$\mathbb{K}$ SEMANTICS FOR ABSTRACTIONS

IRINA MĂRIUCA ASĂVOAE

A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

FACULTY OF COMPUTER SCIENCE
UNIVERSITY ALEXANDRU IOAN CUZA, IAŞI

2012

DECLARATION

No portion of the work referred to in this thesis has been submitted in support of an application for another degree of qualification at this or any other university or institution of learning.

Abstract

This dissertation shows how abstract interpretation's view of program analysis and verification can be instilled in the $\mathbb{K}$ framework in a generic fashion as reflective semantics.

$\mathbb{K}$ is a rewriting-based framework dedicated to defining executable specifications for programming languages semantics. The definitional style proposed by the $\mathbb{K}$ framework is an amalgamation of features from different semantics (e.g., features from operational semantics, continuation based semantics, denotational semantics, a.s.o.). The current $\mathbb{K}$'s desideratum is to demonstrate that its formal specification environment can be effectively used for various classes of programming languages paradigms.

Abstract interpretation provides a well known, standardized and extensively used framework for program analysis and verification. The main idea in abstract interpretation is that program analysis and verification can be achieved by applying fixpoint iterators over sound approximations of program semantics. Solely from a semantics perspective, abstract interpretation is a *reflective* semantics environment where the reflexion of the semantics is called *abstraction*.

The program analysis and verification approaches tackled in $\mathbb{K}$ are either model checking, achieved via Maude's LTL model checker, or deductive verification, achieved via matching logic. However, the third major verification technique, namely abstract interpretation, was not systematically approached in $\mathbb{K}$. Our thesis addresses this omission and covers it. As such, we design a generic method for defining in $\mathbb{K}$ abstract specifications of pushdown systems and fixpoint iterators over these specifications. We demonstrate the efficiency of this design by instantiating it with three case studies of abstractions for data analysis, alias analysis, and shape analysis.

One of the main things about teaching is not what you say but what you don't say. When you hear someone play, you have to work out the way they do things naturally and then leave them alone, because you want the naturalness to be there still.

— Itzhak Perlman. Teaching the Teachers, Strad.

List of Publications

Revised publications directly related with the dissertation

1. Irina Măriuca Asăvoae, and Mihail Asăvoae:
   Collecting Semantics under Predicate Abstraction in the $\mathbb{K}$ Framework,
   Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010, Held as a Satellite Event of ETAPS 2010, Paphos, Cyprus, March 20-21, 2010, Revised Selected Papers, editor P.C. Olveczky, volume 6381 of Lecture Notes in Computer Science, pages 123–139, Springer, 2010,
   ISBN 978–3–642–16309–8.

2. Irina Măriuca Asăvoae, Mihail Asăvoae, and Dorel Lucanu:
   Path Directed Symbolic Execution in the $\mathbb{K}$ Framework
   12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2010, Timisoara, Romania, 23-26 September 2010, editors Tetsuo Ida, Viorel Negru, Tudor Jebelean, Dana Petcu, Stephen M. Watt and Daniela Zaharie, pages 133–141, IEEE Computer Society, 2010,
   ISBN 978–0–7695–4324–6.

3. Irina Măriuca Asăvoae:
   Abstract Semantics for Alias Analysis in $\mathbb{K}$
   Submitted at K 2011.

4. Jurriaan Rot, Irina Măriuca Asăvoae, Frank de Boer, Marcello Bonsangue, and Dorel Lucanu:
   Interacting via the Heap in the Presence of Recursion
   Submitted at ICE 2012.

Technical reports and extended abstracts directly related with the dissertation

1. Irina Măriuca Asăvoae, Frank de Boer, Marcello Bonsangue, Dorel Lucanu, and Jurriaan Rot:
   Bounded Model Checking of Recursive Programs with Pointers in $\mathbb{K}$
   Extended abstract in pre-proceedings of 9th International Workshop on Rewriting Logic and its Applications, WRLA 2012, Tallinn, Estonia, 24-25 March 2012, editor Francisco Durán, pages 26–28,
   `http://wrla2012.lcc.uma.es/wrla/index.html`.

2. Irina Măriuca Asăvoae, Frank de Boer, Marcello Bonsangue, Dorel Lucanu, and Jurriaan Rot:
   Model Checking Programs with Dynamic Linked Structures
   LIACS Technical Report 2012-02, LIACS, Universiteit Leiden, 2012.

3. Irina Măriuca Asăvoae, Frank de Boer, Marcello Bonsangue, Dorel Lucanu, and Jurriaan Rot:
   Bounded Model Checking of Recursive Programs with Pointers in $\mathbb{K}$
   Extended abstract in pre-proceedings of WADT 2012,
   Technical Report TR–08/12, Universidad Complutense de Madrid, Departamento de Sistemas Informáticos y Computación, pages 12–15,
   `http://maude.sip.ucm.es/wadt2012/docs/WADT2012-preproceedings.pdf`

4. Irina Măriuca Asăvoae:
   Systematic Design of Abstractions in $\mathbb{K}$
   Extended abstract in pre-proceedings of WADT 2012,
   Technical Report TR–08/12, Universidad Complutense de Madrid, Departamento de Sistemas Informáticos y Computación, pages 9–11,
   `http://maude.sip.ucm.es/wadt2012/docs/WADT2012-preproceedings.pdf`

Publications co-related with the dissertation

Revised papers:

1. Mihail Asăvoae, and Irina Măriuca Asăvoae:
   Using the Executable Semantics for CFG Extraction and Unfolding
   13th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2011, Timisoara, Romania, 26-29 September 2011, editors Dongming Wang, Viorel Negru, Tetsuo Ida, Tudor Jebelean, Dana Petcu, Stephen M. Watt and Daniela Zaharie, pages 123–127, IEEE Computer Society, 2011,
   ISBN 978–1–4673–0207–4.

2. Mihail Asăvoae, Irina Măriuca Asăvoae, and Dorel Lucanu:
   On Abstractions for Timing Analysis in the $\mathbb{K}$ Framework
   Second International Workshop on Foundational and Practical Aspects of Resource Analysis, FOPARA 2011, Madrid, Spain, May 2011, Revised selected papers, editors Ricardo Peña, Marko van Eekelen, and Olha Shkaravska, volume 7177 of Lecture Notes in Computer Science pages 90–107, Springer, 2011,
   ISBN 978–3–642–32494–9.

Others:

1. Adrián Riesco, Irina Măriuca Asăvoae, and Mihail Asăvoae:
   A Generic Program Slicing Technique based on Language Definitions
   Extended abstract in pre-proceedings of WADT 2012,
   Technical Report TR–08/12, Universidad Complutense de Madrid, Departamento de Sistemas Informáticos y Computación, pages 91–92,
   `http://maude.sip.ucm.es/wadt2012/docs/WADT2012-preproceedings.pdf`

Acknowledgements

To be completed.

This work was supported by the the European Social Fund in Romania, under the responsibility of the Managing Authority for the Sectoral Operational Programme for Human Resources Development 2007-2013 [grant POSDRU/88/1.5/S/47646] and by Contract ANCS POS CCE, O2.1.2, ID nr. 602/12516, ctr.nr 161/15.06.2010, DAK.

Table of Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

Motto: *"In the beginning was the Word,*
and the Word was with God,
and the Word was God."
— John 1:1, The Holy Bible, New International Version Translation.

Along the time, languages, either natural or artificial, were used for communication. The basic principle of language functionality is the association of words with meanings. Languages use composition principles to combine words in order to obtain new, more complex meanings. Hence, languages induce communication by connecting appearance and essence at different levels of complexity. In language theory, the notion of appearance is called *syntax* while the notion of essence in called *semantics*.

In languages there exists a strong property of reflection, inherited through the composition principles. This reflection produced in humans a compelling fascination towards understanding languages in their depth. As such, the languages apprehension and comprehension may very well be the roots of philosophy.

Programming languages are artifacts used as means to communicate with other artifacts, namely the computer machines. As all languages, programming languages inherited the same structural components, namely the syntax and semantics. Programming

language syntax was studied and standardized in early stages in computer science as Chomsky hierarchy [17]. Meanwhile, programming language semantics is a more extensive, still effervescent research area.

A first classification of programming language semantics distinguishes between static semantics and dynamic semantics. Static semantics essentially define restrictions concerning the structure of the texts (programs) written using the syntax of a programming language. For example, in some cases, a restriction could be that a variable must be assigned before it is used. Dynamic semantics, on the other hand, define actual executions of programs on some machine. For example, the semantics may define which are the changes triggered on the machine upon a variable assignment.

Until reaching the machine level of understanding, programs go through various stages of automatic transformations. The entire process of transformation of the "source" program into the "executable" program is called compilation. The classification of semantics in static and dynamic concerns two different stages of programming languages representations: the compilation stage for static semantics and the execution stage for dynamic semantics. Hence, static semantics characterize (parts of) the compilation process while dynamic semantics characterize (parts of) the execution process.

In the current work we are interested in an amalgamation of the two semantics, namely how to obtain static semantics for a programming language via dynamic semantics of some intermediate representation of that programming language. As an element of uniformity, we consider that the intermediate representation of the programming language is also a programming language.

1.1 Motivation

In order to discuss about a mixture of static and dynamic semantics, we first present the environment which uses these semantics. In this way, we lay foundation for the current

work and create the premises to justify it.

Static semantics derives program properties merely only from the program structure. The outcome of static semantics is used for static analysis methods. These methods are applied in the program transformations that a compiler has to create in order to translate the program from the programming language understood by the programmer into the programming language understood by the computer.

There are several approaches towards defining dynamic semantics. The majority of users of the programming languages, i.e., programmers, consult dynamic semantics specifications written in natural language. Moreover, many programming languages have *only* natural language specifications of their semantics.

The natural language semantics specification sufficed for awhile, as long as computers did not play a key role in life-critical systems (i.e., systems whose failure induces life losses, environmental harm, or economic damages). However, life-critical systems became more and more computer-dependant or even computer-based.

Due to the importance of computer applications, there is need of strong reassurance that programs communicate to computers the exact intention of the programmer. Even more, due to the unreliability of the human factor, there is need for the programmer to verify that the program is the correct message to be transmitted to the computer. Hence, reasoning about program semantics becomes an important aspect in the process of human-computer communication.

The standard practice of testing for program correctness has notoriously lack of accuracy. Hence, testing is used only for program validation [55]. For program verification, however, there is need of formal proofs for the mathematical model of the programs. This makes the subject of *formal verification* field which currently receives special attention in academia. Namely, there exists increased, directed academic research efforts to producing formal verification methods for industry. Without even trying to be comprehensive, we point SLAM [110], ASTRÉE [104], and PVS [109].

There exists, however, an infamous result wrt undecidability of program verification. Namely, there is no fully automatic method that can always decide whether a given program in a particular programming language has a particular, nontrivial, property [43]. This result creates a prolific research area which comprises attempts to constructing various methods and methodologies of solving the program verification problem.

In the field of formal verification there are three important directions: model checking, abstract interpretation, and deductive verification. In the following three paragraphs we give a short overview of these formal verification methods. Note that these three overviews have as main source their textbooks [20, 72, 51] and Wikipedia pages [101, 99, 100].

Model checking tackles program verification by solving the following problem: *given a model of a system, verify automatically whether this model meets a given correctness property.* Edmund Clarke, Allen Emerson, and Joseph Sifakis received the Turing Award for their work on model checking [18]. Model checking is mainly used for hardware verification, while in software verification, standard model checking may fail to give a definitive answer. This is due to the fact that model checking relies on the premise of finiteness for the assumed model of the system.

Abstract interpretation approaches the program verification of so called *concrete*, potentially infinite, systems by standardizing a methodology which *performs the verification at some level of abstraction which ignores irrelevant details about the concrete system wrt the property*. This approach is a very prolific field of research, its basis being set by Patrick Cousot and Radhia Cousot [27]. The abstract model is usually finite, though this is not a strict requirement. Nevertheless, abstract interpretation sets the premises foe conducting the verification at the abstract level and then soundly projecting the result back into the concrete system. Note that the concrete is decoupled from the abstract during the verification process. Also, when the abstract system is finite,

model checking is a natural tool for conducting the verification process.

Deductive verification, by comparison with abstract interpretation, creates a tight coupling of the abstract level with the concrete system by inserting abstract code in the concrete program. The abstract code is formed by assertions while the connection between two assertions is made via a proof. As such, for deductive verification *assertions are introduced between the program lines such that the program piece between two assertions is abstracted into the proof connecting these assertions*. Due to the character of deductive verification, this methodology can be used *during* the development stage of the concrete system in order to increase its reliability. The setting of deductive verification were set by Robert Floyd and Tony Hoare and [38, 50].

As mentioned above, the three leading directions in program verification are interdependent and complement each other. In the followings we mention several such interdependent applications.

Firstly, model checking proved to be the most successful approach in industry, mainly due to its automatization degree which facilitates its application in industry. At the same time, the finiteness restriction of model checking is subdued using abstract interpretation or deductive methods (e.g., abstract model checking [19], or interpolation in model checking [61]).

Secondly, abstract interpretation proved to be the most comprehensive approach. Besides providing the means of enhancing model checking, one can see model checking as an instantiation of the fixpoint iteration used in abstract interpretation [22]. Moreover, the most representative deductive verification method, namely Hoare's axiomatic semantics, was proved to be an instantiation of abstract interpretation obtained via consecutive applications of abstractions [23].

Lastly, besides the obvious interconnection with abstract interpretation, deductive verification relies in a certain degree on model checking. Namely, deductive verification methods use theorem provers like PVS [109], Coq [105], Isabelle [107], and so on.

These theorem provers incorporate model checking subroutines which can be used in case-based proofs.

After this incursion in the animated field of formal verification, we pinpoint the following common factor present in each part of the discussion, namely semantics. For model checking the semantics was induced by the notion of "model"; abstract interpretation is the theory of semantic approximation; deductive verification is the theory of semantic parallelization. However, all these three approaches project the programming language semantics expressed in natural language. So, the question "*Are these formal methods accurately verifying the actual program?*" is justified and deserves answering.

A significant amount of research went into formal semantics of programming languages, which allow programs to be specified in a formal manner. Formal semantics create the mathematical, reliable environment which ensures the verification process is accurate. However, there is still the problem of matching the program semantics with the real program in the way it is actually understood by the computer. This problem is similar with certifying that a physics phenomena behaves according to the formula which describes it. The science has already standardized an answer, namely *measuring repeated experiments*.

In order to create proper settings for repeated experiments, one needs to setup the environment for these. However, it is a difficult task to match what the computer understands out of a program by comparison with what the human formalized that the computer is supposed to understand. The difficulty comes from the current complexity of the programs and programming language semantics. To overcome this difficulty, the formal *executable* semantics seem to provide a good solution.

A formal executable semantics is a formal semantics which can be run automatically, on computers. The advantages brought by having a formal executable semantics of a programming language include:

1. We remind that we assume a program is a *text* written in a language used for com-

municating with computers, i.e., a programming language. This text is formed by words for which the "dictionary", i.e., the programming language specification, gives meanings and understanding, i.e., semantics. Note that the dictionary includes definitions for words only. However, a dictionary does not provide definitions for any combination of words. (This would actually be impossible.) Here is where an executable dictionary makes the difference. Namely, for any given text a formal executable semantics can automatically produce the combined meaning defined for that text, according to the dictionary.

2. Recall that compilers are responsible with the translation from the language used by the programmer into the language understood by the computer. Hence a compiler is merely a translator on whose fidelity depends the preservation of the intended meaning of the program. A formal executable semantics for a programming language can also validate compilers. Namely, an interesting property induced by the executability of a formal semantics is the capability of comparing its composed meanings with the meanings induced by a compiler on a representative set of texts.

3. We also reiterate that an important usage of formal semantics resides in the area of program verification. Namely, formal semantics construct a solid mathematical background for the verification process. However, currently verification methods and tools are still evolving in the lab environment of academic research. But the need of verification is actually in the real world, in industry where the programs are produced and still validated with testing and simulation. The current effort of formal verification is to move the process from the lab environment into the real world. The simulator quality of a formal executable semantics and its capabilities of testing make this a promising transfer capsule of the lab conditions, i.e., academia, into the real world, i.e., industry.

Among the projects supporting formal executable semantics specification we men-

tion AsmL [103], Coq [105, 9], Isabelle [107, 76, 73], or Maude [108, 37]. $\mathbb{K}$ is a dedicated framework for specification of formal executable semantics [30, 58]. Among the achievements reported in $\mathbb{K}$ we mention the C specification [36, 34] and the specification of a subset of Verilog [64]. The verification methods employed by these definitions are model checking, via Maude LTL Model Checking tool [33], and deductive verification, via matching logic [30]. There are also ad-hoc approaches to analysis methods as type checking [35, 34] and a more standardized assertion-based analysis framework [47, 45]. However, abstract interpretation remains unexplored methodically in the $\mathbb{K}$ framework.

1.2 Objective

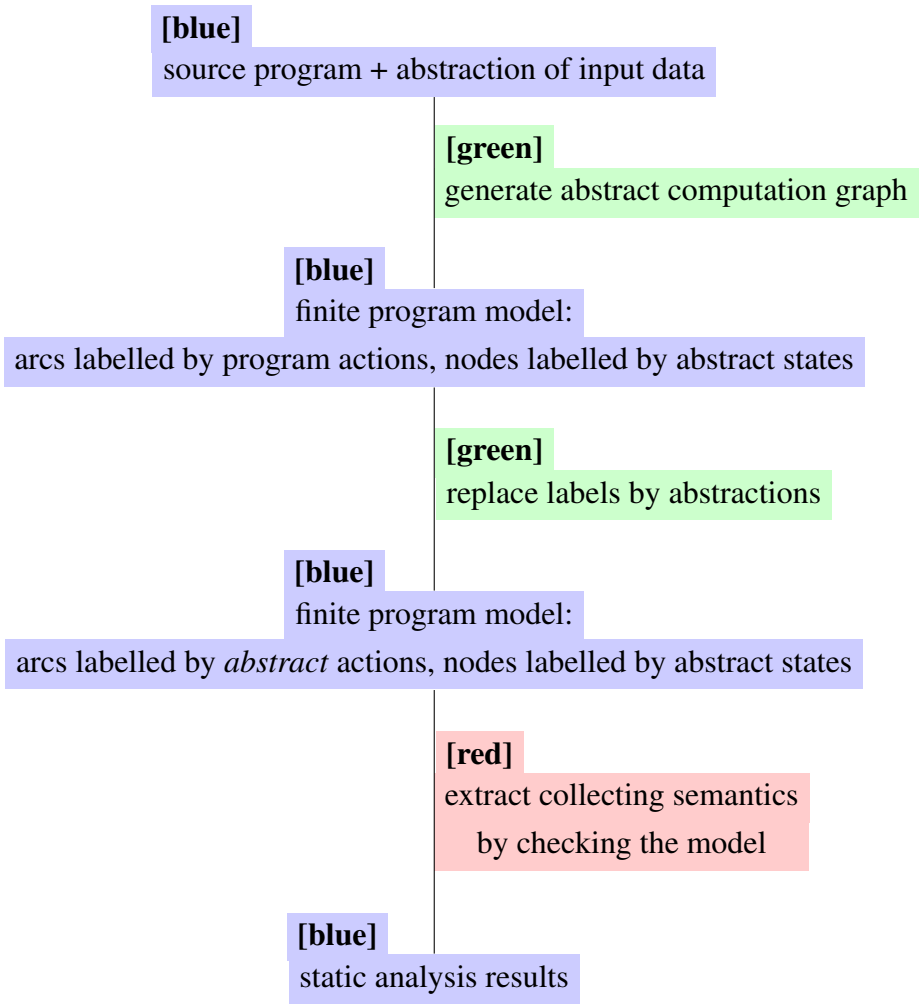
The main objective of the current work is to study how abstract interpretation can be used in $\mathbb{K}$ for program analysis and verification.

Abstract interpretation is a very prolific research area which debuted in [25] as a unified framework for static program analysis. Hence, in the beginnings abstract interpretation was a framework dedicated to defining static semantics. However, along the time, abstract interpretation proved to be applied in many other areas of programming languages. For a comprehensive description of these applications we refer to [21].

As stated before, we aim to obtain a combination of static semantics with dynamic semantics. More to the point, we intend to employ model checking over dynamic semantics of abstractions in order to reproduce results obtained by static analysis via static semantics. In doing this, we rely on the abstract interpretation view given in [94] where a meta-algorithm for representing the analysis methods from abstract interpretation into abstract model checking problems is described.

The meta-algorithm from [94] is reproduced here in Algorithm 1 where the blue labels represent the data with which the meta-algorithm is working, while the green and

Algorithm 1: A Static Analysis Methodology presented in [94]

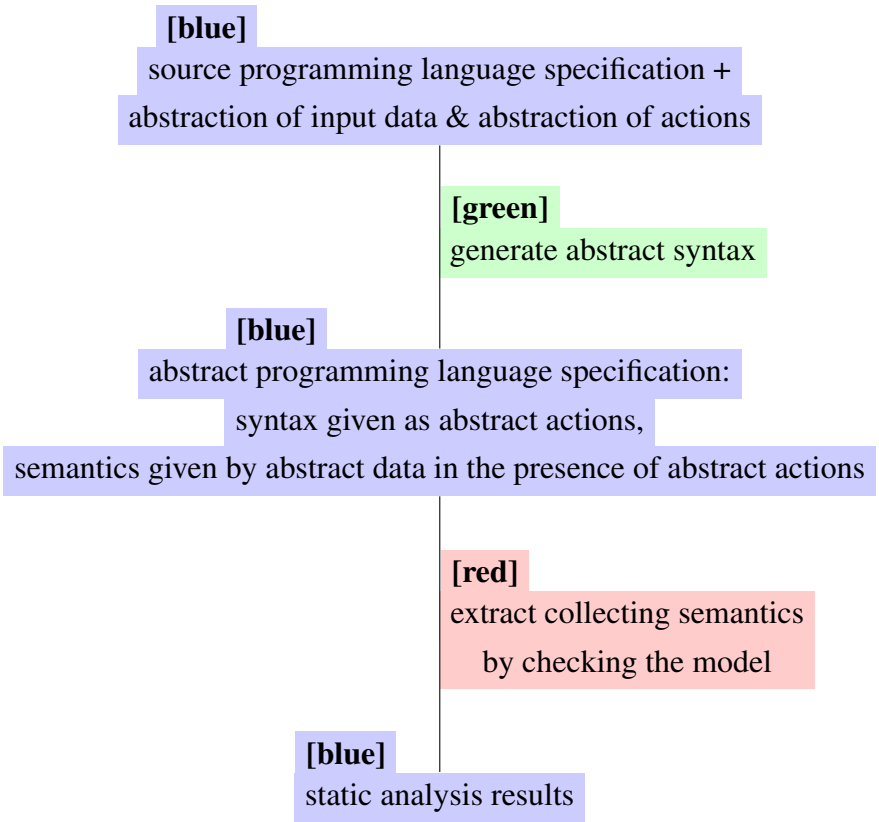


red labels represent the actions that are applied when transforming a data into another. We make the distinction green/red in order to differentiate between the focus attributed in the current work to these actions. Namely, our main concern for the current work is the red labeled action, while the green-labeled actions are collapsed into a single step and covered only for a simple example.

Our view about the Algorithm 1 coordinates with the view available in the tool Astrée [104, 11]. Namely, Astrée is *a comprehensive collection of abstractions used to derive static analysis results for structured C programs, with complex memory usages,*

but without dynamic memory allocation and recursion. However, our main concern is the analysis method itself. Hence, we adopt the decoupling provided by Algorithm 1, and assume that the static analysis methods are parametric in the source programming language. As such, we transform Algorithm 1 into Algorithm 2 in order to provide *a collection of abstract programming languages used to derive static analysis results for real programming languages containing static and dynamic memory allocation in the presence of recursion*.

Algorithm 2: A Static Analysis Methodology adaptation, for the current work, of the meta-algorithm defined in [94] and reproduced in Algorithm 1.



The modifications made by Algorithm 2 concern the fact that we replace conceptually the program with the specification of the programming language in which the program is defined. Moreover, the combination of the two green labeled steps relies on

the fact that abstraction is compositional hence one can see the two steps generation of the abstract program model as one (compositional) step. In any case, our main concern in the current work is the red-labeled step which uses the abstract model, in our case abstract programming language specification, to obtain the static analysis results. This step is based on the collecting semantics, a semantics standardized in abstract interpretation as a means of producing fixpoint iterations. Collecting semantics relies on the operational semantics of the abstraction and collects executions of the targeted program via a forward or backward fixpoint iteration.

Consequently, in Algorithm 2 we provide a generic method for defining various analysis and verification methods using abstract programming languages specifications and some instantiation of collecting semantics. The two-steps Algorithm 2 is the objective of the current work and, out of the two steps, we focus on the red-labeled part.

1.3 Contributions

Towards achieving our objective, the key contributions of this dissertation are:

1. **Generic collecting semantics for pushdown systems abstractions in $\mathbb{K}$**

   We describe a methodology for defining in $\mathbb{K}$ finite abstractions for pushdown systems. We choose pushdown systems because of their relative generality wrt programming languages. Furthermore, we give a generic algorithm for defining collecting semantics over the $\mathbb{K}$ specification of finite pushdown systems. This methodology is instantiated in all subsequent contributions. This contribution is introduced in [2, 3] and presented in Chapter 3.

2. **Data analysis in $\mathbb{K}$**

   We instantiate with predicate abstraction [42], a flow sensitive abstraction for static memory which can be used to verify data invariants. We associate predicate ab-

straction with a collecting semantics which produces the fixpoint iteration. This contribution is published in [4, 5] and presented in Chapter 4.

3. **Alias analysis in $\mathbb{K}$**

   We give the $\mathbb{K}$ specification of an abstract programming language defined in [89, 91]. We use this language with a collecting semantics which defines the fixpoint iteration for producing alias analysis results. This contribution is introduced in [2] and presented in Chapter 5.

4. **Shape analysis in $\mathbb{K}$**

   We give the $\mathbb{K}$ specification of an abstract programming language defined in [90]. We use this language with a collecting semantics which defines a generic invariant model checking algorithm for pushdown system specifications. The state properties employed for the abstract language define shape invariants. We employ these properties to produce demand driven shape analysis. This contribution is briefly described in [8, 6, 90, 7] and detailed in Chapter 6.

Earlier work on program analysis in $\mathbb{K}$ is presented in [47, 45] where the analysis is given as an abstract semantics for a language of program assertions. That work evolved into the deductive verification tool proposed by matching logic [86, 83, 30]. The main difference in the approach presented in the current work is that we propose an abstract semantics which is decoupled from the actual code, in the style of abstract interpretation.

1.4 Organization of the Dissertation

The structure of this dissertation is the following:

Chapter 2. Preliminary notions

We describe here the foundations of the main framework used in this work,

namely $\mathbb{K}$ and abstract interpretation.

Chapter 3. Methodology

This chapter presents a generic approach to integrating analysis and verification methods in the $\mathbb{K}$ framework. In order to have a good degree of generality, we choose pushdown systems for presenting the $\mathbb{K}$ perspective of collecting semantics over operational semantics. We use the theoretical results already available for pushdown systems in order to show the effectiveness of our à la abstract interpretation generic approach to analysis and verification methods.

Chapter 4. Tracing Predicate Abstraction in $\mathbb{K}$

We propose a suitable description in $\mathbb{K}$ for collecting semantics under predicate abstraction of a simple imperative language. We also prove that our $\mathbb{K}$ specification for collecting semantics is a sound approximation of the $\mathbb{K}$ specification for concrete semantics.

Chapter 5. $\mathbb{K}$ Abstract Semantics for Alias Analysis

We present a flow sensitive, context sensitive, interprocedural alias analysis. We achieve this type of alias analysis by means of collecting semantics for an abstract programming language, called *SILK*. *SILK* is an imperative programming language which supports object creation, global variables, static scope and recursive procedures with local variables. The "on paper" semantics of this language is presented [91, 89] as a pushdown system specification. We implement this programming language in $\mathbb{K}$ and also a collecting semantics specification which renders the desired alias analysis results.

Chapter 6. $\mathbb{K}$ Abstract Semantics for Shape Analysis

We introduce a method, and its specification in $\mathbb{K}$, for shape analysis achieved via abstract model checking. Namely, we first present the $\mathbb{K}$ specification of an

abstract language, called *Shylock*, which is focussed on reasoning about the structures maintained in the heap: objects with fields. Shylock is introduced in [90, 8] as "on paper" pushdown system specification. Consequently, we present the $\mathbb{K}$ specification of an algorithm introduced in [95] for model checking pushdown systems. As such, we give the $\mathbb{K}$ specification for a shape analysis abstract model and the $\mathbb{K}$ specification of a generic method of reasoning about it.

Chapter 7. Conclusions

We summarize the current work and give several directions for future work.

Chapter 2

Preliminary notions

Motto: *"Those who read this will not fail to laugh at my gallantries,*
and remark, that after very promising preliminaries,
my most forward adventures concluded by a kiss of the hand:
yet be not mistaken, reader, in your estimate of my enjoyments;
I have, perhaps, tasted more real pleasure in my amours,
which concluded by a kiss of the hand,
than you will ever have in yours, which, at least, begin there."
— Jean-Jacques Rousseau, Confessions.

2.1 $\mathbb{K}$ Background

$\mathbb{K}$ is a rewrite logic-based framework for design and analysis of programming languages. The spark of the $\mathbb{K}$ framework [81] is the observation that the computation is expressed naturally with rewriting. The inspiration source of $\mathbb{K}$ is the Rewriting Logic Semantics project [69, 97, 70] which has the declared purpose of unifying the algebraic denotational semantics and the operational semantics. This unification is achieved by considering the two semantics as different views over the same object. Namely, the

denotational semantics is the view of the rewrite logic specification of a language as a designated model, while the an operational semantics is the executions view of the same specification.

$\mathbb{K}$ is built upon a continuation-based technique and a series of notational conventions to allow for more compact and modular programming language definitions. The $\mathbb{K}$ definitions can be mechanically translated into rewriting logic, and in particular into Maude, to obtain program analysis tools or interpreters based on term rewriting. This versatility makes $\mathbb{K}$ an executable framework with $\mathbb{K}$-Maude its current prototype implementation [29, 28].

A $\mathbb{K}$ definition specifies a transition system while the computations of the transition system are automatically obtained by the execution of its $\mathbb{K}$ definition. Moreover, one can also reuse the $\mathbb{K}$ definition of the transition system to enable richer executions as, for example, sets of computations. When producing the plain computations, $\mathbb{K}$ can be seen as an interpreter. However, when producing sets of computations, $\mathbb{K}$ can also be used as an analyzer/verifier for specified the transition systems.

2.1.1 $\mathbb{K}$ syntax

A $\mathbb{K}$ specification consists of *configurations* and *rules*. The configurations, formed of $\mathbb{K}$ cells, are (potentially) labeled and nested structures that represent program states. The rules in $\mathbb{K}$ are divided into two classes: *computational rules*, that may be interpreted as transitions in a program execution, and *structural rules*, that modify a term to enable the application of a computational rule. The $\mathbb{K}$ framework allows one to define modular and executable programming language semantics.

We present the $\mathbb{K}$ framework by means of an example - a simple imperative language *SIM* with simple integer arithmetic, basic boolean expressions, assignments, if statements, while statements, sequential composition, and blocks. For this purpose we rely extensively on [81].

The $\mathbb{K}$ syntax with annotations and semantics of *SIM* is given in Fig. 2.1. The left column states the *SIM* abstract syntax, the middle column introduces a special $\mathbb{K}$ notation, called strictness attribute, and the right column presents the $\mathbb{K}$ rules for *SIM* language semantics. Because the abstract syntax is given in a standard way, we proceed explaining, via an example, the strictness attribute called *seqstrict* (denoted here as *sq* for space efficiency). The strictness attribute that corresponds to the addition rule $Aexp + Aexp$ is translated into the set of heating/cooling rule pairs: $a_1 + a_2 \rightleftharpoons a_1 \curvearrowright \square + a_2$ and $i_1 + a_2 \rightleftharpoons a_2 \curvearrowright i_1 + \square$. These structural rules state how an arithmetic expression with addition is evaluated sequentially: first the lefthand side term (here a_1) is reduced to an integer, and only then the righthand side term a_2 is reduced to some integer. The resulted integers are added using the internal operation of integer addition $+_{Int}$ as represented by the rule $i_1 + i_2 \rightarrow i_1 +_{Int} i_2$ in the *SIM* semantics (right column). The assignment statement has the attribute $sq(2)$ which means that strictness attribute *seqtrict* is applied only to the second argument.

The $\mathbb{K}$ modeling of a program configuration is a multiset of cells written $\langle c \rangle_l$, where c is the content of a cell and l is the cell label. Examples of labels include: top $\top$, current computation k, store, call stack, output, formal analysis results, etc. The *SIM* program configuration is:

$$Configuration \equiv \langle \langle K \rangle_{\mathsf{k}} \langle Map[Var \mapsto Int] \rangle_{\mathsf{state}} \rangle_{\top}$$

where the top cell $\langle \ldots \rangle_{\top}$ contains two other cells: the computation $\langle K \rangle_{\mathsf{k}}$ and the store $\langle Map[Var \mapsto Int] \rangle_{\mathsf{state}}$. The k cell has a special meaning in $\mathbb{K}$, maintaining computational contents, much as programs or fragments of programs. The computations, i.e. terms of special sort *K*, are lists of computational tasks. Elements of such a list are separated by an associative operator "$\curvearrowright$", as in $s_1 \curvearrowright s_2$, and are processed sequentially: s_2 is computed after s_1. The "$\cdot$" is the identity of "$\curvearrowright$". The contents of state cell is

an element from $Map[Var \mapsto Int]$, namely a mapping from program variables to integer values (maps are easy to define algebraically and, like lists and sets, they are considered builtins in $\mathbb{K}$).

