$\mathbb{K}$—A Semantic Framework for Programming Languages and Formal Analysis*

Xiaohong Chen[0000−0003−3208−4061] and Grigore Roşu[0000−0002−3102−0421]

University of Illinois at Urbana-Champaign, USA
{xc3,grosu}@illinois.edu

Abstract. We give an overview on the applications and foundations of the $\mathbb{K}$ language framework, a semantic framework for programming languages and formal analysis tools. $\mathbb{K}$ represents a 20-year effort in pursuing the ideal language framework vision, where programming languages must have formal definitions, and tools for a given language, such as parsers, interpreters, compilers, semantic-based debuggers, state-space explorers, model checkers, deductive program verifiers, etc., can be derived from just one reference formal definition of the language, which is executable, and no other semantics for the same language should be needed. The correctness of the language tools is guaranteed on a case-by-case basis by proof objects, which encode rigorous mathematical proofs as certificates for every individual tasks that the tools do and can be mechanically checked by third-party proof checkers.

Keywords: $\mathbb{K}$ framework · Matching logic · Formal semantics.

1 What is $\mathbb{K}$?

$\mathbb{K}$ is a language semantic framework and a suite of tools that allow and encourage the language designers to formally define their languages once and for all, using an intuitive and attractive notation, and then obtain language implementations as well as analysis tools for free. This represents a long-standing ideal vision held by the programming languages community. $\mathbb{K}$ is aimed at developing the foundations, techniques, and tools to realize this vision.

1.1 The State-of-the-Art of Programming Languages Design

The state-of-the-art of programming language design is still far from the above ideal vision. The programming languages and formal methods communities still develop language analysis tools for each individual programming language. For example, the C programming language has well-known compilers such as gcc [2] and clang [1], but there are also C interpreters such as TrustInSoft [3] that target detecting undefined behaviors of C programs, model checkers for C such

* This paper follows the lecture notes presented by the second author at the School on Engineering Trustworthy Software Systems (SETSS) in year 2019.

as CBMC [25] that aim at exploring exhaustively the state space of C programs up to a bounded depth, and symbolic execution and deductive verification tools for C such as VCC [15] that formally verify functional properties of C programs. However, the development of these language tools not only for C but also for other languages suffer from the following problems:

- They are built in an ad-hoc fashion, in the sense that language or program analysis experts must rely on their informal understanding of the language to develop the language tools. This informal understanding may not be consistent with the formal definitions of the language, not to mention that most languages do not even have an official formal semantic definition.
- They are time-consuming to develop and may not be thoroughly tested and validated with respect to the formal definition, due to a lack of a mechanized connection between the formal definition and the actual implementation; again, the formal definition might not even exist in the first place;
- Many tools are developed from scratch, sharing very little code or functionality with each other; as a result, not only are there waste of resource and duplicates of work in "re-inventing the wheels", but also we can hardly claim that these tools are implemented for the *same* language;
- They need be updated when the language evolves (e.g., from C11 to C18); in other words, they are inclined to become deprecated;

In conclusion, these language tools that we use to ensure the correctness, reliability, and security of other programs and software systems may themselves be unreliable.

The above story unfolded for various languages over and over again, for more than 50 years, and it is still going on. This is at best uneconomical. Figure 1 shows the state-of-the-art of programming languages design. Suppose we have L programming languages and T tools. Then we need to develop and maintain at least $L \times T$ systems, which share little code or functionality. The cost is waste of talent and resources in doing essentially the same thing, the same tools, but for different languages.

Challenges reinforced by blockchains. The above situation of programming language design is facing more challenges when it comes to the recent burgeoning blockchain industry. Blockchain technology has led to a variety of new programming languages and virtual machines designed specifically for the blockchains, including high-level languages such as Solidity [18] and Vyper [19] to low-level virtual machine bytecode languages such as EVM [23] and IELE [24].

Smart contracts are computer programs running on blockchains that implement communication protocols, often of a kind of digital assets called *cryptocurrencies* that handle economic or financial transactions, withholding a total market capitalization of more than 80 billion US dollars at the time of writing. Therefore, there is enormous demand for formally designing and verifying these highly valuable smart contracts. On the other hand, blockchains and their virtual machine languages have a rapid development cycle with new versions

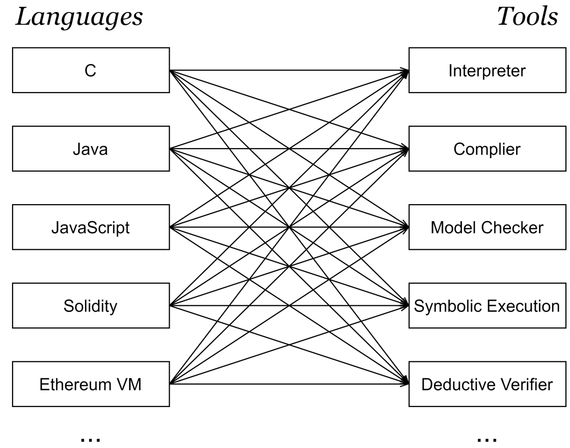

Fig. 1: The state-of-the-art of programming language design

being released on a weekly basis. The state-of-the-art approaches fail to have a canonical reference formal definition of the languages from which language tools are derived. Instead, language designers and tool developers need to implement the language tools and update them whenever a newer version of the languages is released. This has caused a lot of challenges in applying the state-of-the-art approaches to blockchain languages and smart contract formal analysis and verification. As we will see in Section 1.2, these challenges would not exist if one had an ideal language semantic framework.

1.2 The Ideal Language Semantic Framework

Our main motivation is to make programming language design a more organized and scientifically principled process, to reduce duplicated work and waste of resource in programming languages implementation, to increase the reusability and reliability of formal analysis tools, and to increase the reliability and security of the execution, verification, and testing environment of programs and software systems.

We look for an ideal language semantic framework, where all programming languages must be rigorously designed using formal methods and implementations of language tools must be provably correct. We depict this vision of an ideal language framework in Figure 2, where the central yellow bubble denotes the canonical reference formal definition of a given (but arbitrary) programming language, and the surrounding blue bubbles denote language tools for that language, such as interpreters, compilers, state-space explorers, model checkers, deductive program verifiers, etc., which are all derived from the reference formal definition of that language by the framework. We identify the following characteristics of an ideal language framework:

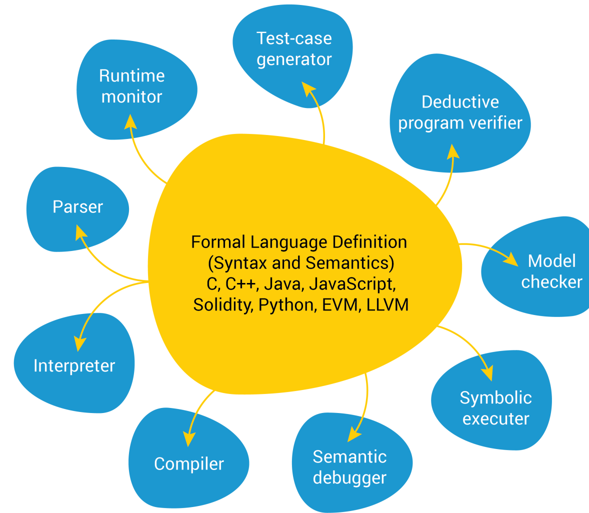

Fig. 2: The vision of an ideal language framework, pursued by $\mathbb{K}$

- The framework should be *language-independent*, in the sense that it uses the same generic method to generate language tools from the formal definitions for all programming languages.
- The framework should be *expressive*, to define the formal syntax and semantics of any programming language, with an intuitive and user-friendly *frontend interface*, so the formal definitions can be understood not only by experts but also by non-semanticists. In particular, the framework should provide easy-to-use facilities that help the language designers handle sophisticated features, such as non-deterministic computations, interaction, concurrency, and more, which are not uncommon in the real-world programming languages (see Section 2).
- The framework should support *modular development*, where formal definitions of large languages can be divided into smaller and more primitive modules. Language features should be loosely coupled, and language designers can easily add new features without revisiting existing definitions.
- The framework should support *testing-driven development*, where basic language tools such as the parser and the interpreter and/or compiler are automatically generated from language definitions for language designers to execute and test the semantics while they are defining it, by running lots of test programs and see if they get the intended results.
- The framework should have a mathematically solid logical foundation, in the sense that every semantic definition yields a *logical theory* of a foundational logic (see Section 4) and all language tools are best-effort implementations of logical reasoning of the foundational logic within the given logical theory.

– The framework should have a *minimal trustbase* that is fully comprehensible and accessible to users. The framework should provide *proof objects* as correctness certificates for all tasks it does. Proof objects can be mechanically and quickly checked by third-party proof checkers, so their correctness can not be compromised.

The 𝕂 *framework* [35,36] (`www.kframework.org`) represents a 20-year effort in pursuing and realizing the ideal language framework shown in Figure 2. There is enough evidence that the ideal vision is within our reach in the short term with 𝕂.

On the theory side, 𝕂 has a solid logical foundation based on matching logic [31,12], which we will discuss in detail in Section 4. Matching logic is an expressive logic that subsumes many important logics, calculi, and models that are used in both mathematics and computer science, in particular in program specification and verification; these include:

– First-order logic (FOL) and its extension with least/greatest fixpoints;
– Separation logic, which is designed specifically to define and reason about mutable data structures on heaps;
– Modal logic and modal μ-logic, as well as the various temporal logic and dynamic logic variants;
– Reachability logic, which supports 𝕂's program verification tools in a language-independent fashion; reachability logic captures the classic Hoare logic as a special instance.

Therefore, matching logic allows us to use 𝕂 to specify and reason about properties written in all the above logics in a systematic and uniform way.

On the practical side, the current 𝕂 implementations take the respective operational semantics of programming languages such as C [22], Java [8], and JavaScript [28] as well as emerging blockchain languages such as EVM [23] as parameters, and automatically generates language tools such as parsers, program interpreters, and program verifiers, for these languages. The auto-generated interpreters have competitive performance against hand-crafted interpreters and the automatic verifiers are capable of verifying challenging heap-manipulating programs at performance comparable to that of state-of-the-art verifiers specifically crafted for those languages. A precursor verifier specialized to the C programming language, MatchC [32] (`http://matching-logic.org`), has a user-friendly online interface for one to verify dozens of predefined or new programs.

The rest of the paper is organized as follows. In Section 2, we discuss some real-world languages whose formal semantics have been completely defined in 𝕂, in order to demonstrate that 𝕂 scales to complex, real languages. In Section 3, we present the complete 𝕂 definitions of two example languages, in order to illustrate the basic 𝕂 features and functionalities, including its parsing and program execution tools. In Section 4, we introduce matching logic, which is the logical foundation of 𝕂. In Section 5, we discuss the language-independent program verification tools of 𝕂. We conclude the paper in Section 6.

This paper is *not* peer reviewed and, indeed, aims at making no novel contributions. It is meant to simply give the students attending the SETSS'19 summer school an overview of the 𝕂 framework and its applications. Specifically, this paper extends the lecture notes of the Marktoberdorf'16 summer school [34], sometimes ad litteram, with material presented in the following papers: [31,13,12,11].

2 𝕂 Scales

Many real programming languages have a formal semantics defined in 𝕂, with their language tools being automatically generated in a correct-by-construction manner. Here we list some representative milestone examples.

C [22]. A complete formal semantics of C11 has been defined, aiming at capturing all the *undefined behaviors* of C. This semantics powers the commercial RV-Match tool, developed and maintained by Runtime Verification Inc. (RV) founded by the second author, aiming at mathematically rigorous dynamic checking of C programs in compliance with the ISO C11 Standard.

Java [9]. A complete formal semantics of Java 1.4 has been defined, which captures all language features and has been extensively tested using a test suite developed in parallel with the semantics, in a test-driven development methodology. The test suite itself was itself an important outcome of the semantics, because at that time Java did not appear to have any publicly available conformance testsuite.

JavaScript [29]. A complete formal semantics of JavaScript has been defined and thoroughly tested against the ECMAScript 5.1 conformance test suite, passing all 2,782 core language tests. The semantics also yields a simple coverage metric for the existing test suites, which is the set of 𝕂 *semantic rules* they exercise; see Section 3. It turned out that the ECMAScript 5.1 conformance test suite was incomplete and failed to cover several semantic rules. The authors of [29] wrote additional tests to exercise those rules and found bugs in commercial JavaScript engines.

Python [21]. Defining the complete formal semantics of Python is one of the first efforts that demonstrated the ability of 𝕂 to formalize complex programming languages. The semantics of Python 3.3 provided an interpreter and several analysis tools for exploring program state space and performing static reasoning and formal verification. The semantics was thoroughly tested against a number of unit tests and was shown to perform as efficiently as CPython, the reference implementation of Python, on those tests.

x86-64 [16] Not being a high-level programming language, x86-64 can also be given a formal semantics in 𝕂 similar to the other high-level languages. The formal semantics of x86-64 faithfully formalizes all the non-deprecated, sequential user-level instructions of the x86-64 Haswell instruction set architecture, including 3,155 instruction variants that correspond to 774 mnemonics. The semantics is fully executable and has been tested against over 7,000 instruction-level test cases and the GCC torture test suite. This extensive testing paid off, revealing bugs in both the x86-64 reference manual and other existing semantics. The formal semantics can be used for formal analyses such as processor verification.

