underspecification, with no way to detect these. In Minuska, one has to directly provide a model – an algebra of (terms over) primitive values – which avoids these problems. The question is then how to describe such a model. Since Minuska is built inside Coq, one can use the full power of the calculus of inductive construction – the logic of Coq that has been proven to be sound with respect to set theory. While it is true that Coq allows its users to assume some axioms without proving them, this can be easily detected by Coq-provided tools (or a simple `grep`). Moreover, such axioms would not live inside a language definition but in the metalevel, so a semantic engineer working inside Minuska has no way of accidentally assuming them.

Minuska comes with a default algebra of primitive values that mimicks the behavior of 𝕂. In the model, primitive values are defined as if by the following snippet:

```
Inductive BuiltInValue :=
| bv_error
| bv_bool (b : bool)
| bv_Z (z : Z)
| bv_sym (s : symbol)
| bv_str (s : string)
| bv_list (m : list GroundTerm)
| bv_finite_dict (m : finite_dict GroundTerm GroundTerm).
```

That is, a built-in value is either an error value, a boolean, an integer, a term-symbol (that is used to build terms), a string, a list of ground terms, or a finite dictionary from ground terms to ground terms. Technically, to satisfy Coq's positivity checker, the last case is implemented as a finite dictionary from positive natural numbers to ground terms (We used the `pmap` type that implements an extensional trie of [5] in the Coq stdpp library), together with the countability property of ground terms and built-in values.

The provided built-in functions are standard; these include some arithmetic operations on integers (based on the `ZArith` module of the Coq standard library), boolean operations (based on the `Bool` module of the Coq standard library), and operations for manipulating dictionaries (based on the `pmap` module of stdpp).

V.2.5 Interpreter

A *one-step interpreter* is a partial function that takes a configuration (represented by a ground term) and performs one computational step within the current rewriting theory (if possible), returning the next configuration. The one-step interpreter of Minuska is implemented entirely in Coq.

Instead of defining a new interpreter for each language, in Minuska we have defined a simple universal interpreter. This interpreter is *universal* in that it takes as an input not only an input configuration but also a rewriting theory Γ (encoding the semantics of a programming language). The interpreter works by simply pattern-matching the left-hand side of a rule, evaluating the side condition in the resulting valuation, and interpreting the right-hand side using that valuation. The process of rule selection is currently somewhat *naive*, based on linear search in the list of rules representing the rewriting theory Γ.

The following algorithms depend on the algorithms $EvaluateCondition$ and $EvaluateExpressionTerm$. These are not shown in the text because they are conceptually simple and technical.

```
function NaiveSelect(Γ : rewriting theory, t : ground term)
  for (p => et if c_1,...,c_k)
    if TryMatch(p, t) matches Some δ
      if EvaluateCondition(δ, c_i) is True for all i ∈ {1, ..., k}
        return ((p => et if c_1,...,c_k), δ)
  return None
function Step(Γ : rewriting theory, t : ground term)
  if NaiveSelect(Γ, t) matches ((p => et if c_1,...,c_k), δ)
    return EvaluateExpressionTerm(δ, et)
  else
    return None
```

Algorithm 1: A naive one-step interpreter

```
function TryMatch(p : symbolic term, t : ground term) : optional Δ
  if p is variable x
    return {(x, p)}
  if p is ground term t'
    if t' = t
      return ∅
    else
      return None
  if t is a primitive value
    return None
  let s'[p_1, ..., p_n'] = p
  let s[t_1, ..., t_n] = t
  if s ≠ s' or n ≠ n'
    return None
  v_1, ..., v_n = TryMatch(p_1,t_1),...,TryMatch(p_n,t_n)
  if v_i is None for some i ∈ {1, ..., n}
    return None
  if v_i(x),v_j(x) are defined but distinct for some distinct i, j ∈ {1, ..., n}
   for some x ∈ X
    return None
  return v_1 ∪ ... ∪ v_n
```

Algorithm 2: Naive pattern matching

The significant benefit of Minuska is that we can prove that its interpreters are always correct in the following sense. We say that an interpreter is *sound* with respect to a well-formed rewriting theory Γ iff for every input ground term t, if the interpreter returns a ground term t', then $t \underset{\Gamma}{\rightsquigarrow} t'$. Conversely, we say that an interpreter is *complete* (with respect to a well-formed rewriting theory Γ) iff for every input ground term t, the interpreter returns some ground term t' unless the input ground term t is stuck – i.e., has no $\underset{\Gamma}{\rightsquigarrow}$-successor. Observe that with such a definition, an interpreter is allowed to resolve non-determinism arbitrarily.

Lemma 103: TryMatch correct

For any ground term t, any symbolic term p, and any valuation δ,

1. if $TryMatch(t, p)$ returns δ, then $\delta, t \vDash_p p$; and
2. if $\delta, t \vDash_p p$, then $TryMatch(t, p)$ returns some δ' such that
   - δ' is defined exactly on the variables of p; and
   - δ extends (non-strictly) δ'

Proof: Straightforward but technical induction on the size of p and case analysis of both p and t. □

Lemma 104: EvaluateCondition correct

For any condition c, $EvaluateCondition(\delta, c)$ returns True if $\delta \vDash_c c$ and False otherwise.

Proof: It is easy to define $EvaluateCondition$ such that the lemma holds by induction on the size of the condition. □

Lemma 105: EvaluateExpressionTerm correct

For any ground term t, an expression term et, and any valuation δ, if $EvaluateExpressionTerm(\delta, et)$ returns t, then $\delta, t \vDash_{et} et$.

Proof: It is easy to define $EvaluateExpressionTerm$ such that the lemma holds by induction on the size of the expression-term. □

Theorem 106: Interpreter correctness

The function `Step` (when applied to a well-formed theory Γ) is a sound and complete interpreter (with respect to Γ).

Proof: For soundness, assume $step(\Gamma, t)$ returns t'. That can happen only if $naiveSelect(\Gamma, t)$ returns some ((p => et if c_1,...,c_k), δ) such that $\mathrm{EvaluateExpressionTerm}(\delta, et)$ returns t'. Because naiveSelect is performing only a simple linear search in Γ, it follows that ((p => et if c_1,...,c_k), δ), $\mathrm{tryMatch}(l, g)$ had to return δ, $\mathrm{evaluate}(\delta, et)$ had to return t', and $\mathrm{evalauteCondition}(\delta, c)$ had to return True. But by correctness of the components (Lemma 104 Lemma 105), we have $\delta, t \vDash_p p$, $\delta, t' \vDash_{et} et$, and $\delta \vDash_c c_i$ for all $i \in \{1, ..., k\}$. But these are exactly the conditions relating t and t' in the rewriting relation.

Completeness is almost the reverse of the above argument, with two caveats.

1. One has to take care of the fact that the semantic relation between t and t' gives us only *some* valuation, not necessarily a minimal one (that would be returned by tryMatch). That is why soundness and completenss of tryMatch are not symmetrical.
2. Another catch is that the interpreter returns only *some* successor of t, not necessarily t', because the language definition might be nondeterministic. That is the reason why the definition of completeness of the interpreter is formulated differently than its soundness.