AExp	::=	*Var* | *Int*		$\langle \underset{\sigma[x]}{x} \cdots \rangle_{\mathsf{k}} \langle \sigma \rangle_{\mathsf{state}}$
	|	$AExp + AExp$	*sq*	$i_1 + i_2 \to i_1 +_{Int} i_2$
	|	$Int * AExp$	$sq(2)$	$i_1 * i_2 \to i_1 *_{Int} i_2$
BExp	::=	$AExp <= AExp$	*sq*	$i_1 <= i_2 \to i_1 \leq_{Int} i_2$
	|	$AExp == AExp$	*sq*	$i_1 == i_2 \to i_1 =_{Int} i_2$
	|	`not` *BExp*	*sq*	`not` $t \to \neg_{Bool} t$
	|	*BExp* `and` *BExp*	$sq(1)$	*true* `and` $b \to b$
				false `and` $b \to$ *false*
Stmt	::=	`skip`		`skip` $\to \cdot$
	|	$Var = AExp$	$sq(2)$	$\langle \underset{\cdot}{x = i} \cdots \rangle_{\mathsf{k}} \langle \cdots \; x \mapsto \underset{i}{\_} \cdots \rangle_{\mathsf{state}}$
	|	*Stmt* ; *Stmt*		$s_1 ; s_2 \curvearrowright s_1 \curvearrowright s_2$
	|	$\{Stmt\}$		$\{s\} \to s$
	|	`if` *BExp* `then`$\{Stmt\}$ `else`$\{Stmt\}$	$sq(1)$	`if` *true* `then` s_1 `else` $s_2 \to s_1$ `if` *false* `then` s_1 `else` $s_2 \to s_2$
	|	`while` *BExp* $\{Stmt\}$		`while` b $\{s\}$ ——— `if` b `then`$\{s;$ `while` b $\{s\}\}$`else`$\{$`skip`$\}$
Pgm	::=	`vars` Set$[Var]$; *Stmt*		$\langle$`vars` $xs; s\rangle_{\mathsf{k}} \langle \underset{xs \mapsto is}{\cdot} \rangle_{\mathsf{state}}$ (under s)

Figure 2.1: $\mathbb{K}$ syntax of *SIM* (left) with annotations (middle) and semantics (right) with $x \in Var, xs \in Set\{Var\}, i, i_1, i_2 \in Int, is \in Set\{Int\}, b \in BExp, s, s_1, s_2 \in Stmt$

The third column in Fig. 2.1 contains the semantic rules of *SIM*. The $\mathbb{K}$ rules generalize the usual rewrite rules, namely $\mathbb{K}$ rules manipulate parts of the rewrite term in different ways: write, read, and don't care. This special type of rewrite rule is conveniently represented in a bidimensional form. In this notation, the lefthand side of the rewrite rule is placed above a horizontal line and the righthand side is placed below. The bidirectional notation is flexible and concise, one could underline only the parts of the term that are to be modified. Ordinary rewrite rules are a special case of $\mathbb{K}$ rules,

when the entire term is replaced; in this case, the standard notation *left*$\rightarrow$*right* is used.

The first $\mathbb{K}$ rule is a *computational rule* using bidimensional notation to describe the variable lookup operation. The underlined term x in cell k means that x is a "write" term, and is to be replaced by $\sigma[x]$ from the state cell. The absence of the horizontal line under σ indicates that this is a "read" term and the state remains unchanged. The notation "$\langle x \ \cdots \rangle$" in cell k says that x is placed in the beginning of the term contained by this cell.

The assignment rule has the statement $x = i$ as the current computation task (the first element in cell k), with a "don't care" value "_" for x *somewhere* in the store (as shown by the notation $\langle \cdots \ x \ \cdots \rangle_{\mathsf{state}}$). The value of x is updated with i in the store and the assignment is replaced by the empty computation.

The variable lookup and assignment rules are computational rules. Recall that structural rules are used to only rearrange the term to enable the application of the computational rules. One such example is the "while" rule from the right column in Fig. 2.1, which unfolds one step of a while-loop statement into a conditional statement. The structural transformation is represented with a dotted line to convey the idea that this transformation is lighter-weight than in computational rules. We recall that the usual rewrite rules are special cases of $\mathbb{K}$ rules and the $\mathbb{K}$ framework proposes "$\rightarrow$" for computational rules, and "$\curvearrowright$" for structural rules. The former notation is used for "if" rules, while the latter is used for the sequential composition. However, in this work we use only bi-dimentional notation.

The application of the initialization rule of a program (last rule in the right column in Fig. 2.1) leaves the computation cell containing the entire set of statements, and the memory cell containing an initial mapping of program variables *xs* into integers. The program terminates when computation is completely consumed, meaning when the computation cell is $\langle \cdot \rangle_{\mathsf{k}}$.

Example 1. *A SIM program pgmX is given in Fig. 2.2 as* `vars x, y, err;` *sX, where*

```
vars x, y, err;
x = 0;  err = x;
while (y <= 0) do {
  x = x + 1;  y = y + x;  x = −1 + x;
  if not (x == 0) then {err = 1} else {skip}
}
```

Figure 2.2: Example of a *SIM* program

sX denotes the statements of the program. In a concrete execution, initialized with $\langle\langle sX\rangle_{\mathsf{k}}\,\langle\cdots\ \mathtt{y}\mapsto -3\ \cdots\rangle_{\mathsf{state}}\rangle_{\top}$*, the first computational rule applied is the rule for assignments, such that the* state *cell becomes* $\langle\cdots\ \mathtt{y}\mapsto -3,\mathtt{x}\mapsto 0\ \cdots\rangle_{\mathsf{state}}$*. This execution terminates with* $\langle\mathtt{x}\mapsto 0,\mathtt{y}\mapsto 0,\mathtt{err}\mapsto 0\rangle_{\mathsf{state}}$*. However, if the while condition in the program is* $(0<=\mathtt{y})$ *and the program is initialized with* $\langle\langle sX\rangle_{\mathsf{k}}\,\langle\cdots\ \mathtt{y}\mapsto 3\ \cdots\rangle_{\mathsf{state}}\rangle_{\top}$*, then the execution does not terminate.*

2.1.2 $\mathbb{K}$ semantics

A $\mathbb{K}$ definition specifies a transition system, with rules specifying transitions between instances of the $\mathbb{K}$ configuration. The $\mathbb{K}$ configuration is a nested bag of cells which defines the structure of the states in the specified transition system. The content of the cells can be one of the predefined types of $\mathbb{K}$, namely *K*, *List*, *Map*, *Set*, or *Bag*. The rules in $\mathbb{K}$ define transitions which, in the transition system, can be either internal or observable. The observable transitions are called *computational rules* and are specified in the $\mathbb{K}$-tool by the rules tagged with attribute **transition**. These rules compile in Maude into rewrite rules. The internal transitions are called *structural rules* and are meant to prepare the current state for the next observable transition which actually represents the next computational step. The internal transitions are defined in the $\mathbb{K}$-tool using the tag **structural** and are compiled in Maude as equations. As such, a $\mathbb{K}$ *definition* is a triple $\mathscr{K} = (\Sigma, S, C)$, where Σ is an algebraic signature providing the infrastructure for the configuration, S is the set of structural rules, and C is the set of computational

rules. Given a ground term σ representing the *current* configuration, the *next* configuration is a ground term obtained from the application of exactly one computational rule. Note that before or after the application of the computational rule any number of structural rules can apply. As such, one can see the current configuration as part of a class of structurally related configurations while a transition in $\mathscr{K}$ is moves to the next class of structurally related rules. For more on the semantics of the $\mathbb{K}$ framework see [81, 96, 30, 58], and more on the $\mathbb{K}$ tool can be found in [29, 28].

2.2 Abstract Interpretation

Abstract interpretation is a theory of semantics-based program analysis and verification. In this section we convey some of the basic notions of abstract interpretation. For an extensive presentation of abstract interpretation, we refer [21, 24, 26, 72, 93].

A complete lattice is a partially ordered set in which all subsets have both a supremum and an infimum. Complete latices are the mathematical environment where monotone functions have certain useful properties. One such property, which is applied in abstract interpretation, is the Knaster–Tarski fixpoint theorem and its consequences. Namely, Knaster–Tarski theorem ensures that the fixpoints of a monotone function can be reached by iterative application of that function over itself, starting with the seed of either the top or the bottom element in the lattice.

However, the effectiveness of the iterative application of a monotone function depends very much on the size of the complete lattice used as function domain. Abstract interpretation gives the mathematical foundation for reducing the size of the computation by a *sound* transportation of the fixpoint computation from a large function domain into a smaller, more efficient one. This process of sound transportation is enabled by Galois connections.

There are several equivalent definitions for a Galois connection $(C, \#, \$, A)$ between

two complete lattices $(C, \subseteq)$ and $(A, \sqsubseteq)$, where $\# : C \to A$ and $\$: A \to C$ are two functions:

a) # and \$ are monotone functions that satisfy $1_C \subseteq \#;\$$ and $\$;\# \sqsubset 1_A$;

b) # and \$ are total functions that satisfy $\#(c) \sqsubseteq a \Leftrightarrow c \subseteq \$(a)$, for any $c \in C$ and $a \in A$;

c) # and \$ are total functions that satisfy $a \sqsubseteq \#(c) \Leftrightarrow \$(a) \subseteq c$, for any $c \in C$ and $a \in A$.

Moreover, a property of Galois connections states that # uniquely determines \$ by $\$(a) = \sqcup\{c \mid \#(c) \sqsubseteq a\}$, and \$ uniquely determines # by $\#(c) = \sqcap\{a \mid c \subseteq \$(a)\}$ and that # is completely additive and \$ is completely multiplicative. A consequence of this result is that a Galois connection can be defined only by either a completely additive #, or a completely multiplicative \$.

There are also several methods of constructing Galois connections by combinations of two or more other Galois connections. Among these we mention only the compositional method, the independent attribute method, and the relational method.

1. The *compositional method* takes two Galois connections $(L_0, \#_1, \$_1, L_1)$ and $(L_1, \#_2, \$_2, L_2)$ and guarantees that $(L_0, \#_1;\#_2, \$_2;\$_1, L_2)$ is also a Galois connection;

2. The *independent attribute method* takes two Galois connections $(C_1, \#_1, \$_1, A_1)$ and $(C_2, \#_2, \$_2, A_2)$ and guarantees that $(C_1 \times C_2, \#, \$, A_1 \times A_2)$ is also a Galois connection, where $\#(c_1, c_2) = (\#_1(c_1), \#_2(c_2))$ and $\$(a_1, a_2) = (\$_1(a_1), \$_2(a_2))$, for any $c_i \in C_i$, and $a_i \in A_i$, with $i = 1..2$;

3. The *relational method* takes two Galois connections $(2^{C_1}, \#_1, \$_1, 2^{A_1})$ and $(2^{C_2}, \#_2, \$_2, 2^{A_2})$ and guarantees that $(2^{C_1 \times C_2}, \#, \$, 2^{A_1 \times A_2})$ is also a Galois con-

nection, where $\#(c) = \cup\{\#_1(\{c_1\}) \times \#_2(\{c_2\}) \mid (c_1, c_2) \in c\}$ and $\$(a) = \{(c_1, c_2) \mid \#_1(\{c_1\}) \times \#_2(\{c_2\}) \subseteq a\}$, for any $c \in C_1 \times C_2$ and any $a \in A_1 \times A_2$.

Furthermore, the most important aspect of Galois connection is the reflective properties of the endomorphisms defined over the two lattices of a Galois connection. Namely, let $(C, \#, \$, A)$ be a Galois connection and let $f : C \to C$, $g : A \to A$ be two monotone functions such that g is an over-approximation of f, i.e., $\$; f; \# \sqsubseteq g$. Then, for all $a \in A$ we have that $g(a) \sqsubseteq a \Rightarrow f(\$(a)) \subseteq \$(a)$. Also $lfp(f) \subseteq \$(lfp(g))$ and $\#(lfp(f)) \sqsubseteq lfp(g)$. This result is applied in abstract interpretation under the name of *sound* abstraction. Practically, this says that if we compute $lfp(g)$ in the abstract lattice A, then we can transport the result into the concrete for $lfp(f)$. Note that the calculation of the abstract least fixpoint, i.e., $lfp(g)$, is known in abstract interpretation under the name of collecting semantics.

From the semantics point of view, a particular analysis/verification method is achieved by defining collecting semantics for the scrutinized transition systems [24, 26]. Traditionally, collecting semantics relies on the operational semantics for defining abstractions of computations which are collected forward or backward via a fixpoint iteration. Hence, the analysis/verification methods are a semantic reflection of the computations.

We give a short description of the analysis case studies approached in the current work, namely data analysis, alias analysis, and shape analysis. These analysis techniques are particularly useful in compilers which apply these analysis in the process of safe code transformation.

Data analysis targets the data in a program with the goal of highlighting useful information and supporting decision making w.r.t. the program transformation produced by compilers. Data analysis has multiple facets and approaches using various techniques. Here we highlight only predicate abstraction, introduced in [42], which is a well known and extensively used instantiation of abstract interpretation. Essentially, predicate abstraction defines a finite set of arithmetic properties over the values of the

program variables and reasons over the program using these arithmetic properties. The important fact is that this reasoning produced by predicate abstraction is a sound over-approximation of the program data.

Both alias and shape analysis focus on the dynamic memory of the program, i.e. the heap. During **alias analysis** the dynamic variables are partitioned into *alias classes* representing disjoint sets of dynamic variables which, at a particular program location, point to the same heap location. Of course, dynamic variables may point to more than one location over time and thus may be in more than one alias class. This means that each program location collects, during alias analysis, a set of alias classes instead of a single alias class. In comparison with alias analysis, **shape analysis** makes more precise inferences about the heap structure by taking into consideration the heap locations a dynamic variable may access via its information contents, i.e., via its fields. Hence the heap properties produced by shape analysis are richer and more refined. As such, shape analysis infers properties about linked lists, dangling pointers, memory leaks, array out of bounds, and so on. However, this richness comes with a high cost of analysis efficiency hence shape analysis is currently mainly approached in research area with less impact in compilers' industry.

Chapter 3

Methodology via Pushdown Systems

$\mathbb{K}$-Specifications

Motto: *"This history of culture will explain to us*
the motives, the conditions of life,
and the thought of the writer or reformer."
— Leo Tolstoy, War and Peace.

This chapter presents an approach to integrating analysis and verification methods in the $\mathbb{K}$ framework. The motivating idea is to promote the semantic reflection available between operational semantics and the analysis and verification methods. This semantic reflection is founded in the abstract interpretation framework. Namely, collecting semantics is an inherent reflection of operational semantics. As such, we rely on collecting semantics to provide the operational semantics for analysis and verification methods. On the other hand, the inherited verification method in $\mathbb{K}$ is given by matching logic which uses the Hoare-Floyd style for program verification. The verification in abstract interpretation style is notoriously orthogonal to the verification in Hoare-Floyd style. Consequently, we use the $\mathbb{K}$ settings proposed with the matching logic approach for program verification, and adjust these to develop the complementary

view of verification given by abstract interpretation. In order to have a good degree of generality, we choose pushdown systems for presenting the $\mathbb{K}$ perspective of collecting semantics over operational semantics. We use the theoretical results already available for pushdown systems in order to show the effectiveness of our à la abstract interpretation generic approach to analysis and verification methods.

3.1 Foundations revisited and revamped

Everything that needs to be said has already been said.
But since no one was listening, everything must be said again.
— André Gide.

In this section we present the generic settings for the representation of pushdown systems in the $\mathbb{K}$ framework. We consider this representation being relevant in $\mathbb{K}$ for few interconnected reasons, hereupon enumerated.

First, we quote Goguen and all: "*The most important and general example of initial many-sorted algebra semantics is semantics for context-free grammars*" [41]. Pushdown systems are the operational counterpart of context-free grammars. Meanwhile, $\mathbb{K}$ adheres to the Rewriting Logic Semantics Project which has as main desiderate the unification between algebraic denotational semantics and operational semantics. Bridging these three ideas, we infer that *the pushdown system specification in* $\mathbb{K}$ *is of importance for its degree of generality*.

Second, the $\mathbb{K}$ framework distills positive aspects from various semantic formalisms as MSOS [71], CHAM [10], reduction semantics with evaluation contexts [102], or continuation based semantics [39]. All these are sewed in a novel notational style given by the cells, the configuration abstraction, the local rewriting, and the current continuation style for program execution inherently embedded in the set-aside cell $\langle\rangle_{\mathsf{k}}$. Among all, we zoom on the fact that the current continuation is usually seen as a denotational se-

mantic counterpart for the stack-based operational semantics. As such, we emphasize here the complementary view which sees the contents of the continuation cell $\langle\rangle_{\mathsf{k}}$ as the stack in the pushdown systems.

Last, but not least, pushdown systems have been employed as formal models for model checking C or Java programs [95, 98]. The decidability results available for analysis and verification of pushdown systems have a better degree of generality than the results for transition systems. The main idea about this degree of generality is that finite pushdown systems are equivalent with potentially infinite transition systems. Meanwhile, one of the goals in the $\mathbb{K}$ framework, besides defining programming language semantics, is to develop formal analysis and verification tools for these definitions. Consequently, we consider of interest specifying pushdown systems in $\mathbb{K}$, such that we can develop afterwards analysis and verification tools for these. We study these aspects in the later sections of the current work.

In the remaining of this section we present a general technique for specifying pushdown systems in $\mathbb{K}$, by means of abstraction.

3.1.1 Pushdown system specifications in $\mathbb{K}$

The intuitive overview of this section is provided by the category fundamental result showing that there is a Galois connection between classes of models and sets of sentences in an institution. Similarly, we instill the same perspective over the Galois connection between a pushdown system and its $\mathbb{K}$-specification. We start the presentation from the pushdown systems (as models) and show how their specifications (i.e., sentences) can be seen reflectively as abstract pushdown systems.

Definition 1. *A* pushdown system *is a quadruple* $\mathscr{P} = (\Delta, \nabla, \hookrightarrow, c_0)$ *where* Δ *is the set of* control locations*,* ∇ *is the* stack alphabet*, and* $\hookrightarrow$ *is a subset of* $(\Delta \times \nabla) \times (\Delta \times \nabla^*)$ *representing the set of* rules*. A* configuration *is a pair* (δ, Γ) *where* $\delta \in \Delta$ *and* $\Gamma \in \nabla^*$.

The set of all configurations is denoted as $Conf(\mathscr{P})$ *and* $c_0 \in Conf(\mathscr{P})$ *is the initial configuration. A pushdown system is said to be* finite *when the above three sets,* Δ, ∇, *and* $\hookrightarrow$, *are finite.*

Example 2. *Let the set of control locations be* $\Delta = \{d_0, d_1, d_2\}$, *the stack alphabet be* $\nabla = \{g_0, g_1, g_2, g_3, g_4\}$, *the initial configuration* $c_0 = (d_0, g_3 g_4)$ *and the relation* $\hookrightarrow$ *be defined as follows:*

$(d_0, g_0) \hookrightarrow (d_0, g_1 g_0 g_2)$	$(d_1, g_0) \hookrightarrow (d_1, g_1 g_0 g_3)$	$(d_2, g_0) \hookrightarrow (d_2, g_1 g_0 g_4)$
$(d_0, g_1) \hookrightarrow (d_1, \varepsilon)$	$(d_1, g_1) \hookrightarrow (d_2, \varepsilon)$	$(d_2, g_1) \hookrightarrow (d_1, \varepsilon)$
$(d_0, g_2) \hookrightarrow (d_0, \varepsilon)$	$(d_1, g_2) \hookrightarrow (d_1, \varepsilon)$	$(d_2, g_2) \hookrightarrow (d_2, \varepsilon)$
$(d_0, g_3) \hookrightarrow (d_0, g_0 g_2)$	$(d_1, g_3) \hookrightarrow (d_1, \varepsilon)$	$(d_2, g_3) \hookrightarrow (d_2, \varepsilon)$
$(d_0, g_4) \hookrightarrow (d_0, \varepsilon)$	$(d_1, g_4) \hookrightarrow (d_1, \varepsilon)$	$(d_2, g_4) \hookrightarrow (d_2, \varepsilon)$

Observation 1. *A pushdown system is equivalently described by its associated transition system* $\mathscr{T}_{\mathscr{P}} = (Conf(\mathscr{P}), \to, c_0)$, *with the relation* $\to \subseteq (\Delta \times \nabla^*) \times (\Delta \times \nabla^*)$ *such that if* $(\delta, \gamma) \hookrightarrow (\delta', \Gamma)$ *then* $(\delta, \gamma\Gamma') \to (\delta', \Gamma\Gamma')$, *for all* $\Gamma' \in \nabla^*$, *where* $\delta, \delta' \in \Delta$, $\gamma \in \nabla$, *and* $\Gamma \in \nabla^*$.

Example 3. *The transition system associated to the pushdown system described in Example 2 has the set of states* $\{(d_i, \Gamma) \mid i \in 0, 1, 2 \text{ and } \Gamma \in \nabla^*\}$. *The transition relation* $\to$ *is defined as:*

$(d_0, g_0\Gamma) \to (d_0, g_1 g_0 g_2 \Gamma)$	$(d_1, g_0\Gamma) \to (d_1, g_1 g_0 g_3 \Gamma)$	$(d_2, g_0\Gamma) \to (d_2, g_1 g_0 g_4 \Gamma)$
$(d_0, g_1\Gamma) \to (d_1, \Gamma)$	$(d_1, g_1\Gamma) \to (d_2, \Gamma)$	$(d_2, g_1\Gamma) \to (d_1, \Gamma)$
$(d_0, g_2\Gamma) \to (d_0, \Gamma)$	$(d_1, g_2\Gamma) \to (d_1, \Gamma)$	$(d_2, g_2\Gamma) \to (d_2, \Gamma)$
$(d_0, g_3\Gamma) \to (d_0, g_0 g_2 \Gamma)$	$(d_1, g_3\Gamma) \to (d_1, \Gamma)$	$(d_2, g_3\Gamma) \to (d_2, \Gamma)$
$(d_0, g_4\Gamma) \to (d_0, \Gamma)$	$(d_1, g_4\Gamma) \to (d_1, \Gamma)$	$(d_2, g_4\Gamma) \to (d_2, \Gamma)$

with $\Gamma \in \nabla^*$ *any finite word formed with the letters from the stack alphabet.*

First we assume we have an algebraic representation for Δ and ∇. Now, the $\mathbb{K}$ configuration describes the structure of $Conf(\mathscr{P})$ as a nested bag of labeled cells, namely $\langle\langle\Delta\rangle_{\mathsf{ctrl}}\langle List_{\curvearrowright}\{\nabla\}\rangle_{\mathsf{k}}\rangle_{\mathscr{P}}$. The continuation-based feature of $\mathbb{K}$ is introduced by the special cell $\langle\rangle_{\mathsf{k}}$ which contains a list of computation tasks from a special sort K. In the $\mathbb{K}$ definition for $\mathscr{P}$, we specify that ∇ is a subsort of K by the $\mathbb{K}$ syntactic declaration $K ::= \nabla$. Hence, when $\Gamma = \gamma_1..\gamma_n \in \nabla^*$ is introduced in the computation cell, it becomes $\langle\gamma_1 \curvearrowright .. \curvearrowright \gamma_n\rangle_{\mathsf{k}}$, where $\curvearrowright$ is the $\mathbb{K}$ separator for the computation tasks. Furthermore, $Conf(\mathscr{P})$ can be consequently defined as $\langle\langle\Delta\rangle_{\mathsf{ctrl}}\langle K\rangle_{\mathsf{k}}\rangle_{\mathscr{P}}$.

The difference between finite and infinite pushdown systems is obvious when specifying the transition relation in $\mathbb{K}$. The main reason is that a $\mathbb{K}$ specification can contain a finite set of rules. Hence, while the transition relation for a finite pushdown system can be defined in $\mathbb{K}$ as it is, we need an abstraction mechanism for describing the transition relation for an infinite pushdown system. Below we explain the difference in detail.

The transition relation of a finite pushdown system is defined by computational rules in $\mathbb{K}$ as follows:

$$\text{For any } \ (\delta,\gamma) \hookrightarrow (\delta',\Gamma) \ \text{ we define the } \mathbb{K} \text{ rule } \langle \underbrace{\ \delta\ }_{\delta'} \rangle_{\mathsf{ctrl}} \langle \underbrace{\ \gamma\ }_{\Gamma} \cdots \rangle_{\mathsf{k}}.$$

Notational elements of $\mathbb{K}$ appearing in the above rule include the ellipses and local rewriting. The ellipses "$\cdots$" appearing by the walls of a cell represent some unspecified content of the cell. For example, the ellipses appearing in the cell $\langle\rangle_{\mathsf{k}}$ make γ the top of the continuation stack while the rest of the stack is encoded by $\cdots$. Note that in the above rule, δ represents the whole content of the control cell $\langle\rangle_{\mathsf{ctrl}}$. The local bi-dimensional rewrites trigger the local changes made to the configuration. As such, the local rewrite

from the cell $\langle\rangle_{\mathsf{k}}$, namely:

$$\langle \underset{\Gamma}{\underset{\downarrow}{\gamma}} \cdots \rangle_{\mathsf{k}}$$

represents a "pop" from the stack followed by a "push" in the stack, while the stack is maintained by the continuation cell $\langle\rangle_{\mathsf{k}}$. The corresponding rule in the associated transition system $\mathscr{T}_{\mathscr{P}}$ is $(\delta, \gamma\Gamma') \to (\delta', \Gamma\Gamma')$ which in $\mathbb{K}$ is represented as:

$$\frac{\langle\delta\rangle_{\mathsf{ctrl}}\,\langle\gamma \curvearrowright \Gamma'\rangle_{\mathsf{k}}}{\langle\delta'\rangle_{\mathsf{ctrl}}\,\langle\Gamma \curvearrowright \Gamma'\rangle_{\mathsf{k}}}$$

The $\mathbb{K}$ representation of $(\delta, \gamma\Gamma') \to (\delta', \Gamma\Gamma')$ in the transition system $\mathscr{T}_{\mathscr{P}}$ completes the configuration abstraction given by the local rewriting and the ellipses in the previously mentioned $\mathbb{K}$-rule:

$$\langle \underset{\delta'}{\underset{\downarrow}{\delta}} \rangle_{\mathsf{ctrl}} \langle \underset{\Gamma}{\underset{\downarrow}{\gamma}} \cdots \rangle_{\mathsf{k}}$$

up to the following standard rewrite rule:

$$\frac{\langle\delta\rangle_{\mathsf{ctrl}}\,\langle\gamma \curvearrowright \Gamma'\rangle_{\mathsf{k}}}{\langle\delta'\rangle_{\mathsf{ctrl}}\,\langle\Gamma \curvearrowright \Gamma'\rangle_{\mathsf{k}}}$$

Note that the standard rewrite rule is here written in the bi-dimentional format for notational uniformity. This rewrite rule is standardly represented as $\langle\delta\rangle_{\mathsf{ctrl}}\,\langle\gamma \curvearrowright \Gamma'\rangle_{\mathsf{k}} \Rightarrow \langle\delta'\rangle_{\mathsf{ctrl}}\,\langle\Gamma \curvearrowright \Gamma'\rangle_{\mathsf{k}}$, a notation available also in $\mathbb{K}$.

Example 4. *The pushdown system described in the Example 2 is defined in* $\mathbb{K}$ *by the following specification:*

$\Delta ::= d_0 \mid d_1 \mid d_2$

$\nabla ::= g_0 \mid g_1 \mid g_2 \mid g_3 \mid g_4$

The initial configuration is $\langle\langle d_0\rangle_{\mathsf{ctrl}}\langle g_3\, g_4\rangle_{\mathsf{k}}\rangle_{\mathscr{P}}$ *while the* $\mathbb{K}$ *rules defining the relation* $\hookrightarrow$ *are:*

$$\begin{array}{c|c|c}
\langle d_0\rangle_{\mathsf{ctrl}}\langle\ \underline{g_0}\ \cdots\rangle_{\mathsf{k}} & \langle d_1\rangle_{\mathsf{ctrl}}\langle\ \underline{g_0}\ \cdots\rangle_{\mathsf{k}} & \langle d_2\rangle_{\mathsf{ctrl}}\langle\ \underline{g_0}\ \cdots\rangle_{\mathsf{k}} \\
g_1\, g_0\, g_2 & g_1\, g_0\, g_3 & g_1\, g_0\, g_4 \\
\langle\ \underline{d_0}\ \rangle_{\mathsf{ctrl}}\langle\ \underline{g_1}\ \cdots\rangle_{\mathsf{k}} & \langle\ \underline{d_1}\ \rangle_{\mathsf{ctrl}}\langle\ \underline{g_1}\ \cdots\rangle_{\mathsf{k}} & \langle\ \underline{d_2}\ \rangle_{\mathsf{ctrl}}\langle\ \underline{g_1}\ \cdots\rangle_{\mathsf{k}} \\
d_1\quad \cdot & d_2\quad \cdot & d_1\quad \cdot \\
\langle d_0\rangle_{\mathsf{ctrl}}\langle\ \underline{g_2}\ \cdots\rangle_{\mathsf{k}} & \langle d_1\rangle_{\mathsf{ctrl}}\langle\ \underline{g_2}\ \cdots\rangle_{\mathsf{k}} & \langle d_2\rangle_{\mathsf{ctrl}}\langle\ \underline{g_2}\ \cdots\rangle_{\mathsf{k}} \\
\cdot & \cdot & \cdot \\
\langle d_0\rangle_{\mathsf{ctrl}}\langle\ \underline{g_3}\ \cdots\rangle_{\mathsf{k}} & \langle d_1\rangle_{\mathsf{ctrl}}\langle\ \underline{g_3}\ \cdots\rangle_{\mathsf{k}} & \langle d_2\rangle_{\mathsf{ctrl}}\langle\ \underline{g_3}\ \cdots\rangle_{\mathsf{k}} \\
g_0\, g_2 & \cdot & \cdot \\
\langle d_0\rangle_{\mathsf{ctrl}}\langle\ \underline{g_4}\ \cdots\rangle_{\mathsf{k}} & \langle d_1\rangle_{\mathsf{ctrl}}\langle\ \underline{g_4}\ \cdots\rangle_{\mathsf{k}} & \langle d_2\rangle_{\mathsf{ctrl}}\langle\ \underline{g_4}\ \cdots\rangle_{\mathsf{k}} \\
\cdot & \cdot & \cdot
\end{array}$$

Note that all the above rules are computational rules, namely tagged with attribute transition *in the* $\mathbb{K}$ *tool. This* $\mathbb{K}$ *specification is obviously behaviorally equivalent with the pushdown system from Example 2.*

The transition relation $\hookrightarrow$ is specified as a finite set of $\mathbb{K}$ rules. For defining the $\mathbb{K}$ rules, we identify a finite set of abstract control locations and stack letters $\breve{\delta}, \breve{\gamma}$ such that $\gamma_1 \breve{\cdot} \gamma_n = \breve{\gamma}_1 . . \breve{\gamma}_n$. We denote with $\breve{\Delta} = \{\breve{\delta}_i \mid i = 0..n\}$ and $\breve{\nabla} = \{\breve{\gamma}_j \mid j = 0..m\}$ the finite sets of abstract control states and stack symbols considered above, where $n, m \in \mathbb{N}$.

The choice for $\breve{\Delta}$ and $\breve{\nabla}$ should be made such that these are complete lattices (with some order relation $\sqsubseteq$) and there exist two Galois connections $(2^{\Delta}, \sharp, \$, \breve{\Delta})$, and $(2^{\nabla}, \sharp, \$, \breve{\nabla})$ such that $\cup\$(\breve{\Delta}) = \Delta$ and $\cup\$(\breve{\nabla}) = \nabla$. Obviously, there always exist such a choice. For

example, the basic one is given by $\breve{\Delta} = \{\emptyset, \Delta\}$ and $\breve{\nabla} = \{\emptyset, \nabla\}$. In the followings, for the "$\emptyset$" abstract element we use the $\mathbb{K}$ notation "$\cdot$", i.e., the unit element for any $\mathbb{K}$ type, including *Set*.

The complete lattice structure of $(\breve{\Delta} \times \breve{\nabla}, \sqsubseteq)$ can be specified in $\mathbb{K}$ as structural rules. We present two methods for this, depending on whether the lattice $(\breve{\Delta} \times \breve{\nabla}, \sqsubseteq)$ is formed using (*i*) the independent attribute method or (*ii*) the relational method:

(*i*) For any $\breve{\delta}_1, \breve{\delta}_2 \in \breve{\Delta}$ and any $\breve{\gamma}_1, \breve{\gamma}_2 \in \breve{\nabla}$

if $\breve{\delta}_1 \sqsubset \breve{\delta}_2$ then there exists a structural rule $\dfrac{\breve{\delta}_1}{\breve{\delta}_2}$, and

if $\breve{\gamma}_1 \sqsubset \breve{\gamma}_2$ then there exists a structural rule $\dfrac{\breve{\gamma}_1}{\breve{\gamma}_2}$.

(*ii*) For any $\breve{\delta}_1, \breve{\delta}_2 \in \breve{\Delta}$ and any $\breve{\gamma}_1, \breve{\gamma}_2 \in \breve{\nabla}$ such that $(\breve{\delta}_1, \breve{\gamma}_1) \sqsubset (\breve{\delta}_2, \breve{\gamma}_2)$

there exists a structural rule $\langle \dfrac{\breve{\delta}_1}{\breve{\delta}_2} \rangle_{\mathsf{ctrl}} \langle \cdots \dfrac{\breve{\gamma}_1}{\breve{\gamma}_2} \cdots \rangle_{\mathsf{k}}$.

Both these methods are described in Chapter 2, Section 2.2. Note however that these structural rules are to be applied only if no other (computational) rule can apply. Hence, the structural rules defining the lattice are to be understood as applying somehow, with the highest precedence, if $\mathbb{K}$ would permit strategies for rule application. Consequently, we present here only the theoretical general approach as the implementation would require features not yet present in the $\mathbb{K}$-tool. Meanwhile, in the current standard $\mathbb{K}$-specifications the structural rules are defined for terms which do not match any other computational rule (see for example the structural rule for while-loop in Chapter 2, Figure 2.1).

The $\mathbb{K}$ computational rules for the transition relation of a pushdown system are defined as:

For any $\check{\delta} \in \check{\Delta}, \check{\gamma} \in \check{\nabla}$, for all $\delta \in \Delta$ matching $\check{\delta}$, for all $\gamma \in \nabla$ matching $\check{\gamma}$,

if $(\delta, \gamma) \hookrightarrow (\delta', \Gamma)$ then there exists exactly one computational rule:

$$\langle \underbrace{\check{\delta}}_{\check{\delta}'} \rangle_{\mathsf{ctrl}} \langle \underbrace{\check{\gamma}}_{\check{\Gamma}} \cdots \rangle_{\mathsf{k}}$$

such that δ' matches $\check{\delta}'$, and Γ matches $\check{\Gamma}$.

Note that by "x matches $\check{x}$" we understand that $x \in \$(\check{x})$, i.e., if we associate to $\check{x}$ the set representation given by the concretization function \$, then the associative matching can be used to identify an element x in $\$(\check{x})$.