EVM [23]. Ethereum virtual machine (EVM) is a bytecode stack-based language that all smart contracts on the Ethereum blockchain are compiled to and then executed by EVM interpreters. A complete formal semantics of EVM, called KEVM, has been defined in 𝕂. The correctness and performance of KEVM have been experimentally evaluated using the official Ethereum test suite, consisting of over 40,000 EVM programs. As a pleasant surprise, the EVM interpreter that is automatically generated by 𝕂 from KEVM is as efficient as the reference JavaScript implementation, suggesting that virtual machines for blockchains (and not only) can realistically be automatically generated from their formal semantics and performance is no longer a main obstacle issue.

IELE [24]. Like EVM, IELE [24] is another virtual machine bytecode language. Unlike EVM, IELE was designed in the spirit of *easy formal verification*, making it significantly different from EVM in various aspects. For example, IELE is a register-based machine instead of a stack-based one; IELE supports unbounded integers, whose reasoning is often easier than bounded integers. IELE was designed in a semantic-driven methodology using 𝕂, and a virtual machine was automatically generated from the formal semantics, making it the first virtual machine whose development and implementation were completely powered by formal methods.

3 Example Language Definitions in 𝕂

In this section, we illustrate the basic features and functionalities of 𝕂 in terms of two example programming languages: one is functional and the other imperative. For more example languages defined in 𝕂, we refer to the online 𝕂 tutorial (`www.kframework.org`).

3.1 LAMBDA: A Functional Language

Here we show the complete 𝕂 definition of a simple functional language definition called LAMBDA. LAMBDA is named after λ-calculus [14], one of the earliest mathematical models of computation, proposed by Alonzo Church in the 1930s, even earlier than when Alan Turing proposed Turing machines. The simplest

form of the λ-calculus is untyped λ-calculus, which consists of only untyped variables, function application, and function abstraction. Function abstraction is also called λ-abstraction, written $\lambda x \,.\, e$, which defines a function object as a process from argument x to return value e, which is a λ-calculus expression that mostly likely contains x. There are many extensions of λ-calculus with *types*. In there, functions can only be applied to arguments of matched types. Typical examples of typed extensions of λ-calculus include the simply-typed and polymorphic typed λ-calculus, as well as type systems, which form the foundations of proof assistants such as Coq [7], Agda [27], and Idris [10].

In the following, we assume readers are familiar with the basic concepts of λ-calculus, such as λ-binder and its binding behavior, α-renaming and α-equivalence, capture-avoiding substitution, and β-reduction. Background knowledge about λ-calculus can be found in [5].

The functional language LAMBDA is a direct incarnation of the untyped λ-calculus in $\mathbb{K}$.

Importing substitution module. We need the predefined substitution module[1] to define β-reduction in λ-calculus (discussed later). We *require* the substitution definition with the command below and then *import* the SUBSTITUTION module in our LAMBDA module below.

```
require "substitution.k"

module LAMBDA
  imports SUBSTITUTION
```

Basic syntax: Call-by-value. We define the conventional call-by-value syntax of λ-calculus, making sure that the λ-abstraction construct `lambda` is declared to be a *binder*, the function application to be *strict*, and the parentheses used for grouping as a *bracket* (explained shortly after).

```
  syntax Val ::= Id
               | "lambda" Id "." Exp  [binder]
  syntax Exp ::= Val
               | Exp Exp              [left, strict]
               | "(" Exp ")"          [bracket]
  syntax KVariable ::= Id
  syntax KResult ::= Val
```

Syntax is defined using the keyword `syntax` and may contain one or more production rules, separated with the vertical bar `|`. Every production rule is defined using the conventional BNF notation, with terminals enclosed in quotes and nonterminals starting with capital letters. Nonterminals are sometimes called *sorts* or *syntactic categories*.

[1] Substitution can be defined fully generically in $\mathbb{K}$ (not shown here) and then used to give semantics to various constructs in various languages.

In the above, `Val` is the syntactic category of the *values* in λ-calculus, which are irreducible λ-calculus expressions. `Exp` is the syntactic category of all λ-calculus expressions. Parentheses are used only for grouping. The `[bracket]` tells $\mathbb{K}$ to not construct internal nodes for parentheses when it generates the parse trees of λ-calculus expressions, so we do not need to bother giving explicit idle semantics for parentheses. `Id`, `KVariable`, and `KResult` are three builtin nonterminals that are predefined in $\mathbb{K}$. `Id` contains all identifiers (in a syntax that is similar to the identifiers in C), which are used to represent λ-calculus variables. `KVariable` is used to define the binding behavior of `lambda`. `KResult` is used to specify the evaluation strategies of $\mathbb{K}$, which are explained below.

Attributes. $\mathbb{K}$ associates the BNF syntax definitions with *attributes*. Attributes are put in square braces `[...]`. Some attributes contain only syntactic meanings and only affect parsing. The other attributes may contain semantic information and can affect program execution. The `bracket` attribute is used for grouping and has been discussed before. The `left` attribute specifies that function application $e_1\, e_2$ is associative to the left, so $\mathbb{K}$ parses $e_1\, e_2\, e_3$ as $(e_1\, e_2)\, e_3$. The `strict` attribute defines *evaluation context* that determines $\mathbb{K}$'s strategy to evaluate expressions and execute programs. Language constructs with a `strict` attribute can evaluate their arguments in any (fully nondeterministic) order. Therefore, $\mathbb{K}$ evaluates the expression $e_1\, e_2$ by first evaluating e_1 to value v_1 and e_2 to value v_2, fully nondeterministically, and finally evaluates $v_1\, v_2$.

`KResult` is a builtin nonterminal predefined in $\mathbb{K}$. It contains all syntactic categories and domain values that should be regarded as *results of computation*. $\mathbb{K}$ uses this information to decide when to continue and stop evaluation. Note that $\mathbb{K}$ does not infer results of computation automatically. The language designer should explicitly specify the results of computation by defining `KResult` properly.

`KVariable` includes all identifiers in `Id` and it tells $\mathbb{K}$ to "hook" λ-calculus variables to $\mathbb{K}$'s internal identifiers. This triggers the capture-avoiding substitution in $\mathbb{K}$, which we will discuss in the next paragraph.

Substitution and β-reduction. Here we define β-reduction in λ-calculus using $\mathbb{K}$'s rewrite rules. Recall that β-reduction refers to the following axiom schema:

$$(\lambda x\,.\,e)\, e' = e[e'/x] \qquad \text{for variable } x \text{ and expressions } e, e'$$

where $e[e'/x]$ denotes capture-avoiding substitution, where bound variables are implicitly renamed (called *α-renaming*) to avoid unintended variable capture during the substitution. For example, consider this instance of the β-reduction schema: $(\lambda x.\lambda y.xy)\, y = (\lambda y.xy)[y/x]$. If we simply replace all occurrences of x for y in $\lambda y.xy$, we would get $\lambda y.yy$, which is *not* the right result of capture-avoiding substitution because the former y is accidentally captured by λy after substitution. To avoid that, capture-avoiding substitution first renames the bound variable y in $\lambda y.xy$ to a fresh variable, say z, and gets $\lambda z.xz$. Then, the substitution $(\lambda z.xz)[y/x]$ will not cause variable capture, and we can get the correct result $\lambda z.yz$.

The above axiom is often called the β-reduction *rule* when it is oriented and applied from left to right. In $\mathbb{K}$, we use *rewrite rules* to implement β-reduction. $\mathbb{K}$ uses the keyword `rule` to define a rewrite rule, or simply a rule. In addition, $\mathbb{K}$ has builtin support for capture-avoiding substitution, which is predefined in the module `SUBSTITUTION` that we imported in the beginning. To use $\mathbb{K}$'s builtin capture-avoiding substitution, we must explicitly tell $\mathbb{K}$ what syntactic category is the one for variables, so $\mathbb{K}$ knows how to generate fresh variables during substitution. This is done by defining the `KVariable` and let it include `Id`, which is used to represent all λ-calculus variables.

The following is the $\mathbb{K}$ rule that implements β-reduction.

```
rule (lambda X:Id . E:Exp) V:Val => E[V / X]
```

Here, `X:Id` is a $\mathbb{K}$ variable, decorated with its syntactic category `Id`, or called *sort* of `X`. Note that we use `V:Val` with sort `Val` instead of `Exp`, because function application is `strict`, so $\mathbb{K}$ will always first evaluate both its arguments to values. In other words, if `V` is not yet a value (i.e., `KResult`), $\mathbb{K}$ does not apply the β-reduction rule. Instead, $\mathbb{K}$ will evaluate `V` further, until it becomes a value.

Nontermination The `strict` attribute drives $\mathbb{K}$'s evaluation strategy. Together with `KResult`, the `strict` attributes offer hints to $\mathbb{K}$ to help it execute programs (i.e., to apply rewrite rules) more efficiently. The `strict` attributes do *not* mean to guarantee the termination of program execution. For example, the following expression `(lambda x . (x x)) (lambda x . (x x))` does not terminate. In fact, it represents the famous λ-calculus Ω combinator, which is the simplest λ-expression whose β-reduction process does not terminate.

Integer and Boolean builtins. We can define arithmetic and Boolean expression constructs, which are simply rewritten to their builtin counterparts once their arguments are evaluated.

```
syntax Val ::= Int | Bool
syntax Exp ::= Exp "*" Exp          [strict, left]
             | Exp "/" Exp          [strict]
             > Exp "+" Exp          [strict, left]
             > Exp "<=" Exp         [strict]
rule I1 * I2 => I1 *Int I2
rule I1 / I2 => I1 /Int I2  requires I2 =/=Int 0
rule I1 + I2 => I1 +Int I2
rule I1 <= I2 => I1 <=Int I2
```

The operations with sort suffixes (such as `*Int` and `/Int`) are $\mathbb{K}$'s builtin arithmetic operations and come with the corresponding builtin sort. Note that the variables appearing in these rules have `Int` sort. That means that these rules will only be applied after the arguments of the arithmetic constructs are fully evaluated to $\mathbb{K}$ results. This happens thanks to their strictness attributes declared as annotations to their syntax declarations.

The keyword `requires` specifies the condition when a rewrite rule can be matched and applied. Therefore, `I1 / I2` is only defined when `I2` is not zero. When `I2` is zero, `I1 / I2` can not be matched by any rewrite rules and thus the execution gets stuck.

Conditional expressions. We can define conditional expressions as follows. Note that the `if` construct is strict only in its first argument. Therefore, 𝕂 will only evaluate its first argument (the condition) to a result and will not touch its second and third argument.

```
syntax Exp ::= "if" Exp "then" Exp "else" Exp    [strict(1)]
rule if true  then E else _ => E
rule if false then _ else E => E
```

Let binder. The let binder is a derived construct, because it can be defined using the λ-binder. The `macro` attribute means that the rule that desugars `let` is applied statically during compilation on all expressions that it is matched, and statically *before* evaluating the given λ-expressions.

```
syntax Exp ::= "let" Id "=" Exp "in" Exp
rule let X = E in E':Exp => (lambda X . E') E         [macro]
```

Letrec binder. Similarly, `letrec` can also be defined in 𝕂. Here, we prefer a definition based on the μ-binder that constructs the fixpoints in λ-calculus.

```
  syntax Exp ::= "letrec" Id Id "=" Exp "in" Exp
               | "mu" Id "." Exp                  [binder]
  rule letrec F:Id X:Id = E in E' => let F = mu F . lambda X . E in E' [macro]
  rule mu X . E => E[(mu X . E) / X]
endmodule
```

Finally, we finish the definition of module LAMBDA with the keyword `endmodule`.

Compiling 𝕂 *definitions and executing programs.* The 𝕂 definition of LAMBDA is now complete. We can compile it using the command

```
$ kompile lambda.k
```

Then we can execute programs, i.e., evaluating λ-expressions using the `krun` command. For example, if the file `factorial.lambda` contains the LAMBDA program

```
letrec f x = if x <= 1 then 1 else (x * (f (x + -1)))
in (f 10)
```

then the command

```
$ krun factorial.k
```

yields the expected result 3628800.

3.2 IMP: An Imperative Language

In this section, we discuss the $\mathbb{K}$ definition of the prototypical IMP language. IMP is a simple imperative language. It is considered as a folklore language, without an official inventor, and has been used in many textbooks and papers, often with slight syntactic variations and often without being called IMP. It includes the most basic imperative language constructs, namely basic constructs for arithmetic and Boolean expressions, and variable assignment, conditional, while loop and sequential composition constructs for statements.

The $\mathbb{K}$ definition of IMP has two modules: IMP-SYNTAX that defines the syntax of IMP and IMP that imports IMP-SYNTAX and defines the formal semantics in terms of $\mathbb{K}$'s rewrite rules.