□

V.2.6 Trust base

The size of the trust base of Minuska depends on the way Minuska is used. With Minuska, one can

- put the language definition into a `*.m` file and invoke the `minuska compile` command to check the language definition for well-formedness and generate an executable interpreter; or
- put the language definition into a `*.m` file and use the `minuska def2coq` command to generate a Coq (`*.v`) file containing the language definition and the associated interpreter, then load the generated file into Coq and use it from within Coq; or
- write the language definition directly in Coq.

In the first situation one needs to trust the hand-written OCaml code that implements the `minuska` executable, as well as Coq's extraction mechanism and the OCaml compiler; in the second case, the trust base consists of the `minuska` executable and Coq's kernel; in the last case, only the kernel of Coq needs to be trusted. In principle, the trust base of the first case could be reduced by using a formally verified extraction mechanism (to OCaml [47] or C [2]) and a formally verified compiler (such as CompCert [15]).

V.3 Evaluation

Let us evaluate Minuska from two perspectives, *expressivity* and *performance*, with respect to the $\mathbb{K}$ implementation of [21], as well as on its own. For performance evaluation with respect to [21], one has to choose benchmarks supported by both Minuska and $\mathbb{K}$. In [21], the authors choose two sets of benchmarks: a rewriting system (named "two-counters") implementing summation from 1 to n of unary-encoded natural numbers, executed with various inputs; and a selection of benchmarks from the rewriting engines competition (REC [40]), with reduced inputs, adapted to $\mathbb{K}$. However, the adapted REC benchmarks use user-defined function symbols, which (as discussed in Section II.4.2) allows for a logical inconsistency in a language definition based on $\mathbb{K}$. Moreover, the REC benchmarks were designed for benchmarking rewriting engines, and not semantic frameworks, and thus exercise neither the idioms used in developments of language definitions using $\mathbb{K}$ (such as *strictness* for specifying evaluation order) nor the supporting infrastructure for these as implemented in both K (*strictness attributes*) and Minuska (*strictness declaration*).

```
@rule [step]:
state[M, N] => state[z.minus(M, [(@builtin-int 1)]), z.plus(N, M)
where z.lt([(@builtin-int 0)],M);
```

Listing 4: "two-counters": the running example of [21] in Minuska.

On the other hand, the "two-counters" example shown in Listing 4 is too simple for Minuska, as can be seen from Table 9 and Table 10. Therefore, to compare Minuska with respect to $\mathbb{K}$, we crafted two new benchmarks that are simple

enough to be handled by $\mathbb{K}$ and simultaneously idiomatic enough in the sense that they use the strictness idioms. As mentioned in [21], their tool does not support the native strictness attribute of $\mathbb{K}$. Therefore, in the $\mathbb{K}$ version of the benchmarks, we emulated this feature manually in the same style as done in the version of the benchmarks for Minuska. Since $\mathbb{K}$ does not support generating proofs for (its) builtin integers, we also restrain from using primitive/builtin values, with the exception of builtin booleans which are used in the side conditions of rules both in $\mathbb{K}$ and in Minuska. The two new benchmarks are: computing the n-th member of the Fibonacci sequence, and computing the factorial of n, both done in the iterative manner over unary-encoded natural numbers.

count-to	**Native compute time [s]**
1′000	0.01
10′000	0.06
100′000	0.45
1′000′000	4.39

Table 9: Benchmark results for native execution of Listing 4. "Native compute time" means wall-clock time of the system executed in an extracted and compiled standalone interpreter.

count-to	**Compute time [s]**	**`coqc` time**
10	0.028	0.75
100	0.044	0.78
1′000	0.213	0.92
10′000	1.871	2.58

Table 10: Benchmark results for Coq-based execution of Listing 4. "Compute time" is the time needed to evaluate the benchmark using Coq's `Compute` command, and "coqc time" is time needed to evaluate a Coq (`*.v`) file containing the command.

V.3.1 Performance of Minuska vs 𝕂

name	𝕂 parameters	𝕂 cert-gen	𝕂 cert-check	M compute	M coqc
tc 10	38.01	1.40	2.71	0.03	0.75
tc 20	79.14	2.41	2.97	0.03	0.75
tc 50	192.64	6.92	2.84	0.04	0.76
tc 100	377.83	14.44	2.81	0.04	0.78
fib 5	95.28	4.92	2.83	0.20	0.89
fact 3	247.50	19.71	3.35	0.48	1.40

Table 11: Running times (in seconds) of shared 𝕂/Minuska benchmarks. The meaning of the columns is as follows. For 𝕂, "𝕂 parameters" is the time required to generate "proof parameters" for a single execution, including time required argument parsing; "𝕂 cert-gen" is the time of certificate generation from those proof parameters; and "𝕂 cert-check" is the time required for checking those certificates. For Minuska, "M compute" is the time to evaluate the benchmark using Coq's Compute mechanism, measured inside Coq using its `Time` command; and "M coqc" is the the time required for Coq to process a file with the benchmark, as measured by the GNU `time` utility.

To benchmark 𝕂, we used the infrastructure of [21], which we modified so as to measure also the time to execute a given program on a given input while generating so-called "proof parameters". We measure the performance on Intel Core I7 10th Gen (model name Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz) in a virtual machine limited to 12 GiB of RAM, running Ubuntu 20.04 (for which the available artifact is intended). To benchmark Minuska, we used the same hardware running virtualized Ubuntu 23.10, with the same memory limit. We used a newer OS for Minuska because it depends on Coq 8.19, which was not available in Ubuntu 20.04.

The benchmarking results are in Table 11; the "two-counters" benchmark is abbreviated as "tc". One can argue that the most interesting measure of the 𝕂 interpreter is the *sum* of all three measured times: to ensure that one gets a valid result, one needs to first run the interpreter while generating proof parameters, then to generate a certificate, and finally to check it. Unforunately, the times required for generating proof parameters are so high because of the rather naive approach applied: the 𝕂 tool is called repeatedly as many times as there are computational steps needed for the execution. Also, note that the "𝕂 parameters" column is not reported in [21], and we had to manually adjust the provided infrastructure to measure it.

Regarding Minuska, the difference between the "Compute" time and the "coqc" time is almost constant, over 0.6 seconds; this difference can be interpreted as the time required to load Coq and the Minuska library. Minuska's certified (but naive) interpreter outperforms the certifying interpreter of $\mathbb{K}$ by a large margin. Perhaps surprisingly, it is faster to start Coq and perform a verified computation inside than to verify the proof certificate generated by K for an equivalent computation.

In my opinion, one reason why Minuska's verified but naive interpreter outperforms $\mathbb{K}$'s certifying interpreter is the size of the generated proof objects. This is likely due to the choice of *matching logic* as the logic of the proof certificates: matching logic is rather low-level and its Hilbert-style proof system makes it hard to come up with compact proofs without resorting to metalevel reasoning. Even a very fast proof checker can hardly compensate for the size of $\mathbb{K}$'s proof certificates, which is often in hundreds of megabytes of text data.