Observation 2. *For any two configurations* $\langle \check{\delta}_1 \rangle_{\mathsf{ctrl}} \langle \check{\gamma}_1 \rangle_{\mathsf{k}}$, $\langle \check{\delta}_2 \rangle_{\mathsf{ctrl}} \langle \check{\gamma}_2 \rangle_{\mathsf{k}}$ *such that* $(\check{\delta}_2, \check{\gamma}_2) \sqsubseteq (\check{\delta}_1, \check{\gamma}_1)$ *and any computational rule:*

$$\langle \underbrace{\check{\delta}_2}_{\check{\delta}'} \rangle_{\mathsf{ctrl}} \langle \underbrace{\check{\gamma}_2}_{\check{\Gamma}'} \cdots \rangle_{\mathsf{k}}$$

then the rule:

$$\langle \underbrace{\check{\delta}_1}_{\check{\delta}'} \rangle_{\mathsf{ctrl}} \langle \underbrace{\check{\gamma}_1}_{\check{\Gamma}'} \cdots \rangle_{\mathsf{k}}$$

is a valid computational rule.

Proof. The proof relies on the Galois connection and the definition of the computational

rules. Namely, by the definition of computational rules, the rule:

$$\langle\ \underbrace{\breve{\delta}_2}_{\breve{\delta}'}\ \rangle_{\mathsf{ctrl}}\langle\ \underbrace{\breve{\gamma}_2}_{\breve{\Gamma}'}\ \cdots\rangle_{\mathsf{k}}$$

represents a subset of $\hookrightarrow$, $\{(\delta_2,\gamma_2)\hookrightarrow(\delta',\Gamma') \mid (\delta_2,\gamma_2)\in\$(\breve{\delta}_2,\breve{\gamma}_2),(\delta',\Gamma')\in\$(\breve{\delta}',\breve{\Gamma}')\}$. Using the Galois connection, from the hypothesis $(\breve{\delta}_2,\breve{\gamma}_2)\sqsubseteq(\breve{\delta}_1,\breve{\gamma}_1)$ we can say that $\$(\breve{\delta}_2,\breve{\gamma}_2)\subseteq\$(\breve{\delta}_1,\breve{\gamma}_1)$. Hence, the rule:

$$\langle\ \underbrace{\breve{\delta}_1}_{\breve{\delta}'}\ \rangle_{\mathsf{ctrl}}\langle\ \underbrace{\breve{\gamma}_1}_{\breve{\Gamma}'}\ \cdots\rangle_{\mathsf{k}}$$

represents a subset of $\hookrightarrow$, $\{(\delta_2,\gamma_2)\hookrightarrow(\delta',\Gamma') \mid (\delta_2,\gamma_2)\in\$(\breve{\delta}_1,\breve{\gamma}_1),(\delta',\Gamma')\in\$(\breve{\delta}',\breve{\Gamma}')\}$.

Consequently, $\langle\ \underbrace{\breve{\delta}_1}_{\breve{\delta}'}\ \rangle_{\mathsf{ctrl}}\langle\ \underbrace{\breve{\gamma}_1}_{\breve{\Gamma}'}\ \cdots\rangle_{\mathsf{k}}$ is a valid computational rule. □

We denote by $k\mathscr{P}$ the described $\mathbb{K}$-definition $\langle\langle\breve{\Delta}\rangle_{\mathsf{ctrl}}\langle\breve{\nabla}\rangle_{\mathsf{k}}\rangle_{\mathscr{P}}$, which is an abstraction of the pushdown system $\mathscr{P}$. Namely, there exists a Galois connection between $2^{Conf(\mathscr{P})}$ and $Conf(k\mathscr{P})$, induced by the Galois connections $(2^{\Delta},\sharp,\$,\breve{\Delta})$ and $(2^{\nabla},\sharp,\$,\breve{\nabla})$. Moreover, we assume that the initial configuration in $k\mathscr{P}$ is the configuration $\langle\breve{\delta}_0\rangle_{\mathsf{ctrl}}\langle\breve{\Gamma}_0\rangle_{\mathsf{k}}$ where $\breve{\delta}_0$ and $\breve{\Gamma}_0$ are any of the smallest elements in $\breve{\Delta}$ and $\breve{\nabla}^*$, respectively, such that c_0 matches $(\breve{\delta}_0,\breve{\Gamma}_0)$.

We use Observation 2 to introduce the notion of "*implicit computational rule*" defined with a strictness-like attribute as follows:

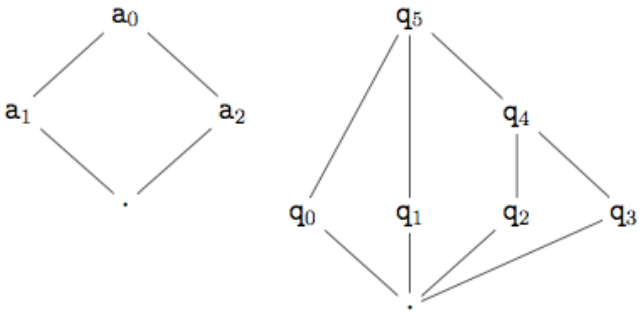
> A $\langle\breve{\delta}_1\rangle_{\mathsf{ctrl}}\langle\breve{\gamma}_1\ \cdots\rangle_{\mathsf{k}}$ is "*implicit*" when the computational rules triggered by $\langle\breve{\delta}_1\rangle_{\mathsf{ctrl}}\langle\breve{\gamma}_1\ \cdots\rangle_{\mathsf{k}}$ are obtained after zero or more steps of *heating* $\langle\breve{\delta}_1\rangle_{\mathsf{ctrl}}\langle\breve{\gamma}_1\ \cdots\rangle_{\mathsf{k}}$ into $\langle\breve{\delta}_2\rangle_{\mathsf{ctrl}}\langle\breve{\gamma}_2\ \cdots\rangle_{\mathsf{k}}$ such that $(\breve{\delta}_2,\breve{\gamma}_2)\sqsubseteq(\breve{\delta}_1,\breve{\gamma}_1)$. After the application of a computational rule triggered by $\langle\breve{\delta}_2\rangle_{\mathsf{ctrl}}\langle\breve{\gamma}_2\ \cdots\rangle_{\mathsf{k}}$, the *cooling* is produced instantaneously.

Example 5. *An abstract $\mathbb{K}$ representation of the pushdown system described in Example 2 can be defined in the traditional abstract interpretation style as:*

$$\breve{\Delta} ::= a_0 \mid a_1 \mid a_2$$
$$\breve{\nabla} ::= q_0 \mid q_1 \mid q_2 \mid q_3 \mid q_4 \mid q_5$$
$$K ::= \breve{\nabla}$$

$$\mathit{Init} \equiv \langle \langle a_2 \rangle_{\mathsf{ctrl}} \langle q_2\, q_3 \rangle_{\mathsf{k}} \rangle \mathscr{P}$$

Note that $\breve{\Delta}$ and $\breve{\nabla}$ are complete lattices, with the order relation provided in the following Hasse diagrams:



The $\mathbb{K}$ structural rules associated with the complete lattice $(\breve{\Delta} \times \breve{\nabla}, \sqsubseteq)$, according to the independent attribute method, are:

$$\frac{a_1}{a_0} \quad \frac{a_2}{a_0} \quad \frac{q_0}{q_5} \quad \frac{q_1}{q_5} \quad \frac{q_2}{q_4} \quad \frac{q_3}{q_4} \quad \frac{q_4}{q_5}$$

Hence, $(\langle \breve{\Delta} \rangle_{\mathsf{ctrl}} \langle \breve{\nabla} \rangle_{\mathsf{k}}, \frac{}{})$ is a complete lattice and we provide the Galois connec-

tion $(2^{\Delta\times\nabla}, \sharp, \$, \langle\breve{\Delta}\rangle_{\mathsf{ctrl}}\langle\breve{\nabla}\rangle_{\mathsf{k}})$ *by defining the concretization function as:*

$$\begin{array}{lll} & \$(\cdot) = \emptyset; & \\ \$(\mathfrak{a}_0) = \{d_0, d_1, d_2\}; & \$(\mathfrak{a}_1) = \{d_1, d_2\}; & \$(\mathfrak{a}_2) = \{d_0\}; \\ \$(\mathfrak{q}_0) = \{g_0\}; & \$(\mathfrak{q}_1) = \{g_1\}; & \$(\mathfrak{q}_2) = \{g_3\}; \\ \$(\mathfrak{q}_3) = \{g_2, g_4\}; & \$(\mathfrak{q}_4) = \{g_2, g_3, g_4\}; & \$(\mathfrak{q}_5) = \{g_0, g_1, g_2, g_3, g_4\}; \end{array}$$

$$\$(\langle a : \breve{\Delta}\rangle_{\mathsf{ctrl}}\langle q : \breve{\nabla}\rangle_{\mathsf{k}}) = (\$(a), \$(q)).$$

To show that $(2^{\Delta\times\nabla}, \sharp, \$, \langle\breve{\Delta}\rangle_{\mathsf{ctrl}}\langle\breve{\nabla}\rangle_{\mathsf{k}})$ *is a Galois connection is enough to prove that* $\$$ *is completely multiplicative in each cell,* $\langle\rangle_{\mathsf{ctrl}}$ *and* $\langle\rangle_{\mathsf{k}}$*. Hence, it is enough to prove the following two equlities:*

$$\$(a : \breve{\Delta} \sqcap a' : \breve{\Delta}) = \$(a) \cap \$(a') \text{ and } \$(q : \breve{\nabla} \sqcap q' : \breve{\nabla}) = \$(q) \cap \$(q')$$

A case reasoning shows that the two equalities hold. We exemplify here one case for each equality:

$$\$(\mathfrak{a}_0 \sqcap \mathfrak{a}_1) = \$(\mathfrak{a}_1) = \{d_1, d_2\} = \{d_0, d_1, d_2\} \cap \{d_1, d_2\} = \$(\mathfrak{a}_0) \cap \$(\mathfrak{a}_1)$$

$$\$(\mathfrak{q}_0 \sqcap \mathfrak{q}_4) = \$(\cdot) = \emptyset = \{g_0\} \cap \{g_2, g_3, g_4\} = \$(\mathfrak{q}_0) \cap \$(\mathfrak{q}_4)$$

Consequently, the abstraction function $\sharp$ *forming, together with* $\$$*, the Galois connection is defined as:*

$$\sharp(C : 2^{\Delta\times\nabla}) = \sqcap\{\langle a : \breve{\Delta}\rangle_{\mathsf{ctrl}}\langle q : \breve{\nabla}\rangle_{\mathsf{k}} \mid C \subseteq \$(\langle a\rangle_{\mathsf{ctrl}}\langle q\rangle_{\mathsf{k}})\}$$

Note that $(2^{\Delta\times\nabla}, \sharp, \$, \langle\breve{\Delta}\rangle_{\mathsf{ctrl}}\langle\breve{\nabla}\rangle_{\mathsf{k}})$ *is actually a Galois insertion because, e.g.,* $\$$ *is an injection. Also, the Galois insertion can be extended to* $(2^{Conf(\mathscr{P})}, \sharp, \$, \langle\breve{\Delta}\rangle_{\mathsf{ctrl}}\langle K\rangle_{\mathsf{k}})$

using the concretization function as it follows:

$$\$(\langle a : \breve{\Delta}\rangle_{\mathsf{ctrl}}\langle q_1 \curvearrowright .. \curvearrowright q_n : K\rangle_{\mathsf{k}}) = (\$(a), \$(q_1)..\$(q_n)),\ n \in \mathbb{N}.$$

Finally, the computational rules are:

$$\text{RULE}\quad \langle a_0\rangle_{\mathsf{ctrl}}\langle \underset{q_1\, q_0\, q_4}{q_0} \cdots\rangle_{\mathsf{k}} \quad [\text{row1}]$$

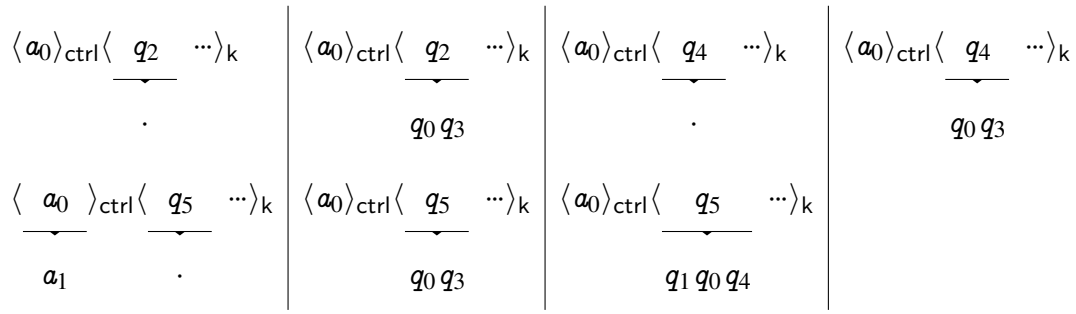
$$\text{RULE}\quad \langle \underset{a_1}{a_0} \rangle_{\mathsf{ctrl}}\langle \underset{\cdot}{q_1} \cdots\rangle_{\mathsf{k}} \quad [\text{row2}]$$

$$\text{RULE}\quad \langle a_0\rangle_{\mathsf{ctrl}}\langle \underset{\cdot}{q_3} \cdots\rangle_{\mathsf{k}} \quad [\text{row3,5}]$$

$$\text{RULE}\quad \langle a_1\rangle_{\mathsf{ctrl}}\langle \underset{\cdot}{q_2} \cdots\rangle_{\mathsf{k}} \quad [\text{row4.col2}-3]$$

$$\text{RULE}\quad \langle a_2\rangle_{\mathsf{ctrl}}\langle \underset{q_0\, q_3}{q_2} \cdots\rangle_{\mathsf{k}} \quad [\text{row4.col1}]$$

Note that we prefer to keep the computational rules minimal. Namely, according to the generic structure of the computational rules for the abstract pushdown systems, the specification has to be completed with the next computational rules for $\langle a_0\rangle_{\mathsf{ctrl}}\langle q_{2,4,5} \cdots\rangle_{\mathsf{k}}$*:*



Instead, we prefer to declare $\langle a_0 \rangle_{\mathsf{ctrl}} \langle q_{2,4,5} \cdots \rangle_{\mathsf{k}}$ *as "*implicit*". This means that whenever the computation reaches one of these ground configurations, the heating mechanism is triggered and the configuration is heated into a lower order configuration, where the order is the one considered for the complete lattice* $(\langle \breve{\Delta} \rangle_{\mathsf{ctrl}} \langle \breve{\nabla} \rangle_{\mathsf{k}}, \_\_\_\_\_)$.

Being in a rewriting environment, the abstract states which form $\breve{\Delta}$ and $\breve{\nabla}$ are naturally described in $\mathbb{K}$ using patterns, given by operators and variables. We exemplify this claim using the following example which is an alternative abstraction to the one proposed in Example 5:

Example 6. *Another abstraction of the pushdown system described in Example 2, more natural for a rewriting environment, can be defined in* $\mathbb{K}$ *as:*

$IndexD ::= 0 \mid 1 \mid 2$

$IndexG ::= IndexD \mid 3 \mid 4$

$\breve{\Delta} ::= d(\ IndexD\ ) \mid d(\odot)$

$\breve{\nabla} ::= g(\ IndexG\ ) \mid g(\odot)$

$Init \equiv \langle \langle d(0) \rangle_{\mathsf{ctrl}} \langle g(3)\, g(4) \rangle_{\mathsf{k}} \rangle_{\mathscr{P}}$

$$\frac{d(0)}{d(\odot)} \quad \frac{d(1)}{d(\odot)} \quad \frac{d(2)}{d(\odot)}$$

$$\frac{g(0)}{g(\odot)} \quad \frac{g(1)}{g(\odot)} \quad \frac{g(2)}{g(\odot)} \quad \frac{g(3)}{g(\odot)} \quad \frac{g(4)}{g(\odot)}$$

RULE $\langle d(i : IndexD) \rangle_{\text{ctrl}} \langle \underline{\quad g(0) \quad} \cdots \rangle_{\text{k}}$ [row1]

$g(1)\,g(0)\,g(i+2)$

RULE $\langle \underline{\quad d(i : IndexD) \quad} \rangle_{\text{ctrl}} \langle \underline{g(1)} \cdots \rangle_{\text{k}}$ [row2]

$d(i \bmod (2)+1)$ $\qquad \cdot$

RULE $\langle d(i : IndexD) \rangle_{\text{ctrl}} \langle \underline{g(j : IndexG)} \cdots \rangle_{\text{k}}$

$\cdot$

when $j ==_{IndexG} 2 \; or_{Bool} \; j >=_{IndexG} 4$ [row3,5]

RULE $\langle d(i : IndexD) \rangle_{\text{ctrl}} \langle \underline{g(3)} \cdots \rangle_{\text{k}}$

$\cdot$

when $i =/=_{IndexD} 0$ [row4.col2,3]

RULE $\langle d(0) \rangle_{\text{ctrl}} \langle \underline{\quad g(3) \quad} \cdots \rangle_{\text{k}}$ [row4.col1]

$g(0)\,g(3)$

In this case, the only "strict" *configuration is* $\langle d(\odot) \rangle_{\text{ctrl}} \langle g(\odot) \cdots \rangle_{\text{k}}$.

Note that using this representation of the abstract states, via variables i and j, we can afford a more precise definition of the abstraction using the `+2` *and* `mod(2)+1` *operations defined over IndexG and IndexD, respectively. Actually, this abstraction is*

bisimilar with the pushdown system described in Example 2, and the $\mathbb{K}$ specification described in 4. The relation giving the bisimulation associates d_i, $i \in \{0,1,2\}$, with $\mathtt{d}(\, i : IndexD\,)$, and g_j, $j \in \{0,1,2,3,4\}$, with $\mathtt{g}(\, j : IndexG\,)$.

We denote by $\mathscr{T}_{k\mathscr{P}}$ the transition system defined by $k\mathscr{P}$. Next, we give the theoretical result which relates $\mathscr{P}$ with $k\mathscr{P}$.

Proposition 1. *$\mathscr{T}_{k\mathscr{P}}$ simulates $\mathscr{T}_{\mathscr{P}}$.*

Proof. We define the relation $\rho \subseteq \mathscr{T}_{\mathscr{P}} \times \mathscr{T}_{k\mathscr{P}}$ as $(\delta,\Gamma)\ \rho\ \langle \check{\delta}\rangle_{\mathsf{ctrl}}\langle \check{\Gamma}\rangle_{\mathsf{k}}$ *iff* δ matches $\check{\delta}$ and Γ matches $\check{\Gamma}$, for any $(\delta,\Gamma) \in Conf(\mathscr{P})$, and any $\langle \check{\delta}\rangle_{\mathsf{ctrl}}\langle \check{\Gamma}\rangle_{\mathsf{k}}$ a ground configuration of $k\mathscr{P}$. We prove that ρ is a simulation.

We consider $(\delta,\Gamma) \in Conf(\mathscr{P})$ and $\langle \check{\delta}\rangle_{\mathsf{ctrl}}\langle \check{\Gamma}\rangle_{\mathsf{k}}$ a ground configuration of $k\mathscr{P}$ such that $(\delta,\Gamma)\ \rho\ \langle \check{\delta}\rangle_{\mathsf{ctrl}}\langle \check{\Gamma}\rangle_{\mathsf{k}}$. Assume that $(\delta,\Gamma) \to (\delta',\Gamma')$ is a transition in $\mathscr{T}_{\mathscr{P}}$. Then, by the definition of $\mathscr{T}_{\mathscr{P}}$, exists $\gamma \in \Delta$ and Γ'' such that $\Gamma = \gamma\Gamma''$ and exists a pushdown system rule $(\delta,\gamma) \hookrightarrow (\delta',\Gamma''')$ such that $\Gamma' = \Gamma'''\Gamma''$.

Since $(\delta,\Gamma)\ \rho\ \langle \check{\delta}\rangle_{\mathsf{ctrl}}\langle \check{\Gamma}\rangle_{\mathsf{k}}$ then δ matches $\check{\delta}$ and Γ matches $\check{\Gamma}$. But $\check{\Gamma} = \check{\gamma}_0 \curvearrowright .. \curvearrowright \check{\gamma}_l$, with $l \in \mathbb{N}$, so Γ matches $\check{\gamma}_0 \curvearrowright .. \curvearrowright \check{\gamma}_l$, hence $\gamma\Gamma''$ matches $\check{\gamma}_0 \curvearrowright .. \curvearrowright \check{\gamma}_l$ which means that γ matches $\check{\gamma}_0$ and Γ'' matches $\check{\gamma}_1 \curvearrowright .. \curvearrowright \check{\gamma}_l$.

From the definition of $k\mathscr{P}$, exists a computational rule:

$$\langle\ \underbrace{\check{\delta}}_{\check{\delta}'}\ \rangle_{\mathsf{ctrl}}\langle\ \underbrace{\check{\gamma}_0}_{\check{\Gamma}_0}\ \cdots\rangle_{\mathsf{k}}$$

where $\langle \check{\delta}'\rangle_{\mathsf{ctrl}}\langle \check{\Gamma}_0\rangle_{\mathsf{k}}$ is a ground configuration in $k\mathscr{P}$ such that δ' matches $\check{\delta}'$ and Γ''' matches $\check{\Gamma}_0$. Recall that the ellipses in the cel $\langle\rangle_{\mathsf{k}}$ denote the remaining of the term representing the contents of the cell $\langle\rangle_{\mathsf{k}}$. Hence, when the ellipses are substituted by

$\breve{\gamma}_1 \curvearrowright .. \curvearrowright \breve{\gamma}_l$, the above computational rule is instantiated with:

$$\frac{\langle \breve{\delta} \rangle_{\mathsf{ctrl}} \langle \breve{\gamma}_0 \curvearrowright \breve{\gamma}_1 \curvearrowright .. \curvearrowright \breve{\gamma}_l \rangle_{\mathsf{k}}}{\langle \breve{\delta}' \rangle_{\mathsf{ctrl}} \langle \breve{\Gamma}_0 \curvearrowright \breve{\gamma}_1 \curvearrowright .. \curvearrowright \breve{\gamma}_l \rangle_{\mathsf{k}}}$$

According to the semantics of $\mathbb{K}$, this is a transition in the transition system $\mathscr{T}_{k\mathscr{P}}$. Since $\breve{\Gamma} = \breve{\gamma}_0 \curvearrowright .. \curvearrowright \breve{\gamma}_l$, this transitions can be rewritten as:

$$\frac{\langle \breve{\delta} \rangle_{\mathsf{ctrl}} \langle \breve{\Gamma} \rangle_{\mathsf{k}}}{\langle \breve{\delta}' \rangle_{\mathsf{ctrl}} \langle \breve{\Gamma}_0 \curvearrowright \breve{\gamma}_1 \curvearrowright .. \curvearrowright \breve{\gamma}_l \rangle_{\mathsf{k}}}$$

Now we are left to prove that $(\delta', \Gamma')\ \rho\ \langle \breve{\delta}' \rangle_{\mathsf{ctrl}} \langle \breve{\Gamma}_0 \curvearrowright \breve{\gamma}_1 \curvearrowright .. \curvearrowright \breve{\gamma}_l \rangle_{\mathsf{k}}$. We already have that δ' matches $\breve{\delta}'$, so we have to show that Γ' matches $\breve{\Gamma}_0 \curvearrowright \breve{\gamma}_1 \curvearrowright .. \curvearrowright \breve{\gamma}_l$. But $\Gamma' = \Gamma'''\Gamma''$, Γ''' matches $\breve{\Gamma}_0$ and Γ'' matches $\breve{\gamma}_1 \curvearrowright .. \curvearrowright \breve{\gamma}_l$. So $\Gamma'''\Gamma''$ matches $\breve{\Gamma}_0 \curvearrowright \breve{\gamma}_1 \curvearrowright .. \curvearrowright \breve{\gamma}_l$.

Hence, exists a transition in $\mathscr{T}_{k\mathscr{P}}$, namely:

$$\frac{\langle \breve{\delta} \rangle_{\mathsf{ctrl}} \langle \breve{\Gamma} \rangle_{\mathsf{k}}}{\langle \breve{\delta}' \rangle_{\mathsf{ctrl}} \langle \breve{\Gamma}_0 \curvearrowright \breve{\gamma}_1 \curvearrowright .. \curvearrowright \breve{\gamma}_l \rangle_{\mathsf{k}}}$$

such that $(\delta', \Gamma')\ \rho\ \langle \breve{\delta}' \rangle_{\mathsf{ctrl}} \langle \breve{\Gamma}_0 \curvearrowright \breve{\gamma}_1 \curvearrowright .. \curvearrowright \breve{\gamma}_l \rangle_{\mathsf{k}}$. So ρ is a simulation. □

Example 7. a) Infinite Case:

We consider an infinite pushdown system $\bar{\mathscr{P}} = (\Delta, \nabla, \hookrightarrow)$ *with the set of control locations* $\Delta = \{d_i \mid i \in \mathbb{N}\}$ *and the stack alphabet as* $\nabla = \{g_j \mid j \in \mathbb{N}\}$. *The* $\hookrightarrow$ *relation is produced by bordering the relation from Example 2 with the sets:*

- $\{(d_i, g_0) \hookrightarrow (d_i, g_1 g_0 g_{i+2}) \mid i \in \mathbb{N}, i > 2\}$;

- $\{(d_i, g_1) \hookrightarrow (d_{i \bmod(2)+1}, \varepsilon) \mid i \in \mathbb{N}, i > 2\}$;

- $\{(d_i, g_j) \hookrightarrow (d_i, \varepsilon) \mid i, j \in \mathbb{N}, i > 2, j \geq 2\}$.

Now, let us observe that the $\mathbb{K}$ specification given in Example 5 is an abstraction of the current pushdown system, as well. The only difference is made in how the concetization function is defined, namely:

$$\$(\cdot) = \emptyset;$$

$$\$(a_0) = \{d_i \mid i \in \mathbb{N}\}; \quad \$(a_1) = \{d_i \mid i \in \mathbb{N}, i > 0\}; \quad \$(a_2) = \{d_0\};$$

$$\$(q_0) = \{g_0\}; \quad \$(q_1) = \{g_1\}; \quad \$(q_2) = \{g_3\};$$

$$\$(q_3) = \{g_2, g_j \mid j \in \mathbb{N}, j > 3\}; \quad \$(q_4) = \{g_j \mid j \in \mathbb{N}, j > 2\}; \quad \$(q_5) = \{g_j \mid j \in \mathbb{N}\};$$

Meanwhile, the $\mathbb{K}$ specification given in Example 6 has to be modified. Namely, the rules can be preserved but the signature has to be changed. For example the change can be made as it follows:

IndexD ::= *Nat*

IndexG ::= *Nat*

$\breve{\Delta}$ ::= `d(` *IndexD* `)` | `d(`$\odot$`)`

$\breve{\nabla}$ ::= `g(` *IndexG* `)` | `g(`$\odot$`)`

This specification produces a pushdown system bisimilar with $\bar{\mathscr{P}}$. However, the pushdown system specified like this is not finite. Another bisimilar but finite pushdown system is specified by, again, the same set of rules over the signature:

IndexD ::= `0` | `1` | `2` | `3`

IndexG ::= *IndexD* | `4` | `5`

$\breve{\Delta}$ ::= `d(` *IndexD* `)` | `d(`$\odot$`)`

$\breve{\nabla}$ ::= `g(` *IndexG* `)` | `g(`$\odot$`)`

b) Nondeterministic Case:

A nondeterministic pushdown system $\tilde{\mathscr{P}}$ can be given by adding $(d_0, g_3) \hookrightarrow (d_0, \varepsilon)$

to the relation $\hookrightarrow$ from $\tilde{\mathscr{P}}$.

In order to adjust the $\mathbb{K}$ specification in Example 5 to be an abstraction of $\tilde{\mathscr{P}}$ we need to replace the rule row4.col2 – 3 *with the following computational rule:*

$$\text{RULE} \quad \langle a_0 \rangle_{\mathsf{ctrl}} \langle \underline{\ q_2\ } \cdots \rangle_{\mathsf{k}} \quad \text{[row4.col1-]}$$

.

Similarly, the changes on the $\mathbb{K}$ specification in Example 6 consist in replacing the syntax declaration as in the case **a)** *of the current example and eliminating the condition of the rule* row4.col2, 3*:*

$$\text{RULE} \quad \langle d(i : \mathit{IndexD}\ ) \rangle_{\mathsf{ctrl}} \langle \underline{g(3)} \cdots \rangle_{\mathsf{k}} \quad \text{[row4.col1-]}$$

.

In the following sections of this chapter we assume that $k\mathscr{P}$ is a $\mathbb{K}$ specification of a *finite* pushdown system.

3.2 Collecting Semantics, Pushdown Systems, and $\mathbb{K}$

"The Master said, 'A true teacher is one who,
keeping the past alive, is also able to understand the present.' "
— Confucius.

The setting proposed in the current section aims to solve the analysis/verification problem $\mathscr{P} \models \Phi$ considering a $\mathbb{K}$ specification for the property Φ. We give next a succinct view of how $\mathscr{P} \models \Phi$ is achieved by this setting.

Firstly, collecting semantics is a *property directed execution* which stores the information collected along the execution. Among collecting semantics applications we

mention: program transformation [27, 54], malware detection [75], model checking [19], and so on. All in all, we can state that collecting semantics is a methodology introduced by abstract interpretation for analysis and verification methods. However, the effectiveness of collecting semantics varies according to the targeted property. For example, model checking is decidable on finite systems but may, in practice, be characterized by combinatorial explosion.

Secondly, abstraction reduces the size of a system in order to improve the chances of an effective analysis/verification process. However, in many cases this reduction introduces additional behaviors and leads to an over-approximation of the system. Abstract interpretation makes precise the fact that the analysis/verfication method can be transported from abstract into concrete only if the abstraction is sound w.r.t. the property of interest. For example, a behaviorally equivalent abstraction is sound w.r.t. any property as long as the property can be equivalently expressed in both concrete and abstract.

In the current section we combine these two ideas to present a generic scheme for $\mathbb{K}$ specifications of collecting semantics. Namely, we present how collecting semantics works in general over $k\mathscr{P}$. In the semantic entailment notation this means that we give the generic $\mathbb{K}$ specification for $k\mathscr{P} \models \Phi$. Note that we use a generic Φ which is passed as a parameter to collecting semantics. Further, assuming that $k\mathscr{P}$ is a sound abstraction of $\mathscr{P}$ w.r.t. Φ, we infer $\mathscr{P} \models \Phi$ from $k\mathscr{P} \models \Phi$.

3.2.1 Why pushdown systems for collecting semantics?

The analysis/verification methods for *finite transition systems* are semantically unified under the collecting semantics in [24]. However, the transition system equivalent to a pushdown system is inherently infinite due to the potential unbounded growth of the stack. Still, there exist well known decidability results for analysis and verification of finite pushdown systems. These results make possible to apply collecting semantics directly on pushdown systems without going through the (finite) transition system

modeling.

The decidability results for finite pushdown systems basically rely on discovering "stack-based loops" in any computation. For example, based on Corollary 5.1. from [14], if there exists an infinite path in $\mathscr{T}_{\mathscr{P}}$, then this is lasso shaped, i.e., there is a prefix of this path that ends in a loop. Namely, an infinite path presents a repetitive stack pattern as follows:

$$c_0 \xrightarrow{*} (\delta, \gamma Y) \xrightarrow{w} (\delta, \gamma XY) \xrightarrow{w} (\delta, \gamma XXY)...$$

Moreover, any infinite path (i.e., lasso shaped) is characterized by a finite prefix (i.e., repetitive) as $c_0 \xrightarrow{*} (\delta, \gamma Y) \xrightarrow{w} (\delta, \gamma X^r Y)$, where X^r is a new term from ∇^* such that $(\delta, \gamma X^r Y) = (\delta, \gamma XY)$. We observe that this characterization is based on the regular structure of the stack which forms the basis of the termination property.

Example 8. *Considering the pushdown system $\mathscr{P}$ described in Example 2, a lasso shape infinite computation is the next one:*

$$\begin{aligned}
&(d_0, g_3 g_4) \rightarrow \\
&(d_0, g_0 g_2 g_4) \rightarrow (d_0, g_1 g_0 g_2 g_2 g_4) \rightarrow \\
&(d_0, g_0 g_2 g_2 g_4) \rightarrow (d_0, g_1 g_0 g_2 g_2 g_2 g_4) \rightarrow \\
&(d_0, g_0 g_2 g_2 g_2 g_4) \dots
\end{aligned}$$

Hence, in this case γ is g_0, Y is $g_2 g_4$, and X is g_2, while the repetitive head is:

$$(d_0, g_3 g_4) \rightarrow (d_0, g_0 g_2 g_4) \rightarrow (d_0, g_1 g_0 g_2 g_2 g_4) \rightarrow (d_0, g_0 g_2^r g_2 g_4)$$

3.2.2 How collecting semantics in $\mathbb{K}$?

Considering the $\mathbb{K}$ encoding of a pushdown system, we aim to develop a technique for reusing the $\mathbb{K}$-definition to obtain the analysis/verification method. We rely on the

methodology given by the collecting semantics, using a forward fixpoint iteration. As such, the configuration for the collecting semantics for $k\mathscr{P}$, denoted $\check{k}\mathscr{P}$, is defined as:

$$\langle\langle\langle\check{\Delta}\rangle_{\mathsf{ctrl}}\langle K\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}*}\rangle_{\mathsf{traces}}\ \langle\langle\langle\check{\Delta}\rangle_{\mathsf{ctrl}}\langle K\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}*}\rangle_{\mathsf{traces}'}\ \langle Bag\rangle_{\mathsf{collect}}$$

where *Bag* is the predefined $\mathbb{K}$ sort.

The cells traces and traces$'$ are meant to guide the rewriting in order to obtain a breadth-first exhaustive execution. As such, traces cell contains the current execution level, while in traces$'$ cell we construct the next level. The breadth-first strategy is tantamount imposing fairness in the application of rewrite rules. As usual, we need the fairness condition to ensure the monotonicity of the fixpoint iteration. The content of the collect cell is customized to maintain the property Φ and the infrastructure to reach the desired outcome of the analysis/verification method. This infrastructure typically consists of an abstract representation of all the computations from $\mathscr{P}$.