Syntax of IMP.

```
module IMP-SYNTAX
```

This module defines the syntax of IMP as shown below.

```
  syntax AExp  ::= Int | Id
                 | AExp "/" AExp              [left, strict]
                 > AExp "+" AExp              [left, strict]
                 | "(" AExp ")"               [bracket]
  syntax BExp  ::= Bool
                 | AExp "<=" AExp  [seqstrict, latex({#1}\leq{#2})]
                 | "!" BExp                   [strict]
                 > BExp "&&" BExp             [left, strict(1)]
                 | "(" BExp ")"               [bracket]
  syntax Block ::= "{" "}"
                 | "{" Stmt "}"
  syntax Stmt  ::= Block
                 | Id "=" AExp ";"            [strict(2)]
                 | "if" "(" BExp ")"
                   Block "else" Block         [strict(1)]
                 | "while" "(" BExp ")" Block
                 > Stmt Stmt                  [left]
  syntax Pgm ::= "int" Ids ";" Stmt
  syntax Ids ::= List{Id,","}
endmodule
```

As in LAMBDA, the syntax of the language is defined using the conventional BNF grammar. Syntax productions are separated by "|" and ">", where "|" means the two productions have the same precedence while ">" means the previous production has higher precedence (binds tighter) than the one that follows. In our example, all language constructs bind tighter than the sequential operator in IMP. `Int` and `Id` are two built-in categories of integers and identifiers (program variables), respectively. `Exp` is the category of expressions, which subsumes `Int` and `Id`, and contains two other productions for plus and minus. `Pgm` is the category of IMP programs.

A wellformed IMP program declares a list of program variables in the beginning and then executes a statement in the state obtained after initializing all those variables to 0. `Ids` is the category for lists of program variables, and it is defined using 𝕂's built-in template `List`. The first argument is the base category `Id`, and second argument is the separating character `","`.

The `seqstrict` attribute specifies that `<=` is sequentially strict, so its arguments will be evaluated *in order* from left to right. `<=` also has a LaTeX attribute making it display as $\leq$, and that `&&` is strict only in its first argument, because we want to give it a short-circuit semantics.

We are done with the definition of IMP's syntax.

Semantics of IMP.

```
module IMP
  imports IMP-SYNTAX
```

The module IMP defines the semantics of IMP as a set of 𝕂 rewrite rules.

Values and results. IMP only has two types of results of computations: integers and Booleans, as defined below:

```
  syntax KResult ::= Int | Bool
```

Configurations. Unlike LAMBDA, the execution of IMP programs requires an execution environment. Specifically, we need to define *program states* that map variables to their values.

In general, 𝕂 uses *configurations* to organize the execution environment. A configuration represents a program execution state, holding all information that is needed for program execution. Configurations are organized into *cells*, which are labeled and can be nested. Simple languages such as IMP have only a few cells, while complex real languages such as C have a lot more. Configurations are defined in XML format as below:

```
  configuration <T color="yellow">
                  <k color="green"> $PGM:Pgm </k>
                  <state color="red"> .Map </state>
                </T>
```

An IMP configuration has two cells: a `<k/>` cell and a `<state/>` cell. For clarity, we gather both cells and put them in a top-level cell `<T/>` cell. For better readability, we color the `<k/>` in green, color the `<state/>` in red, and color the `<T/>` cell in yellow. The `<k/>` cell holds the rest computation (i.e., program fragments) that needs to execute and the `<state/>` cell holds a map from program variables to their values in the memory. Initially, the `<state/>` cell holds the empty map, denoted as `.Map`. In 𝕂, we write "`.`" for "nothing", and `.Map` means the type of the "nothing" is `Map`.

The special configuration variable `$PGM` tells the 𝕂 tool where to place the program. More precisely, the command "`krun file.imp`" parses the IMP program in file `file.imp` and places the resulting 𝕂 abstract syntax tree in the `<k/>` cell before invoking the semantic rules described in the sequel.

Arithmetic and Boolean Expressions. The $\mathbb{K}$ semantics of each arithmetic construct is defined below.

Variable lookup. A program variable X is looked up in the state by matching a binding of the form $X \mapsto I$ in the state cell. If such a binding does not exist, then the rewriting process gets stuck. In other words, we disallow uses of uninitialized variables in IMP. Note that variable lookup is the first task performed while evaluating the statement in the `<k/>` cell (the cell is closed to the left and open to the right, as marked by the "`...`" on the right), while the binding can be anywhere in the `<state/>` cell (the cell is open at both sides, as marked by the "`...`" on both sides). Specifically, "`...`" means something "that exists but does not change in the rewrite". The rule, therefore, says that if a program variable `X:Id` is the current computation fragment in the `<k/>` cell, and `X` binds to the integer `I` somewhere in the `<state/>` cell, then `X:Id` is rewritten to `I` and nothing else should change.

```
rule <k> X:Id => I ...</k> <state>... X |-> I ...</state>
```

In the above, we color the two cells in blue and red, respectively, for readability. The above shows an important characteristic of $\mathbb{K}$'s rewrite rules: $\mathbb{K}$ supports *local rewrites*, which are rewrite rules whose rewrite symbols "`=>`" occur not necessarily at the top but locally at where the rewrites happen. Without local rewrites, the above variables looking up rule has to be written as below:

```
rule <k> X:Id ...</k> <state>... X |-> I ...</state>
  => <k> I    ...</k> <state>... X |-> I ...</state>
```

As we can see, local rewrites avoid writing duplicate expressions on both the LHS and RHS of the rewrites.

Arithmetic operators. We can define the semantics of arithmetic operators in the usual way.

```
rule I1 / I2 => I1 /Int I2  requires I2 =/=Int 0
rule I1 + I2 => I1 +Int I2
```

Note that $\mathbb{K}$'s *configuration abstraction* mechanism is at work here. In other words, rewrite rules do not need to explicitly mention all configuration cells but only those related. $\mathbb{K}$ will infer the implicit cells, compete the configuration automatically, and apply the rewrite rule. Without configuration abstraction, the above rule for arithmetic operators has to be written as:

```
rule <k> I1 + I2 => I1 +Int I2 ... </k> <state> ... </state>
```

Not only is the rule using configuration abstraction more succinct, but it is also more modular. Suppose we need to modify the semantics and add a new configuration cell, we do not need to modify the rules with configuration abstraction because the new added cells can be automatically inferred and completed by $\mathbb{K}$. Configuration abstraction is one of the most important features that makes $\mathbb{K}$ definitions extensible and easy to adapt to language changes.

Boolean expressions. The following rules for Boolean expressions are straightforward.

```
rule I1 <= I2 => I1 <=Int I2
rule ! T => notBool T
rule true && B => B
rule false && _ => false
```

Program Statements.

Blocks. The empty block `{}` is simply dissolved. The dot symbol "`.`" represents the unit of the computation list structure, i.e., the empty task. Similarly, the nonempty blocks are dissolved and replaced by their statement contents, thus effectively giving them a bracket semantics; we can afford to do this only because we have no block-local variable declarations yet in IMP. Since we tagged the rules below with attribute `structural`, 𝕂 structurally erases the block constructs from the computation structure, without considering their erasure as computational steps in the resulting transition systems. In other words, these rules are not regarded as computational steps.

```
rule {} => .   [structural]
rule {S} => S  [structural]
```

Assignments. The variable `X` is assigned a new integer value `I` and then the program state is updated accordingly.

```
rule <k> X = I:Int; => . ...</k> <state>... X |-> (_ => I) ...</state>
```

Sequential composition. Sequential composition is simply structurally translated to 𝕂's builtin task sequentialization operation "`~>`". In other words, the effect of executing the sequential composition statement `S1 S2` is equivalent to the effect of first executing `S1` and then executing `S2`.

```
rule S1:Stmt S2:Stmt => S1 ~> S2  [structural]
```

Conditional statements. The conditional statement has two semantic cases. We have seen them in defining LAMBDA.

```
rule if (true)  S else _ => S
rule if (false) _ else S => S
```

While loops. The semantics of a `while` loop is defined simply by unfolding the loop once.

```
rule while (B) S => if (B) {S while (B) S} else {}  [structural]
```

Note that the above rule works because conditional statement (on the right-hand side) has the attribute `strict(1)`, so the inner `while` loop in the then-branch of the `it`-statement will not be unfolded.

Programs. An IMP program is a list of program variables declarations followed by a statement. The semantics is that the statement is executed in the initial state where all declared variables have value 0. 𝕂's syntactic lists are internally interpreted as cons-lists (i.e., lists constructed with a head element followed by a tail list), we have two cases. One is when the list has at least one element. The other is when the list is empty. In the first case, we initialize the variable to 0 in the state, but only when it is not already declared (we use juxtaposition to denote list concatenation in the following 𝕂 rules). In the second case, we dissolve the residual empty `int;` declaration as a structural cleanup.

```
  rule <k> int (X,Xs => Xs);_ </k>
       <state> Rho:Map (.Map => X|->0) </state>
  requires notBool (X in keys(Rho))
  rule int .Ids; S => S  [structural]
endmodule
```

We have finished the definition of module IMP.

Compiling the Definition and Executing IMP Programs. After compilation with the command `kompile imp.k`, we can execute programs. Suppose `sum.imp` contains the following program:

```
  int n, sum;
  n = 100;
  sum = 0;
  while (!(n <= 0)) {
    sum = sum + n;
    n = n + -1;
  }
```

then `krun sum.imp` yields the following final configuration

```
<T>
  <k> . </k>
  <state>
    n |-> 0
    sum |-> 5050
  </state>
</T>
```

Notice that in the final configuration, the `<k/>` cell is empty, meaning that the program was completely executed, or consumed. In the end of the execution, the program variable `n` has value 0 and `s` has value 5050, which is the total of numbers up to 100, as expected.

𝕂 is able to automatically generate a parser and an interpreter of any language from its formal definition, as we have seen in LAMBDA and IMP. This capability of 𝕂 is crucial for testing language semantics and thus for increasing confidence in its adequacy. The above also illustrates another useful 𝕂 tool: the 𝕂 *unparser*,

which is used by almost any other tool. Indeed, the above configuration result uses concrete language syntax (i.e. the syntax of IMP) to display the cells and their contents, although internally these are all represented as abstract data types.

We point out that the interpreters automatically generated by 𝕂 can be *reasonably efficient*. For example, the formal definition of the Ethereum virtual machine (EVM) bytecode language, one of the most popular virtual machine languages for the blockchain, yields an EVM interpreter that is as efficient as the hand-written reference JavaScript implementation of EVM [23].

4 Matching Logic: The Logical Foundations of 𝕂

In this section, we introduce matching logic [12,31,11], the foundational logic underlying 𝕂. In Section 4.1, we discuss the motivation behind the design of matching logic. In Section 4.2, we formally define the syntax and semantics of matching logic. In Section 4.4, we introduce the Hilbert-style proof system of matching logic, using which we can carry out all logical reasoning in the logics and calculi mentioned in Section 1.2, all of which have been defined as theories and/or notations in matching logic, as shown in Section 4.3.

4.1 Matching Logic: Motivations

One main motivation for the design of matching logic is to give language semantic frameworks, such as 𝕂, a mathematically sound and rigorous logical foundation. Specifically, we want a foundational logic that is able to:

1. specify and reason about static program structures and configurations;
2. specify and reason about dynamic program behaviors and properties;
3. specify and reason about (least/greatest) fixpoints, which occur in both static structures (such as inductive/co-inductive data types) and dynamic properties (such as temporal and reachability properties).

We discuss the three motivations respectively in the following.

Motivation 1: Specifying and reasoning about static program structures and configurations. Traditionally, static structures are specified using first-order logic (FOL) *terms*, which are built from variables, constants, and function symbols and can be used to define data constructors and language constructs. On the other hand, the properties about static structures are specified using FOL *formulas* as logical constraints, which are built from the primitive predicate symbols and composed using logical connectives.

However, such a clear distinction between terms (that represent data) and formulas (that represent the properties of data) can be inconvenient when it comes to specifying and reasoning about *program configurations*.