V.3.2 IMP semantics in Minuska

```
/* imp-count-to-n */
n := $arg ;
sum := 0 ;
while (1 <= n) {
    sum := sum + n ;
    n := n + (-1) ;
};
sum
```

Listing 5: A program for computing $\Sigma_{i=i}^{n} i$ in a simple imperative language IMP with semantics defined in Minuska. The `$arg` part represents a meta-level parameter.

In addition, I give the full definition of a simple imperative language IMP, adapted from [21], to demonstrate that Minuska can do more than the version of $\mathbb{K}$ of [21]: to interpret a program in a programming language, and to do so with a reasonable performance. According to Table 12, one iteration of the main loop of the program in Listing 5 takes about half a second when interpreted by Coq, and about 10 milliseconds when executing natively. The example uses built-in integers, as well as built-in maps; the source code of the language is available in `languages/imp/imp.m` and in Listing 3.

`$arg`	**Compute**	**coqc**	**native**
1	0.66	1.49	0.02
2	1.15	1.86	0.02
3	1.57	2.28	0.03
4	2.03	2.78	0.04
5	2.45	3.13	0.04
6	2.95	3.62	0.05
7	3.41	4.16	0.06

Table 12: Execution times (in seconds) of the program in Listing 5. "Compute" means the time of the `Compute` command within Coq, "coqc" means the total time of the `coqc` command on a file containing the Compute command, and "native" means the time of the native code generated by the OCaml compiler for the interpreter extracted from Coq into OCaml.

V.3.3 Other benchmarks

benchmark	Compute	coqc
native-fib 1	0.026	0.74
native-fib 11	0.042	0.80
native-fib 21	0.058	0.79
native-fib 31	0.068	0.78
native-fib 41	0.079	0.79
native-fib 51	0.110	0.79
native-fib 61	0.125	0.81
native-fib 71	0.141	0.82
native-fib 81	0.152	0.82
native-fib 91	0.161	0.84
native-fib 101	0.172	0.86

Table 13: Performance of Minuska on a fibonacci-based set of benchmarks, with times in seconds.

To get a fuller understanding of Minuska's performance, we have created an additional set of benchmarks based on computing Fibonacci numbers and factorials. The results are in Table 13 and Table 14. Here, "unary-fib" is the same Fibonacci system as in Table 11, and "unary-fact" is the same code for computing factorials, both measured with growing input parameter. The "unary" prefix means that a naive unary encoding of natural numbers is used, with addition of two numbers being computed in a linear number of steps in the value of the first addend. Also, in both cases a naive recursive algorithm is used. The "native-fib" collection of benchmarks exercise a different implementation of the Fibonacci sequence: the code is more imperative in style, always remembering the last two values of the sequence instead of recursing to compute them; moreover, it uses Minuska's built-in integers implemented using Coq's binary integers.

benchmark	Compute	coqc
unary-fib 1	0.094	0.77
unary-fib 2	0.102	0.78
unary-fib 3	0.120	0.80
unary-fib 4	0.151	0.84
unary-fib 5	0.199	0.89
unary-fib 6	0.285	0.98
unary-fib 7	0.428	1.21
unary-fib 8	0.818	1.53
unary-fib 9	1.233	2.02
unary-fib 10	2.171	2.91
unary-fib 11	3.313	4.04
unary-fact 1	0.159	0.90
unary-fact 2	0.222	0.95
unary-fact 3	0.476	1.40
unary-fact 4	1.107	1.85
unary-fact 5	3.108	3.86
unary-fact 6	12.785	13.60

Table 14: Performance of Minuska on a yet another set of benchmarks, with times in seconds.

V.4 Concluding remarks

Minuska internally operates on the given program's abstract syntax; also, a programming language's semantic rules are written with respect to the target language's abstract syntax. This is a simplification with respect to $\mathbb{K}$, which allows the semantic rules to be written using the target language's concrete syntax.

Instead, Minuska provides an interface to connect a user-provided parser for the target language. While this may be perceived as a limitation, one could consider supporting concrete syntax in semantic rules to be orthogonal to what we tried to achieve with Minuska. I do not see any principal problem with adding such support.

The part of Minuska reported so far cost about 4 months for a single person to develop and consists of approximately 16kLOC of Coq code, only approximately 700LOC of which are specifications (of MinusLang etc.), the rest being implementation and proofs. This estimated time includes initial drafting efforts and later refactorings, such as simplifications in the representation of terms. I also explored some other directions in which Minuska could be taken (such as having a symbolic execution engine, a higher-level language-definition language, and a faster concrete execution engine); some code for these is already part of the Minuska code base and cost another 3-4 person-months to develop. That said, since I was interleaving the above-mentioned efforts with paper writing and was not explicitly tracking the time spent on Minuska, the provided estimates need to be taken with a grain of salt.

When it comes to semantic frameworks, having a verified semantic-based interpreter is a good start, but one can go further.

1. As has been demonstrated by [93], semantics frameworks can implement symbolic execution and deductive verification facilities.
2. Although Minuska's verified pattern-matching-based interpreter is more performant than the certifying competition, it is still rather slow. I believe one could reuse some existing work for performant pattern matching that has been done in the context of compilers – for example, [65]. As far as I know, the $\mathbb{K}$ tool already has an unverified implementation of some of these ideas in its LLVM-based backend.
3. In both Minuska and $\mathbb{K}$, the constructs for *evaluation contexts*, *strictness*, and the *order of evaluation* of subexpression have no formal semantics, but are implemented only as a syntactic sugar on top of ordinary rewriting rules. It would be better if these had formal semantics, since it would make reasoning about language definitions more high-level.
4. It would also be interesting to model some of the larger languages defined in $\mathbb{K}$ in Minuska as well.

I discuss some of these ideas in the following chapter.

VI Notes on future research directions

VI.1 On language definitions

Currently, the core formalisms of $\mathbb{K}$ and Minuska are too low-level, and users rely on frontends to raise the level of abstraction. In particular, declarations specifying evaluation order are implemented by maintaining an explicit stack. This is not necessarily a bad way of implementing evaluation order. The problem is that this implementation detail leaks the to descriptions of the programming languages.

On the problem of unnecessary and arbitrary detail in language specification, Plotkin notes:

> I remember attending a seminar at Edinburgh where the intricacies of their PL/I abstract machine were explained. The states of these machines are tuples of various kinds of complex trees and there is also a stack of environments; the transition rules involve much tree traversal to access syntactical control points, handle jumps, and to manage concurrency. I recall not much liking this way of doing operational semantics. It seemed far too complex, burying essential semantical ideas in masses of detail; further, the machine states were too big. The lesson I took from this was that abstract interpreting machines do not scale up well when used as a human-oriented method of specification for real languages […].
>
> — G. D. Plotkin [76]

In a similar spirit, Dana Scott writes

> Certainly the introduction of abstract machines does not settle this difficulty of a multiplicity of translation schemes. After all even for an abstract machine the compiler writer must choose specific ways of keeping track of the order or depth of nesting of computation initiations. There may be different ways of doing this, all of which lead to the same results. The variety of approaches even on an abstract machine may make it quite impossible to define in any meaningful way just what is a single computation step. Therefore, the idea of the state of the computation is not apparently at all well determined by the source language alone. When a specific compiler is fixed, however, we may very well want to investigate how it handles computation steps. But the question remains: do they have anything to do with the semantical definition of the original language? I think not.
>
> — D. S. Scott [87]

Therefore, it would be interesting to find, give formal semantics to, and integrate into a semantics framework a higher-level language for descriptions of programming languages. Ideally, such a language would use concepts already known to users of existing semantics framework, such as *evaluation contexts* and *strictness*, and would come with a formally verified translation to MinusLang.