The initial state in $\check{k}\mathscr{P}$ fills the traces cell with the initial state in $k\mathscr{P}$ as the starting point of the analysis/verification method. The traces$'$ cell is left empty while the collect cell contains the initial settings of the method, such as the property Φ or an implicit form of it.

To obtain the rules for the collecting semantics, we first identify and group the rules in $k\mathscr{P}$ as follows:

> For each $\check{\delta} \in \check{\Delta}$ and $\check{\gamma} \in \check{\nabla}$, consider all configurations $\langle\check{\delta}'\rangle_{\mathsf{ctrl}}\langle\check{\gamma}'\rangle_{\mathsf{k}}$ such that $\langle\check{\delta}'\rangle_{\mathsf{ctrl}}\langle\check{\gamma}'\rangle_{\mathsf{k}}$ is reached from $\langle\check{\delta}\rangle_{\mathsf{ctrl}}\langle\check{\gamma}\rangle_{\mathsf{k}}$ either via zero or more structural rules, or via heating mechanism. This means that $(\check{\delta},\check{\gamma}) \sqsubseteq (\check{\delta}',\check{\gamma}')$ in the lattice $(\check{\Delta}\times\check{\nabla},\sqsubseteq)$. We consider all the rules:

$$\langle\ \underbrace{\check{\delta}'}_{\check{\delta}_i}\ \rangle_{\mathsf{ctrl}}\langle\ \underbrace{\check{\gamma}'}_{\check{\Gamma}_i}\ \cdots\rangle_{\mathsf{k}}$$

$$\text{RULE}\quad \langle \cdots \underline{\langle\langle \breve{\delta} \rangle_{\mathsf{ctrl}} \langle \breve{\gamma} \curvearrowright \breve{\Gamma} \rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}}_{\cdot} \cdots \rangle_{\mathsf{traces}} \langle \cdots \underbrace{\cdot}_{\mathtt{post}\ (\langle \breve{\delta} \rangle_{\mathsf{ctrl}} \langle \breve{\gamma} \curvearrowright \breve{\Gamma} \rangle_{\mathsf{k}})} \cdots \rangle_{\mathsf{traces}'}$$

$$\langle \underbrace{\mho}_{\mathtt{update}(\,\mho, \langle \breve{\delta} \rangle_{\mathsf{ctrl}}\ \langle \breve{\gamma} \curvearrowright \breve{\Gamma} \rangle_{\mathsf{k}}\,)} \rangle_{\mathsf{collect}}$$

$$\text{when}\quad \mho \prec \mathtt{update}(\,\mho, \langle \breve{\delta} \rangle_{\mathsf{ctrl}}\ \langle \breve{\gamma} \curvearrowright \breve{\Gamma} \rangle_{\mathsf{k}}\,) \qquad [\text{unfold}]$$

$$\text{RULE}\quad \langle \cdots \underline{\langle\langle \breve{\delta} \rangle_{\mathsf{ctrl}} \langle \breve{\gamma} \curvearrowright \breve{\Gamma} \rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}}_{\cdot} \cdots \rangle_{\mathsf{traces}} \langle \mho \rangle_{\mathsf{collect}}$$

$$\text{when}\quad \mho \not\prec \mathtt{update}(\,\mho, \langle \breve{\delta} \rangle_{\mathsf{ctrl}}\ \langle \breve{\gamma} \curvearrowright \breve{\Gamma} \rangle_{\mathsf{k}}\,) \qquad [\text{prune}]$$

Figure 3.1: The rules for the $\mathbb{K}$ specification of collecting semantics over $k\mathscr{P}$.

with $0 \leq i \leq n$, where $n \in \mathbb{N}$ is the branching factor for $\langle \breve{\delta} \rangle_{\mathsf{ctrl}} \langle \breve{\gamma} \rangle_{\mathsf{k}}$, and denote the set $\{\langle \breve{\delta}_i \rangle_{\mathsf{ctrl}}\ \langle \breve{\Gamma}_i \rangle_{\mathsf{k}} \mid 0 \leq i \leq n\}$ with $\mathtt{post}\ (\langle \breve{\delta} \rangle_{\mathsf{ctrl}} \langle \breve{\gamma} \rangle_{\mathsf{k}})$.

We extend the $\mathtt{post}$ operator over $\breve{\mathbb{V}}^+$ as follows:

$$\mathtt{post}\ (\langle \breve{\delta} \rangle_{\mathsf{ctrl}} \langle \breve{\gamma} \curvearrowright \breve{\Gamma} \rangle_{\mathsf{k}}) = \{\langle \breve{\delta}_i \rangle_{\mathsf{ctrl}}\ \langle \breve{\Gamma}_i \curvearrowright \breve{\Gamma} \rangle_{\mathsf{k}} \mid 0 \leq i \leq n\}$$

We use the $\mathtt{post}$ operator in the $\mathbb{K}$ rules from Figure 3.1, where $\prec \subseteq \mathit{Bag} \times \mathit{Bag}$ is a well-founded relation over the contents of the collect cell. Note that the set defined by $\mathtt{post}\ (\langle \breve{\delta} \rangle_{\mathsf{ctrl}} \langle \breve{\gamma} \curvearrowright \breve{\Gamma} \rangle_{\mathsf{k}})$ can be obtained with a search command as:

$$\mathtt{search}\ :\ \langle \langle \breve{\delta} \rangle_{\mathsf{ctrl}}\ \langle \breve{\gamma} \curvearrowright \breve{\Gamma} \rangle_{\mathsf{k}} \rangle_{\mathscr{P}}\ \ \mathtt{=>1}\ \ \langle \langle D : \breve{\Delta} \rangle_{\mathsf{ctrl}}\ \langle K : K \rangle_{\mathsf{k}} \rangle_{\mathscr{P}}$$

The collecting rules mimic a shared memory model, where the cell $\langle \rangle_{\mathsf{collect}}$ is the shared memory. The first rule, labeled unfold, encodes the exhaustive step of execution of $k\mathscr{P}$ by consuming a current computation trace from traces and producing new

computation traces in cell $\langle\rangle_{\mathsf{traces}'}$, provided that this step increases the contents of cell $\langle\rangle_{\mathsf{collect}}$. The second collecting rule, labeled prune, covers the case when the currently selected trace is not increasing the content of the collect cell, based on the given `update` operation. We left yet unspecified the contents of the collect cell, because its structure and update operation depend on the targeted analysis/verification method.

Note that the property Φ does not appear explicitly in the rules in Figure 3.1. Instead, we consider Φ embedded in the collect cell. As such υ contains the current form of Φ and the `update` operator may modify it or not, depending on the implemented collecting semantics. For example, usually analysis methods do not need to update the form of Φ, while verification methods as model checking have to update Φ according to the temporal operators forming the property.

The switch to the next level of computations in the breadth-first exhaustive execution is made by the structural rule:

$$\text{RULE}\quad \frac{\langle\ \cdot\ \rangle_{\mathsf{traces}}\ \langle \mathit{NextLevel} : \mathit{Bag}\rangle_{\mathsf{traces}'}}{\mathtt{bag2set}(\mathit{NextLevel})\ \cdot}\quad \text{when}\ \ \mathit{NextLevel} \neq_{\mathit{Bag}} \cdot \qquad [\text{switch2next}]$$

where the operator `bag2set` eliminates from *NextLevel* the trace cells with an empty cell $\langle\rangle_{\mathsf{k}}$, and keeps only one representative from the trace cells with the same content.

The termination of the "exhaustive" execution of $\mathscr{P}$ designed in $k\mathscr{P}$ is ensured by the well-foundedness of the relation $\prec$ and the "fairness" mechanism introduced by the breadth-first-like strategy. As previously mentioned, the infinite computations in pushdown systems present a repetitive pattern given by the lasso shape. The cell $\langle\rangle_{\mathsf{collect}}$ is basically formed by various representations or abstractions of the computations prefixes displayed by the exhaustive breadth-first execution of $k\mathscr{P}$. Hence, the computation traces collected in the collect cell can pivot on the lasso shape, and stop

the `update` as soon as a particular computation identifies a lasso.

With this design for the content of the collect cell, the relation $\prec$ can be simply inclusion because the `update` operation ensures the well-foundedness of the inclusion. As such, the collecting computation terminates due to the fact that the computations in $\mathscr{P}$ are either finite, or reduced to finite prefixes. Note however that we can ensure this in the case when the control locations and the stack alphabet are finite. In the case when we deal with abstractions of infinite pushdown systems, the reasoning is similar but is made for the abstract pushdown system which has finite control locations and stack alphabet.

Theorem 1. *The computation in $\check{k}\mathscr{P}$ is finite.*

Proof. The computation steps in $\check{k}\mathscr{P}$ are triggered by the trace cells, either from traces or from traces$'$ cells. The rules in Figure 3.1 rely on a nonempty cell $\langle\rangle_{\mathsf{traces}}$ to deposit new trace cells in the next level of the breadth first unfolding handled by the cell $\langle\rangle_{\mathsf{traces}'}$. Meanwhile, the rule switch2next refills an empty cell $\langle\rangle_{\mathsf{traces}}$ with the nonempty contents of traces$'$ cell.

First we analyze each rule to check whether it can be a source of infiniteness. The rule switch2next requires only one step, while the operator `bag2set` is finite if the contents of the traces$'$ cell is finite. Hence, the rule switch2next is finite if the contents of the traces$'$ cell is finite. Meanwhile, the rules in Figure 3.1 rely on the `update` and `post` operators which may require more steps. We discuss next the finiteness of these two operators.

The `post` operator represents a computation level in $k\mathscr{P}$. Namely, it captures all possible applications of exactly one computational rule from $k\mathscr{P}$ over the `post` argument. Recall that a computational rule can be accompanied by the application of either several structural rules from $k\mathscr{P}$, or either several steps of heating rules. However, any chain of structural rules, or heating rules in $k\mathscr{P}$ that can be applied starting in a term

$\langle \check{\delta} \rangle_{\mathsf{ctrl}} \langle \check{\gamma} \rangle_{\mathsf{k}}$ is finite. The reason is that, according to the definition of $k\mathscr{P}$, a structural rule can be applied if it produces a strictly greater term in the finite lattice $(\check{\Delta} \times \check{\nabla}, \sqsubseteq)$, while the heating produces a strictly smaller term in the same finite lattice. Taking into account that the branching factor in $k\mathscr{P}$ is finite we conclude that any application of the `post` operator involves the application of a finite number of structural and computational rules.

The `update` operator is generic in this section, following to be instantiated according to the design of $\mho$ and the property to be verified Φ. At this point we can only say that `update` uses the results of `post` to increase $\mho$. However, we can assume that the contents of the collect cell form a complete lattice, and that there is a Galois connection between $(2^{Conf(k\mathscr{P})})^*$ and $Conf(\check{k}\mathscr{P})$. Note that the results presented in Section 3.2.1 ensure the existence of a finite complete lattice $(\langle \mho \rangle_{\mathsf{collect}}, \preceq)$. Also, the contents of collect cell are constructed based on the ground configurations discovered executing $k\mathscr{P}$ with the `post` operator ensures the existence of the mentioned Galois connection.

From the fact that `post` and `update` require a finite number of rule applications, we can deduce that each application of the unfold and prune rules terminate. Now se focus on reasoning on the overall applications of these rules in $\check{k}\mathscr{P}$.

The rule unfold cannot be applied infinitely because $\prec$ is well-founded. The reasoning for this is straightforward, by reductio ad absurdum. Each application of unfold produces an increased content of the collect cell and a finite (bounded) number of new trace cells in $\langle \rangle_{\mathsf{traces}'}$. Hence, the rule switch2next is finite since the `bag2set` operator uses a finite *NextLevel*. Also, the contents of the traces cell cannot grow infinitely (taking also into consideration the fact that switch2next rule dumps the contents of traces' cell into an empty $\langle \rangle_{\mathsf{traces}}$). Consequently, the rule prune can be applied only finite number of times, because each application of this rule decreases the *finite* contents of the traces cell.

Considering the fact that the number of application of the switch2next rule is strictly

smaller than the sum of the applications of rules in Figure 3.1, we conclude that there is a finite number of rule applications in $\check{k}\mathscr{P}$, hence the computation is finite. □

Finally, let us observe that in $\check{k}\mathscr{P}$ we collect computations produced by $k\mathscr{P}$ obtaining in this way an answer for $k\mathscr{P} \models \Phi$. As previously emphasized, $k\mathscr{P}$ is the $\mathbb{K}$ encoding of either the actual finite pushdown system $\mathscr{P}$, or of a sound finite abstraction of $\mathscr{P}$. In the first case it is obvious that the answer for $k\mathscr{P} \models \Phi$ is the same as the answer for $\mathscr{P} \models \Phi$. In the case when $k\mathscr{P}$ is an abstraction then for the analysis methods the result is an approximation of the actual result, while the verification methods are correct if the abstraction is sound, but not complete anymore. Hence, if $k\mathscr{P}$ is a finite pushdown system that is a sound abstraction of $\mathscr{P}$ w.r.t. Φ then the result verified by $\check{k}\mathscr{P}$, namely $k\mathscr{P} \models \Phi$, can be transported into $\mathscr{P} \models \Phi$.

3.2.3 Which collecting semantics?

According to Cousot's semantics hierarchy, the richest collecting semantics is maximal trace semantics [23]. The maximal trace semantics contains two parts: the *infinite traces* and the *maximal finite traces*. We instantiate in our generic collecting semantics only the maximal finite traces because along the current work we present different examples which do not extend over this particular collecting semantics. Hence, in our generic setting for collecting semantics, by trace semantics we understand that the collect cell in $\check{k}\mathscr{P}$ contains at the end of the computation a representation of the *maximal finite traces* in $k\mathscr{P}$.

Example 9. *We instantiate the trace semantics on one of the abstract pushdown systems presented in Example 7. We enlist $k\mathscr{P}$ here, for legibility purposes:*

$\check{\Delta} ::= a_0 \mid a_1 \mid a_2$

$\check{\nabla} ::= q_0 \mid q_1 \mid q_2 \mid q_3 \mid q_4 \mid q_5$

$K ::= \check{\nabla}$

$Init \equiv \langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}}\rangle_{\mathscr{P}}$

a_1	a_2	q_0	q_1	q_2	q_3	q_4
a_0	a_0	q_5	q_5	q_4	q_4	q_5

RULE $\langle a_0\rangle_{\mathsf{ctrl}}\langle \underline{\ q_0\ } \cdots\rangle_{\mathsf{k}}$ [row1]

$q_1\, q_0\, q_4$

RULE $\langle \underline{\ a_0\ }\rangle_{\mathsf{ctrl}}\langle \underline{\ q_1\ } \cdots\rangle_{\mathsf{k}}$ [row2]

$a_1 \qquad \cdot$

RULE $\langle a_0\rangle_{\mathsf{ctrl}}\langle \underline{\ q_3\ } \cdots\rangle_{\mathsf{k}}$ [row3, 5]

$\cdot$

RULE $\langle a_0\rangle_{\mathsf{ctrl}}\langle \underline{\ q_2\ } \cdots\rangle_{\mathsf{k}}$ [row4.col1−]

$\cdot$

RULE $\langle a_2\rangle_{\mathsf{ctrl}}\langle \underline{\ q_2\ } \cdots\rangle_{\mathsf{k}}$ [row4.col1]

$q_0\, q_3$

The "strict" *attribute is given to* $\langle a_0\rangle_{\mathsf{ctrl}}\langle q_{2,4,5} \cdots\rangle_{\mathsf{k}}$ *and the initial configuration for the trace semantics is:*

$$\langle\langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}\rangle_{\mathsf{traces}}\langle\cdot\rangle_{\mathsf{traces}'}\langle\langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{ctrace}}\rangle_{\mathsf{collect}}$$

The computation in $\tilde{k}\mathscr{P}$ *evolves as follows:*

$$\langle\langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{ctrl}}\rangle_{\mathsf{trace}}\rangle_{\mathsf{traces}}\langle\cdot\rangle_{\mathsf{traces}'}$$
$$\langle\langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{ctrace}}\rangle_{\mathsf{collect}}$$

$$\overset{*}{\Rightarrow}\ \langle\cdot\rangle_{\mathsf{traces}}\langle\langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_0\, q_3\, q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}\langle\langle a_0\rangle_{\mathsf{ctrl}}\langle q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}\rangle_{\mathsf{traces}'}$$
$$\langle\langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_2\rangle_{\mathsf{ctrl}}\langle q_0\, q_3\, q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{ctrace}}$$
$$\langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{ctrace}}\rangle_{\mathsf{collect}}$$

$$\overset{*}{\Rightarrow}\ \langle\langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_0\, q_3\, q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}\langle\langle a_0\rangle_{\mathsf{ctrl}}\langle q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}\rangle_{\mathsf{traces}}\langle\cdot\rangle_{\mathsf{traces}'}$$
$$\langle\langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_2\rangle_{\mathsf{ctrl}}\langle q_0\, q_3\, q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{ctrace}}$$
$$\langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{ctrace}}\rangle_{\mathsf{collect}}$$

$$\overset{*}{\Rightarrow}\ \langle\langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_0\, q_3\, q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}\rangle_{\mathsf{traces}}\langle\langle\langle a_0\rangle_{\mathsf{ctrl}}\langle\cdot\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}\rangle_{\mathsf{traces}'}$$
$$\langle\langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_2\rangle_{\mathsf{ctrl}}\langle q_0\, q_3\, q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{ctrace}}$$
$$\langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle\cdot\rangle_{\mathsf{k}}\rangle_{\mathsf{ctrace}}\rangle_{\mathsf{collect}}$$

$$\overset{*}{\Rightarrow}\ \langle\cdot\rangle_{\mathsf{traces}}\langle\langle\langle a_0\rangle_{\mathsf{ctrl}}\langle\cdot\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}\langle\langle a_0\rangle_{\mathsf{ctrl}}\langle q_1\, q_0\, q_4\, q_3\, q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}\rangle_{\mathsf{traces}'}$$
$$\langle\langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_2\rangle_{\mathsf{ctrl}}\langle q_0\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_1\, q_0\, q_4\, q_3\, q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{ctrace}}$$
$$\langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle\cdot\rangle_{\mathsf{k}}\rangle_{\mathsf{ctrace}}\rangle_{\mathsf{collect}}$$

$$\overset{*}{\Rightarrow}\ \langle\langle\langle a_0\rangle_{\mathsf{ctrl}}\langle q_1\, q_0\, q_4\, q_3\, q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}\rangle_{\mathsf{traces}}\langle\cdot\rangle_{\mathsf{traces}'}$$
$$\langle\langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_2\rangle_{\mathsf{ctrl}}\langle q_0\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_1\, q_0\, q_4\, q_3\, q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{ctrace}}$$
$$\langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle\cdot\rangle_{\mathsf{k}}\rangle_{\mathsf{ctrace}}\rangle_{\mathsf{collect}}$$

$$\overset{*}{\Rightarrow}\ \langle\cdot\rangle_{\mathsf{traces}}\langle\langle\langle a_1\rangle_{\mathsf{ctrl}}\langle q_0\, q_4\, q_3\, q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}\rangle_{\mathsf{traces}'}$$
$$\langle\langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_2\rangle_{\mathsf{ctrl}}\langle q_0\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_1\, q_0\, q_4\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright$$
$$\langle a_1\rangle_{\mathsf{ctrl}}\langle q_0\, q_4\, q_3\, q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{ctrace}}$$
$$\langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle\cdot\rangle_{\mathsf{k}}\rangle_{\mathsf{ctrace}}\rangle_{\mathsf{collect}}$$

$$
\begin{aligned}
\overset{*}{\Rightarrow}\ & \langle\langle\langle a_1\rangle_{\mathsf{ctrl}}\langle q_0\, q_4\, q_3\, q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}\rangle_{\mathsf{traces}}\langle\cdot\rangle_{\mathsf{traces}'} \\
& \langle\langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_2\rangle_{\mathsf{ctrl}}\langle q_0\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_1\, q_0\, q_4\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \\
& \quad \langle a_1\rangle_{\mathsf{ctrl}}\langle q_0\, q_4\, q_3\, q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{ctrace}} \\
& \quad \langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle\cdot\rangle_{\mathsf{k}}\rangle_{\mathsf{ctrace}}\rangle_{\mathsf{collect}}
\end{aligned}
$$

$$
\begin{aligned}
\overset{*}{\Rightarrow}\ & \langle\cdot\rangle_{\mathsf{traces}}\langle\langle\langle a_0\rangle_{\mathsf{ctrl}}\langle q_1\, q_0\, q_4\, q_4\, q_3\, q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}\rangle_{\mathsf{traces}'} \\
& \langle\langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_2\rangle_{\mathsf{ctrl}}\langle q_0\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_1\, q_0\, q_4\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \\
& \quad \langle a_1\rangle_{\mathsf{ctrl}}\langle q_0\, q_4\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_1\, q_0\, q_4\, q_4\, q_3\, q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{ctrace}} \\
& \quad \langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle\cdot\rangle_{\mathsf{k}}\rangle_{\mathsf{ctrace}}\rangle_{\mathsf{collect}}
\end{aligned}
$$

$$
\begin{aligned}
\overset{*}{\Rightarrow}\ & \langle\langle\langle a_0\rangle_{\mathsf{ctrl}}\langle q_1\, q_0\, q_4\, q_4\, q_3\, q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}\rangle_{\mathsf{traces}}\langle\cdot\rangle_{\mathsf{traces}'} \\
& \langle\langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_2\rangle_{\mathsf{ctrl}}\langle q_0\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_1\, q_0\, q_4\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \\
& \quad \langle a_1\rangle_{\mathsf{ctrl}}\langle q_0\, q_4\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_1\, q_0\, q_4\, q_4\, q_3\, q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{ctrace}} \\
& \quad \langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle\cdot\rangle_{\mathsf{k}}\rangle_{\mathsf{ctrace}}\rangle_{\mathsf{collect}}
\end{aligned}
$$

$$
\begin{aligned}
\overset{*}{\Rightarrow}\ & \langle\cdot\rangle_{\mathsf{traces}}\langle\langle\langle a_1\rangle_{\mathsf{ctrl}}\langle q_0\, q_4\, q_4\, q_3\, q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}\rangle_{\mathsf{traces}'} \\
& \langle\langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_2\rangle_{\mathsf{ctrl}}\langle q_0\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_1\, q_0\, q_4\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \\
& \quad \langle a_1\rangle_{\mathsf{ctrl}}\langle q_0\, q_4\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_1\, q_0\, q_4\, q_4\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \\
& \quad \langle a_1\rangle_{\mathsf{ctrl}}\langle q_0\, q_4\, q_4\, q_3\, q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{ctrace}} \\
& \quad \langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle\cdot\rangle_{\mathsf{k}}\rangle_{\mathsf{ctrace}}\rangle_{\mathsf{collect}}
\end{aligned}
$$

$$
\begin{aligned}
\overset{*}{\Rightarrow}\ & \langle\langle\langle a_1\rangle_{\mathsf{ctrl}}\langle q_0\, q_4\, q_4\, q_3\, q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}\rangle_{\mathsf{traces}}\langle\cdot\rangle_{\mathsf{traces}'} \\
& \langle\langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_2\rangle_{\mathsf{ctrl}}\langle q_0\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_1\, q_0\, q_4\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \\
& \quad \langle a_1\rangle_{\mathsf{ctrl}}\langle q_0\, q_4\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_1\, q_0\, q_4\, q_4\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \\
& \quad \langle a_1\rangle_{\mathsf{ctrl}}\langle q_0\, q_4\, q_4\, q_3\, q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{ctrace}} \\
& \quad \langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle\cdot\rangle_{\mathsf{k}}\rangle_{\mathsf{ctrace}}\rangle_{\mathsf{collect}}
\end{aligned}
$$

$$
\begin{aligned}
\overset{*}{\Rightarrow}\ & \langle\cdot\rangle_{\mathsf{traces}}\langle\langle\langle a_0\rangle_{\mathsf{ctrl}}\langle q_1\, q_0\, q_4\, q_4\, q_4\, q_3\, q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}\rangle_{\mathsf{traces}'} \\
& \langle\langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_2\rangle_{\mathsf{ctrl}}\langle q_0\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_1\, q_0\, q_4\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \\
& \quad \langle a_1\rangle_{\mathsf{ctrl}}\langle q_0\, q_4\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_1\, q_0\, q_4\, q_4\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \\
& \quad \langle a_1\rangle_{\mathsf{ctrl}}\langle q_0\, q_4\, q_4\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_1\, q_0\, q_4\, q_4\, q_4\, q_3\, q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{ctrace}} \\
& \quad \langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle\cdot\rangle_{\mathsf{k}}\rangle_{\mathsf{ctrace}}\rangle_{\mathsf{collect}}
\end{aligned}
$$

$$
\begin{aligned}
\overset{*}{\Rightarrow}\ & \langle\langle\langle a_0\rangle_{\mathsf{ctrl}}\langle q_1\, q_0\, q_4\, q_4\, q_4\, q_3\, q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}\rangle_{\mathsf{traces}}\langle\cdot\rangle_{\mathsf{traces}'} \\
& \langle\langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_2\rangle_{\mathsf{ctrl}}\langle q_0\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_1\, q_0\, q_4\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \\
& \quad \langle a_1\rangle_{\mathsf{ctrl}}\langle q_0\, q_4\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_1\, q_0\, q_4\, q_4\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \\
& \quad \langle a_1\rangle_{\mathsf{ctrl}}\langle q_0\, q_4\, q_4\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_1\, q_0\, q_4\, q_4\, q_4\, q_3\, q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{ctrace}} \\
& \quad \langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle\cdot\rangle_{\mathsf{k}}\rangle_{\mathsf{ctrace}}\rangle_{\mathsf{collect}}
\end{aligned}
$$

$$
\begin{aligned}
\overset{*}{\Rightarrow}\ & \langle\cdot\rangle_{\mathsf{traces}}\langle\cdot\rangle_{\mathsf{traces}'} \\
& \langle\langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_2\rangle_{\mathsf{ctrl}}\langle q_0\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_1\, q_0\, q_4\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \\
& \quad \langle a_1\rangle_{\mathsf{ctrl}}\langle q_0\, q_4\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_1\, q_0\, q_4\, q_4\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \\
& \quad \langle a_1\rangle_{\mathsf{ctrl}}\langle q_0\, q_4^r\, q_4\, q_3\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_1\, q_0\, q_4\, q_4\, q_4\, q_3\, q_3\rangle_{\mathsf{k}}\rangle_{\mathsf{ctrace}} \\
& \quad \langle\langle a_2\rangle_{\mathsf{ctrl}}\langle q_2\, q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle q_3\rangle_{\mathsf{k}} \curvearrowright \langle a_0\rangle_{\mathsf{ctrl}}\langle\cdot\rangle_{\mathsf{k}}\rangle_{\mathsf{ctrace}}\rangle_{\mathsf{collect}}
\end{aligned}
$$

3.3 $\mathbb{K}$-specifications *as* pushdown systems

"The end may justify the means as long as
there is something that justifies the end."
— Leon Trotsky.

We now investigate the relevance of studying pushdown systems in the context of programming language semantics. The $\mathbb{K}$ framework is specially designed for the specification of programming language semantics. The great advantage in having this specification is the fact that we have a language interpreter directly based on the semantics.

However, note that the interpreter can be seen at work only when the semantics is instantiated for some program. At this point pushdown systems show theoretical relevance, because a program is behaviorally equivalent with a pushdown system. Hence, designing a method for analysis/verification of pushdown systems in $\mathbb{K}$ is tantamount to giving a methodology for analysis/verification for programming languages defined in $\mathbb{K}$.

Consider the $\mathbb{K}$ specification $\mathscr{S}$ for the semantics of a language and a program P in this programming language. According to the methodology for designing $\mathscr{S}$ provided in [30], the k-cell behaves as the stack, while the control location is maintained by the cells containing the memory and the program. In this view, the semantics $\mathscr{S}$ is tantamount to the abstract specification of the pushdown systems appointed by the syntactic part of $\mathscr{S}$ under the assumptions of a particular memory model.

According to the description in Section 3.1.1, the abstract representation is given only for infinite pushdown systems. Obviously, the infiniteness is not necessarily a constraint and abstract representations can be provided also for finite pushdown systems, if the settings are conductive to this. As such, the specification $\mathscr{S}$ is an abstraction irrespective of the finiteness of the pushdown system induced by the program P.

The finiteness of the pushdown system produced for a program P with a specification $\mathscr{S}$ is, nonetheless, worth discussing from the point of view of the analysis/verification methods. Namely, we can produce a method for analysis/verification based on "exhaustive" execution only in the case when both the reachable control locations and the stack alphabet are finite.

Infinite pushdown systems are usually handled by abstract interpretation via a sound finite projection which is expressive enough to render the desired result for the analysis/verification method of interest. Following the perspective of abstract interpretation for state abstractions, the control locations of a pushdown system are coerced into a finite frame by means of a meta-operator α. Because the program P is part of the control location, we observe that the meta-operator must be applied to the syntactic part of $\mathscr{S}$

as well. This is the reason why, in [26], providing the abstraction meta-operator induces a transformation of the syntactic elements. Moreover, the stack language contains the syntactic elements in $\mathscr{S}$. Hence, the mere syntactic transformation of P is not enough because the rules in $\mathscr{S}$ rely on the original syntax. Consequently, when providing an abstraction one has to be able to automatically transform the rules in $\mathscr{S}$ into equivalent (abstract) rules to be applied in the abstract semantics.

Let us consider $\mathscr{S}[P]$ the instantiation of the semantics $\mathscr{S}$ with the program P. We construct the pushdown system associated to P as follows: Δ is the set of states defined by $\mathscr{S}$, while the stack alphabet ∇ is the closure of the syntactic elements in the program P. The transition relation $\hookrightarrow$ is defined as:

$(\delta, \gamma) \hookrightarrow (\delta', \Gamma)$

iff

$\langle \underbrace{\ \delta\ }_{\delta'} \rangle_{\mathsf{state}} \langle \underbrace{\ \gamma\ }_{\Gamma} \cdots \rangle_{\mathsf{k}}$ is the instantiation of a rule $\langle \underbrace{\ S\ }_{S'} \rangle_{\mathsf{state}} \langle \underbrace{\ K\ }_{K'} \cdots \rangle_{\mathsf{k}}$

from $\mathscr{S}$ (i.e., δ matches S, δ' matches S', γ matches K, and Γ matches K').

Let us denote the above defined pushdown system as $\mathscr{P}[P]$ and let $\mathscr{T}[P]$ denote its associated transition system.

Conjecture 1. *$\mathscr{S}[P]$ and $\mathscr{P}[P]$ are behaviorally equivalent, i.e., bisimilar.*

Proof. Note that due to the generality of the settings for the semantics $\mathscr{S}$ we cannot identify exactly which cells form the state, which are the syntactic elements appearing in the k cell, or which behavioral equivalence to use for the conjecture. Hence we can only provide a proof schemata, which we consider available for any instantiation of the conjecture.

The function $b : \mathscr{S}[P] \longrightarrow \mathscr{T}[P]$, defined as

$$b(\langle S \rangle_{\mathsf{state}} \langle K \rangle_{\mathsf{k}}) = \{(\delta, \Gamma) \mid \delta \text{ matches } S \text{ and } \Gamma \text{ matches } K\}$$

defines an equivalence (i.e., bisimulation, observational equivalence, trace equivalence, aso) on the set of reachable states. The proof uses structural induction on the computation. □

3.4 Conclusions

In this chapter we proposed a technique for defining in $\mathbb{K}$ analysis and verification methods over the $\mathbb{K}$ specifications of an abstract semantics. Namely, we employed the methodology provided by abstract interpretation with collecting semantics. As such, we described a blueprint for the $\mathbb{K}$ specification of the analysis/verification of pushdown systems specifications.

In Section 3.1 we presented an infrastructure for the $\mathbb{K}$ specification of analysis/verification methods for pushdown systems. This infrastructure brings in $\mathbb{K}$ the collecting semantics ideas following the footsteps of matching logic. In more detail, in Section 3.1.1 we proposed a generic representation in $\mathbb{K}$ of pushdown systems. We used the proposed $\mathbb{K}$ specification of a pushdown system as support for deriving the analysis/verification infrastructure in Section 3.2. We also argued the opportunity to consider pushdown systems and their $\mathbb{K}$ specification in Section 3.3.

With the study of collecting semantics for pushdown systems we intend to complement the $\mathbb{K}$-based verification perspective brought by matching logic [86, 83]. By adopting a different viewpoint on the problem of analysis/verification in $\mathbb{K}$, we hope to bring evidence of the feasibility for the implementation of a verification tool in the $\mathbb{K}$ framework. The choice of pushdown systems as a focal point for this study is based on the generality of the notion, the already available theoretical results, and the close resemblance with the $\mathbb{K}$ definitions. By the latter similitude we mean that the continuation-based technique used in $\mathbb{K}$ gives the stack aspect to the k-cell, while the $\mathbb{K}$ rules ultimately rely on a pushed down stack mechanism.

Chapter 4

Tracing Predicate Abstraction in $\mathbb{K}$

Motto: *"So obscure are the greatest events,*
as some take for granted any hearsay, whatever its source,
others turn truth into falsehood,
and both errors find encouragement with posterity. "
— Tacitus, The Annals of Imperial Rome.

This chapter introduces intraprocedural data analysis into the $\mathbb{K}$ framework by means of a $\mathbb{K}$ specification for invariant model checking with predicate abstraction technique. To express this technique in $\mathbb{K}$, we go to the foundations of predicate abstraction, that is abstract interpretation, and use its collecting semantics. As such, we propose a suitable description in $\mathbb{K}$ for collecting semantics under predicate abstraction of the simple imperative language *SIM* introduced in Chapter 2, Section 2.1. Next, we prove that our $\mathbb{K}$ specification for collecting semantics is a sound approximation of the $\mathbb{K}$ specification for the concrete semantics of *SIM*. This work makes a step towards the development of program verification methodologies in rewriting logic semantics project in general and the $\mathbb{K}$ framework in particular.

Programs are expected to work correctly with respect to certain requirements. To ensure their desired behavior, one needs to be able to formally reason about programs

and about programming languages. Existing formal approaches range from manually-constructed proofs to highly automated techniques. The latter includes model checking [20] and static analysis [72] as methods of ensuring correctness and finding certain classes of bugs.

In the context of software model checking, a program is translated, via convenient abstractions, into a state transition system. Abstraction helps to reduce the space size and therefore to improve the chances of a runnable/terminating verification process. However, the reduction in the number of states introduces additional behaviors, and leads to an over-approximation of the initial program.