Consider as an example the program shown in Figure 3, which reads `n` elements and outputs them in reversed order. The reader need not to understand

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

void list_read_write(int n) {
rule ⟨$Pgm ⇒ return; ···⟩code ⟨A ⇒ · ···⟩in ⟨··· · ⇒ rev(A)⟩out ∧ n = len(A)
  int i=0;
  struct listNode *x=0;
inv ⟨β ∧ len(β) = n − i ∧ i ≤ n ···⟩in ⟨list(x, α) ···⟩heap ∧ A = rev(α)@β
  while (i < n) {
    struct listNode *y = x;
    x = (struct listNode*) malloc(sizeof(struct listNode));
    scanf("%d", &(x->val));
    x->next = y;
    i += 1; }
inv ⟨··· α⟩out ⟨list(x, β) ···⟩heap ∧ rev(A) = α@β
  while (x) {
    struct listNode *y;
    y = x->next;
    printf("%d␣",x->val);
    free(x);
    x = y; }
}
```

Fig. 3: Reading, storing, and reverse writing a sequence of integers

all details; we will explain what are necessary below. The formal specifications of the program are given in gray. Note that in the specifications, we need to match an *abstract* sequence of n elements in the input buffer, and then to match its reverse at the end of the output buffer when the function terminates. Furthermore, in order to state the invariants of the two loops, we need to identify a singly linked list pattern in the heap, which is a finitely-supported partial map. Many such sequence or map patterns, as well as functions and operations on them, can be defined using conventional algebraic data types (ADTs) and/or FOL terms.

However, there are limitations. A major limitation is that function symbols must be interpreted as functions in models, which sometimes is insufficient. For example, a two-element linked list in the heap starting with location 7 and holding values 9 and 5, written as $list(7, 9@5)$, can allow infinitely many heap values, one for each location where the value 5 may be stored. So we cannot define *list* as an operation symbol $Int \times Seq \to Map$. The FOL alternative is to define *list* as a predicate $Int \times Seq \times Map$, taking an additional heap argument. but mentioning the map all the time as an argument makes specifications verbose and hard to read, use and reason about. An alternative, proposed by separation logic [30], is to fix and move the map domain from explicit in models to implicit in the logic, so that $list(7, 9@5)$ is interpreted as a predicate but the non-deterministic map choices are implicit in the logic. The drawback of that, is that we may need customized separation logics for different languages that require different varia-

tions of map models or different configurations making use of different kinds of resources. This may also require specialized separation logic tools and provers, or otherwise encodings that need to be proved correct. Finally, since the map domain is not available as data, one cannot use FOL variables to range over maps and thus proof rules like "heap framing" need to be added to the logic explicitly.

Matching logic avoids the limitations of the approaches above, by interpreting its terms/formulae uniformly as *sets* of values. Matching logic's formulas, called *patterns*, are built using variables, symbols from a signature, and FOL connectives and quantifiers, and their semantics are the sets of values that *match* them; see Section 4.2.

Motivation 2: Specifying and reasoning about dynamic program behaviors and properties. Traditionally, dynamic behaviors and properties can be specified in modal logic and/or modal μ-logic, as well as their temporal logics and dynamic logics fragments. Modal logic uses *modal operators* to specify various dynamic properties of transition systems. For example, the "next" operator $\circ\varphi$ holds on a state if the next state satisfies φ; $\Box\varphi$ holds if φ always holds; the "eventually" operator $\diamond\varphi$ holds if φ eventually holds; etc.

A major limitation of modal logic is that it has no direct support for specifying the static structures of states. Indeed, in modal logic models, which are transition systems, states are structureless "points". Therefore, it is insufficient to specify and reason about program configurations, especially 𝕂 configurations, which are nested structures built from basic mathematical domain values, data constructors, language constructs, and configuration cells.

Matching logic overcomes this limitation by defining modal logic operators uniformly using *symbols*. Recall that symbols and matching logic patterns are interpreted as the sets of elements that match them, so matching logic symbols are naturally interpreted as relations and can thus be used to capture the transition relations in transition systems. In addition, matching logic can use symbols and its FOL connectives and quantifiers to re-construct FOL formulas and structures, which are ideal in defining program configurations.

Motivation 3: Specifying and reasoning about least/greatest fixpoints Fixpoints, especially least and greatest fixpoints, play an important role in programming languages semantics. Many real-world programming languages support inductive data types, which are mathematical domains that are defined as the smallest sets closed under user-defined constructors. Some programming languages, such as Haskell, support co-inductive data types (also called infinite data types). Both inductive and co-inductive data types are special instances of least/greatest fixpoints about static structures.

Fixpoints also play an important role in defining dynamic program behaviors and properties. For example, modal operators $\Box\varphi$ (always φ), $\diamond\varphi$ (eventually φ), $\varphi_1 \mathbin{\mathrm{U}} \varphi_2$ (φ_1 until φ_2, meaning that φ_1 holds from now until the first time φ_2 holds), can all be defined using least/greatest fixpoints from the basic transition

relations, i.e., the "next" operator $\circ\varphi$. For program verification, we define *reachability properties* $\varphi_1 \Rightarrow \varphi_2$, read "$\varphi_1$ reaches φ_2", to mean that φ_1 can reach φ_2 on some finite execution paths, which corresponds to the partial correctness semantics in the traditional Hoare-style verification; see Section 5.3.

Matching logic provides built-in support for fixpoint reasoning that occurs in both static structures and configurations and dynamic behaviors and properties, and in particular program verification.

4.2 Matching Logic: Syntax and Semantics

Matching logic formulas, called *patterns*, are built using variables, symbols, propositional connectives, FOL quantifiers, and fixpoints. Matching logic has a *powerset semantics*, where patterns are interpreted as the sets of elements that match them.

We assume readers are familiar with the basic notions and concepts about FOL and modal μ-logic.

Definition 1. *A* matching logic signature *or simply a* signature *is a triple* (S, V, Σ)*, where*

- S *is a nonempty set of* sorts *written* $s_1, s_2, \ldots$*;*
- $V = EV \cup SV$ *with* $EV \cap SV = \emptyset$ *is a disjoint union of two sets of variables, where* $EV = \{EV_s\}_{s \in S}$ *contains* sorted element variables *written* $x{:}s, y{:}s, \ldots$ *and* $SV = \{SV_s\}_{s \in S}$ *contains* sorted set variables *written* $X{:}s, Y{:}s, \ldots$*;*
- $\Sigma = \{\Sigma_{s_1 \ldots s_n, s}\}_{s_1, \ldots, s_n, s \in S}$ *is a set of many-sorted symbols written* $\sigma \in \Sigma_{s_1 \ldots s_n, s}$*, where* $s_1, \ldots, s_n$ *are called the argument sorts and* s *is called the return sort.*

Given a signature (S, V, Σ)*, matching logic* patterns *are inductively defined as follows for all* $s, s' \in S$*:*

$$\begin{aligned}\varphi_s ::= \; & x{:}s \mid X{:}s \mid \sigma(\varphi_{s_1}, \ldots, \varphi_{s_n}) \quad \textit{where } \sigma \in \Sigma_{s_1 \ldots s_n, s} \\ & \mid \varphi_s \wedge \varphi'_s \mid \neg \varphi_s \mid \exists x{:}s' \,.\, \varphi_s \mid \mu X{:}s \,.\, \varphi_s\end{aligned}$$

where $\mu X{:}s \,.\, \varphi_s$ *requires all free occurrences of* $X{:}s$ *are under an even number of negations in* φ_s*. The logical connectives* $\vee, \rightarrow, \leftrightarrow, \forall$ *are defined in the usual way.*

ML patterns are interpreted on an underlying carrier set of elements, and each pattern is then interpreted as a *set of elements*, which are those that *match* the pattern. This is called the *pattern matching semantics* of ML, and is what inspired the name "matching logic". For example, pattern *zero* is matched by the natural number 0; pattern *succ*(*zero*) is matched by the number 1; the (disjunctive) pattern $\mathit{zero} \vee \mathit{succ}(\mathit{zero})$ is matched by 0 and 1; to put it another way, an element a matches $\mathit{zero} \vee \mathit{succ}(\mathit{zero})$, if a matches *zero or* a matches *succ*(*zero*). Intuitively, $\varphi_s \wedge \varphi'_s$ is matched by those matching both φ_s and φ'_s. Pattern $\neg \varphi_s$

is matched by the elements (of sort s) that do not match φ_s. Element variable $x{:}s$ is a pattern that is matched by exactly one element to which $x{:}s$ evaluates (evaluation of variables is defined later). Set variable $X{:}s$ is a pattern that is matched by exactly the elements in the set to which $X{:}s$ evaluates to.

The meaning of $\sigma(\varphi_{s_1}, \ldots, \varphi_{s_n})$, called a symbol application, depends on how we interpret σ. For example, if σ is interpreted as a *constructor*, then $\sigma(\varphi_{s_1}, \ldots, \varphi_{s_n})$ is matched by the *structures* built by σ on elements matching $\varphi_{s_1}, \ldots, \varphi_{s_n}$, respectively. If σ is interpreted as a *function*, then $\sigma(\varphi_{s_1}, \ldots, \varphi_{s_n})$ is matched by the *return values* obtained by applying σ on elements matching $\varphi_{s_1}, \ldots, \varphi_{s_n}$. If σ is interpreted as a *relation* (such as the modal operators in modal logic). then $\sigma(\varphi_{s_1}, \ldots, \varphi_{s_n})$ is matched by the elements that have the relation σ with elements matching $\varphi_{s_1}, \ldots, \varphi_{s_n}$, respectively. In conclusion, matching logic symbols can be used to uniformly represent constructors, functions, and relations (predicates).

There are two *binders* in matching logic. The $\exists$-binder binds element variables and builds *abstraction* $\exists x{:}s' . \varphi_s$, which is matched by the elements that match φ_s for *some valuations* of $x{:}s'$. In other words, it "abstracts away" the irrelevant part (i.e., $x{:}s'$) from the matched part (i.e., φ_s). Note that the sort of the binding variable $x{:}s'$ needs not to be the same as the sort of the pattern φ_s.

The μ-binder builds *least fixpoints*. Intuitively, φ_s with free occurrences of $X{:}s$ defines a function $\mathcal{F}_{\varphi_s, X:S}$ that maps (the set of elements matching) $X{:}s$ to (the set of elements matching) φ_s. Since $X{:}s$ occurs positively in φ_s, we can verify that $\mathcal{F}_{\varphi_s, X:s}$ is a monotone function, so it has a unique least fixpoint denoted as $\mu\mathcal{F}_{\varphi_s, X:s}$, guaranteed by the Knaster-Tarski fixpoint theorem (Theorem 1). The least fixpoint pattern $\mu X{:}s . \varphi_s$ is then matched by the elements in set $\mu\mathcal{F}_{\varphi_s, X:s}$.

We define the notions of free variables, capture-avoiding substitution, α-renaming, etc. in the usual way. We use $\varphi[\psi/x{:}s]$ (resp. $\varphi[\psi/X{:}s]$ to denote the result of substituting ψ for $x{:}s$ (resp. $X{:}s$) in φ, where α-renaming happens implicitly to prevent variable captures.

We review the Knaster-Tarski fixpoint theorem [37].

Theorem 1 (Knaster-Tarski). *Let M be a nonempty set and $\mathcal{P}(M)$ be the powerset of M. Let $\mathcal{F}\colon \mathcal{P}(M) \to \mathcal{P}(M)$ be a monotone function, i.e., $\mathcal{F}(A) \subseteq \mathcal{F}(B)$ for all subsets $A \subseteq B$ of M. Then $\mathcal{F}$ has a unique least fixpoint, written $\mu\mathcal{F}$, and a unique greatest fixpoint, written $\nu\mathcal{F}$, given as:*

$$\mu\mathcal{F} = \bigcap\{A \in \mathcal{P}(M) \mid \mathcal{F}(A) \subseteq A\},$$
$$\nu\mathcal{F} = \bigcup\{A \in \mathcal{P}(M) \mid A \subseteq \mathcal{F}(A)\}.$$

We call A a pre-fixpoint *of $\mathcal{F}$ whenever $\mathcal{F}(A) \subseteq A$, and a* post-fixpoint *of $\mathcal{F}$ whenever $A \subseteq \mathcal{F}(A)$.*

We now define matching logic models and interpretations of patterns.

Definition 2. *An (S, V, Σ)*-model *is a pair $M = (\{M_s\}_{s\in S}, \{\sigma_M\}_{\sigma\in\Sigma})$, consisting of a nonempty carrier set M_s for every $s \in S$ and an interpretation*

$\sigma_M : M_{s_1} \times \cdots \times M_{s_n} \to \mathcal{P}(M_s)$ *for every* $\sigma \in \Sigma_{s_1 \ldots s_n, s}$. *We extend* σ_M *to its* pointwise extension, $\sigma_M : \mathcal{P}(M_{s_1}) \times \cdots \times \mathcal{P}(M_{s_n}) \to \mathcal{P}(M_s)$, *defined as*

$$\sigma_M(A_1, \ldots, A_n) = \bigcup_{a_i \in A_i, 1 \le i \le n} \sigma_M(a_1, \ldots, a_n)$$

for $A_i \subseteq M_{s_i}$, $1 \le i \le n$. *An* M*-valuation* $\rho \colon V \to M \cup \mathcal{P}(M)$ *is one such that* $\rho(x{:}s) \in M_s$ *and* $\rho(X{:}s) \subseteq M_s$ *for all* $x{:}s, X{:}s \in V$. *Its* extension $\bar{\rho}$ *interprets* (S, V, Σ)*-patterns to sets as follows:*

- $\bar{\rho}(x{:}s) = \{\rho(x{:}s)\}$;
- $\bar{\rho}(\sigma(\varphi_{s_1}, \ldots, \varphi_{s_n})) = \sigma_M(\bar{\rho}(\varphi_{s_1}), \ldots, \bar{\rho}(\varphi_{s_n}))$ *for all* $\sigma \in \Sigma_{s_1 \ldots s_n, s}$
- $\bar{\rho}(X{:}s) = \rho(X{:}s)$
- $\bar{\rho}(\varphi_s \wedge \varphi'_s) = \bar{\rho}(\varphi_s) \cap \bar{\rho}(\varphi'_s)$
- $\bar{\rho}(\exists x{:}s' . \varphi_s) = \bigcup_{a \in M_{s'}} \overline{\rho[a/x{:}s']}(\varphi_s)$
- $\bar{\rho}(\neg \varphi_s) = M \setminus \bar{\rho}(\varphi_s)$
- $\bar{\rho}(\mu X{:}s . \varphi_s) = \mu \mathcal{F}^{\rho}_{\varphi, X:s}$ *with* $\mathcal{F}^{\rho}_{\varphi, X:s}(A) = \overline{\rho[A/X{:}s]}(\varphi_s)$ *for* $A \subseteq M_s$