Consider something like the following. Syntactically, a language definition would consist of

- a list c_v of *value side conditions* (as in MinusLang), with some x_v being the only variable occurring in each side condition;
- a list D of *declarations*, each one being either a *rewrite rule declaration* of the shape

  $$\texttt{rule}\ (l_c, l_s) \overset{a}{\Rightarrow} (r_c, r_s)\ \texttt{if}\ cs$$

  (where $l_c, l_s, r_c, r_s \in P$ and cs is a list of side conditions) or a *context declaration* of the shape

  $$\texttt{context}\ c\ \texttt{with}\ h\ \texttt{if}\ cs$$

  (where $c \in P$ is called the *context*, $h \in X$ called the *hole variable* occurs exactly once in c, and cs is a list of side conditions).

(For technical reasons it would be required that x_v would be distinct from all hole variables and that hole variables do not occur in side conditions.) The semantics of such a language definition is given by the rules in Figure 15.

$$\frac{}{(c_1, s_1) \overset{\varepsilon}{\rightsquigarrow} (c_1, s_1)}\ \text{refl}$$

$$\frac{(c_1, s_1) \overset{w}{\rightsquigarrow} (c_2, s_2) \qquad (c_2, s_2) \overset{w'}{\rightsquigarrow} (c_3, s_3)}{(c_1, s_1) \overset{w \cdot w'}{\rightsquigarrow} (c_3, s_3)}\ \text{trans}$$

$$\frac{\begin{array}{c}\texttt{rule}\ (l_c, l_s) \overset{a}{\Rightarrow} (r_c, r_s)\ \texttt{if}\ cs \in D \qquad \rho, c_1 \vDash l_c \qquad \rho, c_2 \vDash r_c \qquad \rho, s_1 \vDash l_s \\ \rho, s_2 \vDash r_s \qquad \rho \vDash cs\end{array}}{(c_1, s_1) \overset{a}{\rightsquigarrow} (c_2, s_2)}\ \text{rule}$$

,

$$\frac{\begin{array}{c}\texttt{context}\ c\ \texttt{with}\ h\ \texttt{if}\ cs \in D \qquad \rho_1 \vDash cs \qquad \rho_1, c_1 \vDash c\left[\frac{r}{h}\right] \\ (r, s_1) \overset{w}{\rightsquigarrow} (v, s_2) \qquad agree(c, h, \rho_1, \rho_2) \qquad \rho_2 \vDash c_v\left[\frac{e}{x_v}\right] \qquad \rho_2, c_2 \vDash c\left[\frac{v}{h}\right]\end{array}}{(c_1, s_1) \overset{w}{\rightsquigarrow} (c_2, s_2)}\ \text{context}$$

Figure 15: The rules defining the semantics of a language definition, where v, r, c_1, c_2, s_1, s_2 range over ground terms, a over actions, w, w_1, w_2 over action words, and $agree(c, h, \rho_1, \rho_2)$ means that ρ_1 and ρ_2 have the same value for all variables other than h occurring in c.

To elaborate on the semantics:

- the relation $_ \overset{_}{\rightsquigarrow} _$ is ternary, between a *source configuration*, an *action word*, and a *target configuration*;
- the rules `refl` and `trans` implement reflexivity and transitivity, respectively, with ε being the empty word;
- the rule `rule` gives meaning to ordinary rewriting rules, where each step happens under a single action `a`; and

- the rule `context` gives meaning to context declarations by means of lifting transitions into context `c`.

Now, the claim would be that
1. this language captures the intuition of context declarations of $\mathbb{K}$, and
2. definitions in this languages can be naturally converted to the language used in Chapter V.

However, developing the argument goes beyond the scope of this thesis.

VI.2 On separation logic

Reachability logic (RL) was designed as an alternative to Hoare logic. However, it is not easy to use Hoare logics for reasoning about pointer-manipulating programs:

> The use of shared mutable data structures, i.e., of structures where an updatable field can be referenced from more than one point, is widespread in areas as diverse as systems programming and artificial intelligence. Approaches to reasoning about this technique have been studied for three decades, but the result has been methods that suffer from either limited applicability or extreme complexity, and scale poorly to programs of even moderate size.
>
> — J. C. Reynolds [78]

This limitation of Hoare logics is also inherited by reachability logic. *Separation logic* [78] (SL) is an extension of Hoare logic adressing the same problem. Therefore, it would be interesting to combine RL and SL to gain a logic that is both language-parametric and allows reasoning about pointer-manipulating programs.

SL enables scaleable control over pointer aliasing by means of a binary operator ($\star$) known as "separating star". Intuitively, this operator splits the program configuration into two parts and evaluates the operands. Moreover, proof systems of SL are equipped with (variants of) a *framing rule*

$$\frac{\{P\}C\{Q\}}{\{P \star R\}C\{Q \star R\}}\,\text{Framing}$$

that allows *local reasoning*: intuitively, if a program behaves in some way, it will behave in the same way in any larger configuration.

The problem is: the intuition behind the Framing rule does not apply to all languages.

For an example, consider a language where reads from memory locations not present in the configuration have some defined semantics – like, returning a default value or raising an exception. The behavior of such a memory read is not preserved under configuration extension, and thus Framing is violated. Or, consider a language where one semantic rule compares (using equality) the whole configuration with another configuration; or, a semantic rule that computes the number of allocated memory addresses; or, a rule which finds the least address not present in the configuration. The last rule could be useful for dynamic memory

allocation. The question at hand is therefore: how to limit the expressivity of a semantics framework such that the Framing rule is honored, while keeping the ability to express interesting languages?

One possible solution is based on the notions of a "separation algebra" and a "local action" of C. Calcagno, P. W. O'Hearn, and H. Yang [19], as refined by R. Dockins, A. Hobor, and A. W. Appel [39]. Intuitively, the idea is that the structure of configurations of a given programming language would form a separation algebra (a set equipped with the notion of a *compatability* of two elements, a *join* of two compatible elements, and a *sub-element* relation), and semantic rules would be *local* with respect to this algebra. This could easily be done if the structure of primitive values is already a kind of the separation algebra, and if the language of the rule side conditions is restricted enough to prohibit arbitrary equality comparions, negations, and other problematic constructs.

VII Final remarks

I have presented a few ways of improving the ecosystem of programming language semantics frameworks. Three of these improvements form the core of this thesis:

1. Cartesian reachability logic – an improvement over Cartesian Hoare logic and reachability logic that combines the strong points of both;
2. a support for interactive reasoning in matching logic – a logic used as the basis of $\mathbb{K}$-framework, an existing real-world semantics framework; and
3. Minuska – a new, formally specified and verified semantics framework that avoids some of the issues present in the design and implementation of $\mathbb{K}$.