Abstract interpretation [25] makes precise the fact that formal verification of the concrete program is reduced to verification of the simplified, abstract program, if the abstraction is sound. A consistent number of program reasoning methods make use of collecting semantics. Essentially, collecting semantics [25] abstracts each program point by a set of states, and stores the collected information according to the property of interest.

In this chapter we explore the potential of the $\mathbb{K}$ framework to define program reasoning methods. More specifically, we use $\mathbb{K}$ to define model checking with predicate abstraction. Because of the semantics-based characteristic of $\mathbb{K}$ specifications, we use collecting semantics as a means of delivering the work. As such, we propose a $\mathbb{K}$ description for the collecting semantics under predicate abstraction for a simple imperative language, *SIM*. In order to check the consistency of this $\mathbb{K}$ specification, we go along the lines of abstract interpretation standards, and prove that the defined programs' abstract executions are a sound approximation of the programs' concrete executions (the latter is also specified in $\mathbb{K}$ [81]). Even though we frame our work in this chapter within $\mathbb{K}$ for notational convenience, since $\mathbb{K}$ can be "desugared" to a large extent into rewriting logic (see [80, 81]), the results apply very well also to rewriting logic. In fact, we fully adhere to the rewriting logic project [69, 97].

4.1 Predicate Abstraction in $\mathbb{K}$

"The one duty we owe to history is to rewrite it."

— Oscar Wilde, The Critic as Artist.

In this section we describe how predicate abstraction can be defined in $\mathbb{K}$ as a push-down system specification.

Predicate abstraction [42] is a popular technique to build abstract models for programs which relies on defining a set of predicates over program variables. Valid executions in the abstract representation correspond to valid executions in the original program, whereas invalid runs in the abstract semantics need to be checked for feasibility in the concrete counterpart.

We recall the notion of abstract computation in the predicate abstraction environment. Abstraction is a mapping from a set of concrete states to an abstract state. An abstract transition between two abstract states exists if there is at least a concrete transition between concrete states from the preimage of each abstract state, respectively.

In predicate abstraction, an abstract state is represented by a predicate φ. The formal definition for φ is $\varphi ::= p \in \Pi \mid \neg p \mid \varphi \wedge \varphi \mid \mathit{true} \mid \mathit{false}$, where Π is a finite set of predicates from *AtomPreds* - all atomic predicates of interest over program variables. In other words, $\varphi \in \mathscr{L}(\Pi) =$ the lattice generated by the atomic predicates from Π. Formally, this lattice is defined as $\langle \mathscr{L}(\Pi), \sqcap, \sqcup, \bot, \top \rangle$ where $\sqcap$ stands for the logic operator $\vee$, $\sqcup$ stands for the logic operator $\wedge$, while $\bot$ and $\top$ stand for *true* and *false*, respectively (more details on this can be found in [72]).

Intuitively, an abstract state φ corresponds to the set of concrete states for which the values of the program variables make the formula φ true. The correspondence between the concrete states and the predicates φ is provided by a Galois connection from the powerset of all concrete states to $\mathscr{L}(\Pi)$. Additional details on this Galois connection are given in Section 4.3 where are of use in the formal reasoning about the specification.

We denote a transition in predicate abstraction by a function $\texttt{next}^\sharp$ standing for the abstract transition $(\varphi, s) \hookrightarrow \texttt{next}^\sharp\ (\varphi, s)$, where s is an abstract action. The formal definition of $\texttt{next}^\sharp$ is in Figure 4.2. We do not elaborate on it now, since it makes use of notations introduced later in this section. Of importance here is to have an understanding of the abstract computation with predicate abstraction, as this is a component of the collecting semantics described in Section 4.2.

In the remaining of the current section we describe the design in $\mathbb{K}$ of the predicate abstraction abstract actions (i.e., the stack alphabet in a pushdown system) and the abstract state (i.e., the control state in a pushdown system).

The k cell maintains the "abstract" computation of the program to be verified. This is in essence the control flow graph of a program, or program fragment. More formally, we provide the definition of an abstract computation $K^\sharp$ as follows:

$$\langle K^\sharp \rangle_{\mathsf{k}} \left\{ \begin{array}{l} \mathit{Label} ::= \mathit{Nat} \\ \mathit{Var} ::= \mathit{Qid} \\ \mathit{Asg} ::= \mathit{Var} := \mathit{AExp} \\ \mathit{Cnd} ::= [\mathit{BExp}] \\ \mathit{TransAsg} ::= \mathit{Label} : \mathit{Asg} \\ \mathit{TransCnd} ::= \mathit{Label} : \mathit{Cnd} \\ \mathit{Trans} ::= \mathit{TransAsg} \mid \mathit{TransCnd} \\ \mathit{Ks} ::= \mathit{TransAsg} \\ \qquad \mid \mathit{TransCnd}(\mathit{Ks} + \mathit{Ks}) \\ \qquad \mid \mathit{TransCnd}\ \texttt{while}_{\mathit{Label}} :: (\mathit{Ks} \curvearrowright \texttt{while}_{\mathit{Label}}) \\ \qquad \mid \texttt{flag} \mid \mathit{List}_{\curvearrowright}\{\mathit{Ks}\} \\ K^\sharp ::= \mathit{Ks} \curvearrowright \lfloor \mathit{Label} \rfloor \end{array} \right.$$

There is no obvious novelty in the abstract computation defined by $K^\sharp$ besides adding

$$stmt^\sharp : Label \times Stmt \to LabelK^\sharp Pair \qquad k^\sharp : Pgm \to K^\sharp$$
$$|\_,\_| : K^\sharp \times Label \to LabelK^\sharp Pair \qquad \lfloor \_ \rfloor : Label \to K^\sharp$$
$$\_ : \_ : Label \times (Asg \cup Cnd) \to Trans \qquad \_ \curvearrowright \_ : K^\sharp \times K^\sharp \to K^\sharp$$

$$stmt^\sharp(\ell_{in},\ \cdot) = |\cdot,\ \ell_{in}|$$
$$stmt^\sharp(\ell_{in},\ \texttt{skip};\ S) = stmt^\sharp(\ell_{in},\ S)$$
$$stmt^\sharp(\ell_{in},\ X = A;\ S) = |\ell_{in} : X := A \curvearrowright K,\ \ell_{fin}|$$
$$\quad \textbf{if } |K,\ \ell_{fin}| := stmt^\sharp(\ell_{in}+1,\ S)$$
$$stmt^\sharp(\ell_{in},\ S_1;\ S_2) = |K_1 \curvearrowright K_2,\ \ell_{fin}|$$
$$\quad \textbf{if } |K_1,\ \ell_{aux}| := stmt^\sharp(\ell_{in},\ S_1) \text{ and } |K_2,\ \ell_{fin}| := stmt^\sharp(\ell_{aux},\ S_2)$$
$$stmt^\sharp(\ell_{in},\ \{S\}) = stmt^\sharp(\ell_{in},\ S)$$
$$stmt^\sharp(\ell_{in},\ \texttt{if } B \texttt{ then } \{S_1\} \texttt{ else } \{S_2\}) = |\ell_{in} : [B](K_1 + K_2),\ \ell_{fin}|$$
$$\quad \textbf{if } |K_1,\ \ell_{aux}| := stmt^\sharp(\ell_{in}+1,\ S_1) \text{ and } |K_2,\ \ell_{fin}| := stmt^\sharp(\ell_{aux},\ S_2)$$
$$stmt^\sharp(\ell_{in},\ \texttt{while } B\ \{S\}) = |\ell_{in} : [B]\ \texttt{while}_{\ell_{in}} :: (K \curvearrowright \texttt{while}_{\ell_{in}}),\ \ell_{fin}|$$
$$\quad \textbf{if } |K,\ \ell_{fin}| := stmt^\sharp(\ell_{in}+1,\ S)$$

$$k^\sharp(\texttt{vars } Xs;\ S) = K \curvearrowright \lfloor \ell_{fin} \rfloor \ \textbf{if } |K,\ \ell_{fin}| := stmt^\sharp(1, S)$$

Figure 4.1: The rewrite-based rules for abstract computation $K^\sharp$ of a *SIM* program

labels to each basic statement - assignments and conditions. However, a closer look shows that we categorize the statements into basic statements (*Trans*, involved in computational rules in the collecting semantics proposed in Figure 4.4), and composed statements (branches and loops, involved in structural rules in Figure 4.4). Moreover, note that after constructing the list of abstract computational tasks *Ks* we finalize by tailing $\lfloor Label \rfloor$ to it ($\lfloor Label \rfloor$ marks the end of the program, and provides base case computational rules in Figure 4.4).

In Figure 4.1 we give an explicit rewrite-based method for labeling a *SIM* program, and transforming it into an abstract computation (or the associated control flow graph).

Example 10. *The abstract computation for the program in Figure 2.2 is:*

$$1 : \texttt{x} := 0 \curvearrowright 2 : \texttt{err} := \texttt{x} \curvearrowright 3 : [\texttt{y} \leq 0]\ \texttt{while}_3 :: (4 : \texttt{x} := \texttt{x} + 1 \curvearrowright 5 : \texttt{y} := \texttt{y} + \texttt{x}$$
$$\curvearrowright 6 : \texttt{x} := -1 + \texttt{x} \curvearrowright 7 : [\neg(\texttt{x} = 0)](8 : \texttt{err} := 1 + \cdot) \curvearrowright \texttt{while}_3) \curvearrowright \lfloor 9 \rfloor$$

Proposition 2. *Let P be a SIM program, then $k^\sharp(P)$ terminates and produces the control*

flow graph associated to P.

Proof. The $stmt^\sharp$ function traverses the $AST(P)$ the abstract syntax tree associated with the term P. Since $AST(P)$ is finite, then $stmt^\sharp$ terminates, hence $k^\sharp(P)$ terminates.

Each step in $stmt^\sharp$ associates an unique label ℓ to either an assignment, or a condition. Knowing that a *SIM* program contains only assignments, "`if then else`" branching statements, and "`while`" loops then, by structural induction, the result produced by $k^\sharp(P)$ is a representation of the control flow graph of P.

More to the point, $\ell : [B](K_1 + K_2)$ encodes a branching structure where K_1 is executed in case the guard B can be evaluated to *true*, K_2 is executed in case B can be evaluated to *false*. The marker $\texttt{while}_\ell$ represents the beginning of a loop starting at node labeled by ℓ. Namely, we use tail recursion to encode the "`while`" loops in the control flow graph representation. □

A ctrl cell is an abstract state which actually stands for a subset of states in the concrete execution. Since we use predicate abstraction with atomic predicates Π, the abstract state is a formula $\varphi \in \mathscr{L}(\Pi)$. However, here we prefer an equivalent representation which writes a formula $\varphi \in \mathscr{L}(\Pi) - \{\top\}$ as $\wedge_{p\in\Pi} op(\varphi, p)$, where op is defined as:

$$op(\varphi, p) = \begin{cases} p, & \text{if } \varphi \Rightarrow p \\ \neg p, & \text{if } \varphi \Rightarrow \neg p \\ \bot_p, & \text{otherwise} \end{cases}$$

Obviously, *true* is $(\bot_p)_{p\in\Pi}$, and for example, if the set of atomic predicates Π is:

$$\{\texttt{x} \geq 0,\ \texttt{x} = 0,\ \texttt{y} = 1\}$$

then the formula $\varphi := \mathtt{x} > 0$ is defined with the above representation as

$$\langle (\mathtt{x} \geq 0)\ \neg(\mathtt{x} = 0)\ \bot_{(\mathtt{y}=1)} \rangle_{\mathsf{ctrl}}$$

Also, we recall that this abstract state corresponds to the set of concrete states which map x to a positive integer. Hence, we use the equivalent representation of an abstract state:

$$\langle \mathit{State}^\sharp \rangle_{\mathsf{ctrl}} \begin{cases} \mathit{State}^\sharp ::= \mathit{Valid} \mid \mathit{False} \\ \mathit{Valid} ::= \mathit{Map}\{\Pi \mapsto \{(), \neg(), \bot_{()}\}\} \end{cases}$$

In this representation, the set of predicates Π defining the abstraction is implicitly contained in the ctrl cell. Note that *False* stands for the top element of $\mathscr{L}(\Pi)$, and is actually the abstract state corresponding to the empty set of concrete states. Meanwhile, *Valid* stands for any element from $\mathscr{L}(\Pi) - \{\top\}$. Because it represents a nonempty set of concrete states, we say that $\check{\delta} \in \mathit{Valid}$ is a "valid" abstract state. Moreover, we make the implicit assumption that $\mathit{false} \in \mathit{False}$ is always differentiated from a valid abstract state $\check{\delta}$, as if *false* and $\check{\delta}$ are cells with distinct labels (while ctrl cell can contain either of them). However, for simplicity, we do not embellish the notation any further. Finally, in Figure 4.2 we define $\mathtt{next}^\sharp$, the update relation for the pushdown system specification given by predicate abstraction.

Proposition 3. *The pushdown system specification*

$$\mathscr{P}(P) = (\mathit{State}^\sharp, \mathit{Ks}, \mathtt{next}^\sharp, (\check{\delta}_0, k^\sharp(P)))$$

associated to a SIM program P produces a finite *pushdown system.*

Proof. The set of control states $\mathit{State}^\sharp$ is finite being an one-to-one encoding of $\mathscr{L}(\Pi)$. The stack alphabet produced by $k^\sharp(P)$ is also finite, i.e., $k^\sharp(P)$ produces a finite number

$\texttt{next}^\sharp \subseteq (\mathit{Valid} \times K^\sharp) \times (\mathit{State}^\sharp \times K^\sharp)$ is defined as follows:

$$\texttt{next}^\sharp\ (\check{\delta}, \ell : x := a) = \ (\texttt{post}^\sharp_\bot\ (\check{\delta}, x := a), \cdot)$$

$$\texttt{next}^\sharp\ (\check{\delta}, \ell : [b](k_1 + k_2) =$$

$$\begin{cases} (\texttt{post}^\sharp_\bot\ (\check{\delta}, [b]), k_1), & \text{if } \texttt{post}^\sharp_\bot\ (\check{\delta}, [b]) \neq \mathit{false} \\ (\texttt{post}^\sharp_\bot\ (\check{\delta}, [\neg b]), k_2), & \text{if } \texttt{post}^\sharp_\bot\ (\check{\delta}, [\neg b]) \neq \mathit{false} \end{cases}$$

$$\texttt{next}^\sharp\ (\check{\delta}, \ell : [b]\ \texttt{while}_\ell :: (k_1 \curvearrowright \texttt{while}_\ell)) =$$

$$\begin{cases} (\texttt{post}^\sharp_\bot\ (\check{\delta}, [b]), k_1 \curvearrowright \ell : [b] \curvearrowright \texttt{while}_\ell :: (\ldots)), & \text{if } \texttt{post}^\sharp_\bot\ (\check{\delta}, [b]) \neq \mathit{false} \\ (\texttt{post}^\sharp_\bot\ (\check{\delta}, [\neg b]), \cdot), & \text{if } \texttt{post}^\sharp_\bot\ (\check{\delta}, [\neg b]) \neq \mathit{false} \end{cases}$$

$\texttt{post}^\sharp_\phi : \mathit{Valid} \times (\mathit{Asg} \cup \mathit{Cnd}) \longrightarrow \mathit{State}^\sharp$ is defined as follows:

$$\texttt{post}^\sharp_\phi\ (\check{\delta}, s) = \begin{cases} \wedge_{p \in \Pi}\ \texttt{post}_{(\check{\delta}, s)}\ (p), & \text{if } \wedge_{p \in \Pi}\ \texttt{post}_{(\check{\delta}, s)}\ (p) \Rightarrow \phi \\ \mathit{false}, & \text{otherwise} \end{cases}$$

$$\text{with } \texttt{post}_{(\check{\delta}, [b])}\ (p) = \begin{cases} \mathit{false}, & \text{if } \wedge_{p \in \Pi} \check{\delta}(p) \wedge b \Rightarrow \mathit{false} \\ p, & \text{if } \wedge_{p \in \Pi} \check{\delta}(p) \wedge b \Rightarrow p \\ \neg p, & \text{if } \wedge_{p \in \Pi} \check{\delta}(p) \wedge b \Rightarrow \neg p \\ \bot_p, & \text{otherwise} \end{cases}$$

$$\text{and } \texttt{post}_{(\check{\delta}, x := a)}\ (p) = \begin{cases} p, & \text{if } \wedge_{p \in \Pi} \check{\delta}(p) \wedge (x' = a) \Rightarrow p[x'/x] \\ \neg p, & \text{if } \wedge_{p \in \Pi} \check{\delta}(p) \wedge (x' = a) \Rightarrow \neg p[x'/x] \\ \bot_p, & \text{otherwise} \end{cases}$$

Figure 4.2: The update relation for the abstract pushdown system

of labeled abstract actions of sort *Trans*. Then $\mathit{Reach}(\mathscr{P}(P))$ is finite so the pushdown system produced by $\mathscr{P}(P)$ is finite. □

4.2 Trace semantics with predicate abstraction

"The ages live in history through their anachronisms."

— Oscar Wilde, Complete Works of Oscar Wilde.

Collecting semantics defines the set of program executions from the property of in-

terest point of view and has several instantiations: computation traces, transitive closure of the program transition relation, reachable states, and so on. In this chapter we *collect forward abstract computation traces* using predicate abstraction. We describe next the details of this setting.

We proceed to define in $\mathbb{K}$ the program meta-executions using collecting semantics under a fixed predicate abstraction. The finite set of predicates Π is given, as well as the property of interest: the state invariant φ. Note that we refer meta-executions also as abstract executions.

The *abstract configuration* in $\mathbb{K}$ is defined as:

$$\mathit{Configuration} \equiv \langle \langle \langle K^\sharp \rangle_{\mathsf{k}} \langle \mathit{State}^\sharp \rangle_{\mathsf{ctrl}} \langle \mathit{List}_{\cdot} \{\mathit{Label}\} \rangle_{\mathsf{path}} \rangle_{\mathsf{trace}*} \rangle_{\mathsf{traces}} \langle \langle \mathit{Store}^\sharp \rangle_{\mathsf{store}} \langle \varphi \rangle_{\mathsf{inv}} \rangle_{\mathsf{collect}}$$

In order to have the intuition behind *Configuration*, imagine that *PdcT* is a *p*arallel *d*ivide and *c*onquer algorithm performing the *t*raversal of a digraph. *PdcT* traverses the digraph in a standard fashion but, when it encounters a node with more than one neighbor, it is going to clone itself on each neighboring direction. The instances of *PdcT* communicate via a shared memory where everyone deposits its own visited nodes. When an instance of *PdcT* encounters a node existing in the shared memory, it terminates its job, as that part of the digraph is already under the administration of another instance. *Configuration* is similar to a state in the running of *PdcT*. Namely, a trace cell resembles with the state of an instance of *PdcT*, and the store cell resembles with the state of the shared memory. Moreover, the rules for collecting semantics under predicate abstraction, from Figure 4.4, are similar with the transitions between states of *PdcT*.

In Section 4.1 we describe the cells concerning the finite pushdown system in *Configuration*, namely the k cell maintaining the stack of the pushdown system and the ctrl cell maintaining the control state. Next, we continue the description of each cell of the $\mathbb{K}$ configuration for collecting semantics under predicate abstraction.

A cell of type path is a list of labels which represents a trace of a possible abstract execution. Note that we refer to this as *trace* because many details are cut out from the abstract execution. Instead, we keep as representative the program points where the abstract execution took place, in their order of appearance. Hence, the cells k and ctrl capture the *abstract computation*, the cell path stands for *trace*, while *forward* comes from $\texttt{post}^\sharp$, the abstract state update operator. Note that $\texttt{trace}*$ in *Configuration* indicates the existence of many trace cells.

The contents of the store cell, denoted as Σ, is a set of pairs $(\!|\ell, \breve{\delta}|\!)$ of labels from the abstract computation and elements from $\mathscr{L}(\Pi) - \{\top\}$, defined in $\mathbb{K}$ as:

$$\mathit{Store}^\sharp ::= \mathit{Set}\{(\!|\mathit{Label},\ \mathit{Valid}|\!)\}$$

The abstract store update is denoted as $\Sigma[\ell \rightharpoonup \breve{\delta}] = \Sigma \cup \{(\!|\ell, \breve{\delta}|\!)\}$.

A more standard definition for the abstract store would involve a mapping, such as:

$$\mathit{Store}^\sharp ::= \mathit{Map}\{\mathit{Label} \mapsto \mathit{Set}\{\mathit{Valid}\}\}$$

Note that the two representations of $\mathit{Store}^\sharp$ are equivalent. However, we prefer the former one in order to suggest the *collecting* nature of the current semantics.

An inv cell maintains the state invariant formula to be validated, where $\phi ::= p \in \mathit{AtomPreds} \mid \neg p \mid \phi \wedge \phi \mid \mathit{false} \mid \mathit{true}$. On short, ϕ is a state invariant is translated as "formula ϕ is satisfied in any (abstract) state, on any computational path". Φ is defined similarly with $\mathit{State}^\sharp$ (Π is replaced by the set of atomic predicates from ϕ). Usually, Π includes the atomic predicates from ϕ.

Figure 4.3 provides the $\mathbb{K}$ structural rules for initialization and termination of the abstract executions, where pgm^ϕ_Π is a shorthand for the input cell containing the program $\mathit{pgm} = \texttt{vars}\ \mathit{Xs};\ S$, the abstraction predicates set Π, and the formula to verify ϕ.

$$\textit{Initialization} \equiv \frac{pgm^{\phi}_{\Pi}}{\langle\langle\langle k^{\sharp}(s)\rangle_{\mathsf{k}}\langle\sqcap\{\varphi\in\mathscr{L}(\Pi)\,|\,\varphi\Rightarrow\phi\}\rangle_{\mathsf{ctrl}}\langle\cdot\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}}\rangle_{\mathsf{traces}}\langle\langle\cdot\rangle_{\mathsf{store}}\langle\phi\rangle_{\mathsf{inv}}\rangle_{\mathsf{collect}}}$$

$$\textit{Termination} \equiv \begin{cases} \dfrac{\langle\cdots\ \langle\langle\cdot\rangle_{\mathsf{k}}\langle\cdot\rangle_{\mathsf{ctrl}}\langle \mathit{path}\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}}\ \cdots\rangle_{\mathsf{traces}}\langle\_\rangle_{\mathsf{collect}}}{\langle \mathit{path}\rangle_{\mathsf{CE}}} \\ \dfrac{\langle\cdot\rangle_{\mathsf{traces}}\langle\_\rangle_{\mathsf{collect}}}{\langle\cdot\rangle_{\mathsf{CE}}} \end{cases}$$

Figure 4.3: Initialization and termination for $\mathbb{K}$ abstract executions of *SIM*

The initialization of an execution in collecting semantics for the program *pgm* has one trace cell containing the abstract computation of the program, an initial abstract state corresponding to the best over-approximation of the property ϕ in the lattice $\mathscr{L}(\Pi)$, an empty path, an empty abstract store, and the property to be verified upon the program. (Recall that we consider "$\cdot$" as the unit element for any cell.) The choice of the initial ctrl cell, $\sqcap\{\varphi \in \Pi |\ \phi \Rightarrow \varphi\}$, is the abstract representation of all concrete states δ_0, where ϕ is true (i.e., $\{\delta_0 \mid \delta_0 \vDash \phi\}$). As a matter of fact, we could as well generalize the initial abstract state cell ctrl to contain any element from the lattice $\mathscr{L}(\Pi)$.

The termination of an execution in collecting semantics is expected to provide a path representing a potential counterexample to the validity of the property ϕ for *pgm*. In the case when there is no counterexample, the property is valid in the abstract model, and also in the program. Otherwise, no conclusion could be derived with respect to the validity of property ϕ for the given program.

The semantic rules for an execution with collecting semantics under predicate abstraction are described in Figure 4.4. Note that in these rules the cells are considered to appear in the inner most environment wrapping them, according with the *locality principle* [81]. Next, we explain in details each of these rules.

The first two rules, (R1-2), deal with the case when the abstract computation reaches the final label of the program either with a valid abstract state $\breve{\delta}$ or with *false*. If the abstract state is valid then its containing trace cell is voided (because there is no abstract

RULE $\langle\langle \lfloor \ell \rfloor \rangle_{\mathsf{k}}\ \langle \breve{\delta} \rangle_{\mathsf{ctrl}}\ \langle \_ \rangle_{\mathsf{path}} \rangle_{\mathsf{trace}}$ $\cdot$ $[(\mathrm{R1})_{\square}]$

RULE $\langle\ \lfloor \ell \rfloor\ \rangle_{\mathsf{k}} \langle \mathit{false} \rangle_{\mathsf{ctrl}} \langle \cdots\ \cdot\ \rangle_{\mathsf{path}}$ ℓ $[(\mathrm{R2})_{\boxtimes}]$

RULE $\langle\langle \ell : \_ \cdots \rangle_{\mathsf{k}}\ \langle \breve{\delta} \rangle_{\mathsf{ctrl}}\ \langle \_ \rangle_{\mathsf{path}} \rangle_{\mathsf{trace}} \langle \cdots\ (\!|\ell, \breve{\delta}|\!)\ \cdots \rangle_{\mathsf{store}}$ $\cdot$ $[(\mathrm{R3})_{\square\checkmark}]$

RULE $\langle \ell : \_ \curvearrowright \_ \rangle_{\mathsf{k}} \langle \mathit{false} \rangle_{\mathsf{ctrl}} \langle \cdots\ \cdot\ \rangle_{\mathsf{path}}$ ℓ $[(\mathrm{R4})_{\boxtimes}]$

RULE $\langle \ell : x := a\ \cdots \rangle_{\mathsf{k}} \langle\ \breve{\delta}\ \rangle_{\mathsf{ctrl}} \langle \cdots\ \cdot\ \rangle_{\mathsf{path}} \langle\ \Sigma\ \rangle_{\mathsf{store}} \langle \phi \rangle_{\mathsf{inv}}$
$\cdot$ $\quad \mathtt{post}^{\sharp}_{\phi}(\breve{\delta}, x := a)$ $\quad \ell$ $\quad \Sigma[\ell \rightarrowtail \breve{\delta}]$
when $(\!|\ell, \breve{\delta}|\!) \notin \Sigma$ $[(\mathrm{R5})_{\rightarrowtail}]$

RULE $\langle \ell : [b](k_1 + k_2) \curvearrowright k \rangle_{\mathsf{k}} \langle\ \breve{\delta}\ \rangle_{\mathsf{ctrl}} \langle\ p\ \rangle_{\mathsf{path}}$
$\mathtt{flag} \curvearrowright k_1$ $\quad \mathtt{post}^{\sharp}_{\phi}(\breve{\delta}, [b])$ $\quad p, \ell$
$$\frac{\cdot}{\langle\langle \mathtt{flag} \curvearrowright k_2 \curvearrowright k \rangle_{\mathsf{k}} \langle \mathtt{post}^{\sharp}_{\phi}(\breve{\delta}, [\neg b]) \rangle_{\mathsf{ctrl}} \langle p, \ell \rangle_{\mathsf{path}} \rangle_{\mathsf{trace}}} \quad \frac{\langle\ \Sigma\ \rangle_{\mathsf{store}} \langle \phi \rangle_{\mathsf{inv}}}{\Sigma[\ell \rightarrowtail \breve{\delta}]}$$
when $(\!|\ell, \breve{\delta}|\!) \notin \Sigma$ $[(\mathrm{R6})_{\rightarrowtail}]$

RULE $\langle \cdots\ \langle \mathtt{flag}\ \cdots \rangle_{\mathsf{k}} \langle \mathit{false} \rangle_{\mathsf{ctrl}}\ \cdots \rangle_{\mathsf{trace}}$ $\cdot$ $[(\mathrm{R7})_{\curlyvee}]$

RULE $\langle \mathtt{flag}\ \cdots \rangle_{\mathsf{k}} \langle \breve{\delta} \rangle_{\mathsf{ctrl}}$ $\cdot$ $[(\mathrm{R8})_{\curlyvee}]$

RULE $\langle\ \ell : [b]\mathtt{while}_{\ell} :: (k \curvearrowright \mathtt{while}_{\ell})\ \cdots \rangle_{\mathsf{k}}$ $(\ell : [b]((k \curvearrowright \ell : [b] \curvearrowright \mathtt{while}_{\ell} :: (k \curvearrowright \mathtt{while}_{\ell})) + \cdot)$ $[(\mathrm{R9})_{\multimap}]$

Figure 4.4: $\mathbb{K}$ rules for collecting semantics under predicate abstraction of *SIM*

computation left for it, and along the current abstract trace only valid states were encountered, meaning that the property ϕ is satisfied in any abstract state along this trace). If the abstract state is *false* then an error is found just before the end of the program. Whenever an error is found, meaning an abstract state where ϕ is not valid, we end the abstract execution from that particular trace cell and keep its representation in the path cell as a witness to the potential discovery of a bug (so called counterexample). This happens in the rules annotated with $\boxtimes$. (Note that $\square$ annotation stands for "good" termination.)

The rules (R3-4) present two other base cases, when the statement labeled ℓ is at the top of the abstract computation (i.e. $\langle \ell : \_ \ \cdots \rangle_{\mathsf{k}}$). The rule (R3) covers the case when a particular program point is reached again, with the same abstract state $\breve{\delta}$. This is expressed by the fact that the abstract store, store cell, contains the pair $(\!|\ell, \breve{\delta}|\!)$. We can void the current trace cell, because this particular abstract trace will not increment the store cell any further. However, if a particular program point is reached with the *false* abstract state, as in (R4), we maintain the path as a counterexample.

Rule (R5)$_{\curvearrowright}$ performs an abstract execution of an assignment statement encountered at the top of the abstract computation in the k cell. This means that the abstract state is updated by the abstract postcondition $\texttt{post}^\sharp$, while the current abstract state is used to update store by adding the pair $(\!|\ell, \breve{\delta}|\!)$ to it. Note that this addition is made only if the pair is not already in the abstract store, according to the definition of the store update.

The rules (R7-8)$_{\curlyvee}$, containing `flag` at the top of the abstract computation, are both following a branching rule (R6)$_{\curvearrowright}$. When the abstract execution encounters a branching condition, denoted by $\ell : [b](K_1 + K_2)$, the rule (R6)$_{\curvearrowright}$ spawns another abstract trace. In this way, the current abstract trace maintains the "then" branch, with the boolean condition b, while the new trace maintains the "else" branch, with the boolean condition $\neg b$. However, it might be the case that not both branches are possible executions (e.g. if the boolean condition is *false*, then only the "else" branch is feasible). In order to filter

these cases, when spawning the two traces, we also add a `flag` at the top of the abstract computation. The structural rules $(\text{R7-8})_{\curlyvee}$ filter the `flag`: if the abstract state obtained by adding the conditional evaluates to *false*, then we remove this trace cell, otherwise we continue the execution removing the `flag`.

The last rule, $(\text{R9})_{\multimap}$ unfolds the loops once. Note that the last three rules, (R7-9), are structural rules that transform the abstract computation. Also, we emphasize the R's annotations provide additional rules' classification.