We say φ_s *holds in* M, *written* $M \vDash \varphi_s$, *iff* $\bar{\rho}(\varphi_s) = M_s$ *for all* ρ. *A* theory *is a set* Γ *of patterns. We write* $M \vDash \Gamma$, *iff* $M \vDash \varphi$ *for all* $\varphi \in \Gamma$. *We write* $\Gamma \vDash \varphi_s$, *iff* $M \vDash \varphi_s$ *for all models with* $M \vDash \Gamma$.

Predicate Patterns A difference between FOL formulas and ML patterns is that FOL formulas can only be interpreted as either true or false, while ML patterns can be interpreted as any subsets of the carrier set. To represent the (logical) true and false using patterns, we identify two special sets M and $\emptyset$, and use M to represent the logical truth and $\emptyset$ to represent the logical false. Obviously, not all patterns are interpreted as M or $\emptyset$. Given a model M, we call φ an M*-predicate*, if $\bar{\rho}(\varphi) \in \{\emptyset, M\}$ for all ρ. We call φ a *predicate* (or *predicate pattern*), if it is an M-predicate in all M. Predicate patterns can be built from $\bot$, $\top$, and ML logical constructs. More interesting patterns can be built from symbols and application. We will see more predicate patterns in Section 4.3 and throughout the paper. Roughly speaking, predicate patterns are the ML counterparts of FOL formulas. They make "statements", and can take only two possible values: M if the statements are facts, and $\emptyset$ if the statements are not facts.

4.3 Matching Logic Expressiveness

In this section, we discuss the expressiveness of matching logic by showing that FOL, inductive data types, transition systems, temporal logics, and reachability logic (for language-independent program verification) can be defined as theories and/or notations.

Important Mathematical Instruments. Several mathematical instruments of practical importance, such as definedness, totality, equality, membership, set containment, functions and partial functions, and constructors, can all be defined/axiomatized in matching logic.

Definition 3. *For any (not necessarily distinct) sorts s, s', let us consider a unary symbol $\lceil _ \rceil_s^{s'} \in \Sigma_{s,s'}$, called the* definedness symbol*, and the pattern/axiom $\lceil x{:}s \rceil_s^{s'}$, called* (Definedness)*. We define* totality *"$\lfloor _ \rfloor_s^{s'}$",* equality *"$=_s^{s'}$",* membership *"$\in_s^{s'}$", and* set containment *"$\subseteq_s^{s'}$" as derived constructs:*

$$\lfloor \varphi \rfloor_s^{s'} \equiv \neg \lceil \neg \varphi \rceil_s^{s'} \qquad \varphi_1 =_s^{s'} \varphi_2 \equiv \lfloor \varphi_1 \leftrightarrow \varphi_2 \rfloor_s^{s'}$$
$$x \in_s^{s'} \varphi \equiv \lceil x \wedge \varphi \rceil_s^{s'} \qquad \varphi_1 \subseteq_s^{s'} \varphi_2 \equiv \lfloor \varphi_1 \rightarrow \varphi_2 \rfloor_s^{s'}$$

and feel free to drop the (not necessarily distinct) sorts s, s'.

Intuitively, the axiom (Definedness) states that every individual element x is defined. This is true, because x is matched by *exactly one element* to which it evaluates. Therefore, in any model that validates (Definedness), $\lceil x \rceil$ is interpreted as the total set, according to ML validity (Definition 2). Now, consider any pattern φ that is defined, and that φ is matched by one element, say x. By *pointwise extension* (Definition 2), the interpretation of $\lceil \varphi \rceil$ must include the interpretation of $\lceil x \rceil$, which we know is the total set. Therefore, $\lceil \varphi \rceil$ is also interpreted as the total set, which is intended. On the other hand, if φ is *undefined*, its interpretation is the empty set, and by pointwise extension, $\lceil \varphi \rceil$ is also interpreted as the empty set. The above intuition is made formal below.

Proposition 1. *Let M be a matching logic model satisfying* (Definedness)*. Let ρ be any valuation. Then the following hold:*

- $\bar{\rho}(\lceil \varphi_s \rceil) = M_s$ *if* $\bar{\rho}(\varphi_s) \neq \emptyset$*, i.e.,* φ_s *is defined;*
- $\bar{\rho}(\lceil \varphi_s \rceil) = \emptyset$ *if* $\bar{\rho}(\varphi_s) = \emptyset$*, i.e.,* φ_s *is not defined;*
- $\bar{\rho}(\lfloor \varphi_s \rfloor) = M_s$ *if* $\bar{\rho}(\varphi_s) = M_s$*, i.e.,* φ_s *is total;*
- $\bar{\rho}(\lfloor \varphi_s \rfloor) = \emptyset$ *if* $\bar{\rho}(\varphi_s) \neq M_s$*, i.e.,* φ_s *is not total;*
- $\bar{\rho}(\varphi_s =_s^{s'} \varphi'_s) = M_{s'}$ *if* $\bar{\rho}(\varphi_s) = \bar{\rho}(\varphi'_s)$*;*
- $\bar{\rho}(\varphi_s =_s^{s'} \varphi'_s) = \emptyset$ *if* $\bar{\rho}(\varphi_s) \neq \bar{\rho}(\varphi'_s)$*;*
- $\bar{\rho}(x{:}s \in_s^{s'} \varphi_s) = M_{s'}$ *if* $\rho(x{:}s) \in \bar{\rho}(\varphi_s)$*;*
- $\bar{\rho}(x{:}s \in_s^{s'} \varphi_s) = \emptyset$ *if* $\rho(x{:}s) \notin \bar{\rho}(\varphi_s)$*;*
- $\bar{\rho}(\varphi_s \subseteq_s^{s'} \varphi'_s) = M_{s'}$ *if* $\bar{\rho}(\varphi_s) \subseteq \bar{\rho}(\varphi'_s)$*;*
- $\bar{\rho}(\varphi_s \subseteq_s^{s'} \varphi'_s) = \emptyset$ *if* $\bar{\rho}(\varphi_s) \not\subseteq \bar{\rho}(\varphi'_s)$*.*

As seen in Definition 2, symbols in matching logic are interpreted as relations. Specifically speaking, consider a symbol $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ and its interpretation $\sigma_M \colon M_{s_1} \times \cdots \times M_{s_n} \rightarrow \mathcal{P}(M_s)$. Obviously, functions and partial functions are special instances of matching logic symbols. Functions are when $|\sigma_M(a_1, \ldots, a_n)| = 1$ for all $a_1 \in M_{s_1}, \ldots, a_n \in M_{s_n}$. Partial functions are when

$|\sigma_M(a_1, \ldots, a_n)| \leq 1$ for all $a_1 \in M_{s_1}, \ldots, a_n \in M_{s_n}$. In the following, we show that functions and partial functions can be defined by axioms:

$$\text{(FUNCTION)} \qquad \exists y \,.\, \sigma(x_1, \ldots, x_n) = y$$
$$\text{(PARTIAL FUNCTION)} \qquad \exists y \,.\, \sigma(x_1, \ldots, x_n) \subseteq y$$

Intuitively, (FUNCTION) requires $\sigma(x_1, \ldots, x_n)$ to contain exactly one element and (PARTIAL FUNCTION) requires it to contain at most one element (recall that variable y evaluates to a singleton set). For brevity, we use the function notation $\sigma \colon s_1 \times \cdots \times s_n \to s$ to mean we automatically assume the (FUNCTION) axiom of σ. Similarly, partial functions are written as $\sigma \colon s_1 \times \cdots \times s_n \rightharpoonup s$.

First-Order Logic. We can use the above definitions of functions to capture first-order logic (FOL) in matching logic. Specifically, given a FOL signature (S, Σ, Π) with *function symbols* Σ and *predicate symbols* Π, the *syntax* of FOL is given by:

$$t_s ::= x \in \mathrm{VAR}_s \mid f(t_{s_1}, \ldots, t_{s_n}) \text{ with } f \in \Sigma_{s_1 \ldots s_n, s}$$
$$\varphi ::= \pi(t_{s_1}, \ldots, t_{s_n}) \text{ with } \pi \in \Pi_{s_1 \ldots s_n} \mid \varphi \to \varphi \mid \neg\varphi \mid \forall x . \varphi$$

To capture FOL, we define a matching logic signature $\mathbb{\Sigma}^{\mathsf{FOL}} = (S^{\mathsf{FOL}}, \Sigma^{\mathsf{FOL}})$ where $S^{\mathsf{FOL}} = S \cup \{Pred\}$ contains all FOL sorts plus a distinguished sort *Pred* for FOL formulas and $\Sigma^{\mathsf{FOL}} = \{f \colon s_1 \times \cdots \times s_n \to s \mid f \in \Sigma_{s_1 \ldots s_n, s}\} \cup \{\pi \in \Sigma^{\mathsf{FOL}}_{s_1 \ldots s_n, Pred} \mid \pi \in \Pi_{s_1 \ldots s_n}\}$ contains FOL function symbols as matching logic functions and FOL predicate symbols as matching logic symbols that return *Pred*. Let Γ^{FOL} be the resulting ML theory of signature $\mathbb{\Sigma}^{\mathsf{FOL}}$.

Proposition 2. *All FOL formulas φ are $\mathbb{\Sigma}^{\mathsf{FOL}}$-patterns of sort Pred, and we have $\vDash_{\mathsf{FOL}} \varphi$ iff $\Gamma^{\mathsf{FOL}} \vDash \varphi$ (see [31]), where $\vDash_{\mathsf{FOL}} \varphi$ means that φ is valid in FOL.*

Inductive Data Structures. Here we show how configurations and inductive data structures can be *precisely axiomatized* in matching logic.

Definition 4. *Let $\mathbb{\Sigma} = (\{Term\}, \Sigma)$ be a signature with one sort Term and at least one constant.* $\mathbb{\Sigma}$-terms *are defined as:*

$$t ::= c \in \Sigma_{\lambda, Term} \mid c(t_1, \ldots, t_n) \textit{ for } c \in \Sigma_{Term \ldots Term, Term}$$

The $\mathbb{\Sigma}$-term algebra $T^{\mathbb{\Sigma}} = (\{T^{\mathbb{\Sigma}}_{Term}\}, \{c_{T^{\mathbb{\Sigma}}}\}_{c \in \Sigma})$ *consists of:*

- *a carrier set $T^{\mathbb{\Sigma}}_{Term}$ of all $\mathbb{\Sigma}$-terms;*
- *a function $c_{T^{\mathbb{\Sigma}}} : T^{\mathbb{\Sigma}}_{Term} \times \cdots \times T^{\mathbb{\Sigma}}_{Term} \to T^{\mathbb{\Sigma}}_{Term}$ for all $c \in \Sigma_{Term \ldots Term, Term}$ defined as $c_{T^{\mathbb{\Sigma}}}(t_1, \ldots, t_n) = c(t_1, \ldots, t_n)$.*

Proposition 3. *Let $\mathbb{\Sigma} = (\{Term\}, \Sigma)$ be a signature with one sort Term and at least one constant. Define a $\mathbb{\Sigma}$-theory $\Gamma^{\mathsf{term}}_{\mathbb{\Sigma}}$ with* (Function) *axioms for all constructors, plus the following axioms:*

(No Confusion I) *for all $i \neq j$ and $s_i = s_j$:*

$$\neg(c_i(x_i^1, \ldots, x_i^{m_i}) \wedge c_j(x_j^1, \ldots, x_j^{m_j}))$$

(No Confusion II) *for all $1 \leq i \leq n$:*

$$(c_i(x_i^1, ..., x_i^{m_i}) \wedge c_i(y_i^1, ..., y_i^{m_i})) \to c_i(x_i^1 \wedge y_i^1, ..., x_i^{m_i} \wedge y_i^{m_i})$$

(Inductive Domain)

$$\mu D. \bigvee_{c \in \Sigma} c(D, \ldots, D)$$

Then for all $\mathbb{\Sigma}$-models $M \vDash \Gamma^{\mathsf{term}}_{\mathbb{\Sigma}}$, M is isomorphic to $T^{\mathbb{\Sigma}}$.

Intuitively, (Inductive Domain) forces that for all models M, the carrier set M_{Term} must be the *the smallest set* that is closed under all symbols in Σ, while (Function) and (No Confusion) force all symbols in Σ to be interpreted as injective functions, and different symbols construct different terms.

Transition Systems. At a high level, every $\mathbb{K}$ definition defines a transition system over program configurations. Here we show how to specify and reason about transition systems in matching logic. We first recall the definition of transition systems.

Definition 5. *A transition system $\mathbb{S} = (S, R)$ consists of a nonempty set S of states/configurations and a binary relation $R \subseteq S \times S$ called transition relation. For $s, t \in S$ such that $s\,R\,t$, we say that s is an R-predecessor of t and t is an R-successor of s.*

To capture transition systems in matching logic, we define a signature $\mathbb{\Sigma}^{\mathsf{TS}} = (\{State\}, \{\bullet \in \Sigma^{\mathsf{TS}}_{State,State}\})$ where *State* is the sort of states and $\bullet \in \Sigma^{\mathsf{TS}}_{State,State}$ is a symbol called *one-path next*.

An important observation is that matching logic models of the signature $\mathbb{\Sigma}^{\mathsf{TS}}$ are *exactly* the transition systems, where $\bullet \in \Sigma^{\mathsf{TS}}_{State,State}$ is interpreted as the transition relation R. Specifically, for any transition system $\mathbb{S} = (S, R)$, we can regard $\mathbb{S}$ as a model where S is the carrier set of *State* and $\bullet_{\mathbb{S}}(t) = \{s \in S \mid s\,R\,t\}$ contains all *R-predecessors* of t. The intuition is illustrated as follows:

$$\begin{array}{cccccl} \cdots & s & \xrightarrow{R} & s' & \xrightarrow{R} s'' \cdots & \text{// states} \\ & \bullet\bullet\varphi & & \bullet\varphi & \varphi & \text{// patterns} \end{array}$$

In other words, $\bullet\varphi$ is matched by states that have a next state matching φ.

Other dynamic properties about transition systems can be defined as patterns. As an example, let us define *all-path next* $\circ\varphi \equiv \neg\bullet\neg\varphi$. It is straightforward

to show that $\circ\varphi$ is matched by states if all their R-successors matching φ. In particular, if s has no R-successor, i.e. it is terminating, then s matches $\circ\varphi$ for any φ. In other words, the pattern $\circ\bot$ is matched by exactly states that are terminating.

We define more dynamic properties as patterns. In the following, φ, φ_1, φ_2, and X have sort *State*.

$$\begin{aligned}
\text{“all-path next”}\quad & \circ\varphi \equiv \neg\bullet\neg\varphi \\
\text{“eventually”}\quad & \diamond\varphi \equiv \mu X\,.\,\varphi \vee \bullet X \\
\text{“always”}\quad & \Box\varphi \equiv \nu X\,.\,\varphi \wedge \circ X \\
\text{“(strong) until”}\quad & \varphi_1 \,\mathrm{U}\, \varphi_2 \equiv \mu X\,.\,\varphi_2 \vee (\varphi_1 \wedge \bullet X) \\
\text{“well-founded”}\quad & \mathsf{WF} \equiv \mu X\,.\circ X \quad // \text{ no infinite paths}
\end{aligned}$$

The following proposition justifies the above definitions.

Proposition 4. *Let $\mathbb{S} = (S, R)$ be a transition system regarded as a Σ^{TS}-model, and let ρ be any valuation and $s \in S$. Then:*

- *$s \in \bar\rho(\bullet\varphi)$ if there exists $t \in S$ such that $s\,R\,t$, $t \in \bar\rho(\varphi)$; in particular, $s \in \bar\rho(\bullet\top)$ if s has an R-successor;*
- *$s \in \bar\rho(\circ\varphi)$ if for all $t \in S$ such that $s\,R\,t$, $t \in \bar\rho(\varphi)$; in particular, $s \in \bar\rho(\circ\bot)$ if s has no R-successor;*
- *$s \in \bar\rho(\diamond\varphi)$ if there exists $t \in S$ such that $s\,R^*\,t$, $t \in \bar\rho(\varphi)$;*
- *$s \in \bar\rho(\Box\varphi)$ if for all $t \in S$ such that $s\,R^*\,t$, $t \in \bar\rho(\varphi)$;*
- *$s \in \bar\rho(\varphi_1\,U\varphi_2)$ if there exists $n \geq 0$ and $t_1, \ldots, t_n \in S$ such that $sRt_1R\cdots Rt_n$, $t_n \in \bar\rho(\varphi_2)$, and $s, t_1, \ldots, t_{n-1} \in \bar\rho(\varphi_1)$;*
- *$s \in \bar\rho(\mathsf{WF})$ if s is R-well-founded, meaning that there is no infinite sequence $t_1, t_2, \dots \in S$ with $s\,R\,t_1\,R\,t_2\,R\ldots$;*

where $R^* = \bigcup_{i\geq 0} R^i$ is the reflexive transitive closure of R.

Modal μ-Logic and Temporal Logics. We have seen that transition systems can be captured in matching logic by the one-path next symbol $\bullet \in \Sigma_{State,State}$. Here, we show that we can define modal μ-logic and various temporal logics such as linear temporal logic (LTL) and computation tree logic (CTL) as matching logic theories, whose axioms constrain the underlying transition relations. The resulting theories are simple, intuitive, and faithfully capture both the syntax (provability) and the semantics of these temporal logics.