There is more work to do. From a semantics engineer's point of view, it would be helpful if programming language semantics frameworks offered more higher-level formalisms for specifying a programming language's semantics. A formal verification engineer would benefit from improved abilities of reachability logics for reasoning about pointer-manipulating programs. I tried to sketch out what I mean by these, but the development of these ideas is left for future work.

There is even more work to do. A long-standing issue with semantics frameworks is their performance: generated interpreters and program verifiers are typically slower than hand-crafted ones. Another idea that is sometimes informally discussed is the ability of semantics frameworks to partially specialize a language semantics with a program and then generate compilers producing program-specific executables that would outperform program-generic interpreters. Language-parametric abstract interpretation is yet another idea. All these are likely related.

But there is always more work to do.

Bibliography

[1] S. Agrawal and B. Bonakdarpour, "Runtime Verification of k-Safety Hyperproperties in HyperLTL," in *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, IEEE Computer Society, 2016, pp. 239–252. doi: 10.1109/CSF.2016.24.

[2] A. Anand *et al.*, "CertiCoq : A verified compiler for Coq," in *CoqPL workshop*, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:9607775

[3] A. W. Appel, "Verified Software Toolchain," in *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*, A. Goodloe and S. Person, Eds., in Lecture Notes in Computer Science, vol. 7226. Springer, 2012, p. 2. doi: 10.1007/978-3-642-28891-3_2.

[4] A. W. Appel and S. Blazy, "Separation logic for small-step Cminor," in *International Conference on Theorem Proving in Higher Order Logics*, 2007. doi: 10.1007/978-3-540-74591-4_3.

[5] A. W. Appel and X. Leroy, "Efficient Extensional Binary Tries," *J. Autom. Reason.*, vol. 67, no. 1, p. 8, 2023, doi: 10.1007/S10817-022-09655-X.

[6] A. Arusoaie, D. Nowak, V. Rusu, and D. Lucanu, "A Certified Procedure for RL Verification," in *19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2017, Timisoara, Romania, September 21-24, 2017*, T. Jebelean, V. Negru, D. Petcu, D. Zaharie, T. Ida, and S. M. Watt, Eds., IEEE Computer Society, 2017, pp. 129–136. doi: 10.1109/SYNASC.2017.00031.

[7] R. Bahlke and G. Snelting, "The PSG System: From Formal Language Definitions to Interactive Programming Environments," *ACM Trans. Program. Lang. Syst.*, vol. 8, no. 4, pp. 547–576, 1986, doi: 10.1145/6465.20890.

[8] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure Information Flow by Self-Composition," in *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*, IEEE Computer Society, 2004, pp. 100–114. doi: 10.1109/CSFW.2004.17.

[9] P. Bereczky, X. Chen, D. Horpácsi, T. B. Mizsei, L. Peña, and J. Tušil, "Mechanizing Matching Logic in Coq," in *Proceedings of the Sixth Working Formal Methods Symposium, FROM 2022, "Al. I. Cuza University", Iasi, Romania, 19-20 September, 2022*, V. Rusu, Ed., in EPTCS, vol. 369. 2022, pp. 17–36. doi: 10.4204/EPTCS.369.2.

[10] P. Bereczky, X. Chen, D. Horpácsi, T. B. Mizsei, L. Peña, and J. Tušil, "Matching logic formalization." [Online]. Available: https://github.com/harp-project/AML-Formalization/releases/tag/v1.0.15

[11] Y. Bertot, "Filters on CoInductive Streams, an Application to Eratosthenes' Sieve," in *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005, Proceedings*, P. Urzyczyn, Ed., in Lecture Notes in Computer Science, vol. 3461. Springer, 2005, pp. 102–115. doi: 10.1007/11417170_9.

[12] L. Bettini, *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition*, 2nd ed. Packt Publishing, 2016.

[13] S. Blazy and X. Leroy, "Formal Verification of a Memory Model for C-Like Imperative Languages," in *Formal Methods and Software Engineering, 7th International Conference on Formal Engineering Methods, ICFEM 2005, Manchester, UK, November 1-4, 2005, Proceedings*, K.-K. Lau and R. Banach, Eds., in Lecture Notes in Computer Science, vol. 3785. Springer, 2005, pp. 280–299. doi: 10.1007/11576280_20.

[14] R. Bohrer, V. Rahli, I. Vukotic, M. Völp, and A. Platzer, "Formally verified differential dynamic logic," in *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, in CPP 2017. Association for Computing Machinery, 2017. doi: 10.1145/3018610.3018616.

[15] S. Boldo, J.-H. Jourdan, X. Leroy, and G. Melquiond, "A Formally-Verified C Compiler Supporting Floating-Point Arithmetic," in *21st IEEE Symposium on Computer Arithmetic, ARITH 2013, Austin, TX, USA, April 7-10, 2013*, A. Nannarelli, P.-M. Seidel, and P. T. P. Tang, Eds., IEEE Computer Society, 2013, pp. 107–115. doi: 10.1109/ARITH.2013.30.

[16] R. Bornat and B. Sufrin, "Animating formal proof at the surface: The Jape Proof Calculator," *The Computer Journal*, 1999, doi: 10.1093/comjnl/42.3.177.

[17] J. Buisson and S. Rehab, "Automatic Transformation from Ecore Metamodels towards Gallina Inductive Types," in *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development*, in MODELSWARD 2018. Funchal, Madeira, Portugal: SCITEPRESS - Science, Technology Publications, Lda, 2018, pp. 488–495. doi: 10.5220/0006608604880495.

[18] J. Buisson and S. Rehab, "Generation of Inductive Types from Ecore Metamodels," in *Model-Driven Engineering and Software Development*, S. Hammoudi, L. F. Pires, and B. Selic, Eds., Cham: Springer International Publishing, 2019, pp. 308–334.

[19] C. Calcagno, P. W. O'Hearn, and H. Yang, "Local Action and Abstract Separation Logic," in *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*, IEEE Computer Society, 2007, pp. 366–378. doi: 10.1109/LICS.2007.30.

[20] A. Charguéraud, "The locally nameless representation," *J. Autom. Reason.*, 2012, doi: 10.1007/s10817-011-9225-2.

[21] X. Chen, Z. Lin, M.-T. Trinh, and G. Roşu, "Towards a Trustworthy Semantics-Based Language Framework via Proof Generation," in *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*, A. Silva and K. R. M. Leino, Eds., in Lecture Notes in Computer Science, vol. 12760. Springer, 2021, pp. 477–499. doi: 10.1007/978-3-030-81688-9_23.

[22] X. Chen, D. Lucanu, and G. Roşu, "Matching logic explained," *J. Log. Algebraic Methods Program.*, vol. 120, p. 100638, 2021, doi: 10.1016/J.JLAMP.2021.100638.

[23] X. Chen and G. Roşu, "The K Vision for the Future of Programming Language Design and Analysis," in *Formal Methods in Outer Space - Essays Dedicated to Klaus Havelund on the Occasion of His 65th Birthday*, E. Bartocci, Y. Falcone, and M. Leucker, Eds., in Lecture Notes in Computer Science, vol. 13065. Springer, 2021, pp. 3–9. doi: 10.1007/978-3-030-87348-6_1.