Example 11. *For the program in Figure 2.2 and the invariant* $(\texttt{err}=0)$ *the abstract execution with the predicate abstraction given by* $\Pi=\{\texttt{err}=0\}$ *terminates with* $\langle 1,2,3\rangle_{\mathsf{CE}}$*, while if* $\Pi=\{\texttt{err}=0,\texttt{x}=0\}$ *the abstract execution terminates with* $\langle 1,2,3,4,5,6,7,8,3\rangle_{\mathsf{CE}}$*. However, with the predicate abstraction given by* $\Pi=\{\texttt{err}=0,\texttt{x}=0,\texttt{x}=1\}$ *the abstract execution ends with* $\langle\cdot\rangle_{\mathsf{CE}}$*.*

The abstract execution with $\Pi=\{\texttt{err}=0,\texttt{x}=0\}$ *starts with the abstract computation described in Example 10, and proceeds as described in the followings.*

$$\langle\langle\langle 1:\texttt{x}:=0\ \cdots\rangle_{\mathsf{k}}\langle(\texttt{err}=0)\bot_{(\texttt{x}=0)}\rangle_{\mathsf{ctrl}}\langle\cdot\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}}\rangle_{\mathsf{traces}}\langle\langle\cdot\rangle_{\mathsf{store}}\langle\texttt{err}=0\rangle_{\mathsf{inv}}\rangle_{\mathsf{collect}}$$

$$\overset{R5}{\Rrightarrow}\ \langle\langle 2:\texttt{err}:=\texttt{x}\ \cdots\rangle_{\mathsf{k}}\langle(\texttt{err}=0)(\texttt{x}=0)\rangle_{\mathsf{ctrl}}\langle 1\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}}$$
$$\langle\langle(\!|1,(\texttt{err}=0)\bot_{(\texttt{x}=0)}|\!)\rangle_{\mathsf{store}}\langle\texttt{err}=0\rangle_{\mathsf{inv}}\rangle_{\mathsf{collect}}$$

$$\overset{R5}{\Rrightarrow}\ \langle\langle 3:[\texttt{y}\leq 0]\texttt{while}_3::(K\curvearrowright\texttt{while}_3)\ \cdots\rangle_{\mathsf{k}}\langle(\texttt{err}=0)(\texttt{x}=0)\rangle_{\mathsf{ctrl}}\langle 1,2\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}}$$
$$\langle\langle(\!|1,(\texttt{err}=0)\bot_{(\texttt{x}=0)}|\!)(\!|2,(\texttt{err}=0)(\texttt{x}=0)|\!)\rangle_{\mathsf{store}}\langle\texttt{err}=0\rangle_{\mathsf{inv}}\rangle_{\mathsf{collect}}$$

where K is $4:\texttt{x}:=\texttt{x}+1\curvearrowright 5:\texttt{y}:=\texttt{y}+\texttt{x}\curvearrowright 6:\texttt{x}:=-1+\texttt{x}$
$$\curvearrowright 7:[\neg(\texttt{x}=0)](8:\texttt{err}:=1+\cdot)$$

$$\overset{R9}{\Rrightarrow}\ \langle\langle 3:[\texttt{y}\leq 0]((K\curvearrowright K')+\cdot)\ \cdots\rangle_{\mathsf{k}}\langle(\texttt{err}=0)(\texttt{x}=0)\rangle_{\mathsf{ctrl}}\langle 1,2\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}}$$
$$\langle\langle(\!|1,(\texttt{err}=0)\bot_{(\texttt{x}=0)}|\!)(\!|2,(\texttt{err}=0)(\texttt{x}=0)|\!)\rangle_{\mathsf{store}}\langle\texttt{err}=0\rangle_{\mathsf{inv}}\rangle_{\mathsf{collect}}$$

where K' is $3:[\texttt{y}\leq 0]\texttt{while}_3::(K\curvearrowright\texttt{while}_3)$

$$\overset{R6}{\Rightarrow} \langle\langle \texttt{flag} \curvearrowright (K \curvearrowright K') \ \cdots\rangle_{\mathsf{k}}\langle(\texttt{err}=0)(\texttt{x}=0)\rangle_{\mathsf{ctrl}}\langle 1,2,3\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}}$$
$$\langle\langle \texttt{flag} \curvearrowright \lfloor 9 \rfloor\rangle_{\mathsf{k}}\langle(\texttt{err}=0)(\texttt{x}=0)\rangle_{\mathsf{ctrl}}\langle 1,2,3\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}}$$
$$\langle (\!| 1,(\texttt{err}=0)\bot_{(\texttt{x}=0)} |\!)(\!| 2,(\texttt{err}=0)(\texttt{x}=0) |\!)(\!| 3,(\texttt{err}=0)(\texttt{x}=0) |\!)\rangle_{\mathsf{store}}$$

$$\overset{R8}{\Rightarrow} \langle\langle K \ \cdots\rangle_{\mathsf{k}}\langle(\texttt{err}=0)(\texttt{x}=0)\rangle_{\mathsf{ctrl}}\langle 1,2,3\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}}$$
$$\langle\langle \texttt{flag} \curvearrowright \lfloor 9 \rfloor\rangle_{\mathsf{k}}\langle(\texttt{err}=0)(\texttt{x}=0)\rangle_{\mathsf{ctrl}}\langle 1,2,3\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}}$$
$$\langle (\!| 1,(\texttt{err}=0)\bot_{(\texttt{x}=0)} |\!)(\!| 2,(\texttt{err}=0)(\texttt{x}=0) |\!)(\!| 3,(\texttt{err}=0)(\texttt{x}=0) |\!)\rangle_{\mathsf{store}}$$

$$\overset{R8}{\Rightarrow} \langle\langle 4 : \texttt{x} := \texttt{x}+1 \ \cdots\rangle_{\mathsf{k}}\langle(\texttt{err}=0)(\texttt{x}=0)\rangle_{\mathsf{ctrl}}\langle 1,2,3\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}}$$
$$\langle\langle \lfloor 9 \rfloor\rangle_{\mathsf{k}}\langle(\texttt{err}=0)(\texttt{x}=0)\rangle_{\mathsf{ctrl}}\langle 1,2,3\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}}$$
$$\langle (\!| 1,(\texttt{err}=0)\bot_{(\texttt{x}=0)} |\!)(\!| 2,(\texttt{err}=0)(\texttt{x}=0) |\!)(\!| 3,(\texttt{err}=0)(\texttt{x}=0) |\!)\rangle_{\mathsf{store}}$$

$$\overset{R1}{\Rightarrow} \langle\langle 4 : \texttt{x} := \texttt{x}+1 \ \cdots\rangle_{\mathsf{k}}\langle(\texttt{err}=0)(\texttt{x}=0)\rangle_{\mathsf{ctrl}}\langle 1,2,3\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}}$$
$$\langle (\!| 1,(\texttt{err}=0)\bot_{(\texttt{x}=0)} |\!)(\!| 2,(\texttt{err}=0)(\texttt{x}=0) |\!)(\!| 3,(\texttt{err}=0)(\texttt{x}=0) |\!)\rangle_{\mathsf{store}}$$

$$\overset{R5}{\Rightarrow} \langle\langle 5 : \texttt{y} := \texttt{y}+\texttt{x} \ \cdots\rangle_{\mathsf{k}}\langle(\texttt{err}=0)\neg(\texttt{x}=0)\rangle_{\mathsf{ctrl}}\langle 1,2,3,4\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}}$$
$$\langle\cdots \ (\!| 4,(\texttt{err}=0)(\texttt{x}=0) |\!)\rangle_{\mathsf{store}}$$

$$\overset{R5}{\Rightarrow} \langle\langle 6 : \texttt{x} := -1+\texttt{x} \ \cdots\rangle_{\mathsf{k}}\langle(\texttt{err}=0)\neg(\texttt{x}=0)\rangle_{\mathsf{ctrl}}\langle 1,2,3,4,5\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}}$$
$$\langle\cdots \ (\!| 5,(\texttt{err}=0)\neg(\texttt{x}=0) |\!)\rangle_{\mathsf{store}}$$

$$\overset{R5}{\Rightarrow} \langle\langle (7 : [\neg\texttt{x}=0](8 : \texttt{err} := 1+\cdot) \ \cdots\rangle_{\mathsf{k}}\langle(\texttt{err}=0)\bot_{(\texttt{x}=0)}\rangle_{\mathsf{ctrl}}\langle\cdots \ 6\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}}$$
$$\langle\cdots \ (\!| 6,(\texttt{err}=0)\neg(\texttt{x}=0) |\!)\rangle_{\mathsf{store}}$$

$$\overset{R6}{\Rightarrow} \langle\langle \texttt{flag} \curvearrowright 8 : \texttt{err} := 1 \ \cdots\rangle_{\mathsf{k}}\langle(\texttt{err}=0)\neg(\texttt{x}=0)\rangle_{\mathsf{ctrl}}\langle\cdots \ 7\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}}$$
$$\langle\langle \texttt{flag} \curvearrowright \cdot \ \cdots\rangle_{\mathsf{k}}\langle(\texttt{err}=0)(\texttt{x}=0)\rangle_{\mathsf{ctrl}}\langle\cdots \ 7\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}}$$
$$\langle\cdots \ (\!| 7,(\texttt{err}=0)\bot_{(\texttt{x}=0)} |\!)\rangle_{\mathsf{store}}$$

$$\overset{R8}{\Rightarrow} \langle\langle \texttt{flag} \curvearrowright 8 : \texttt{err} := 1 \ \cdots\rangle_{\mathsf{k}}\langle(\texttt{err}=0)\neg(\texttt{x}=0)\rangle_{\mathsf{ctrl}}\langle\cdots \ 7\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}}$$
$$\langle\langle K' \ \cdots\rangle_{\mathsf{k}}\langle(\texttt{err}=0)(\texttt{x}=0)\rangle_{\mathsf{ctrl}}\langle\cdots \ 7\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}}$$
$$\langle\cdots \ (\!| 3,(\texttt{err}=0)(\texttt{x}=0) |\!) \ \cdots\rangle_{\mathsf{store}}$$

$$\overset{R9}{\Rrightarrow}\ \langle\langle \texttt{flag} \curvearrowright 8 : \texttt{err} := 1 \ \cdots\rangle_{\mathsf{k}}\langle(\texttt{err} = 0)\neg(\texttt{x} = 0)\rangle_{\mathsf{ctrl}}\langle\cdots\ 7\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}}$$
$$\langle\langle 3 : [\texttt{y} \leq 0]((K \curvearrowright K') + \cdot)\ \cdots\rangle_{\mathsf{k}}\langle(\texttt{err} = 0)(\texttt{x} = 0)\rangle_{\mathsf{ctrl}}\langle\cdots\ 7\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}}$$
$$\langle\cdots\ (\!|3, (\texttt{err} = 0)(\texttt{x} = 0)|\!)\ \cdots\rangle_{\mathsf{store}}$$

$$\overset{R3}{\Rrightarrow}\ \langle\langle \texttt{flag} \curvearrowright 8 : \texttt{err} := 1 \ \cdots\rangle_{\mathsf{k}}\langle(\texttt{err} = 0)\neg(\texttt{x} = 0)\rangle_{\mathsf{ctrl}}\langle\cdots\ 7\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}}$$
$$\langle\cdots\ (\!|3, (\texttt{err} = 0)(\texttt{x} = 0)|\!)\ \cdots\rangle_{\mathsf{store}}$$

$$\overset{R8}{\Rrightarrow}\ \langle\langle 8 : \texttt{err} := 1 \ \cdots\rangle_{\mathsf{k}}\langle(\texttt{err} = 0)\neg(\texttt{x} = 0)\rangle_{\mathsf{ctrl}}\langle\cdots\ 7\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}}$$
$$\langle\cdots\ (\!|3, (\texttt{err} = 0)(\texttt{x} = 0)|\!)\ \cdots\rangle_{\mathsf{store}}$$

$$\overset{R5}{\Rrightarrow}\ \langle\langle K' \ \cdots\rangle_{\mathsf{k}}\langle \mathit{false}\rangle_{\mathsf{ctrl}}\langle\cdots\ 8\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}}$$
$$\langle\cdots\ (\!|8, (\texttt{err} = 0)\neg(\texttt{x} = 0)|\!)\rangle_{\mathsf{store}}$$

$$\overset{R4}{\Rrightarrow}\ \langle\langle\cdot\rangle_{\mathsf{k}}\langle\cdot\rangle_{\mathsf{ctrl}}\langle 1,2,3,4,5,6,7,8,3\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}}\langle\ldots\rangle_{\mathsf{store}}$$

$$\Rrightarrow\ \langle 1,2,3,4,5,6,7,8,3\rangle_{\mathsf{CE}}$$

4.3 Connecting concrete and collecting semantics with predicate abstraction

"Now produce your explanation and pray make it improbable."

— Oscar Wilde, The Importance of Being Earnest.

In this section we focus on proving the correctness of the $\mathbb{K}$ definition of *SIM* collecting semantics under predicate abstraction with respect to the $\mathbb{K}$ definition of the concrete semantics. In other words, we investigate if our $\mathbb{K}$ description of model checking with predicate abstraction can be soundly used to prove certain properties for *SIM* programs.

We revise first some basics of predicate abstraction, namely the Galois connection, in the context where the concrete system in a *SIM* program *pgm*. The Galois connection is defined as $(2^{\Delta}, \sharp, \$, \mathscr{L}(\Pi))$ where $\Delta = \{\delta : V \mapsto \mathbf{Z}\}$ is the set of all states for the *SIM*

program *pgm*. The abstraction-concretization pair $(\sharp, \$)$ is defined as follows:

$$\sharp(S) = \sqcap\{\varphi \mid (\forall \delta \in S)\ \delta \vDash \varphi\}, \text{ for any subset of states } S \subseteq \Delta$$

$$\$(\varphi) = \{\delta \in \Delta \mid \delta \vDash \varphi\}, \text{ for any formula } \varphi \in \mathscr{L}(\Pi)$$

It is easy to verify that $(2^{\Delta}, \sharp, \$, \mathscr{L}(\Pi))$ forms a Galois connection. Also, it is standard that $\texttt{post}^{\sharp}$ is a sound approximation of the strongest postcondition (i.e., if $\delta \vDash \varphi$ and $\delta' \in post(\delta)$ then $\delta' \vDash \texttt{post}^{\sharp}(\varphi)$). More on these can be found in [42].

In what follows we prove a similar property about $\mathbb{K}$ executions of programs in concrete semantics and collecting semantics under predicate abstraction, respectively. In other words, we check that any concrete execution of a program can be retrieved from the meta-execution, and we state what conditions need to be satisfied such that we can derive from the meta-execution the validity of the property of interest for the given *SIM* program. We assume as given the *SIM* program $pgm = \texttt{vars}\ xs; s$, the finite set of predicates Π, and the state invariant ϕ. Moreover, to simplify the presentation, we assume that the *Configuration* described in Section 4.2 is wrapped in an outer cell $\langle\rangle_{\top^{\sharp}}$.

Theorem 2. *Any $\mathbb{K}$ execution in collecting semantics under predicate abstraction is finite.*

This theorem essentially ensures the termination of the program verification method described in the previous section.

Lemma 1. *For any $\langle \cdots\ \langle \Sigma_1 \rangle_{\mathsf{store}}\ \cdots \rangle_{\top^{\sharp}} \overset{*}{\Rrightarrow} \langle \cdots\ \langle \Sigma_2 \rangle_{\mathsf{store}}\ \cdots \rangle_{\top^{\sharp}}$, a fragment of execution in collecting semantics, we have $\Sigma_1 \subseteq \Sigma_2$.*

Proof. This is easy to see from the fact that any rule (R1-9) produces transitions that preserve the ascending inclusion of the store terms. Namely, any rule either preserves the term in the store cell or increases the term Σ in cell store by using the operator $\Sigma[\ell \rightarrowtail \check{\delta}] = \Sigma \cup \{(\!|\ell, \check{\delta}|\!)\}$ when $(\!|\ell, \check{\delta}|\!) \notin \Sigma$. □

Lemma 2. *If the $\mathbb{K}$ execution in collecting semantics encounters a transition that does not change the* store *cell, as $\langle \cdots \ \langle \Sigma_1 \rangle_{\mathsf{store}} \ \cdots \rangle_{\top^\sharp} \overset{Ri}{\Rrightarrow} \langle \cdots \ \langle \Sigma_1 \rangle_{\mathsf{store}} \ \cdots \rangle_{\top^\sharp}$ where $i = 7,8,9$, then the execution evolves either into a terminal configuration, with the rules $(R1-4)$, or into a configuration $\langle \cdots \ \langle \Sigma_2 \rangle_{\mathsf{store}} \ \cdots \rangle_{\top^\sharp}$ where $\Sigma_1 \subset \Sigma_2$, with the rules $(R5-6)$.*

This lemma ensures that any structural rule enables a computational rule, and, consequently, there is no execution in collecting semantics with a suffix that does not increment the content of the store cell.

Proof. Each rule (R7-9) changes a particular trace cell. That cell either disappears at the next step, with (R7), or places on top of the k cell an element which enables a computational rule after at most yet another structural rule. Namely, after (R8) is applied at most another structural rule, (R9), then the (R1-6), while after (R9) can be applied only (R3) or (R6). □

Proof of Theorem 2. The proof follows from Lemma 1 and Lemma 2, coupled with the fact that there is an upper bound for any store term (because any *SIM* program produces via $k^\sharp$ a finite number of labels ℓ, and Π has a finite number of predicates, hence $\mathscr{L}(\Pi)$ has a finite number of elements). □

Theorem 3. *If the concrete execution initialized with a $\langle \delta_0 \rangle_{\mathsf{state}}$ where $\delta_0 \vDash \phi$ evolves into a concrete configuration with $\langle \delta \rangle_{\mathsf{state}}$, namely*

$$\langle \langle s \rangle_{\mathsf{k}} \langle \delta_0 \rangle_{\mathsf{state}} \rangle_{\top} \overset{*}{\Rrightarrow} \langle \cdots \ \langle \delta \rangle_{\mathsf{state}} \ \cdots \rangle_{\top}$$

where $\delta \vDash \phi$, and if $\delta_0 \vDash \breve{\delta}_0$ with $\breve{\delta}_0 \Rightarrow \phi$, then the abstract execution starting with $\langle \breve{\delta}_0 \rangle_{\mathsf{ctrl}}$ evolves into an abstract configuration containing $\langle \breve{\delta} \rangle_{\mathsf{ctrl}}$, namely

$$\langle \cdots \ \langle k^\sharp(s) \rangle_{\mathsf{k}} \langle \breve{\delta}_0 \rangle_{\mathsf{ctrl}} \ \cdots \rangle_{\top^\sharp} \overset{*}{\Rrightarrow} \langle \cdots \ \langle \breve{\delta} \rangle_{\mathsf{ctrl}} \ \cdots \rangle_{\top^\sharp}$$

where $\langle \cdots \ \langle \delta \rangle_{\mathsf{state}} \ \cdots \rangle_{\top} \vDash \langle \cdots \ \langle \breve{\delta} \rangle_{\mathsf{ctrl}} \ \cdots \rangle_{\top^\sharp}$ and $\breve{\delta} \Rightarrow \phi$ hold true.

This theorem states that any $\mathbb{K}$ invariant preserving execution in the concrete semantics is sinked into a $\mathbb{K}$ execution in the collecting semantics under predicate abstraction.

Remark 1. *By* $\langle \cdots \ \langle \delta \rangle_{\mathsf{state}} \ \cdots \rangle_{\top} \vDash \langle \cdots \ \langle \check{\delta} \rangle_{\mathsf{ctrl}} \ \cdots \rangle_{\top^\sharp}$ *we understand that* $\delta \vDash \check{\delta}$*, and that the abstract computation* $\langle k^\sharp : K^\sharp \rangle_{\mathsf{k}}$ *from the* trace *cell containing* $\langle \check{\delta} \rangle_{\mathsf{ctrl}}$ *is the abstract computation of the program fragment obtained from the cell* $\langle k : K \rangle_{\mathsf{k}}$ *in* $\langle \cdots \ \langle \delta \rangle_{\mathsf{state}} \ \cdots \rangle_{\top}$.

Lemma 3. *For any* $\langle \cdots \ \langle \delta \rangle_{\mathsf{state}} \ \cdots \rangle_{\top} \Rrightarrow \langle \cdots \ \langle \delta' \rangle_{\mathsf{state}} \ \cdots \rangle_{\top}$ *a concrete transition in a concrete execution, if* $\delta \vDash \phi$ *and* $\delta' \vDash \phi$ *then for any abstract configuration* $\langle \cdots \ \langle \check{\delta} \rangle_{\mathsf{ctrl}} \ \cdots \rangle_{\top^\sharp}$ *such that* $\langle \cdots \ \langle \delta \rangle_{\mathsf{state}} \ \cdots \rangle_{\top} \vDash \langle \cdots \ \langle \check{\delta} \rangle_{\mathsf{ctrl}} \ \cdots \rangle_{\top^\sharp}$ *there is* $\langle \cdots \ \langle \check{\delta}' \rangle_{\mathsf{ctrl}} \ \cdots \rangle_{\top^\sharp}$ *an abstract configuration satisfying the following two properties:*

(1) $\langle \cdots \ \langle \check{\delta} \rangle_{\mathsf{ctrl}} \ \cdots \rangle_{\top^\sharp} \overset{*}{\Rrightarrow} \langle \cdots \ \langle \check{\delta}' \rangle_{\mathsf{ctrl}} \ \cdots \rangle_{\top^\sharp}$ *and*

(2) $\langle \cdots \ \langle \delta' \rangle_{\mathsf{state}} \ \cdots \rangle_{\top} \vDash \langle \cdots \ \langle \check{\delta}' \rangle_{\mathsf{ctrl}} \ \cdots \rangle_{\top^\sharp}$.

Proof. The proof goes by case analysis over the rules in the concrete semantics. For example, let us consider that $\langle \cdots \ \langle \delta \rangle_{\mathsf{state}} \ \cdots \rangle_{\top} \Rrightarrow \langle \cdots \ \langle \delta' \rangle_{\mathsf{state}} \ \cdots \rangle_{\top}$, the concrete transition from the hypothesis, is the result of the application of the assignment rule $x = i$ where i is the evaluation of $a : AExp$ in δ, namely $\delta(a)$. Hence $\delta' = \delta[x \mapsto \delta(a)]$.

Knowing that $\langle \cdots \ \langle \delta \rangle_{\mathsf{state}} \ \cdots \rangle_{\top} \vDash \langle \cdots \ \langle \check{\delta} \rangle_{\mathsf{ctrl}} \ \cdots \rangle_{\top^\sharp}$, then the transition in the abstract semantics is made via application of the rule (R5) for a trace cell where the computation stack is $\langle \ell : x := a \ \cdots \rangle_{\mathsf{k}}$. In the next configuration the ctrl cell of the same trace contains $\check{\delta}' = \mathtt{post}^\sharp_\phi(\check{\delta}, x := a)$. From the definition of $\mathtt{post}^\sharp_\phi$, in Figure 4.2, we can see that $\check{\delta} \wedge (x' = a) \Rightarrow \check{\delta}'[x'/x]$ which is equivalent to $\check{\delta}[x'/x] \wedge (x = a[x'/x]) \Rightarrow \check{\delta}'$. (Note that we assume x' is a fresh variable.)

From hypothesis we have $\delta \vDash \check{\delta}$, so $\delta[x \mapsto \delta(a)] \vDash \check{\delta}[x'/x] \wedge (x = a[x'/x])$. Hence $\delta' \vDash \check{\delta}[x'/x] \wedge (x = a[x'/x])$ and since $\check{\delta}[x'/x] \wedge (x = a[x'/x]) \Rightarrow \check{\delta}'$ we have $\delta' \vDash \check{\delta}'$.

Note that (R5) is the only rule that can be applied in this case though also (R3) could match when the store cell contains $\check{\delta}$. However, it can be proved that (R3) always

follows the application of (R9) because $\texttt{while}_\ell$ is the only repeating head in the stack language $K^\sharp$ using the transition relation given by the rules $(R1-9)$ from Figure 4.4. □

Proof of Theorem 3. The proof uses induction on the length of the concrete derivation and the inductive step is proved by Lemma 3. □

Theorem 4. *For any pgm,* Π*, and state invariant* ϕ*, if we have the abstract execution*

$$pgm^{\phi}_{\Pi} \Rrightarrow \langle\langle\langle\langle k^\sharp(s)\rangle_{\mathsf{k}}\ \langle \sqcap\{\varphi \mid \varphi \Rightarrow \phi\}\rangle_{\mathsf{ctrl}}\ \langle\cdot\rangle_{\mathsf{path}}\rangle_{\mathsf{trace}}\rangle_{\mathsf{traces}}\ \langle\langle\cdot\rangle_{\mathsf{store}}\ \langle\phi\rangle_{\mathsf{inv}}\rangle_{\mathsf{collect}}\rangle_{\top^\sharp}$$
$$\overset{*}{\Rrightarrow} \langle\langle\cdot\rangle_{\mathsf{traces}}\langle\langle\_\rangle_{\mathsf{store}}\ \langle\phi\rangle_{\mathsf{inv}}\rangle_{\mathsf{collect}}\rangle_{\top^\sharp} \Rrightarrow \langle\cdot\rangle_{\mathsf{CE}}$$

then, for all $\langle\delta_0\rangle_{\mathsf{state}}$ *and* $\langle\delta\rangle_{\mathsf{state}}$ *states from a concrete execution*

$$pgm \rightharpoonup \langle\langle s\rangle_{\mathsf{k}}\langle\delta_0\rangle_{\mathsf{state}}\rangle_{\mathsf{top}} \xrightarrow{*} \langle\cdots\ \langle\delta\rangle_{\mathsf{state}}\ \cdots\rangle_{\top}$$

if $\delta_0 \vDash \sqcap\{\varphi \mid \phi \Rightarrow \varphi\}$*i, then* $\delta \vDash \phi$ *holds true and* ϕ *is a state invariant.*

This theorem says that if the $\mathbb{K}$ execution in collecting semantics under predicate abstraction terminates without finding any counterexample, then the property ϕ is an invariant for any concrete execution of the program *pgm*.

Proof. We observe that since all trace terms disappear in the final state of the abstract execution, then it means that rules $(\text{R2,4})_{\boxtimes}$ were never executed. We can apply Theorem 2 (we know that $\delta_0 \vDash \sqcap\{\varphi \mid \varphi \Rightarrow \phi\}$ so $\delta_0 \vDash \phi$). Hence, there exists a valid abstract state $\breve{\delta}$ such that $\delta \vDash \breve{\delta}$. Moreover, because any intermediate $\langle\breve{\delta}\rangle_{\mathsf{ctrl}}$ is a $\texttt{post}^\sharp_\phi$ result, it means that $\breve{\delta} \Rightarrow \phi$. From these two, namely $\delta \vDash \breve{\delta}$ and $\breve{\delta} \Rightarrow \phi$, we conclude that $\delta \vDash \phi$. □

It is notorious that model checking with abstraction is not a complete procedure. Essentially, this comes from the false negative answers the model checking with ab-

straction can issue. Nevertheless, the incompleteness of abstract model checking gives rise to a bundle of work known as ”abstraction refinement”.

4.4 Related Work

There is extensive work in *software model checking*, and most of it spawned from abstract interpretation [25] and model checking [20]. If we follow the line of *predicate abstraction* that emerged in [42], we could mention only a few and important forward steps in improving the technique with counterexample-based refinement [31], localized, on-demand abstraction refinement [44], and then optimization of abstract computation using Craig interpolants [62].

Rewriting logic [65, 60] *theories* allow for nondeterminism and concurrency, while a LTL Model Checker for Maude is described in [33], and used in [1] for Java programs. A methodology for equational abstraction in the context of rewrite logic, with direct application to the Maude model checker, is proposed in [67]. An alternative (to ours) *predicate abstraction* approach is introduced to model checking under rewriting logic in [74]. A comparative study of various program semantics defined in the context of rewriting logic can be found in [97].

The $\mathbb{K}$ framework is extensively described in [80, 81], and it is used to define a series of languages, such as Scheme in [63], and a non-trivial object oriented language called KOOL in [46], as well as type systems, explicit state model checkers, and Hoare style program verifier [88]. The latest development within the $\mathbb{K}$ framework is $\mathbb{K}$-Maude, a rewriting based tool for semantics of programing languages, introduced in [29].

Having the above brief history map, let us pinpoint a few correlations with the current work. In the $\mathbb{K}$ framework area, this chapter contributes with incorporation of model checking under predicate abstraction as program meta-executions, showing that the $\mathbb{K}$ definitional style for concrete semantics of programming languages can be con-

sistently used for program verification methods. However, since $\mathbb{K}$ is a rewrite-based framework, a question arises about how is our work positioned with respect to previously described abstractions in rewriting logic systems. Firstly, the equational abstraction creates the abstract state space via equivalence classes, which inherently introduces the overhead of equivalence checking in the infrastructure. In our case, the abstract state is denoted and calculated in the traditional predicate abstraction style (as predicates conjunction), the overhead being transferred to the specialized SMT-solver for the calculation of the abstract transition. Here we need to boast only the potential benefits our approach could bring into rewriting logic from state of the art abstraction based model checking techniques. Secondly, predicate abstraction support is already introduced in rewriting logic systems by [74]. There, the concrete transitional system is provided as a rewrite theory, which is injected with the abstraction predicates to produce the rewrite theory for the abstract transitional system. Then, model checking is performed on the latter theory. There is an important aspect of our approach, which does not seem to be easy to address with the technique in [74]: we are able to also obtain the inverse transformation, from abstract to concrete. This aspect is particularly important when the model checking in the abstract system fails to verify the property, and a refinement of the abstraction needs to be performed, approached in our ongoing work [5].

4.5 Conclusions

In this chapter we studied the embedding of predicate abstraction model checking into the $\mathbb{K}$ framework. This work makes two contributions: first, it shows that model checking with predicate abstraction can be incorporated as a formal analysis approach following the very same definitional style used for concrete semantics of programming languages in $\mathbb{K}$; second, it shows how to relate the concrete semantics and the predicate based collecting semantics, and proves that the latter is correct for the original language.

Chapter 5

$\mathbb{K}$ Abstract Semantics for Alias Analysis

Motto: *"Who am I then?*
Tell me that first, and then,
if I like being that person, I'll come up:
if not, I'll stay down here till I'm somebody else"
— Lewis Carroll, Alice in Wonderland.

In this chapter we describe a case study which instantiates the collecting semantics for pushdown system specifications with the alias analysis. We emphasize that alias analysis is a benchmark for analysis and verification methods because of its extensive application in the practical areas as compiler optimizations.

Two important alias analysis problems are *may alias* which finds aliases that occur during *some* execution, and *must alias* which finds aliases that occur on *all* executions. May alias is undecidable, while must alias is uncomputable, even for languages with if statements, while loops, dynamic storage, and recursive data structures [57].

In this context, abstract interpretation provides the methodology for deriving decidable alias analysis by means of sound approximations, i.e., abstractions. As such,

depending on the type of abstraction used, alias analysis is usually distinguished in flow-sensitive or flow-insensitive, context-sensitive or context-insensitive and interprocedural or intraprocedural.

We present a flow sensitive, context sensitive, interprocedural alias analysis. We achieve this type of alias analysis by means of collecting semantics for an abstraction defined as a pushdown system specification. This pushdown system specification is presented in [91, 89] as the algebraic semantics of an imperative programming language which supports object creation, global variables, static scope and recursive procedures with local variables. Here we describe in Sections 5.1, 5.2 the $\mathbb{K}$ specification of this language, according to the semantics proposed in [91, 89], and provide in Section 5.3 the alias analysis as an instantiation of collecting semantics for this language .

5.1 A simple imperative language with object creation

"The important work of moving the world forward
does not wait to be done by perfect men."
— George Eliot.

In this section we introduce *SILK* - the $\mathbb{K}$ specification of a simple block-structured programming language which supports object creation, global variables, static scope and recursive procedures with local variables. The language is introduced and studied in [91, 89]. There the language associated with its semantics is described as finite pushdown system specification. Here we present the $\mathbb{K}$ representation of *SILK* syntax, in Figure 5.1, and discuss how and why this language is a suitable abstraction w.r.t. alias analysis for a subset of real programming languages like C or Java.

A *SILK* program consists of a finite set of procedures acting on some global and local set of reference variables which are statically scoped (i.e., all variables are visible through the entire program). Upon a procedure's call, the body of the procedure is

$Pgm^\sharp ::= \texttt{gvars:}\ Ids\ \texttt{lvars:}\ Ids\ \{\ Procs\ \}$

$Ids ::= List\{Id,","\}$

$Procs ::= ProcId\ \texttt{::}\ Ks \mid Procs\ \ Procs$

$ProcId ::= Id \mid \texttt{while(}\ Int\ \texttt{)}$

$VExp ::= Id \mid \texttt{ref(}\ Id\ \texttt{)}$

$BExp ::= VExp\ \texttt{=}\ VExp \mid VExp\ \texttt{/=}\ VExp$

$$\begin{aligned} Ks ::= {} & VExp\ \texttt{:=}\ VExp \mid VExp\ \texttt{:=}\ \texttt{new} \mid VExp\ \texttt{:=}\ \bot \\ & \mid Ks\ \texttt{;}\ Ks \mid \texttt{[}\ BExp\ \texttt{]}\ Ks \mid Ks\ \texttt{+}\ Ks \\ & \mid ProcId \mid \texttt{join(}\ Int\ \texttt{)} \end{aligned}$$

$IntBot ::= Int \mid \bot$

Figure 5.1: The *SILK* syntax

executed with the same global variables and a fresh instantiation of the local variables. Upon a procedure return, the changes to the global variables are preserved, while the local variables from the procedure's call point are restored.

The procedure body is a sequential composition of assignment, object creation, procedure call, test, and branching statements. The assignment, object creation, and procedure call statements induce changes in the state while test and branching statements give the flow. Next we describe an informal semantics for each of these statements.

The assignment statement x `:=` y passes the identity stored in y to x. Note that the association between variables and their respective identities forms the state. The set of possible identities is given by type *IntBot* which, besides integers, contains a special element $\bot$ for the not yet created objects. The statement x `:=new` creates a new object that will be referred to by the variable x.

Sequential composition B_1 `;` B_2 and conditional statements `[` b `]` B have the standard interpretation. Nondeterministic choice is implemented as two computational rules which reduce B_1 + B_2 to either B_1 or B_2. Note that collecting semantics takes into con-

sideration both nondeterministic choices. Since *SILK* is designed mainly for analysis/verification purposes, i.e., we use collecting semantics over *SILK*, for the moment we do not concern ourselves with the veridicality of the implementation of nondeterministic computations as two rewrite rules.

Given a particular program, the stack alphabet contains the closure of the statements appearing in procedure bodies together with the procedure calls and returns. Note that the procedure return statements is parameterized by the state at call site. Consequently, if the set of states is finite then the stack alphabet is finite. In [89] is described a semantics for *SILK* which is proved to give a finite set of states. The $\mathbb{K}$ specification of this semantics is described in Section 5.2. For the moment, let us observe that, based on the results in [89], *SILK* instantiated to a program produces a finite stack alphabet.

Although very simple, the language is powerful enough to encode the control flow of high-level imperative programming languages like C, and also object-oriented programs like Java. Moreover, it keeps the necessary information for alias analysis, avoiding the burdening of the state space. In the followings we survey a few common elements of imperative languages.

Ordinary statements like while, skip, and if-then-else can be expressed easily in the language using recursive procedures, conditional statements and nondeterministic choice. The call-by-value parameters of the procedures can be modeled by means of global variables assignments. Moreover, the method calls can be transformed into procedure calls by introducing the called object as an additional parameter. For more details on these, see [89].

SILK does not support any concrete data for objects, and does not have syntax for object fields or explicit object destruction. The language can be easily extended with object fields, by simply adding expressions of the form $x.f$ as variables, this feature being added in Chapter 6. Object destruction, as well as on-the-fly object declaration can be easily supported by allowing assignments like $x := \perp$. We advocate against adding

data in order to avoid the complications induced by arithmetic. Data arithmetic might add flow precision but we can still obtain good alias analysis results without it. There is however an issue w.r.t. pointer arithmetic which in the current settings is neglected. Namely, without pointer arithmetic the alias analysis produces an approximation which can guarantee only positive results, i.e., two objects are aliasing, but not negative results.

Finally, *SILK* is a sequential language. It is relatively straightforward to add parallelism or concurrency constructs. Using an interleaving semantics, these additions would only add to the branching syntactic constructs. However, in order to preserve the decidability of the alias analysis we need to work with bounds assumptions over the branching factor induced by the parallelism and/or concurrency.