We assume readers are familiar with the basic syntax of modal μ-logic and the various temporal logic. The following table summarizes the assumptions that these logics make on the traces of the underlying transition systems, and the corresponding matching logic axioms that capture the assumptions.

Target logic	Assumption on traces	Matching logic axioms
Modal μ-logic	Any traces, no assumptions	No axioms
Infinite-trace LTL	Infinite and linear traces	(INF) + (LIN)
Finite-trace LTL	Finite and linear traces	(FIN) + (LIN)
CTL	Infinite traces	(INF)

Rule	Axiom / Inference
(Propositional Tautology)	φ if φ is a propositional tautology over patterns of the same sort
(Modus Ponens)	$\dfrac{\varphi_1 \quad \varphi_1 \to \varphi_2}{\varphi_2}$
($\exists$-Quantifier)	$\varphi[y/x] \to \exists x \,.\, \varphi$
($\exists$-Generalization)	$\dfrac{\varphi_1 \to \varphi_2}{(\exists x.\varphi_1) \to \varphi_2}$ if $x \notin \mathrm{FV}(\varphi_2)$
(Propagation$_\bot$)	$C_\sigma[\bot] \to \bot$
(Propagation$_\vee$)	$C_\sigma[\varphi_1 \vee \varphi_2] \to C_\sigma[\varphi_1] \vee C_\sigma[\varphi_2]$
(Propagation$_\exists$)	$C_\sigma[\exists x.\varphi] \to \exists x.C_\sigma[\varphi]$ if $x \notin \mathrm{FV}(C_\sigma[\exists x.\varphi])$
(Framing)	$\dfrac{\varphi_1 \to \varphi_2}{C_\sigma[\varphi_1] \to C_\sigma[\varphi_2]}$
(Existence)	$\exists x \,.\, x$
(Singleton Variables)	$\neg(C_1[x \wedge \varphi] \wedge C_2[x \wedge \neg\varphi])$ where C_1 and C_2 are nested symbol contexts.
(Set Variable Substitution)	$\dfrac{\varphi}{\varphi[\psi/X]}$
(Pre-Fixpoint)	$\varphi[\mu X \,.\, \varphi/X] \to \mu X \,.\, \varphi$
(Knaster-Tarski)	$\dfrac{\varphi[\psi/X] \to \psi}{\mu X \,.\, \varphi \to \psi}$

Fig. 4: Matching Logic Proof System

where (Inf) is the pattern/axiom $\bullet\top$ stating that all states are non-terminal states, (Fin) is the pattern/axiom $\mathsf{WF} \equiv \mu X \,.\, \circ X$ stating that all states are well-founded, and (Lin) is the pattern/axiom $\bullet X \to \circ X$ enforcing the linear paths: X holds on one next state implies X holds on all next states.

In conclusion, modal μ-logic is the empty theory over one-path next $\bullet \in \Sigma_{State,State}$ that contains no axioms. Adding (Inf) yields precisely CTL. Adding (Inf) yields precisely infinite-trace LTL and replacing (Inf) with (Fin) yields finite-trace LTL. Therefore, matching logic over the one-path next symbol $\bullet$ gives a playground for defining variants of temporal logics.

It also shows that matching logic can serve as a convenient and uniform framework to define and study temporal logics. For example, finite-trace CTL (which is not shown in the above) can be trivially obtained as the theory containing only the axiom (Fin); LTL with both finite and infinite traces is the theory containing only the axiom (Lin), etc.

Reachability logic (Program verification in 𝕂). We can define reachability properties as patterns using one-path next $\bullet \in \Sigma_{State,State}$. We will discuss it and 𝕂's program verification tools in Section 5.

4.4 Matching Logic Proof System

We have discussed the syntax and semantics of matching logic and have seen many important mathematical instruments as well as other important logics

and models can be defined as theories/notations using patterns. In this section, we discuss the proof system of matching logic; that is, how to carry out formal reasoning in matching logic.

We first need the following definition of contexts.

Definition 6. *A* context C *is a pattern with a distinguished placeholder variable* $\square$. *We write* $C[\varphi]$ *to mean the result of* replacing $\square$ with φ without any α-renaming, *so free variables in* φ *may become bound in* $C[\varphi]$, different *from capture-avoiding substitution. A* single symbol context *has the form*

$$C_\sigma \equiv \sigma(\varphi_1, \ldots, \varphi_{i-1}, \square, \varphi_{i+1}, \ldots, \varphi_n)$$

where $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ *and* $\varphi_1, \ldots, \varphi_{i-1}, \varphi_{i+1}, \ldots, \varphi_n$ *are patterns of appropriate sorts. A* nested symbol context *is inductively defined as follows:*

- $\square$ *is a nested symbol context, called the* identity context*;*
- *if* C_σ *is a single symbol context, and* C *is a nested symbol context, then* $C_\sigma[C[\square]]$ *is a nested symbol context.*

Intuitively, a context C is a nested symbol context iff the path to $\square$ in C contains only symbols and no logic connectives.

Figure 4 shows the Hilbert-style proof system of matching logic. It has four categories of proof rules. The first category (containing the first four rules) consists of all FOL proof rules. This makes the normal FOL reasoning available in matching logic. The second category (containing the next four rules) that supports *framing reasoning*, which allows one to lift the local reasoning in a context (in particular a symbol) to the top level. Separation logic, for example, has a specific framing rule for heap reasoning that allows one to lift the reasoning over a heap fragment to the entire heap. Matching logic, on the other hand, supports generic frame reasoning for all symbols and structures, where heap reasoning is just an special instance. The third category contains two technical proof rules (Existence) and (Singleton Variables) that are needed for certain completeness result (see [12]). The last category contains three proof rules borrowed from modal μ-logic that support fixpoint reasoning. The (Knaster-Tarski) proof rule is a logical incarnation of the Knaster-Tarski fixpoint theorem (Theorem 1) that is the key proof rule for carrying out inductive and co-inductive reasoning; it is known as (Park Induction) in some literature.

Definition 7. *Let* Γ *be a theory and* φ *be a pattern. We write* $\Gamma \vdash \varphi$ *if* φ *can be proved by the proof system shown in Figure 4 with patterns in* Γ *regarded as additional axioms.*

As we have seen earlier, matching logic can capture precisely inductive data structures. As a consequence, matching logic can capture precisely natural numbers (which are inductive data structures built from two constructors *zero* and *succ*) and define the addition and multiplication of natural numbers using pattern axioms in the usual way. Therefore, the proof system of matching logic cannot be both sound and complete for all theories. Some completeness results

have been shown for some theories or fragments of matching logic in [12]. In the following, we only state the soundness theorem of the matching logic proof system.

Theorem 2 (Soundness). *$\Gamma \vdash \varphi$ implies $\Gamma \vDash \varphi$.*

Since the proof system of matching logic contains the normal proof rules for FOL reasoning, framing reasoning, and fixpoint reasoning as in modal μ-logic, all these reasonings are sound and available in matching logic, too.

5 Program Verification in $\mathbb{K}$

Here we discuss the logical foundations and the $\mathbb{K}$ tools for program verification. We first review the classic approaches to program verification in Section 5.1 and then show the $\mathbb{K}$ tools for program verification in Section 5.2. Finally, we discuss the logical foundations of $\mathbb{K}$'s verification tools, reachability logic, and show that is can be defined in matching logic in Section 5.3.

5.1 Classic Approaches to Program Verification

Program verification is a decision problem that asks if a given program satisfies a given specification. If so, a collection of proof objects is expected to be generated as evidence. If not, counterexamples are given often in the form of concrete program execution traces that violate the specification.

Hoare-Style Program Verification. Hoare-style program verification refers to the program verification approaches where the formal semantics of a programming language is given as a *program logic*, which has several proof rules that are specific to the constructs of that language. The program logic, which is often called an axiomatic semantics or the Hoare logic of the language, derives sentences called *Hoare triples* that have the form $\{\varphi\}P\{\varphi'\}$ where P is the program, φ is a logic formula called the *pre-condition* of the triple, and φ' is a formula called the *post-condition*. The semantics of the Hoare triple is that if P is executed on a state satisfying the pre-condition φ, and if P terminates on a final state, then the final state satisfies the post-condition φ'. The requirement that P terminates implies that the Hoare triple unconditionally holds if P does not terminate. This is known as *partial correctness* in literature.

Hoare logic remains one of the most popular program logics since the day it was born. Obviously, Hoare logic is a *language-specific* logic because different languages must have their own variants of Hoare logic. This makes the development of verification tools based on Hoare logic difficult to adapt to language changes. Such inconvenience is being made worse when it comes to blockchain languages that have a rapid development cycle with new versions of the languages being released on a weekly basis.

Another notable characteristic of Hoare logic is that it is not directly executable. This makes it difficult to test Hoare logic semantics. In practice, language semanticists may need to define a separate trusted operational semantics that is executable, and carry out complex proofs of equivalence between the two semantics, which can take years to complete.

All the above makes language design with Hoare logic a highly expensive task, and *changing* the language rather inconvenient and demotivating, as it requires a thorough change of the Hoare logic proof system for that language and thus of all the related verification tools. If a trusted operational semantics is given, it needs to change, too, and a new proof of equivalence between the new Hoare logic and the new operational semantics should be carried out. This high cost brings us poor *reusability* of verification tools. Considering the fact that these tools often need several man-years to develop, the lack of reusability leads to a remarkable waste of resources and talent, as well as to duplicate work.

In $\mathbb{K}$, such drawbacks are overcome by using only one *language-independent* proof system to verify any programs written in any programming languages, given that the formal language definitions are given in $\mathbb{K}$. We will explain it in detail in Section 5.2.

Intermediate Verification Languages. A common alternative practice to Hoare-style verification is to design *intermediate verification languages* (IVL) such as Boogie [6] and Why [20], to develop verification tools for these IVL languages, and to translate the target languages to IVL. This brings some reusability, as verification tools are designed and implemented for IVL, in isolation from the target languages. However, correct program translation can be hard to develop. The proof of its correctness (called *soundness proof*) often involves the usage of higher-order theorem provers such as Coq [26] and Isabelle [38], not to mention that many real languages such as Java do not even have an official formal specification of the semantics. Thus, research about language-specific program logics and IVL tools sometimes have to compromise and claim "no intention of formally proving the soundness result" [4].

5.2 Program Verification by Reachability Logic

$\mathbb{K}$'s program verification tools are based on *reachability logic* [33], which has been shown to be a fragment of matching logic in [12]. One appealing aspect of reachability logic is that it is *language independent*, that is, it uses one fixed proof system to reason about any programs written in any programming languages, given that their formal semantics have been defined in $\mathbb{K}$. Some selected proof rules of reachability logic are shown in Fig. 5. The proof system derives judgments of the form $A \vdash_C \varphi_1 \Rightarrow \varphi_2$, where $\varphi_1 \Rightarrow \varphi_2$ is a *reachability rule* that specifies that any configurations matching φ_1 will eventually reach a configuration matching φ_2, on termination. (Readers who are more familiar with the traditional Hoare-style verification can intuitively regard φ_1 as the *pre-condition* and φ_2 as the *post-condition*). A and C are two sets of reachability rules, where

$$\text{(Axiom)}\quad \frac{\varphi_1 \Rightarrow \varphi_2 \in A}{A \vdash_C \varphi_1 \Rightarrow \varphi_2}$$

$$\text{(Transitivity)}\quad \frac{A \vdash_C \varphi_1 \Rightarrow \varphi_2 \quad A \cup C \vdash \varphi_2 \Rightarrow \varphi_3}{A \vdash_C \varphi_1 \Rightarrow \varphi_3}$$

$$\text{(Consequence)}\quad \frac{M^{\mathsf{cfg}} \vDash \varphi_1 \rightarrow \varphi_1' \quad A \vdash_C \varphi_1' \Rightarrow \varphi_2' \quad M^{\mathsf{cfg}} \vDash \varphi_2' \rightarrow \varphi_2}{A \vdash_C \varphi_1 \Rightarrow \varphi_2}$$

$$\text{(Circularity)}\quad \frac{A \vdash_{C \cup \{\varphi_1 \Rightarrow \varphi_2\}} \varphi_1 \Rightarrow \varphi_2}{A \vdash_C \varphi_1 \Rightarrow \varphi_2}$$

Fig. 5: Some selected proof rules in the proof system of reachability logic

rules in A are considered as *axioms* and can be directly used to discharge the proof obligations, rules in C are called *circularities* and cannot be directly used. The distinguished proof rule (Circularity) adds the current proof obligation to the circularity set, which is then flushed to the axiom set by (Transitivity). In other words, circularities become axioms after making any progress on program execution.

We use the following `sum` program as an example to illustrate program verification in 𝕂 by reachability logic.