[24] X. Chen and G. Roşu, "Matching μ-Logic", in *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*, IEEE, 2019, pp. 1–13. doi: 10.1109/LICS.2019.8785675.

[25] X. Chen and G. Roşu, "Matching mu-Logic: Foundation of K Framework," in *8th Conference on Algebra and Coalgebra in Computer Science (CALCO 2019)*, M. Roggenbach and A. Sokolova, Eds., in Leibniz International Proceedings in Informatics (LIPIcs), vol. 139. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, pp. 1–4. doi: 10.4230/LIPIcs.CALCO.2019.1.

[26] X. Chen and G. Roşu, "Matching μ-Logic". [Online]. Available: http://hdl.handle.net/2142/102281

[27] X. Chen and G. Roşu, "Applicative matching logic: Semantics of K," 2019. [Online]. Available: http://hdl.handle.net/2142/104616

[28] X. Chen and G. Roşu, "A Language-Independent Program Verification Framework," in *Leveraging Applications of Formal Methods, Verification and Validation. Verification - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part II*, T. Margaria and B. Steffen, Eds., in Lecture Notes in Computer Science, vol. 11245. Springer, 2018, pp. 92–102. doi: 10.1007/978-3-030-03421-4_7.

[29] Ş. Ciobâcă, "Reducing Partial Equivalence to Partial Correctness," in *16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2014, Timisoara, Romania, September 22-25, 2014*, F. Winkler, V. Negru, T. Ida, T. Jebelean, D. Petcu, S. M. Watt, and D. Zaharie, Eds., IEEE Computer Society, 2014, pp. 164–171. doi: 10.1109/SYNASC.2014.30.

[30] Ş. Ciobâcă, D. Lucanu, V. Rusu, and G. Roşu, "A Language-Independent Proof System for Mutual Program Equivalence," in *Formal Methods and Software Engineering - 16th International Conference on Formal Engineering Methods, ICFEM 2014, Luxembourg, Luxembourg, November 3-5, 2014. Proceedings*, S. Merz and J. Pang, Eds., in Lecture Notes in Computer Science, vol. 8829. Springer, 2014, pp. 75–90. doi: 10.1007/978-3-319-11737-9_6.

[31] Ş. Ciobâcă, D. Lucanu, V. Rusu, and G. Roşu, "A language-independent proof system for full program equivalence," *Formal Aspects Comput.*, vol. 28, no. 3, pp. 469–497, 2016, doi: 10.1007/S00165-016-0361-7.

[32] M. R. Clarkson, B. Finkbeiner, M. Koleini, K. K. Micinski, M. N. Rabe, and C. Sánchez, "Temporal Logics for Hyperproperties," in *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, M. Abadi and S. Kremer, Eds., in Lecture Notes in Computer Science, vol. 8414. Springer, 2014, pp. 265–284. doi: 10.1007/978-3-642-54792-8_15.

[33] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *J. Comput. Secur.*, vol. 18, no. 6, pp. 1157–1210, 2010, doi: 10.3233/JCS-2009-0393.

[34] N. Coenen, B. Finkbeiner, C. Sánchez, and L. Tentrup, "Verifying Hyperliveness," in *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, I. Dillig and S. Tasiran, Eds., in Lecture Notes in Computer Science, vol. 11561. Springer, 2019, pp. 121–139. doi: 10.1007/978-3-030-25540-4_7.

[35] S. Coupet-Grimal, "An axiomatization of linear temporal logic in the Calculus of Inductive Constructions," *Journal of Logic and Computation*, 2003, doi: 10.1093/logcom/13.6.801.

[36] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, R. M. Graham, M. A. Harrison, and R. Sethi, Eds., ACM, 1977, pp. 238–252. doi: 10.1145/512950.512973.

[37] O. Danvy and L. R. Nielsen, "Refocusing in Reduction Semantics," *BRICS Report Series*, vol. 11, 2004, [Online]. Available: https://api.semanticscholar.org/CorpusID:17048255

[38] S. Dasgupta, D. Park, T. Kasampalis, V. S. Adve, and G. Roşu, "A complete formal semantics of x86-64 user-level instruction set architecture," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, K. S. McKinley and K. Fisher, Eds., ACM, 2019, pp. 1133–1148. doi: 10.1145/3314221.3314601.

[39] R. Dockins, A. Hobor, and A. W. Appel, "A Fresh Look at Separation Algebras and Share Accounting," in *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*, Z. Hu, Ed., in Lecture Notes in Computer Science, vol. 5904. Springer, 2009, pp. 161–177. doi: 10.1007/978-3-642-10672-9_13.

[40] F. Durán and H. Garavel, "The Rewrite Engines Competitions: A RECtrospective," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Beyer, M. Huisman, F. Kordon, and B. Steffen, Eds., Cham: Springer International Publishing, 2019, pp. 93–100.

[41] R. Dyckhoff, "Contraction-free sequent calculi for intuitionistic logic," *The Journal of Symbolic Logic*, 1992, doi: 10.2307/2275431.

[42] M. Felleisen, R. B. Findler, and M. Flatt, *Semantics Engineering with PLT Redex*. in The MIT Press. London, England: MIT Press, 2009.

[43] M. Felleisen and R. Hieb, "The Revised Report on the Syntactic Theories of Sequential Control and State," *Theor. Comput. Sci.*, vol. 103, no. 2, pp. 235–271, 1992, doi: 10.1016/0304-3975(92)90014-7.

[44] B. Finkbeiner, L. Haas, and H. Torfah, "Canonical Representations of k-Safety Hyperproperties," in *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*, IEEE, 2019, pp. 17–31. doi: 10.1109/CSF.2019.00009.

[45] B. Finkbeiner, M. N. Rabe, and C. Sánchez, "Algorithms for Model Checking HyperLTL and HyperCTL ^*," in *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, D. Kroening and C. S. Pasareanu, Eds., in Lecture Notes in Computer Science, vol. 9206. Springer, 2015, pp. 30–48. doi: 10.1007/978-3-319-21690-4_3.

[46] R. W. Floyd, "Assigning meaning to programs," 1967. [Online]. Available: https://api.semanticscholar.org/CorpusID:62710333

[47] Y. Forster, M. Sozeau, and N. Tabareau, "Verified Extraction from Coq to OCaml," *Proc. ACM Program. Lang.*, vol. 8, no. PLDI, Jun. 2024, doi: 10.1145/3656379.

[48] G. Gentzen, "Investigations into logical deduction," *The Collected Papers of Gerhard Gentzen*. in Studies in logic and the foundations of mathematics. Elsevier, 1969. doi: 10.1016/S0049-237X(08)70822-X.

[49] M. J. C. Gordon, *The denotational description of programming languages - an introduction*. Springer, 1979.

[50] M. Hills, T. Şerbănuţă, and G. Roşu, "A Rewrite Framework for Language Definitions and for Generation of Efficient Interpreters," *Electronic Notes in*

Theoretical Computer Science, vol. 176, no. 4, pp. 215–231, 2007, doi: https://doi.org/10.1016/j.entcs.2007.06.017.

[51] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969, doi: 10.1145/363235.363259.