For a lightweight exemplification, we transform *SIM* language described in Figure 2.1 into *SIMP* language by adding few elements to the specification. Namely, we add procedures, statically scoped pointer variables with creation and deletion, and comparisons between these pointer variables. These additions are described in Figure 5.2.

$$
\begin{array}{lcl}
\mathit{AExp} & ::= & \mathit{Var} \mid \mathit{Int} \mid \mathit{AExp} + \mathit{AExp} \mid \mathit{Int} * \mathit{AExp} \\
\mathit{BExp} & ::= & \mathit{AExp} <= \mathit{AExp} \mid \mathit{AExp} == \mathit{AExp} \\
 & \mid & \mathit{Id} == \mathit{Id} \mid \mathit{Id} =/= \mathit{Id} \\
 & \mid & \texttt{not}\ \mathit{BExp} \mid \mathit{BExp}\ \texttt{and}\ \mathit{BExp} \\
\mathit{Stmt} & ::= & \texttt{skip} \\
 & \mid & \mathit{Var} = \mathit{AExp} \\
 & \mid & \mathit{Id} = \mathit{Id} \mid \mathit{Id} = \&\mathit{Var} \\
 & \mid & \mathit{Id} = \texttt{new(Int)} \mid \mathit{Id} = \texttt{delete(Int)} \\
 & \mid & \mathit{Id}() \\
 & \mid & \mathit{Stmt}\ ;\ \mathit{Stmt} \mid \{\mathit{Stmt}\} \\
 & \mid & \texttt{if}\ \mathit{BExp}\ \texttt{then}\{\mathit{Stmt}\}\ \texttt{else}\{\mathit{Stmt}\} \\
 & \mid & \texttt{while}\ \mathit{BExp}\ \{\mathit{Stmt}\} \\
\mathit{Prc} & ::= & \mathit{Id}()\ \{\mathit{Stmt}\} \\
\mathit{Prcs} & ::= & \mathit{List}\{\mathit{PrcD}, \text{`` ''}\} \\
\mathit{Pgm} & ::= & \texttt{vars}\ \mathit{Set}\{\mathit{Var}\}; \texttt{pvars}\ \mathit{Set}\{\mathit{Id}\}; \texttt{lpvars}\ \mathit{Set}\{\mathit{Id}\}; \mathit{Prcs}
\end{array}
$$

Figure 5.2: $\mathbb{K}$ syntax of *SIMP*

$stmt^\sharp : Label \times Stmt \times Procs \to KsLabelProcs \quad procs^\sharp : Label \times Prcs \to Procs$
$|\_,\_,\_| : Label \times Stmt \times Procs \to KsLabelProcs \quad k^\sharp : Pgm \to Pgm^\sharp$

$stmt^\sharp(\ell_{in},\ \cdot,\ WPs) = |\ \cdot,\ \ell_{in},\ WPs|$
$stmt^\sharp(\ell_{in},\ X = A;\ S,\ WPs) = |K,\ \ell_{fin},\ WPs'|$
  if $|K,\ \ell_{fin},\ WPs'| := stmt^\sharp(\ell_{in},\ S,\ WPs)$
$stmt^\sharp(\ell_{in},\ R_1 = R_2;\ S,\ WPs) = |R_1 := R_2\ ;\ K,\ \ell_{fin},\ WPs'|$
  if $|K,\ \ell_{fin},\ WPs'| := stmt^\sharp(\ell_{in},\ S,\ WPs)$
$stmt^\sharp(\ell_{in},\ R = \&X;\ S,\ WPs) = |R := \texttt{ref}(X)\ ;\ K,\ \ell_{fin},\ WPs'|$
  if $|K,\ \ell_{fin},\ WPs'| := stmt^\sharp(\ell_{in},\ S,\ WPs)$
$stmt^\sharp(\ell_{in},\ R = \texttt{new(Int)};\ S,\ WPs) = |R := \texttt{new}\ ;\ K,\ \ell_{fin},\ WPs'|$
  if $|K,\ \ell_{fin},\ WPs'| := stmt^\sharp(\ell_{in},\ S,\ WPs)$
$stmt^\sharp(\ell_{in},\ R = \texttt{delete(Int)};\ S,\ WPs) = |R := \bot\ ;\ K,\ \ell_{fin},\ WPs'|$
  if $|K,\ \ell_{fin},\ WPs'| := stmt^\sharp(\ell_{in},\ S,\ WPs)$
$stmt^\sharp(\ell_{in},\ S_1;\ S_2,\ WPs) = |K_1\ ;\ K_2,\ \ell_{fin},\ WPs_2|$
  if $|K_1,\ \ell_{aux},\ WPs_1| := stmt^\sharp(\ell_{in},\ S_1 WPs)$
  and $|K_2,\ \ell_{fin},\ WPs_2| := stmt^\sharp(\ell_{aux},\ S_2,\ WPs_1)$
$stmt^\sharp(\ell_{in},\ \{S\},\ WPs) = stmt^\sharp(\ell_{in},\ S,\ WPs)$
$stmt^\sharp(\ell_{in},\ \texttt{if}\ B\ \texttt{then}\ \{S_1\}\ \texttt{else}\ \{S_2\},\ WPs) = |[B](K_1 + K_2)\ ;\ \texttt{join}(\ell_{in}),\ \ell_{fin},\ WPs_2|$
  if $|K_1,\ \ell_{aux},\ WPs_1| := stmt^\sharp(\ell_{in}+1,\ S_1,\ WPs)$
  and $|K_2,\ \ell_{fin},\ WPs_2| := stmt^\sharp(\ell_{aux},\ S_2,\ WPs_1)$
  and ($B := Id == Id$ or $B := Id =/= Id$)
$stmt^\sharp(\ell_{in},\ \texttt{if}\ B\ \texttt{then}\ \{S_1\}\ \texttt{else}\ \{S_2\},\ WPs) = |(K_1 + K_2)\ ;\ \texttt{join}(\ell_{in}),\ \ell_{fin},\ WPs_2|$
  if $|K_1,\ \ell_{aux},\ WPs_1| := stmt^\sharp(\ell_{in}+1,\ S_1,\ WPs)$
  and $|K_2,\ \ell_{fin},\ WPs_2| := stmt^\sharp(\ell_{aux},\ S_2,\ WPs_1)$
  and ($B := AExp <= AExp$ or $B := AExp == AExp$)
$stmt^\sharp(\ell_{in},\ \texttt{while}\ B\ \{S\},\ WPs) = |[B]\ \texttt{while}_{\ell_{in}},\ \ell_{fin},\ WPs'\ \ \texttt{while}_{\ell_{in}} :: (K\ ;\ \texttt{while}_{\ell_{in}})|$
  if $|K,\ \ell_{fin},\ WPs'| := stmt^\sharp(\ell_{in}+1,\ S,\ WPs)$
$stmt^\sharp(\ell_{in},\ P();\ S,\ WPs) = |P\ ;\ K,\ \ell_{fin},\ WPs'|$
  if $|K,\ \ell_{fin},\ WPs'| := stmt^\sharp(\ell_{in},\ S,\ WPs)$

$procs^\sharp(\ell_{in}, PId()\{S\}\ Ps) = PId\ ::\ K\ WPs\ procs^\sharp(\ell_{fin}, PId()\{S\}\ Ps)$
  if $|K,\ \ell_{fin},\ WPs| := stmt^\sharp(\ell_{in},\ S,\ \cdot)$
$procs^\sharp(\ell_{in},\ \cdot) = \cdot$

$k^\sharp(\texttt{vars}\ Xs;\ \texttt{pvars}\ Ps;\ \texttt{lpvars}\ Ls;\ Prcs) =$
  $\texttt{gvars}\ list(Ps)\ \ \texttt{lvars}\ list(Ls), list(\texttt{ref(}\ Xs\ \texttt{)})\ procs^\sharp(Prcs)$

Figure 5.3: The rewrite-based rules for abstracting a *SIMP* program into *SILK*

5.2 *SILK* abstract semantics

"What a different result one gets by changing the metaphor!"

— George Eliot, The Mill on the Floss.

The design of program analysis methods is based on abstract semantics that are approximations for the collecting semantics [26]. In this section we describe the $\mathbb{K}$ specification of the *SILK* abstract semantics for alias analysis.

The $\mathbb{K}$ configuration used for the abstract semantics of *SILK* is the following:

$$\langle K \rangle_{\mathsf{k}}\ \langle Map \rangle_{\mathsf{heap}}\ \langle \langle Set \rangle_{\mathsf{G}}\ \langle Set \rangle_{\mathsf{L}}\ \langle Map \rangle_{\mathsf{prcs}} \rangle_{\mathsf{pgm}}\ \langle K \rangle_{\mathsf{kAbs}}$$

The k-cell maintains the continuation, the heap-cell contains the current heap state, while the pgm-cell is designated as a program container.

The abstraction modifies the statements concerning the memory allocation, namely the object creation, procedure call and return. We devise a mechanism for defining abstractions which maintains the syntactic elements as they are in the k cell and dispatches the abstract computation in a special cell kAbs, exemplified in Fig. 5.4. As such, the abstract semantics for each statement is specified in two stages: `ping` (i.e., processing) and `ped` (i.e., processed). The `ping` stage is implemented by a structural rule which pushes in the kAbs cell the processing of the next abstract state. The `ped` stage recognizes the fact of having received the next processed abstract state in the kAbs cell, hence it performs the transition which updates the memory and the top of the k cell.

The `ping` operator has an equational implementation which ends in the `ped` normal form. Consequently, the `ping`-`ped` mechanism transforms the abstraction à la abstract interpretation into an equational abstraction. This transformation reflects the inherited orthogonality of the two abstractions. Namely, the equational abstraction is carried on an enhanced signature $\Sigma \cup \Sigma'$, where Σ is the signature of the specification for the

$$\text{RULE}\quad \langle X := \texttt{new}\ \cdots\rangle_{\mathsf{k}}\ \langle S\rangle_{\mathsf{state}}\ \langle N\rangle_{\mathsf{size}}\ \langle\ \underset{\texttt{ping}\ \langle\langle S\rangle_{\mathsf{sigma}}\langle N\rangle_{\mathsf{size}}\langle X\rangle_{\mathsf{var}}\rangle_{\mathsf{new}}}{\cdot}\ \rangle_{\mathsf{kAbs}}$$

[structural]

$$\text{RULE}\quad \langle \underset{\cdot}{X := \texttt{new}}\ \cdots\rangle_{\mathsf{k}}\ \langle\ \underset{S}{\_}\ \rangle_{\mathsf{state}}\ \langle \underset{\cdot}{\texttt{ped}\ S}\rangle_{\mathsf{kAbs}}$$

[transition]

$$\text{RULE}\quad \langle P\ \cdots\rangle_{\mathsf{k}}\ \langle S\rangle_{\mathsf{state}}\ \langle G\rangle_{\mathsf{gvars}}\ \langle L\rangle_{\mathsf{lvars}}\ \langle\ \underset{\texttt{ping}\ \langle\langle S\rangle_{\mathsf{sigma}}\langle G\rangle_{\mathsf{gs}}\langle L\rangle_{\mathsf{ls}}\rangle_{\mathsf{cal}}}{\cdot}\ \rangle_{\mathsf{kAbs}}$$

[structural]

$$\text{RULE}\quad \langle\ \underset{B \curvearrowright \texttt{restore(}\ S'\ \texttt{)}}{P}\ \cdots\rangle_{\mathsf{k}}\ \langle\ \underset{S}{S'}\ \rangle_{\mathsf{state}}\ \langle \underset{\cdot}{\texttt{ped}\ S}\rangle_{\mathsf{kAbs}}\ \langle \cdots\ \ P \mapsto B\ \cdots\rangle_{\mathsf{pgm}}$$

[transition]

$$\text{RULE}\quad \left(\langle\ \underset{\texttt{ping}\ \langle\langle S\rangle_{\mathsf{sigma}}\langle S'\rangle_{\mathsf{sigma1}}\langle Set(G)\rangle_{\mathsf{gi-gn}}\langle Set(G)\rangle_{\mathsf{g1-gn}}\langle S'\rangle_{\mathsf{sigmai}}\rangle_{\mathsf{ret}}}{\langle\texttt{restore(}\ S'\ \texttt{)}\ \cdots\rangle_{\mathsf{k}}\ \langle S\rangle_{\mathsf{state}}\ \langle G\rangle_{\mathsf{gvars}}\ \cdot}\ \rangle_{\mathsf{kAbs}}\right)$$

[structural]

$$\text{RULE}\quad \langle \underset{\cdot}{\texttt{restore(}\ \_\ \texttt{)}}\ \cdots\rangle_{\mathsf{k}}\ \langle\ \underset{S}{\_}\ \rangle_{\mathsf{state}}\ \langle \underset{\cdot}{\texttt{ped}\ S}\rangle_{\mathsf{kAbs}}$$

[transition]

Figure 5.4: The `ping`-`ped` abstraction mechanism.

"concrete" semantics. The restriction imposed for this particular equational abstraction is that, besides the structural rules initializing the `ping` stage, all the equations added for the abstraction are over the terms in Σ'.

For decidability reasons, the collecting semantics has to work with a finite state model. In particular, the abstraction for alias analysis for *SILK* presented in [89, 91] uses a memory allocation protocol with *abstract* memory addresses. As such, the state can be seen as maintaining *abstract memory addresses* which represent the alias equivalence classes. The abstract memory allocation protocol introduces the so-called freeze variables, which are used to compare the alias partitions of variables before and after executing a procedure. The freeze variables copy the abstract addresses of the global

$K ::= \texttt{ping}\ \mathit{BagItem} \mid \texttt{ped}\ \mathit{Map}$

RULE $\texttt{ping}\ \langle \cdots\ \langle \cdot \rangle_{\mathsf{gi-gn}}\ \langle Sn \rangle_{\mathsf{sigmai}}\ \cdots \rangle_{\mathsf{ret}} \Rightarrow \texttt{ped}\ Sn$
[end structural]

RULE $\langle \cdots\ Gi \mapsto \bot\ \cdots \rangle_{\mathsf{sigma}}\ \langle \cdots\ \underset{\cdot}{Gi}\ \cdots \rangle_{\mathsf{gi-gn}}\ \langle \cdots\ Gi \mapsto \underset{\bot}{\_}\ \cdots \rangle_{\mathsf{sigmai}}$
[step1. structural]

RULE $\langle \cdots\ Gi \mapsto K\ \ Gj \mapsto K\ \cdots \rangle_{\mathsf{sigma}}\ \langle \underset{\cdot}{Gi}\ \ Gs \rangle_{\mathsf{gi-gn}}\ \langle \cdots\ Gj\ \cdots \rangle_{\mathsf{g1-gn}}$

$\langle \cdots\ Gj \mapsto K'\ \ Gi \mapsto \underset{K'}{\_}\ \cdots \rangle_{\mathsf{sigmai}}$ when $\neg_{Bool}\ Gj\ \texttt{in}\ Gi\ \ Gs$
[step2. structural]

RULE $\langle Gi \mapsto K\ \ \texttt{frz(}\ G\ \texttt{)} \mapsto K\ \ S \rangle_{\mathsf{sigma}}\ \langle \cdots\ G \mapsto K'\ \cdots \rangle_{\mathsf{sigma1}}$
$\langle \underset{\cdot}{Gi}\ \ Gs \rangle_{\mathsf{gi-gn}}\ \ \langle G\ \ GGs \rangle_{\mathsf{g1-gn}}\ \ \langle \cdots\ Gi \mapsto \underset{K'}{\_}\ \cdots \rangle_{\mathsf{sigmai}}$
when $\neg_{Bool}\ K\ \texttt{in}\ S\ \texttt{(}\ G\ \ GGs\ -_{Set}\ Gi\ \ Gs\ \texttt{)}$
[step3. structural]

RULE $\langle Gi \mapsto K\ \ S \rangle_{\mathsf{sigma}}\ \langle \underset{\cdot}{Gi}\ \ Gs \rangle_{\mathsf{gi-gn}}\ \langle GGs \rangle_{\mathsf{g1-gn}}$

$\langle \underset{Si\ \texttt{[}\ \texttt{nextFreeValue(}\ S\ \texttt{(}\ Gi\ \texttt{)}\ \texttt{,}\ \texttt{|}\ \texttt{values}\ S\ \texttt{|}\ \texttt{,}\ S\ \texttt{)}\ /\ Gi\ \texttt{]}}{Si} \rangle_{\mathsf{sigmai}}$
when $\neg_{Bool}\ K\ \texttt{in}\ S\ \texttt{(}\ \texttt{frzSet(}\ GGs\ \texttt{))} \wedge_{Bool} \neg_{Bool}\ K\ \texttt{in}\ S\ \texttt{(}\ GGs\ -_{Set}\ Gi\ \ Gs\ \texttt{)}$
$\wedge_{Bool}\ K \neq_{Bool} \bot$
[steps4-5. structural]

Figure 5.5: The `ping`-`ped` structural rules for the abstract procedure return statement.

variables upon the procedure call such that the procedure body maintains a sound representation of alias equivalence classes w.r.t. the procedure's return. Hence, the abstract addresses space is $\{\bot\} \cup \{1..2|V_g| + |V_l|\}$, where V_g and V_l are the sets of global and local variables, respectively. Consequently, the abstract state space of *SILK* programs is finite because the programs have a finite number of variables, either global or local.

The procedure return is the most interesting abstract operator and constitutes the essence of the abstraction. The idea of this step is to leave the local variables as they

were at the procedure's call point and focus on reassigning the global variables according to the current alias partition. This operator is described in [89, 91] as an iterative algorithm which reads as in Algorithm 3.

Algorithm 3: The iterative algorithm for obtaining the next state upon the procedure's return as described in [89, 91].

- let σ be the current state and σ' be the state from the procedure call point (maintained in the k cell in the `restore()` operator);
- let n be the number of global variables $g_1..g_n$ and $\sigma_0 = \sigma'$;
- **for** $1 \leq i \leq n$ do the following **if-then-else** steps:
  1. **if** $\sigma(g_i) = \perp$ **then** $\sigma_i = \sigma_{i-1}[\perp/g_i]$ **else**
  2. **if** $\sigma(g_i) = \sigma(g_j)$, for some $j < i$, **then** $\sigma_i = \sigma_{i-1}[\sigma_{i-1}(g_j)/g_i]$ **else**
  3. **if** $\sigma(g_i) = \sigma(g')$, for some freeze variable g', **then** $\sigma_i = \sigma_{i-1}[\sigma'(g)/g_i]$ **else**
  4. **if** in σ_{i-1} all indices except $\perp$ are used **then** $\sigma_i = \sigma_{i-1}$ **else**
  5. $\sigma_i = \sigma_{i-1}[k/g_i]$, where k is the smallest abstract address not used by σ_{i-1}.

In Fig. 5.5 we present the specification of this algorithm in $\mathbb{K}$. Namely, the cells $\langle\rangle_{\mathsf{sigma}}, \langle\rangle_{\mathsf{sigma1}}, \langle\rangle_{\mathsf{sigmai}}$ contain the maps σ, σ' and σ_i, respectively. The iteration is maintained via the cell $\langle\rangle_{\mathsf{gi-gn}}$ which contains the global variables that are left to be processed by the algorithm. The cell $\langle\rangle_{\mathsf{g1-gn}}$ maintains all the global variables, while the freeze variables g' are represented via the `frz()` operator. Note that all the $\mathbb{K}$ rules specifying Algorithm 3 are structural rules. This helps ensuring that the *SILK* definition is a pushdown system specification.

Several considerations regarding the algorithm in Fig. 5.5: The iterative processing is implemented via the gi$-$gn cell. The first rule finalizes the iteration by sending the

result to the ped operator. The concordance between the steps 1.-5. in the algorithm are reflected by the rule attributes. The if-then-else cascade is implemented via matching and conditions.

We emphasize the following arguments w.r.t. the accuracy of the implementation:

- In the step2. rule, if K is $\perp$ (i.e., $\sigma(g_i) = \perp$) then, by an well-founded inductive reasoning, also K' is $\perp$ (i.e., $\sigma_i(g_j) = \perp$, because $\sigma(g_j) = \perp$ and $j < i$). Hence, the update in the sigmai cell is the same in both step1. and step2. rules, so we do not need to enforce the condition $K \neq_{Bool} \perp$ in step2.

- We argue the omission of the condition $K \neq_{Bool} \perp$ in the step3. rule with a similar reasoning as above. The condition in the step3. rule enforces the application of the step3. only on the "else branch" of the step2.

- The merge of the cases 4. and 5. of the algorithm into the steps4-5. rule is induced by the operator `nextFreeValue`.

Lemma 4. *The application of the* `ping-ped` *structural rules in Fig. 5.5 reaches the* `ped` *Map normal form after* $n+1$ *steps and the Sn term from* `ped` *Sn is the* σ_n *map computed by the algorithm 5.*

Proof. At each step the rules in Fig. 5.5 consume and element from the gi − gn cell, which initially contains the set og global variables. While the cell gi − gn is nonempty, one of the rules step* rules can still apply because of the conditions and the matching on the sigma cell cover all possible cases. After n applications of the step* rules, the gi − gn cell becomes empty, and the fin rule can apply, rendering the ped form of the next state.

Note, however that this reasoning does not take into consideration the rules defining `nextFreeValue`. In order not to complicate the presentation any further, we assume that the rules for `nextFreeValue` apply up to the normal form (an integer), and then

similarly for the *Map* substitution operator. Enforcing a clean implementation would either involve yet more `ping-ped` calls for `nextFreeValue` an so on, or would need the function constructs. □

Theorem 5. *The SILK specification is bisimilar with the abstract semantics defined in [91, 89].*

Proof. The main idea is that the state in the abstract semantics is maintained in the `state` cell, while the stack in maintained in the `k` cell. The only transition rules in the $\mathbb{K}$ specification for *SILK* are the ones in Fig. 5.4, corresponding one-to-one with the transition schemas defined for the abstract semantics presented in [91, 89]. Moreover, all the additional functionality described for the transitions in the abstract semantics are specified in $\mathbb{K}$ using the `ping-ped` mechanisms, similar to the one discussed in lemma 4.

We do not insist on thoroughly covering the proof of this result because it would involve reloading all the context in [91, 89] and a tedious structural induction on the computation. □

Observation 3. *The abstract lattice is the flat lattice formed by all aliasing equivalence classes over the* `heap` *cell. Note that these equivalence classes are formed by the partition produced over the set of all variables by the map representing the content of the* `heap` *cell.*

Furthermore, the Galois connection is defined as follows. The abstraction # *maps a concrete state in SIMP into its afferent aliasing equivalence class, while the concretization* $ *maps an aliasing equivalence class into the set of all concrete states preserving that aliasing. We do not insist on further formalization because we did not explicitly present the concrete configuration and the concrete state.*

5.3 Alias analysis for *SILK*

"I desire no future that will break the ties of the past."

— George Eliot, The Mill on the Floss.

Having the abstract semantics for *SILK*, we can present the alias analysis as an instantiation of the collecting semantics settings introduced in Section 3.2. Namely, we describe the content of the collect cell and the relation $\ll$.

Alias analysis is a technique used in compiler theory to determine if a storage location can be accessed in more than one way. Two pointers are said to be aliased if they point to the same location. The syntax of *SILK* determines the nature of the derived alias analysis. Namely, we present a flow sensitive, interprocedural alias analysis.

The collect cell maintains two cells, heads and aliases as follows:

$$\langle\langle\langle\langle K\rangle_{\mathsf{k}}\langle Map\rangle_{\mathsf{state}}\rangle_{\mathsf{head}*}\rangle_{\mathsf{heads}}\ \langle\langle Map\rangle_{\mathsf{state}*}\rangle_{\mathsf{aliases}}\rangle_{\mathsf{collect}}$$

The heads cell contains information used by the well-founded relation $\ll$ to stop the exhaustive execution induced by the collecting semantics. At the end of the exhaustive execution, the aliases cell contains all the necessary aliasing information. Hence, the aliases cell can be analyzed, either post-mortem or on-the-fly, with queries like $p \stackrel{?}{=} q$ for a demand-driven alias analysis.

Recall that the collecting semantics performs an exhaustive execution by means of fixpoint iteration. So, the relation $\ll$ helps in realizing that certain computations reached a "partial" fixpoint, i.e., they cannot contribute anymore to the collected result. Hence, we rely on the repetitive stack pattern guaranteed to be discovered in the pushdown systems and define the relation $\ll$ using matching as follows:

- $\ll$ is false, i.e., the currently considered computation trace which matches the pattern $\langle\langle P \curvearrowright X \curvearrowright X \curvearrowright Y\rangle_{\mathsf{k}}\langle S\rangle_{\mathsf{state}}\rangle_{\mathsf{trace}}$, where P is a procedure name, does

not contribute to the `update` of the collect cell, when the computation trace is a repetition of a previously collected head. Hence, the second rule in Fig. 3.1 is instantiated as:

$$\text{RULE}\quad \frac{\langle\langle P \curvearrowright X \curvearrowright X \curvearrowright Y\rangle_{\mathsf{k}}\langle S\rangle_{\mathsf{state}}\rangle_{\mathsf{trace}}}{\cdot}\ \langle\langle P \curvearrowright X \curvearrowright Y\rangle_{\mathsf{k}}\langle S\rangle_{\mathsf{state}}\rangle_{\mathsf{head}}$$

- $\ll$ is true when the computation trace is not a repetition of any of the previously collected heads. Hence, the first rule in Fig. 3.1 reads as:

$$\text{RULE}\quad \frac{\langle\langle P \curvearrowright K\rangle_{\mathsf{k}}\langle S\rangle_{\mathsf{state}}\rangle_{\mathsf{trace}}}{\cdot}\ \frac{\cdot}{\langle\langle \texttt{next(}\ P \curvearrowright K\ \texttt{)}\rangle_{\mathsf{k}}\langle \texttt{next(}\ S\ \texttt{)}\rangle_{\mathsf{state}}\rangle_{\mathsf{trace}}}$$

$$\langle \cdots\ \frac{\cdot}{\langle S\rangle_{\mathsf{state}}}\ \cdots\rangle_{\mathsf{aliases}}\ \langle Hs\ \frac{\cdot}{\langle\langle P \curvearrowright K\rangle_{\mathsf{k}}\langle S\rangle_{\mathsf{state}}\rangle_{\mathsf{head}}}\rangle_{\mathsf{heads}}$$

when $\langle P \curvearrowright K\rangle_{\mathsf{k}}\langle S\rangle_{\mathsf{state}} \not\in^{\texttt{rep}} Hs$

Note that, for efficiency reasons, we apply the $\not\in^{\texttt{rep}}$ test only when the current computation is a procedure call while in all the other cases the $\ll$ relation is considered to be true. The reason for this simplification is the fact that the procedure call is the only source of infiniteness in *SILK* computations.

Theorem 6. *The exhaustive execution given by the collecting semantics terminates and produces the reachable state set in the* aliases *cell in the configuration term with no* trace *cell.*

Proof. The proof follows directly from the result in [14]. □

Both must and may alias analysis can be defined as predicates over the contents of the aliases cell. The simplest but not always efficient choice for the implementation is to insert these predicates in a rule which enforces all emptiness of the traces cell, when the entire set of reachable states is available. This can be seen as a post-mortem alias

analysis. In the case of must alias analysis, the exhaustive execution (i.e., the computation of the reachable state set) could be earlier stopped. This involves the coupling of the exhaustive execution with the property check which resides in a different, richer specification for the operator `update(`$\mho$`,_)`.

5.3.1 Example

We exemplify a demand driven must analysis on the program in Figure 5.6, with the query $x \stackrel{?}{=} y$. Note that `mark` stands for `join`$(I : Int)$.

```
macro pgmEx =
gvars:  x
lvars:  y
{ main ::  y := new; (x := y + x := new); mark; main; y := x }
```

Figure 5.6: A simple *SILK* program.

Note that in this example we use `mark` as a point of interest for answering the query "What is the alias information at that particular point in the program?". A global alias analysis will use markings for each program point.

$$
\begin{aligned}
& \langle\langle \texttt{main} \rangle_{\mathsf{k}} \langle \texttt{frz(x)} \mapsto \bot\ \texttt{x} \mapsto \bot\ \texttt{y} \mapsto \bot \rangle_{\mathsf{state}} \rangle_{\mathsf{trace}} \langle \cdot \rangle_{\mathsf{heads}} \langle \cdot \rangle_{\mathsf{aliases}} \\
\Rrightarrow\ & \langle \langle B_0 \curvearrowright \texttt{restore(frz(x)} \mapsto \bot\ \texttt{x} \mapsto \bot\ \texttt{y} \mapsto \bot \texttt{)} \rangle_{\mathsf{k}} \\
& \quad \langle \texttt{frz(x)} \mapsto \bot\ \texttt{x} \mapsto \bot\ \texttt{y} \mapsto \bot \rangle_{\mathsf{state}} \rangle_{\mathsf{trace}} \\
& \quad \langle \langle \langle \texttt{main} \rangle_{\mathsf{k}} \langle \texttt{frz(x)} \mapsto \bot\ \texttt{x} \mapsto \bot\ \texttt{y} \mapsto \bot \rangle_{\mathsf{state}} \rangle_{\mathsf{head}} \rangle_{\mathsf{heads}} \\
& \quad \text{where } B_0 \text{ is } \texttt{y} := \texttt{new}; (\texttt{x} := \texttt{y} + (\texttt{x} := \texttt{new}; \texttt{mark})); \texttt{main}; \texttt{y} := \texttt{x} \\
\Rrightarrow\ & \langle \langle \texttt{y} := \texttt{new} \curvearrowright B_1 \curvearrowright \texttt{restore(frz(x)} \mapsto \bot\ \texttt{x} \mapsto \bot\ \texttt{y} \mapsto \bot \texttt{)} \rangle_{\mathsf{k}} \\
& \quad \langle \texttt{frz(x)} \mapsto \bot\ \texttt{x} \mapsto \bot\ \texttt{y} \mapsto \bot \rangle_{\mathsf{state}} \rangle_{\mathsf{trace}} \\
& \quad \text{where } B_1 \text{ is } (\texttt{x} := \texttt{y} + \texttt{x} := \texttt{new}); \texttt{mark}; \texttt{main}; \texttt{y} := \texttt{x} \\
\Rrightarrow\ & \langle \langle B_1 \curvearrowright \texttt{restore(frz(x)} \mapsto \bot\ \texttt{x} \mapsto \bot\ \texttt{y} \mapsto \bot \texttt{)} \rangle_{\mathsf{k}} \\
& \quad \langle \texttt{frz(x)} \mapsto \bot\ \texttt{x} \mapsto \bot\ \texttt{y} \mapsto 0 \rangle_{\mathsf{state}} \rangle_{\mathsf{trace}}
\end{aligned}
$$

$$
\begin{aligned}
\Rrightarrow\ & \langle\langle(\mathtt{x} := \mathtt{y} + \mathtt{x} := \mathtt{new}) \curvearrowright B_2 \curvearrowright \mathtt{restore}(\mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto \bot\ \mathtt{y} \mapsto \bot)\rangle_{\mathsf{k}} \\
& \langle \mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto \bot\ \mathtt{y} \mapsto 0 \rangle_{\mathsf{state}} \rangle_{\mathsf{trace}} \\
& \text{where } B_2 \text{ is } \mathtt{mark}; \mathtt{main}; \mathtt{y} := \mathtt{x} \\
\Rrightarrow\ & \langle\langle \mathtt{x} := \mathtt{new} \curvearrowright B_2 \curvearrowright \mathtt{restore}(\mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto \bot\ \mathtt{y} \mapsto \bot)\rangle_{\mathsf{k}} \\
& \langle \mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto \bot\ \mathtt{y} \mapsto 0 \rangle_{\mathsf{state}} \rangle_{\mathsf{trace}} \\
& \langle\langle \mathtt{x} := \mathtt{y} \curvearrowright B_2 \curvearrowright \mathtt{restore}(\mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto \bot\ \mathtt{y} \mapsto \bot)\rangle_{\mathsf{k}} \\
& \langle \mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto \bot\ \mathtt{y} \mapsto 0 \rangle_{\mathsf{state}} \rangle_{\mathsf{trace}} \\
\Rrightarrow\ & \langle\langle B_2 \curvearrowright \mathtt{restore}(\mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto \bot\ \mathtt{y} \mapsto \bot)\rangle_{\mathsf{k}} \\
& \langle \mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto 1\ \mathtt{y} \mapsto 0 \rangle_{\mathsf{state}} \rangle_{\mathsf{trace}} \cdots \\
\Rrightarrow\ & \langle\langle \mathtt{mark} \curvearrowright B_3 \curvearrowright \mathtt{restore}(\mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto \bot\ \mathtt{y} \mapsto \bot)\rangle_{\mathsf{k}} \\
& \langle \mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto 1\ \mathtt{y} \mapsto 0 \rangle_{\mathsf{state}} \rangle_{\mathsf{trace}} \cdots \\
& \text{where } B_3 \text{ is } \mathtt{main}; \mathtt{y} := \mathtt{x} \\
\Rrightarrow\ & \langle\langle B_3 \curvearrowright \mathtt{restore}(\mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto \bot\ \mathtt{y} \mapsto \bot)\rangle_{\mathsf{k}} \\
& \langle \mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto 1\ \mathtt{y} \mapsto 0 \rangle_{\mathsf{state}} \rangle_{\mathsf{trace}} \cdots \\
& \langle\langle \mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto 1\ \mathtt{y} \mapsto 0 \rangle_{\mathsf{state}} \rangle_{\mathsf{aliases}} \\
\Rrightarrow\ & \langle\langle \mathtt{main} \curvearrowright B_4 \curvearrowright \mathtt{restore}(\mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto \bot\ \mathtt{y} \mapsto \bot)\rangle_{\mathsf{k}} \\
& \langle \mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto 1\ \mathtt{y} \mapsto 0 \rangle_{\mathsf{state}} \rangle_{\mathsf{trace}} \cdots \\
& \langle\langle \mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto 1\ \mathtt{y} \mapsto 0 \rangle_{\mathsf{state}} \rangle_{\mathsf{aliases}} \\
& \text{where } B_4 \text{ is } \mathtt{y} := \mathtt{x} \\
\Rrightarrow\ & \langle\langle B_0 \curvearrowright \mathtt{restore}(\mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto 1\ \mathtt{y} \mapsto 0) \\
& \quad \curvearrowright B_4 \curvearrowright \mathtt{restore}(\mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto \bot\ \mathtt{y} \mapsto \bot)\rangle_{\mathsf{k}} \\
& \langle \mathtt{frz}(\mathtt{x}) \mapsto 1\ \mathtt{x} \mapsto 1\ \mathtt{y} \mapsto \bot \rangle_{\mathsf{state}} \rangle_{\mathsf{trace}} \cdots \\
& \langle \cdots \langle\langle \mathtt{main} \curvearrowright B_4 \curvearrowright \mathtt{restore}(\mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto \bot\ \mathtt{y} \mapsto \bot)\rangle_{\mathsf{k}} \\
& \quad \langle \mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto 1\ \mathtt{y} \mapsto 0 \rangle_{\mathsf{state}} \rangle_{\mathsf{head}} \cdots \rangle_{\mathsf{heads}}
\end{aligned}
$$