```
int n, sum;
n = N;
sum = 0;
while (!(n <= 0)) {
  sum = sum + n;
  n = n + -1;
}
```

We will use 𝕂's generic program verification tool to prove that the above `sum` program correctly computes the total of `1` to `N`, where `N` is a symbolic value denoting any natural number.

The first step to verify `sum` using reachability logic and 𝕂 is to formally define the specifications as 𝕂's rewrite rules.

```
module SUM_SPEC
  imports IMP

  rule     // invariant spec
      <k> while(n){ s = s + n; n = n - 1; } => .K ... </k>
      <state>
        n |-> (N:Int => 0)
        s |-> (S:Int => S +Int ((N +Int 1) *Int N /Int 2)
      </state>
  requires N >=Int 0

  rule     // main spec
```

```
      <k> int n, s; n = N:Int; while(n){ s = s + n; n = n - 1; }
          => .K
      </k>
      <state> .Map =>
        n |-> 0
        s |-> ((N +Int 1) *Int N /Int 2)
      </state>
  requires N >=Int 0
endmodule
```

The above specification contains two sub-specifications as reachability claims. The first is the invariant reachability claim that specifies the behavior of the while-loop. It is provided as a *lemma* to prove the main claim. The second claim is the main verification claim. It specifies that if the `sum` program (where `n` is now initialized to a symbolic value n, written as a $\mathbb{K}$ variable `N:Int`) terminates, then the final value of `s` equals $n(n+1)/2$. The condition after the keyword `requires` has the similar meaning of a pre-condition in Hoare logic. It asks $\mathbb{K}$ to prove the mentioned reachability claim given that $n \geq 0$.

Then, $\mathbb{K}$ proves the claims via *circular proofs*, based on reachability logic proof system (see Figure 5). We take the proof of the invariant claim as an example. We put the formal proof in Fig. 6 and explain it in the following. $\mathbb{K}$ starts with a configuration with a while-loop in the `<k/>` cell and a state that maps `n` to n and `s` mapping to s, as required by the left-hand side of the claim. Then, $\mathbb{K}$ rewrites the configuration *symbolically* using exactly the same rewrite rules used to execute IMP programs. After the rewrites, the while-loop is desugared to an if-statement and the two assignments are resolved accordingly. After that, $\mathbb{K}$ reaches a configuration with the same while-loop in `<k/>` cell, but in the `<state/>` cell, `n` maps to $n-1$ and `s` maps to $s+n$. For clarity, let us denote *that* configuration as γ and let $n' = n-1$ and $s' = s+n$. At this point, the (Circularity) proof rule of the reachability logic proof system (see, Figure 5) is applied, and the invariant claim itself becomes a regular *axiom* which can be used in further proofs. Therefore, we can *instantiate* the variables n and s in the invariant claim by n' and s', yielding exactly the configuration γ, and the invariant claim immediately tells us that γ will terminate at a state where `n` maps to 0 and `s` maps to $s' + n'(n'+1)/s$. And this tells us that the initial configuration, with `n` mapping to n and `s` mapping to s, can reach γ and then terminate at the same state. Finally, $\mathbb{K}$ calls SMT solvers (such as Z3 [17]) to prove that $s'+n'(n'+1)/2 = s+n(n+1)/2$, and concludes the proof successfully.

5.3 Reachability Logic is a Fragment of Matching Logic

We have seen a program verification example using reachability logic. In this section, we show that we can faithfully capture reachability logic in matching logic, and all reachbaility logic reasoning, including the key proof rule (Circularity), can be derived by the matching logic proof system. In other words, reachability logic is the fragment of matching logic for program verification.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\cdot}{(S+N)+((N-1)+1)(N-1)/2 = S+(N+1)N/2}}{A \cup C \vdash_\emptyset \langle \texttt{WHILE} \rangle_{\mathsf{k}} \langle \texttt{N} \mapsto N-1, \texttt{S} \mapsto S+N \rangle_{\mathsf{state}} \Rightarrow \langle \texttt{.K} \rangle_{\mathsf{k}} \langle \texttt{N} \mapsto 0, \texttt{S} \mapsto S+(N+1)N/2 \rangle_{\mathsf{state}}}}{A \cup C \vdash_\emptyset \langle \texttt{n=n-1;WHILE} \rangle_{\mathsf{k}} \langle \texttt{N} \mapsto N, \texttt{S} \mapsto S+N \rangle_{\mathsf{state}} \Rightarrow \langle \texttt{.K} \rangle_{\mathsf{k}} \langle \texttt{N} \mapsto 0, \texttt{S} \mapsto S+(N+1)N/2 \rangle_{\mathsf{state}}}}{A \cup C \vdash_\emptyset \langle \texttt{s=s+n;n=n-1;WHILE} \rangle_{\mathsf{k}} \langle \texttt{N} \mapsto N, \texttt{S} \mapsto S \rangle_{\mathsf{state}} \Rightarrow \langle \texttt{.K} \rangle_{\mathsf{k}} \langle \texttt{N} \mapsto 0, \texttt{S} \mapsto S+(N+1)N/2 \rangle_{\mathsf{state}}}}{A \cup C \vdash_\emptyset \langle \texttt{IF} \rangle_{\mathsf{k}} \langle \texttt{N} \mapsto N, \texttt{S} \mapsto S \rangle_{\mathsf{state}} \Rightarrow \langle \texttt{.K} \rangle_{\mathsf{k}} \langle \texttt{N} \mapsto 0, \texttt{S} \mapsto S+(N+1)N/2 \rangle_{\mathsf{state}}}}{A \vdash_C \langle \texttt{WHILE} \rangle_{\mathsf{k}} \langle \texttt{N} \mapsto N, \texttt{S} \mapsto S \rangle_{\mathsf{state}} \Rightarrow \langle \texttt{.K} \rangle_{\mathsf{k}} \langle \texttt{N} \mapsto 0, \texttt{S} \mapsto S+(N+1)N/2 \rangle_{\mathsf{state}}}}{A \vdash_\emptyset \langle \texttt{WHILE} \rangle_{\mathsf{k}} \langle \texttt{N} \mapsto N, \texttt{S} \mapsto S \rangle_{\mathsf{state}} \Rightarrow \langle \texttt{.K} \rangle_{\mathsf{k}} \langle \texttt{N} \mapsto 0, \texttt{S} \mapsto S+(N+1)N/2 \rangle_{\mathsf{state}}}$$

where $\texttt{WHILE} \equiv$ `while(n){s=s+n; n=n-1;}` and $\texttt{IF} \equiv$ `IF(n){s=s+n; n=n-1; WHILE}{}`. We use A to denote the axiom set that contains all semantic rules of IMP and let $C = \{\langle \texttt{WHILE} \rangle_{\mathsf{k}} \langle \texttt{N} \mapsto N, \texttt{S} \mapsto S \rangle_{\mathsf{state}} \Rightarrow \langle \texttt{.K} \rangle_{\mathsf{k}} \langle \texttt{N} \mapsto 0, \texttt{S} \mapsto S+(N+1)N/2 \rangle_{\mathsf{state}}\}$ contain the original invariant proof goal, which is added to C by (Circularity) in the first proof step and moved to A by (Transitivity) in the second proof step. This circularity pattern is then used in the second to last proof step, where we instantiate N by $N-1$ and S by $S+N$. The last proof step is done by calling external SMT solvers such as Z3 [17].

Fig. 6: Reachability logic proof of the invariant of sum program.

Reachability Logic Preliminaries. Reachability logic is a "top-most" logic that builts on top of matching logic (without μ). Reachability logic is parametric in a model of configurtaions. Specifically, fix a signature (of static program configurations) Σ^{cfg} which may have various sorts and symbols, among which there is a distinguished sort *Cfg*. A model of signature Σ^{cfg}, denoted M^{cfg}, is called the configuration model where M^{cfg}_{Cfg} is the set of all configurations. Reachability logic formulas are called *reachability rules* of the form $\varphi_1 \Rightarrow \varphi_2$ where φ_1, φ_2 are matching logic patterns matched by the (static) program configurations. A *reachability system* S is a finite set of rules, which yields a transition system $\mathbb{S} = (M^{\mathsf{cfg}}_{Cfg}, R)$ where $s\,R\,t$ iff there exist a rule $\varphi_1 \Rightarrow \varphi_2 \in S$ and an M^{cfg}-valuation ρ such that $s \in \bar{\rho}(\varphi_1)$ and $t \in \bar{\rho}(\varphi_2)$. A rule $\psi_1 \Rightarrow \psi_2$ is S-*valid*, denoted $S \vDash_{\mathsf{RL}} \psi_1 \Rightarrow \psi_2$, iff for all M^{cfg}_{Cfg}-valuations ρ and configurations $s \in \bar{\rho}(\psi_1)$, either there is an infinite trace $s\,R\,t_1\,R\,t_2\,R \ldots$ in $\mathbb{S}$ or there is a configuration t such that $s\,R^*\,r$ and $t \in \bar{\rho}(\psi_2)$. Therefore, validity in RL is defined in the spirit of *partial correctness*.

The reachability logic proof system (Figure 5) derives *reachability logic sequents* of the form $A \vdash_C \varphi_1 \Rightarrow \varphi_2$ where A (called *axioms*) and C (called *circularities*) are finite sets of rules. Initially, we start with $A = S$ and $C = \emptyset$. As the proof proceeds, more rules can be added to C via (Circularity) and then moved to A via (Transitivity), which can then be used via (Axiom). We write $S \vdash_{\mathsf{RL}} \psi_1 \Rightarrow \psi_2$ to mean that $S \vdash_\emptyset \psi_1 \Rightarrow \psi_2$. Notice (Consequence) consults the configuration model M^{cfg} for validity, so the completeness result is *relative to* M^{cfg}. We recall the following result [33] that shows that program verification with reachability logic is relative complete to the reason about the static program configurations.

Theorem 3. *For all reachability systems S satisfying some reasonable technical assumptions (see [33]) and all rules $\psi_1 \Rightarrow \psi_2$, we have $S \vDash_{\mathsf{RL}} \psi_1 \Rightarrow \psi_2$ iff $S \vdash_{\mathsf{RL}} \psi_1 \Rightarrow \psi_2$.*

Defining Reachability Logic in Matching Logic. As how we define modal μ-logic and the various temporal logics, we can faithfully define reachability logic in matching logic using the one-path next $\bullet \in \Sigma_{Cfg,Cfg}$ that captures the underlying transition relation defined by the rewrite rules. Specifically, we define the following reachability properties as patterns:

$$\text{“weak eventually”} \quad \diamond_w \varphi \equiv \nu X.\, \varphi \vee \bullet X \quad // \text{ equal to } \neg\mathsf{WF} \vee \diamond\varphi$$
$$\text{“reaching star”} \quad \varphi_1 \Rightarrow^* \varphi_2 \equiv \varphi_1 \to \diamond_w \varphi_2$$
$$\text{“reaching plus”} \quad \varphi_1 \Rightarrow^+ \varphi_2 \equiv \varphi_1 \to \bullet \diamond_w \varphi_2$$

Notice that the “weak eventually” $\diamond_w\varphi$ is defined similarly to the “eventually” $\diamond\varphi \equiv \mu X\,.\,\varphi \vee \bullet X$, but instead of using least fixpoint μ-binder, we define it as a greatest fixpoint. One can prove that $\diamond_w\varphi = \neg\mathsf{WF} \vee \diamond\varphi$, that is, a configuration γ satisfies $\diamond_w\varphi$ if either it satisfies $\diamond\varphi$, or it is not well-founded, meaning that there exists an infinite execution path from γ. Also notice that “reaching plus” $\varphi_1 \Rightarrow^+ \varphi_2$ is a stronger version of “reaching star”, requiring that $\diamond_w\varphi_2$ should hold *after at least one step*. This *progressive condition* is crucial to the soundness of RL reasoning: as shown in (Transitivity), circularities are flushed into the axiom set only *after one reachability step is established*. This leads us to the following translation from RL sequents to MmL patterns.