[52] C. A. R. Hoare, "An axiomatic definition of the programming language PASCAL," in *International Sympoisum on Theoretical Programming, Novosibirsk, Russia, August 7-11, 1972, Proceedings*, A. P. Ershov and V. A. Nepomniaschy, Eds., in Lecture Notes in Computer Science, vol. 5. Springer, 1972, pp. 1–16. doi: 10.1007/3-540-06720-5_1.

[53] C. A. R. Hoare and P. E. Lauer, "Consistent and Complementary Formal Theories of the Semantics of Programming Languages," *Acta Informatica*, vol. 3, pp. 135–153, 1974, doi: 10.1007/BF00264034.

[54] G. Kahn, "Natural Semantics," in *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*, F.-J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, Eds., in Lecture Notes in Computer Science, vol. 247. Springer, 1987, pp. 22–39. doi: 10.1007/BFB0039592.

[55] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C: A software analysis perspective," *Formal Aspects Comput.*, vol. 27, no. 3, pp. 573–609, 2015, doi: 10.1007/S00165-014-0326-7.

[56] C. Klein *et al.*, "Run your research: on the effectiveness of lightweight mechanization," in *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, J. Field and M. Hicks, Eds., ACM, 2012, pp. 285–296. doi: 10.1145/2103656.2103691.

[57] C. Klein, J. A. McCarthy, S. Jaconette, and R. B. Findler, "A Semantics for Context-Sensitive Reduction Semantics," in *Programming Languages and Systems - 9th Asian Symposium, APLAS 2011, Kenting, Taiwan, December 5-7, 2011. Proceedings*, H. Yang, Ed., in Lecture Notes in Computer Science, vol. 7078. Springer, 2011, pp. 369–383. doi: 10.1007/978-3-642-25318-8_27.

[58] P. Klint, "A Meta-Environment for Generating Programming Environments," *ACM Trans. Softw. Eng. Methodol.*, vol. 2, no. 2, pp. 176–201, 1993, doi: 10.1145/151257.151260.

[59] R. Krebbers *et al.*, "MoSeL: A general, extensible modal framework for interactive proofs in separation logic," *Proceedings of the ACM on Programming Languages*, 2018, doi: 10.1145/3236772.

[60] R. Krebbers, A. Timany, and L. Birkedal, "Interactive proofs in higher-order concurrent separation logic," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France,*

January 18-20, 2017, G. Castagna and A. D. Gordon, Eds., ACM, 2017, pp. 205–217. doi: 10.1145/3009837.3009855.

[61] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, 2009, doi: 10.1145/1538788.1538814.

[62] X. Leroy, "A Formally Verified Compiler Back-end," *J. Autom. Reason.*, vol. 43, no. 4, pp. 363–446, 2009, doi: 10.1007/S10817-009-9155-4.

[63] Z. Lin, X. Chen, M.-T. Trinh, J. Wang, and G. Roşu, "Generating Proof Certificates for a Language-Agnostic Deductive Program Verifier," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, pp. 56–84, 2023, doi: 10.1145/3586029.

[64] D. Lucanu, V. Rusu, A. Arusoaie, and D. Nowak, "Verifying Reachability-Logic Properties on Rewriting-Logic Specifications," in *Logic, Rewriting, and Concurrency - Essays dedicated to José Meseguer on the Occasion of His 65th Birthday*, N. Martí-Oliet, P. C. Ölveczky, and C. L. Talcott, Eds., in Lecture Notes in Computer Science, vol. 9200. Springer, 2015, pp. 451–474. doi: 10.1007/978-3-319-23165-5_21.

[65] L. Maranget, "Compiling pattern matching to good decision trees," in *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, in ML '08. Victoria, BC, Canada: Association for Computing Machinery, 2008, pp. 35–46. doi: 10.1145/1411304.1411311.

[66] J. McCarthy and J. Painter, "An Analysis of Example," in *Mathematical Aspects of Computer Science 1*, in Proceedings of Symposia in Applied Mathematics, vol. 19. 1967.

[67] A. McCreight, "Practical tactics for separation logic," in *International Conference on Theorem Proving in Higher Order Logics*, 2009. doi: 10.1007/978-3-642-03359-9_24.

[68] E. Mendelson, *Introduction to Mathematical Logic, Fourth Edition.* in Discrete Mathematics and Its Applications. Taylor & Francis, 1997. [Online]. Available: https://books.google.cz/books?id=ZO1p4QGspoYC

[69] R. Milner and R. Weyhrauch, "Proving compiler correctness in a mechanised logic," *Machine Intelligence*, vol. 7, no. 3, pp. 51–73, 1972.

[70] D. Monniaux, L. Gourdin, S. Boulmé, and O. Lebeltel, "Testing a Formally Verified Compiler," in *Tests and Proofs - 17th International Conference, TAP 2023, Leicester, UK, July 18-19, 2023, Proceedings*, V. Prevosto and C. Seceleanu, Eds., in Lecture Notes in Computer Science, vol. 14066. Springer, 2023, pp. 40–48. doi: 10.1007/978-3-031-38828-6_3.

[71] B. M. Moore, L. Peña, and G. Roşu, "Program Verification by Coinduction," in *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20,*

2018, Proceedings, A. Ahmed, Ed., in Lecture Notes in Computer Science, vol. 10801. Springer, 2018, pp. 589–618. doi: 10.1007/978-3-319-89884-1_21.

[72] L. M. de Moura and N. S. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, C. R. Ramakrishnan and J. Rehof, Eds., in Lecture Notes in Computer Science, vol. 4963. Springer, 2008, pp. 337–340. doi: 10.1007/978-3-540-78800-3_24.

[73] F. J. Pelletier and A. Hazen, "Natural deduction systems in logic," *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, 2022.

[74] J. von Plato, *Elements of Logical Reasoning*. Cambridge University Press, 2014.

[75] G. D. Plotkin, "A structural approach to operational semantics," *J. Log. Algebraic Methods Program.*, pp. 17–139, 2004.

[76] G. D. Plotkin, "The origins of structural operational semantics," *J. Log. Algebraic Methods Program.*, pp. 3–15, 2004, doi: 10.1016/J.JLAP.2004.03.009.

[77] J. F. Power and C. Webster, "Working with linear logic in Coq." [Online]. Available: https://mural.maynoothuniversity.ie/6461/1/JP-Working-Linear-Logic.pdf

[78] J. C. Reynolds, "Separation Logic: A Logic for Shared Mutable Data Structures," in *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, IEEE Computer Society, 2002, pp. 55–74. doi: 10.1109/LICS.2002.1029817.

[79] G. Roşu, "Matching Logic," *Log. Methods Comput. Sci.*, vol. 13, no. 4, 2017, doi: 10.23638/LMCS-13(4:28)2017.

[80] G. Roşu and A. Stefanescu, "Towards a Unified Theory of Operational and Axiomatic Semantics," in *Automata, Languages, and Programming - 39th International Colloquium, ICALP 2012, Warwick, UK, July 9-13, 2012, Proceedings, Part II*, A. Czumaj, K. Mehlhorn, A. M. Pitts, and R. Wattenhofer, Eds., in Lecture Notes in Computer Science, vol. 7392. Springer, 2012, pp. 351–363. doi: 10.1007/978-3-642-31585-5_33.