Definition 8. *Given a rule $\varphi_1 \Rightarrow \varphi_2$, define the MmL pattern $\boxdot(\varphi_1 \Rightarrow \varphi_2) \equiv \Box(\varphi_1 \Rightarrow^+ \varphi_2)$ and extend it to a rule set A as follows: $\boxdot A \equiv \bigwedge_{\varphi_1 \Rightarrow \varphi_2 \in A} \boxdot(\varphi_1 \Rightarrow \varphi_2)$. Define the translation RL2ML from RL sequents to MmL patterns as follows:*

$$RL2ML(A \vdash_C \varphi_1 \Rightarrow \varphi_2) = (\forall \boxdot A) \wedge (\forall \circ \boxdot C) \to (\varphi_1 \Rightarrow^\star \varphi_2)$$

where $\star = *$ if C is empty and $\star = +$ if C is nonempty. We use $\forall\varphi$ as a shorthand for $\forall\vec{x}\,.\,\varphi$ where $\vec{x} = \mathrm{FV}(\varphi)$. Recall that the “$\circ$” in $\forall\circ\boxdot C$ is “all-path next”.

Hence, the translation of $A \vdash_C \varphi_1 \Rightarrow \varphi_2$ depends on whether C is empty or not. When C is nonempty, the RL sequent is *stronger* in that it requires *at least one step* being made in $\varphi_1 \Rightarrow \varphi_2$. Axioms (those in A) are also *stronger* than circularities (those in C) in that axioms *always* hold, while circularities only hold *after at least one step* because of the leading all-path next “$\circ$”; and since the “next” is an “all-path” one, it does not matter which step is actually made, as circularities hold on *all* next states.

Theorem 4. *Let Γ^{RL} be the set of all matching logic patterns of sort Cfg that hold in the configuration model M^{cfg}. For all reachability systems S and rules $\varphi_1 \Rightarrow \varphi_2$ satisfying the same technical assumptions in [33], the following are equivalent: (1) $S \vdash_{\mathsf{RL}} \varphi_1 \Rightarrow \varphi_2$; (2) $S \vDash_{\mathsf{RL}} \varphi_1 \Rightarrow \varphi_2$; (3) $\Gamma^{\mathsf{RL}} \vdash RL2ML(S \vdash_\emptyset \varphi_1 \Rightarrow \varphi_2)$; (4) $\Gamma^{\mathsf{RL}} \vDash RL2ML(S \vdash_\emptyset \varphi_1 \Rightarrow \varphi_2)$.*

Therefore, given an oracle for validity of matching logic patterns in the configuration model M^{cfg}, the matching logic proof system is capable of deriving any reachability property that can be derived with the reachability logic proof system. This result makes matching logic an even more fundamental logic foundation for the 𝕂 framework and thus for programming language specification and verification than reachability logic, because it can express significantly more properties than partial correctness reachability.

6 Conclusion

We have discussed the ideal language framework vision pursued by 𝕂, where programming languages must have formal definitions and language tools are automatically generated by the framework from the definitions at no additional costs. Then, we presented two example languages, LAMBDA and IMP, to illustrate the basic features and functionality of the 𝕂 tools, such as parsers, interpreters, etc. Next, we presented in detail the foundational logic of the 𝕂 framework, matching logic, and showed that many important mathematical instruments as well as other common logics and/or models can be faithfully captured in matching logic. Finally, we discussed the language-independent program verification tools of 𝕂 and showed that its logical foundation, reachability logic, can also be faithfully captured by matching logic.

References

1. Clang: A C language family frontend for LLVM, `https://clang.llvm.org/`
2. GCC, the GNU compiler collection, `https://gcc.gnu.org/`
3. TrustInSoft—Cybersecurity and safety provider, `https://trust-in-soft.com/`
4. Ahrendt, W., Beckert, B., Bubel, R., Hahnle, R., H. Schmitt, P., Ulbrich, M.: Deductive Software Verification—The KeY Book. Springer (2016)
5. Barendregt, H.: The lambda calculus: Its syntax and semantics. Studies in Logic and the Foundations of Mathematics, Elsevier Science Publishers (1984)
6. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: Revised Lectures of the 4$^{\text{th}}$ International Symposium on Formal Methods for Components and Objects (FMCO'05). vol. 4111, pp. 364–387. Springer (2006)
7. Bertot, Y., Castran, P.: Interactive theorem proving and program development: Coq'Art the calculus of inductive constructions. Springer (2010)
8. Bogdănaş, D., Roşu, G.: K-Java: A Complete Semantics of Java. In: Proceedings of the 42$^{\text{nd}}$ Symposium on Principles of Programming Languages (POPL'15). pp. 445–456. ACM (Jan 2015). https://doi.org/http://dx.doi.org/10.1145/2676726.2676982
9. Bogdănaş, D., Roşu, G.: K-Java: A complete semantics of Java. In: Proceedings of the 42$^{\text{nd}}$ Symposium on Principles of Programming Languages (POPL'15). pp. 445–456. ACM (2015). https://doi.org/10.1145/2676726.2676982
10. Brady, E.: IDRIS — Systems programming meets full dependent types. In: Proceedings of the 5$^{\text{th}}$ ACM Workshop on Programming Languages Meets Program Verification (PLPV '11). pp. 43–54. ACM (2011). https://doi.org/10.1145/1929529.1929536

11. Chen, X., Roşu, G.: Applicative matching logic. Tech. Rep. http://hdl.handle.net/2142/104616, University of Illinois at Urbana-Champaign (Jul 2019)
12. Chen, X., Roşu, G.: Matching μ-logic. In: Proceedings of the 34$^{\text{th}}$ Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'19) (2019)
13. Chen, X., Roşu, G.: A language-independent program verification framework. In: Proceedings of the 8$^{\text{th}}$ International Symposium on Leveraging Applications of Formal Methods (ISoLA'18). vol. 11245, pp. 92–102. Springer (2018). https://doi.org/10.1007/978-3-030-03421-4
14. Church, A.: The calculi of lambda-conversion. Princeton University Press (1941)
15. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Theorem Proving in Higher Order Logics. pp. 23–42. Springer (2009)
16. Dasgupta, S., Park, D., Kasampalis, T., Adve, V.S., Roşu, G.: A complete formal semantics of x86-64 user-level instruction set architecture. In: Proceedings of the 40$^{\text{th}}$ ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19). pp. 1133–1148. ACM (Jun 2019). https://doi.org/http://dx.doi.org/10.1145/3314221.3314601
17. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proceedings of the 14$^{\text{th}}$ International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08). pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
18. Ethereum: Solidity documentation (2019), `http://solidity.readthedocs.io`
19. Ethereum: Vyper documentation (2019), `https://vyper.readthedocs.io`
20. Filliâtre, J., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Proceedings of the 19$^{\text{th}}$ International Conference on Computer Aided Verification (CAV'07). vol. 4590, pp. 173–177. Springer (2007)
21. Guth, D.: A formal semantics of Python 3.3. Master's thesis, University of Illinois at Urbana-Champaign (2013), `http://hdl.handle.net/2142/45275`
22. Hathhorn, C., Ellison, C., Roşu, G.: Defining the undefinedness of c. In: Proceedings of the 36$^{\text{th}}$ ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15). pp. 336–345. ACM (Jun 2015). https://doi.org/http://dx.doi.org/10.1145/2813885.2737979
23. Hildenbrandt, E., Saxena, M., Zhu, X., Rodrigues, N., Daian, P., Guth, D., Moore, B., Zhang, Y., Park, D., Ştefănescu, A., Roşu, G.: KEVM: A complete semantics of the Ethereum virtual machine. In: Proceedings of the 2018 IEEE Computer Security Foundations Symposium (CSF'18). IEEE (2018), `http://jellopaper.org`
24. Kasampalis, T., Guth, D., Moore, B., Şerbănuţă, T.F., Zhang, Y., Filaretti, D., Şerbănuţă, V., Johnson, R., Roşu, G.: IELE: A rigorously designed language and tool ecosystem for the blockchain. In: Proceeding of the 23$^{\text{rd}}$ International Symposium on Formal Methods (FM'19) (2019)
25. Kroening, D., Tautschnig, M.: CBMC—C bounded model checker. In: Proceedings of the 20$^{\text{th}}$ International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14). pp. 389–391. Springer (2014)
26. The Coq development team: The Coq proof assistant reference manual. LogiCal Project (2004), `http://coq.inria.fr`
27. Norell, U.: Dependently typed programming in Agda. In: Proceedings of the 6$^{\text{th}}$ International Conference on Advanced Functional Programming (AFP'09). pp. 230–266. Springer (2009)

28. Park, D., Ştefănescu, A., Roşu, G.: KJS: A complete formal semantics of JavaScript. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15). pp. 346–356. ACM (Jun 2015). https://doi.org/http://dx.doi.org/10.1145/2737924.2737991
29. Park, D., Ştefănescu, A., Roşu, G.: KJS: A complete formal semantics of JavaScript. In: Proceedings of the 36th annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15). pp. 346–356. ACM (2015). https://doi.org/10.1145/2737924.2737991
30. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02). pp. 55–74. IEEE (2002). https://doi.org/10.1109/lics.2002.1029817
31. Roşu, G.: Matching logic. Logical Methods in Computer Science **13**(4), 1–61 (2017). https://doi.org/10.23638/lmcs-13(4:28)2017
32. Roşu, G., Ştefănescu, A.: Checking reachability using matching logic. In: Proceedings of the 27th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12). pp. 555–574. ACM (Oct 2012). https://doi.org/http://dl.acm.org/citation.cfm?doid=2384616.2384656
33. Roşu, G., Ştefănescu, A., Ciobâcă, Ş., Moore, B.M.: One-path reachability logic. In: Proceedings of the 28th Symposium on Logic in Computer Science (LICS'13). pp. 358–367. IEEE (2013). https://doi.org/10.1109/lics.2013.42
34. Rosu, G.: K—A semantic framework for programming languages and formal analysis tools. In: Dependable Software Systems Engineering. IOS Press (2017)
35. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. Journal of Logic and Algebraic Programming **79**(6), 397–434 (2010). https://doi.org/10.1016/j.jlap.2010.03.012
36. Rosu, G., Serbanuta, T.F.: K overview and simple case study. In: Proceedings of International K Workshop (K'11). ENTCS, vol. 304, pp. 3–56. Elsevier (Jun 2014). https://doi.org/http://dx.doi.org/10.1016/j.entcs.2014.05.002
37. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics **5**(2), 285–309 (1955). https://doi.org/10.2140/pjm.1955.5.285
38. The Isabelle development team: Isabelle (2018), `https://isabelle.in.tum.de/`