[81] G. Roşu and A. Stefanescu, "Checking reachability using matching logic," in *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, G. T. Leavens and M. B. Dwyer, Eds., ACM, 2012, pp. 555–574. doi: 10.1145/2384616.2384656.

[82] G. Roşu, A. Stefanescu, Ş. Ciobâcă, and B. M. Moore, "One-Path Reachability Logic," in *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, IEEE Computer Society, 2013, pp. 358–367. doi: 10.1109/LICS.2013.42.

[83] G. Roşu and T.-F. Şerbănuţă, "An overview of the K semantic framework," *J. Log. Algebraic Methods Program.*, vol. 79, no. 6, pp. 397–434, 2010, doi: 10.1016/J.JLAP.2010.03.012.

[84] V. Rusu and A. Arusoaie, "Proving Reachability-Logic Formulas Incrementally," in *Rewriting Logic and Its Applications - 11th International Workshop, WRLA 2016, Held as a Satellite Event of ETAPS, Eindhoven, The Netherlands, April 2-3, 2016, Revised Selected Papers*, D. Lucanu, Ed., in Lecture Notes in Computer Science, vol. 9942. Springer, 2016, pp. 134–151. doi: 10.1007/978-3-319-44802-2_8.

[85] V. Rusu, G. Grimaud, and M. Hauspie, "Proving partial-correctness and invariance properties of transition-system models," *Sci. Comput. Program.*, vol. 186, 2020, doi: 10.1016/J.SCICO.2019.102342.

[86] V. Rusu and D. Nowak, "(Co)inductive proof systems for compositional proofs in reachability logic," *J. Log. Algebraic Methods Program.*, vol. 118, p. 100619, 2021, doi: 10.1016/J.JLAMP.2020.100619.

[87] D. S. Scott, "Mathematical concepts in programming language semantics," in *American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1972 Spring Joint Computer Conference, Atlantic City, NJ, USA, May 16-18, 1972*, in AFIPS Conference Proceedings, vol. 40. AFIPS, 1972, pp. 225–234. doi: 10.1145/1478873.1478903.

[88] P. Sewell *et al.*, "Ott: effective tool support for the working semanticist," in *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, in ICFP '07. Freiburg, Germany: Association for Computing Machinery, 2007, pp. 1–12. doi: 10.1145/1291151.1291155.

[89] P. Sewell *et al.*, "Ott: Effective tool support for the working semanticist," *J. Funct. Program.*, vol. 20, no. 1, pp. 71–122, 2010, doi: 10.1017/S0956796809990293.

[90] M. Sousa and I. Dillig, "Cartesian hoare logic for verifying k-safety properties," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, C. Krintz and E. D. Berger, Eds., ACM, 2016, pp. 57–69. doi: 10.1145/2908080.2908092.

[91] M. Sousa and I. Dillig, "Cartesian hoare logic for verifying k-safety properties," 2016. [Online]. Available: https://www.cs.utexas.edu/users/isil/chl-extended.pdf

[92] A. Stefanescu, Ş. Ciobâcă, R. Mereuta, B. M. Moore, T.-F. Şerbănuţă, and G. Roşu, "All-Path Reachability Logic," *Log. Methods Comput. Sci.*, vol. 15, no. 2, 2019, doi: 10.23638/LMCS-15(2:5)2019.

[93] A. Stefănescu, D. Park, S. Yuwen, Y. Li, and G. Roşu, "Semantics-based program verifiers for all languages," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, E. Visser and Y. Smaragdakis, Eds., ACM, 2016, pp. 74–91. doi: 10.1145/2983990.2984027.

[94] M. H. Sørensen and P. Urzyczyin, "Chapter 5 - Proofs as combinators," *Lectures on the Curry-Howard Isomorphism*, vol. 149. in Studies in Logic and the Foundations of Mathematics, vol. 149. Elsevier, pp. 103–125, 2006. doi: https://doi.org/10.1016/S0049-237X(06)80006-6.

[95] The Coq Development Team, "The Coq Proof Assistant." Zenodo, Jul. 2023. doi: 10.5281/zenodo.8161141.

[96] J. Tušil, P. Bereczky, and D. Horpácsi, "Interactive Matching Logic Proofs in Coq," in *Theoretical Aspects of Computing - ICTAC 2023 - 20th International Colloquium, Lima, Peru, December 4-8, 2023, Proceedings*, E. Ábrahám, C. Dubslaff, and S. L. T. Tarifa, Eds., in Lecture Notes in Computer Science, vol. 14446. Springer, 2023, pp. 139–157. doi: 10.1007/978-3-031-47963-2_10.

[97] J. Tušil and J. Obdržálek, "Minuska: Towards a Formally Verified Programming Language Framework," in *Software Engineering and Formal Methods*, A. Madeira and A. Knapp, Eds., Cham: Springer Nature Switzerland, 2025, pp. 200–214.

[98] J. Tušil, T. Şerbănuţă, and J. Obdržálek, "Cartesian Reachability Logic: A Language-parametric Logic for Verifying k-Safety Properties," in *LPAR 2023: Proceedings of 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Manizales, Colombia, 4-9th June 2023*, R. Piskac and A. Voronkov, Eds., in EPiC Series in Computing, vol. 94. EasyChair, 2023, pp. 405–456. doi: 10.29007/1874.

[99] M. Voelter, "Language and IDE Modularization and Composition with MPS," in *Generative and Transformational Techniques in Software Engineering IV: International Summer School, GTTSE 2011, Braga, Portugal, July 3-9, 2011. Revised Papers*, R. Lämmel, J. Saraiva, and J. Visser, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 383–430. doi: 10.1007/978-3-642-35992-7_11.

[100] A. van Wijngaarden *et al.*, "Revised Report on the Algorithmic Language ALGOL 68," *Acta Informatica*, vol. 5, pp. 1–236, 1975, doi: 10.1007/BF00265077.

[101] B. Xavier, C. Olarte, G. Reis, and V. Nigam, "Mechanizing focused linear logic in Coq," *Electronic Notes in Theoretical Computer Science*, 2018, doi: 10.1016/j.entcs.2018.10.014.

[102] V. Zamfir *et al.*, "VLSM: Validating Labelled State Transition and Message Production Systems," *CoRR*, 2022, [Online]. Available: https://arxiv.org/abs/2202.12662

[103] T.-F. Şerbănuţă and G. Roşu, "K-Maude: A Rewriting Based Tool for Semantics of Programming Languages," in *Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010, Held as a Satellite Event of ETAPS 2010, Paphos, Cyprus, March 20-21, 2010, Revised Selected Papers*, P. C. Ölveczky, Ed., in Lecture Notes in Computer Science, vol. 6381. Springer, 2010, pp. 104–122. doi: 10.1007/978-3-642-16310-4_8.

[104] T.-F. Şerbănuţă, G. Roşu, and J. Meseguer, "A rewriting logic approach to operational semantics," *Inf. Comput.*, vol. 207, no. 2, pp. 305–340, 2009, doi: 10.1016/J.IC.2008.03.026.