$$
\begin{aligned}
\Rrightarrow\ & \langle\langle \mathtt{y} := \mathtt{new} \curvearrowright B_1 \curvearrowright \mathtt{restore}(\mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto 1\ \mathtt{y} \mapsto 0) \dots \\
& \quad \curvearrowright B_4 \curvearrowright \mathtt{restore}(\mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto \bot\ \mathtt{y} \mapsto \bot) \rangle\rangle_{\mathsf{k}} \\
& \quad \langle \mathtt{frz}(\mathtt{x}) \mapsto 1\ \mathtt{x} \mapsto 1\ \mathtt{y} \mapsto \bot \rangle_{\mathsf{state}} \rangle_{\mathsf{trace}} \dots \\
\Rrightarrow\ & \langle\langle B_1 \curvearrowright \mathtt{restore}(\mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto 1\ \mathtt{y} \mapsto 0) \\
& \quad \curvearrowright B_4 \curvearrowright \mathtt{restore}(\mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto \bot\ \mathtt{y} \mapsto \bot) \rangle\rangle_{\mathsf{k}} \\
& \quad \langle \mathtt{frz}(\mathtt{x}) \mapsto 1\ \mathtt{x} \mapsto 1\ \mathtt{y} \mapsto 0 \rangle_{\mathsf{state}} \rangle_{\mathsf{trace}} \dots \\
\overset{*}{\Rrightarrow}\ & \langle\langle \mathtt{main} \curvearrowright B_4 \curvearrowright \mathtt{restore}(\mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto 1\ \mathtt{y} \mapsto 0) \\
& \quad \curvearrowright B_4 \curvearrowright \mathtt{restore}(\mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto \bot\ \mathtt{y} \mapsto \bot) \rangle\rangle_{\mathsf{k}} \\
& \quad \langle \mathtt{frz}(\mathtt{x}) \mapsto 1\ \mathtt{x} \mapsto 2\ \mathtt{y} \mapsto 0 \rangle_{\mathsf{state}} \rangle_{\mathsf{trace}} \dots \\
\Rrightarrow\ & \langle\langle B_0 \curvearrowright \mathtt{restore}(\mathtt{frz}(\mathtt{x}) \mapsto 1\ \mathtt{x} \mapsto 2\ \mathtt{y} \mapsto 0) \\
& \quad \curvearrowright B_4 \curvearrowright \mathtt{restore}(\mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto 1\ \mathtt{y} \mapsto 0) \\
& \quad \curvearrowright B_4 \curvearrowright \mathtt{restore}(\mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto \bot\ \mathtt{y} \mapsto \bot) \rangle\rangle_{\mathsf{k}} \\
& \quad \langle \mathtt{frz}(\mathtt{x}) \mapsto 2\ \mathtt{x} \mapsto 2\ \mathtt{y} \mapsto \bot \rangle_{\mathsf{state}} \rangle_{\mathsf{trace}} \dots \\
\overset{*}{\Rrightarrow}\ & \langle\langle \mathtt{main} \curvearrowright B_4 \curvearrowright \mathtt{restore}(\mathtt{frz}(\mathtt{x}) \mapsto 1\ \mathtt{x} \mapsto 2\ \mathtt{y} \mapsto 0) \\
& \quad \curvearrowright B_4 \curvearrowright \mathtt{restore}(\mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto 1\ \mathtt{y} \mapsto 0) \\
& \quad \curvearrowright B_4 \curvearrowright \mathtt{restore}(\mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto \bot\ \mathtt{y} \mapsto \bot) \rangle\rangle_{\mathsf{k}} \\
& \quad \langle \mathtt{frz}(\mathtt{x}) \mapsto 2\ \mathtt{x} \mapsto 1\ \mathtt{y} \mapsto 0 \rangle_{\mathsf{state}} \rangle_{\mathsf{trace}} \dots \\
\overset{*}{\Rrightarrow}\ & \langle\langle \mathtt{main} \curvearrowright B_4 \curvearrowright \mathtt{restore}(\mathtt{frz}(\mathtt{x}) \mapsto 2\ \mathtt{x} \mapsto 1\ \mathtt{y} \mapsto 0) \\
& \quad \curvearrowright B_4 \curvearrowright \mathtt{restore}(\mathtt{frz}(\mathtt{x}) \mapsto 1\ \mathtt{x} \mapsto 2\ \mathtt{y} \mapsto 0) \\
& \quad \curvearrowright B_4 \curvearrowright \mathtt{restore}(\mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto 1\ \mathtt{y} \mapsto 0) \\
& \quad \curvearrowright B_4 \curvearrowright \mathtt{restore}(\mathtt{frz}(\mathtt{x}) \mapsto \bot\ \mathtt{x} \mapsto \bot\ \mathtt{y} \mapsto \bot) \rangle\rangle_{\mathsf{k}} \\
& \quad \langle \mathtt{frz}(\mathtt{x}) \mapsto 1\ \mathtt{x} \mapsto 2\ \mathtt{y} \mapsto 0 \rangle_{\mathsf{state}} \rangle_{\mathsf{trace}} \dots
\end{aligned}
$$

$$
\begin{aligned}
\overset{*}{\Rightarrow}\ & \langle\langle \texttt{main} \curvearrowright B_4 \curvearrowright \texttt{restore}(\texttt{frz(x)}\mapsto 1\ \texttt{x}\mapsto 2\ \texttt{y}\mapsto 0) \\
& \quad \curvearrowright B_4 \curvearrowright \texttt{restore}(\texttt{frz(x)}\mapsto 2\ \texttt{x}\mapsto 1\ \texttt{y}\mapsto 0) \\
& \quad \curvearrowright B_4 \curvearrowright \texttt{restore}(\texttt{frz(x)}\mapsto 1\ \texttt{x}\mapsto 2\ \texttt{y}\mapsto 0) \\
& \quad \curvearrowright B_4 \curvearrowright \texttt{restore}(\texttt{frz(x)}\mapsto \bot\ \texttt{x}\mapsto 1\ \texttt{y}\mapsto 0) \\
& \quad \curvearrowright B_4 \curvearrowright \texttt{restore}(\texttt{frz(x)}\mapsto \bot\ \texttt{x}\mapsto \bot\ \texttt{y}\mapsto \bot)\rangle_{\mathsf{k}} \\
& \quad \langle \texttt{frz(x)}\mapsto 2\ \texttt{x}\mapsto 1\ \texttt{y}\mapsto 0\rangle_{\mathsf{state}}\rangle_{\mathsf{trace}} \cdots \\
\overset{*}{\Rightarrow}\ & \langle\langle \texttt{main} \curvearrowright B_4 \curvearrowright \texttt{restore}(\texttt{frz(x)}\mapsto 2\ \texttt{x}\mapsto 1\ \texttt{y}\mapsto 0) \\
& \quad \curvearrowright B_4 \curvearrowright \texttt{restore}(\texttt{frz(x)}\mapsto 1\ \texttt{x}\mapsto 2\ \texttt{y}\mapsto 0) \\
& \quad \curvearrowright B_4 \curvearrowright \texttt{restore}(\texttt{frz(x)}\mapsto 2\ \texttt{x}\mapsto 1\ \texttt{y}\mapsto 0) \\
& \quad \curvearrowright B_4 \curvearrowright \texttt{restore}(\texttt{frz(x)}\mapsto 1\ \texttt{x}\mapsto 2\ \texttt{y}\mapsto 0) \\
& \quad \curvearrowright B_4 \curvearrowright \texttt{restore}(\texttt{frz(x)}\mapsto \bot\ \texttt{x}\mapsto 1\ \texttt{y}\mapsto 0) \\
& \quad \curvearrowright B_4 \curvearrowright \texttt{restore}(\texttt{frz(x)}\mapsto \bot\ \texttt{x}\mapsto \bot\ \texttt{y}\mapsto \bot)\rangle_{\mathsf{k}} \\
& \quad \langle \texttt{frz(x)}\mapsto 1\ \texttt{x}\mapsto 2\ \texttt{y}\mapsto 0\rangle_{\mathsf{state}}\rangle_{\mathsf{trace}} \cdots
\end{aligned}
$$

This trace terminates with the application of the rule:

$$\textsc{rule}\quad \frac{\langle\langle P \curvearrowright X \curvearrowright X \curvearrowright Y\rangle_{\mathsf{k}}\ \langle S\rangle_{\mathsf{state}}\rangle_{\mathsf{trace}}}{\cdot}\ \langle\langle P \curvearrowright X \curvearrowright Y\rangle_{\mathsf{k}}\langle S\rangle_{\mathsf{state}}\rangle_{\mathsf{head}}$$

where P is matched by `main`, the state S is matched by $\texttt{frz(x)}\mapsto 1\ \texttt{x}\mapsto 2\ \texttt{y}\mapsto 0$, X is matched by

$$B_4 \curvearrowright \texttt{restore}(\texttt{frz(x)}\mapsto 2\ \texttt{x}\mapsto 1\ \texttt{y}\mapsto 0) \curvearrowright B_4 \curvearrowright \texttt{restore}(\texttt{frz(x)}\mapsto 1\ \texttt{x}\mapsto 2\ \texttt{y}\mapsto 0)$$

and Y is matched by

$$B_4 \curvearrowright \texttt{restore}(\texttt{frz(x)}\mapsto \bot\ \texttt{x}\mapsto 1\ \texttt{y}\mapsto 0) \curvearrowright B_4 \curvearrowright \texttt{restore}(\texttt{frz(x)}\mapsto \bot\ \texttt{x}\mapsto \bot\ \texttt{y}\mapsto \bot)\rangle_{\mathsf{k}}$$

Note that on the particular trace we exhibit in the above unfolding, the `mark` point is never reached by a state with $\mathtt{x} = \mathtt{y}$ (i.e., namely both `x` and `y` mapped to the same value). Hence the query $\mathtt{x} \stackrel{?}{=} \mathtt{y}$ is not true on this trace.

At branching points, we assign to each trace a unique watermark (as a Boolean sequence) which is propagated to the state cells in aliases. As such, at the end of the analysis we can reason about the execution paths. For example, until the end of the analysis, the branch with $\mathtt{x} := \mathtt{new}$ will produce in the aliases cell the states: $(\bot,1,0)_1, (1,2,0)_{11}, (2,1,0)_{111}, (1,2,0)_{1111}, (2,1,0)_{11111}$ (where by $(a,b,c)_i$ we understand $\langle \mathtt{frz}(\mathtt{x}) \mapsto a\ \mathtt{x} \mapsto b\ \mathtt{y} \mapsto c \rangle_{\mathsf{state}}$, while i is the watermark). A query for must alias $\mathtt{x} \stackrel{?}{=} \mathtt{y}$ identifies this execution path and answers "no" to the query. Meanwhile, the same query for may alias identifies the collected state $(\bot,0,0)_0$ and answers "yes" to the same query.

5.4 Related work

The core of the current work is the semantics study given in [91, 89] and the ideas promoted in the discussions with the authors of [91, 89, 6]. We take this opportunity to extend our thanks for the enriching collaboration on [6] where *SILK* is extended with fields and the programs are verified, using the Maude LTL Model Checker [33], via bounded model checking for a regular language of heap properties.

The program analysis procedure approached here is a refinement of the idea presented in [4]. Nevertheless, that work and the current one subscribe to the lines promoted by the $\mathbb{K}$ framework and the matching logic [30].

The collecting semantics case study we use is alias analysis for imperative or object oriented languages. Alias analysis is a technique in compiler theory, used to determine if a storage location may be accessed in more than one way [48]. Two pointers are said to be aliased if they point to the same location. Alias analysis techniques are usually

classified by flow-sensitivity and context-sensitivity [49]. These techniques determine may-alias [114] or must-alias [52] information. Due to the abstraction choice, the current work presents a quite reach alias analysis which is flow sensitive, context sensitive, and renders both may and must aliasing information.

The forwarding semantics for the $\mathbb{K}$ framework is the $\mathbb{K}$ specification of C described in [36]. There, the semantics is tested for validity and is also made to work for program verification, using the inherited model checking capabilities provided in Maude [33]. In this context and regarding alias analysis, we bring as witness the work in [114, 106] where C is abstracted into a context free language of equalities and the demand-driven, flow insensitive alias analysis is mapped into reachability. A similar approach, i.e., demand-driven may alias analysis for Java, is described in [113] where Java is abstracted into a context sensitive language of balanced parenthesis. Our approach in *SILK* goes along the same lines being demand-driven alias analysis, either may or must , the only difference being that the abstraction allows for interprocedural and flow sensitive alias analysis.

5.5 Conclusions

In this chapter we apply the technique presented in Chapter 3 to alias analysis for an abstract semantics of *SILK*.

In Section 5.1 we present a case study of an abstract imperative programming language with procedures and objects, *SILK*, via its $\mathbb{K}$ specification. *SILK* is of interest in the context of the $\mathbb{K}$ framework for the following reasons:

- This is a research language introduced in [91, 89] with several bisimilar semantics. In Section 5.2 we present the $\mathbb{K}$ specification of one of these semantics which is described as *abstract*. This particular semantics has the useful feature of producing a finite reachable state space. Besides this important feature which we

discuss below, the abstract semantics exhibits algorithmic details which emphasize the versatility of $\mathbb{K}$ in the area of algorithm formulation.

- The abstract semantics for *SILK* ensures a finite reachable state space. Consequently, a benefit brought by using this semantics is the decidability of the analysis/verification methods. In Section 5.3 we present the alias analysis as an instantiation of the general method for analysis/verification from Section 3.2, and exemplify this in Section 5.3.1.

- In Section 5.1 we also discuss how *SILK* can be used as an abstraction for real programming languages. We emphasize that the decidability of alias analysis is preserved in the proved environment of *SILK* and its abstract semantics. Consequently, the syntactic projection promoted by the abstract interpretation is the natural approach for relating real programming languages with *SILK* and its inherited verification methods.

Chapter 6

$\mathbb{K}$ Abstract Semantics for Shape Analysis

Motto: *"If you prick us, do we not bleed?*
if you tickle us, do we not laugh?
if you poison us, do we not die?
and if you wrong us, shall we not revenge?"
— William Shakespeare, The Merchant of Venice.

In this chapter we introduce a method, and its specification in $\mathbb{K}$, for shape analysis achieved via abstract model checking. Namely, we first present in Section 6.1 the $\mathbb{K}$ specification of an abstract language, called *Shylock*, which is focussed on reasoning about the structures maintained in the heap: objects with fields. Shylock is introduced in [90, 8] as "on paper" pushdown system specification. Consequently, we present in Section 6.2 the $\mathbb{K}$ specification of an algorithm introduced in [95] for model checking pushdown systems. As such, we give the $\mathbb{K}$ specification for a shape analysis abstract model and the $\mathbb{K}$ specification of a generic method of reasoning about it.

In [92] the authors present a framework for shape analysis that can be instantiated to define different shape analysis algorithms with various degrees of efficiency and preci-

sion. This framework enlists some fundamental concepts which need to be formalized in order to be able to carry out shape analysis. In Table 6.1 we quote this list. Further in this chapter we describe how each of the items in Table 6.1 is tackled in our approach for shape analysis.

“

- an encoding (or representation) of stores, so that we can talk precisely about store elements and the relationships among them;
- a language in which to state properties that store elements may or may not possess;
- a way to extract the properties of stores and store elements;
- a definition of the concrete semantics of the programming language, and, in particular, one that makes it possible to track how properties change as the execution of a program statement changes the store; and
- a technique for creating abstractions of stores so that abstract interpretation can be applied.

”

Table 6.1: The list of concepts used in shape analysis as enlisted in [92].

In a succinct preview of the Table's 6.1 instantiation in the current work, we claim that the first and the last two items are covered in Section 6.1 while the the other two, the second and third item as well as part of the last item, are covered in Section 6.2.

More to the point, in Section 6.1 we introduce the $\mathbb{K}$ specification for Shylock - a simple imperative programming language with pointers and recursive procedures which use local and global variables with fields to store references in the heap. The recursion allows unbounded object allocation, hence the heap size may grow unboundedly making the direct verification of such programs more challenging. We advertise Shylock as a simple but expressive enough language to represent an abstraction for imperative or object oriented programming languages like C or Java. Namely promote the abstract interpretation perspective, [26], that advocates applying analysis or verification meth-

ods to an abstract, simplified model. The advantage of the abstract model consists in the enhanced effectiveness for verification, or even existence of decidability results in certain cases. Moreover, the verification results obtained on the abstract model automatically have concrete interpretation, based on the relation between the real/concrete programming language and the abstract one. To show the relevance of Shylock, we plan to apply meta-programming techniques for generating suitable Shylock abstract models for C and Java programs.

The value of Shylock abstract semantics concerns its pushdown system characterization for which there are standardized model checking algorithms [95]. In Section 6.2 we propose embedding the $\mathbb{K}$ specification for Shylock abstract semantics into a $\mathbb{K}$ specification for Shylock collecting semantics. The Shylock collecting semantics renders an exhaustive execution of Shylock's abstract semantics, is terminating, and produces the reachable state space and/or the reachability automaton. The approach sets the basis of a *generic algebraic method* for model checking pushdown systems.

6.1 A $\mathbb{K}$ specification for Shylock

"You speak an infinite deal of nothing."

— William Shakespeare, The Merchant of Venice.

In this section we introduce the $\mathbb{K}$ specification for Shylock - a simple imperative programming language with pointers and recursive procedures which use local and global variables with fields to store references in the heap. Shylock is defined in [90, 8] as "on paper" pushdown system specification.

Shylock represents the syntactic extension of *SILK* namely, Shylock adds fields for variables. Hence, if we we consider *SIMPF* being the extension of *SIMP* with fields for objects, the syntax-based abstraction of *SIMPF* into Shylock is a mere extension of the rules in Figure 5.3. We do not insist on this matter in the current chapter.

For legibility reasons, we give the $\mathbb{K}$ specification of Shylock syntax in Figure 6.1. However, even if Shylock maintains the original program instructions, the abstract semantics has to be redesigned due to the change in the state structure. Consequently, the abstract semantics of Shylock presents actual novelty in comparison with *SILK*. We use Shylock as a witness to the fact that the semantics does not have to maintain the incremental design of the syntax.

Pgm ::= `gvars:` *Ids* `lvars:` *Ids* `flds:` *Ids* { *Procs* }

Ids ::= *List*{*Id*,","}

Procs ::= *ProcId* `::` *Ks* | *Procs* *Procs*

ProcId ::= *Id* | `while(` *Int* `)`

VExp ::= *Id* | `ref(` *Id* `)`

FExp ::= *Id*

VFExp ::= *VExp* | *VExp* `.` *FExp*

BExp ::= *VFExp* `=` *VFExp* | *VFExp* `/=` *VFExp*

Ks ::= *VFExp* `:=` *VFExp* | *VExp* `:=` `new` | *VExp* `:=` ⊥
| *Ks* `;` *Ks* | `[` *BExp* `]` *Ks* | *Ks* `+` *Ks*
| *ProcId* | `join(` *Int* `)`

IntBot ::= *Int* | ⊥

Figure 6.1: The Shylock syntax

There are two semantics for Shylock described in [90]: the *concrete* semantics and the *abstract* semantics. The abstraction process, i.e., deriving the abstract semantics from the concrete semantics, is made via changing the semantics for some syntactic elements. Namely, once identified the syntactic sources of infiniteness in the concrete semantics, the abstract semantics modifies the rules for these syntactic sources such

that the infiniteness source is eliminated. These two semantics are proved bisimilar. We present in the current section a "modular" $\mathbb{K}$ specification where we reuse the concrete semantics for the syntactic parts where the rules in concrete do not change for abstract .

6.1.1 Shylock configuration

Here we present Shylock's $\mathbb{K}$-configuration which is faithful to its "on paper" definition from [90, 8]. Further, we discuss Shylock's $\mathbb{K}$-configuration as an abstraction for shape analysis of $\mathbb{K}$ specifications.

The $\mathbb{K}$ configuration used for the abstract semantics of Shylock is the following:

$$\langle K\rangle_{\mathsf{k}}\ \langle\langle Map\rangle_{\mathsf{var}}\ \langle\langle Map\rangle_{\mathsf{fld}^\star}\rangle_{\mathsf{h}}\rangle_{\mathsf{heap}}\ \langle\langle Set\rangle_{\mathsf{G}}\ \langle Set\rangle_{\mathsf{L}}\ \langle Set\rangle_{\mathsf{F}}\ \langle Map\rangle_{\mathsf{prcs}}\rangle_{\mathsf{pgm}}\ \langle Bag\rangle_{\mathsf{abs}}$$

The k-cell maintains the continuation, the heap-cell contains the current heap state, while the pgm-cell is designated as a program container. The abs-cell is introduced for the modularity of the implementation and it is formed as $\langle\langle Int\rangle_{\mathsf{id}}\langle List_{\rightsquigarrow}\{K\}\rangle_{\mathsf{kAbs}}\rangle_{\mathsf{abs}}$. In this way, we maintain in the cell k only the "pure" syntactic elements of the language (used by both concrete and abstract semantics), and we move to the kAbs-cell the additional computational effort used by the abstract semantics for object creation, as well as for procedure call and return.

Even if the $\mathbb{K}$-configuration above has the necessary items for representing Shylock pushdown system specification, we need to clarify the relation with $\mathscr{P} = (C, \Sigma, \rightarrow)$ defined in [90, 8]. Firstly, we present how $Conf(\mathscr{P})$ relates to the $\mathbb{K}$-configuration. This relation, represented in the $\mathbb{K}$ notation as $A ::= B_1 \mid B_2$, is introduced via subsorting, and it means that both B_1, B_2 are a subsort of A. More to the point, $K ::= V \mid F \mid \Sigma \mid \mathbb{N}_\perp$, where $V, F, \mathbb{N}_\perp$ are as in [90, 8]. Thus, not only Σ can be used in the k-cell, but also the pair $C = ((V \mapsto \mathbb{N}_\perp), h : F \mapsto (\mathbb{N}_\perp \mapsto \mathbb{N}_\perp))$ becomes elementwise subsorted to *Map*. We represent C in the two cells var and h. We introduce the heap-cell, which contains

var and h, only for the control state localization, namely $\langle C \rangle_{\mathsf{heap}}$.

The $\mathbb{K}$ framework already offers the semantic specification of languages such as C [36] and KOOL [46]. However, it is notoriously difficult and not recommended to alter a concrete semantics for such a purpose as abstraction. Instead, the abstract interpretation standardized approach advocates for the following separation of concerns: the concrete and the abstract models are defined in isolation, the analysis and verification methods are developed in the abstract and are projected in the concrete using the a priori proved relation between concrete and abstract. To convey the same perspective in $\mathbb{K}$, we can consider the C specification as the concrete model, the Shylock specification as the abstract model, and we are left to prove the simulation relation between the two.

In order to define an abstraction for shape analysis, we first restrict the state space to the heap only. As such, the heap contains objects represented as *object-variables* and fields for object represented as *field-variables*. Because we target an interprocedural analysis, the object-variables are either *global* or *local*, and we call them directly *global variables* and *local variables*. Obviously, the local variables are seen as local to the program procedures. Also, the field variables are called directly fields.

Observation 4. *The abstract lattice is the flat lattice formed by all graph-isomorphic equivalence classes over the* heap *cell. Note that these equivalence classes are infinite.*

Furthermore, the Galois connection is defined as follows. The abstraction # *maps a concrete state in SIMPF into its afferent graph-isomorphic equivalence class, while the concretization* $ *maps an equivalence class into the set of all concrete states with a graph isomorphic structure. Again, we do not insist on further formalization because we did not explicitly present the concrete configuration and the concrete state.*

6.1.2 Shylock rules

In the followings we present the $\mathbb{K}$ rules which implement both concrete and abstract semantics of Shylock. We implement semantics-reusability as a mutual exclusion protocol. Namely, we label each semantics with its own id cell and we attribute each semantic rule to a particular semantics. Further, we use the top of the kAbs cell as a handler for the execution priority given to each rule. This mechanism is described in more detail this section.

The definition of the abstract operators in abstract interpretation is, in our settings, tantamount to giving semantics for Shylock. The abstract semantics from [90, 8] introduces a novel mechanism for managing the object identities in the heap. This mechanism resides in a combination of *memory reuse* upon generation of fresh object identities and a *renaming scheme* to resolve possible resulting clashes. The memory reuse allows the allocation of any object identity which is unreachable in the context of the current procedure but not necessarily in the context of pending local environments on the stack. The renaming scheme resolves possible resulting name clashes upon changing the context, i.e., upon procedure call/return. The proposed mechanism significantly simplifies, comparing to the standard approach in [15], the heap transformation operations performed on procedure calls and returns.

The transition relation in the pushdown system is encoded in $\mathbb{K}$ by semantic rules which basically express changes to the configuration triggered by the execution of an atomic piece of syntax. An example of such encoding is the following rule for assignment:

$$\text{RULE}\quad \langle \underbrace{x.f := y}_{\bullet} \cdots \rangle_{\mathsf{k}}\ \langle \cdots\ v(x) \mapsto \underbrace{\_}_{v(y)} \cdots \rangle_{\mathsf{fld}(f)}\ \langle v \rangle_{\mathsf{var}}\ \langle\ \underbrace{0}_{\bullet} \cdots \rangle_{\mathsf{kAbs}}$$

when $v(x) \neq_{Bool} \bot$

[asgnFieldLeft transition]

which reads as: if the current computation (the first element in the cell k) is the assignment $x.f := y$, then the map associated to the field f (i.e., the content of the cell $\mathsf{fld}(f)$) is modified by replacing whatever object was pointed by $v(x)$ with the object associated to the variable y (i.e., $v(y)$), where v is the content of the cell var (i.e. the current variable assignment), only when the object associated to x is not $\perp$. Note that this rule is conditional, the reason for introducing the condition is the preservation of the invariant $h(f)(\perp) = \perp$, for any field f. Next, we explain the meaning of the id and kAbs cells.

In the current section we describe an interleaved execution of the two semantics presented in [90, 8] which, being bisimilar, we consider to be at the same semantic level wrt abstraction. The interleaving of the two semantics for Shylock is designed using a simplistic message passing mechanism. Namely, the cells k and heap are the shared resource, while we synchronize the id and kAbs cells to say which abstraction is currently locking the resource. For example, the assignment rule has the same behavior in both initial, $\langle 0 \rangle_{\mathsf{id}}$, and improved, $\langle 1 \rangle_{\mathsf{id}}$, semantics. Hence, the rule is attributed to the initial semantics while the improved semantics passes the computation charge to the original semantics. This is specified using the following structural rules:

$$\text{RULE} \quad \langle x.f := y \ \cdots \rangle_{\mathsf{k}} \ \langle 1 \rangle_{\mathsf{id}} \ \langle \ \underset{0 \rightsquigarrow 1}{\bullet} \ \cdots \rangle_{\mathsf{kAbs}} \quad [\text{structural}]$$

$$\text{RULE} \quad \langle \_ \ \cdots \rangle_{\mathsf{k}} \ \langle 0 \rangle_{\mathsf{id}} \ \langle \ \underset{0}{\bullet} \ \cdots \rangle_{\mathsf{kAbs}} \quad [\text{structural}]$$

Next we present how the message passing mechanism works for the syntactic elements for which the two semantics have different behaviors, namely object creation, procedure call and return. Let us consider for example the case of object creation. The common behavior is the following: if the current computation (the first element in the cell k) is "x:=`new`", then whatever object was pointed by x in the var-cell is replaced

with the new object "*oNew*" obtained from the cell $\langle \_ \rangle_{\mathsf{kAbs}}$. Also, the field assignment map h, i.e., the content of h-cell, is updated by the addition of a new map item "$oNew \mapsto \bot$". We emphasize that in the implementation the h-cell, the field assignment, maintains for each field a partial map with the currently already created objects. Also, note that one of the differences between the two semantics is in the calculation of the *oNew* value. Hence, the calculation of this value is handled by the kAbs-cell. The object creation rules in the two semantics are:

$$\text{RULE}\quad \langle x := \texttt{new}\ \cdots \rangle_{\mathsf{k}}\ \langle\ \underset{\texttt{oNew}(n) \rightsquigarrow \texttt{nextNew}(n +_{Int} 1)}{0 \rightsquigarrow \texttt{nextNew}(n)}\ \cdots \rangle_{\mathsf{kAbs}} \quad [\text{structural}]$$

$$\text{RULE}\quad \langle x := \texttt{new}\ \cdots \rangle_{\mathsf{k}}\ \langle H \rangle_{\mathsf{heap}}\ \langle 1 \rangle_{\mathsf{id}}\ \langle\ \underset{\texttt{min}(\mathscr{R}(H)^c)}{\bullet}\ \rangle_{\mathsf{kAbs}} \quad [\text{structural}]$$

$$\text{RULE}\quad \langle \underset{\bullet}{x := \texttt{new}}\ \cdots \rangle_{\mathsf{k}}\ \langle \cdots\ x \mapsto \underset{n}{\_}\ \cdots \rangle_{\mathsf{var}}\ \langle\ \underset{\texttt{update}\ H_h\ \texttt{with}\ n \mapsto \bot}{H_h}\ \rangle_{\mathsf{h}}$$

$$\langle \underset{\bullet}{\texttt{oNew}(n)}\ \cdots \rangle_{\mathsf{kAbs}}$$

[objCreation transition]

where $\texttt{min}(\mathscr{R}(H)^c)$ is implemented equationally such that when it finds n, the first integer not in $\mathscr{R}(H)$, $\texttt{min}(\mathscr{R}(H)^c)$ becomes $\texttt{oNew}(n)$, where oNew is defined as an operator which wraps a reference into K any element, i.e., $K ::=$ "oNew" "(" $\mathbb{N}_\bot$ ")". Note that all the computation in the kAbs-cell is implemented equationally, by means of structural rules. In fact, we find a very good practice to move any underwired computation in the kAbs-cell. As such, the update *Bag* with *MapItem* should be computed in the

kAbs-cell, hence the *objCreation* rule is replaced by the following two rules:

$$\text{RULE}\quad \langle x := \texttt{new}\ \cdots\rangle_{\mathsf{k}}\ \langle H_h\rangle_{\mathsf{h}}\ \langle \texttt{oNew}(n) \rightsquigarrow \underset{\texttt{update}\ H_h\ \texttt{with}\ n\mapsto\bot}{\bullet}\ \cdots\rangle_{\mathsf{kAbs}}$$

[objCreationPipe1 structural]

$$\text{RULE}\quad \langle \underset{\bullet}{x := \texttt{new}}\ \cdots\rangle_{\mathsf{k}}\ \langle \cdots\ x \mapsto \underset{n}{\_}\ \cdots\rangle_{\mathsf{var}}\ \langle\ \underset{H'_h}{H_h}\ \rangle_{\mathsf{h}}$$

$$\underset{\bullet}{\langle \texttt{oNew}(n) \rightsquigarrow \texttt{updated}(H'_h)\ \cdots\rangle_{\mathsf{kAbs}}}$$

[objCreationPipe2 transition]

The renaming scheme defined for resolving name clashes in Shylock is based on the concept of *cut points* as introduced in [78]. Cut points are objects in the heap that are referred to from both local and global variables, and as such, are subject to modifications during a procedure call. Recording cut points in extra logical variables allows for a sound return in the calling procedure, enabling a precise abstract execution w.r.t. object identities. Hence, under the assumption of a bounded visible heap, the programs can be represented by finitary structures, namely *finite pushdown systems*. For more details on the implementation where the cut points are used, we present Figure 6.2 where we list the rules for procedure call and return.

The quality of solely Shylock's $\mathbb{K}$ specification lies in the rules for fresh object creation, which implement a memory reuse mechanism, and procedure call/return, which implement a renaming scheme. Each element in this entire mechanism is implemented equationally, i.e., by means of structural $\mathbb{K}$ rules which have equational interpretation when compiled in Maude. Hence, if we interpret Shylock as an abstract model for C or Java, the $\mathbb{K}$ specification for Shylock's abstract semantics renders an equational ab-

straction. As such, Shylock is yet another witness to the versatility of the equational abstraction methodology [68, 66].

Theorem 7. *The transition systems generated by the* $\mathbb{K}$ *specification containing* $\langle 0 \rangle_{\mathsf{id}}$, *and* $\langle 1 \rangle_{\mathsf{id}}$, *respectively, in the* abs *cell are respectively bisimilar, on the reachable state space.*

Proof. The specification containing $\langle 0 \rangle_{\mathsf{id}}$ is the faithful $\mathbb{K}$ specification of the concrete semantics given in [90]. Also, the $\mathbb{K}$ specification of the memory reuse and the renaming scheme (i.e., object creation and procedure call/return) are faithfully specified according to the abstract semantics defined in [90]. As mentioned before, the concrete and abstract semantics in [90] are proved bisimilar. Hence, the $\mathbb{K}$ specifications containing in the abs cell $\langle 0 \rangle_{\mathsf{id}}$ and $\langle 1 \rangle_{\mathsf{id}}$, respectively, are two bisimilar $\mathbb{K}$ specifications. □

6.1.3 Shylock abstract computation example

Here we give an example of a computation in Shylock's abstract semantics $\langle 1 \rangle_{\mathsf{id}}$ for a program `pgmTest`. This example emphasizes the functionality of the abstract semantics from [90, 8] in the context of various object creation cases and procedure call and return with cut points.

```
macro pgmTest =
 gvars: g1, g2, g3  lvars: l1, l2, l3  flds: f1, f2
 { main ::
     // testing object creation and reachable set
     g1:=new;  g2:=new;  g2.f1:= g1;  g1.f1:=g2;
     //setting the scene for the procedure call for globals
     g3:=g1.f1; g3:=new; g3.f1:=g1; g3:=new;
     g2:=new; g2.f1:=g2; g2.f2:=g1;
     // setting the scene for procedure call and cut-point set
```

RULE $\langle p \;\cdots\rangle_{\mathsf{k}} \langle H \rangle_{\mathsf{heap}} \; \langle L \rangle_{\mathsf{L}} \; \langle \cdots \;\; p \mapsto B \;\; \cdots \rangle_{\mathsf{prcs}} \langle \underbrace{0}_{\texttt{update}\; H \;\texttt{with}\; L \mapsto \bot} \cdots \rangle_{\mathsf{kAbs}}$

[structural]

RULE $\langle p \;\cdots\rangle_{\mathsf{k}} \langle H \rangle_{\mathsf{heap}} \; \langle L \rangle_{\mathsf{L}} \; \langle G \rangle_{\mathsf{G}} \; \langle F \rangle_{\mathsf{F}} \; \langle \cdots \;\; p \mapsto B \;\; \cdots \rangle_{\mathsf{prcs}}$
$\langle 1 \rangle_{\mathsf{id}} \; \langle \underbrace{\bullet}_{\texttt{processingCP}(\texttt{setCutPoints}(H,L,G,F))} \cdots \rangle_{\mathsf{kAbs}}$

[structural]

RULE $\langle H \rangle_{\mathsf{heap}} \; \langle 1 \rangle_{\mathsf{id}} \; \langle \underbrace{\texttt{processedCP}(CP)}_{\texttt{updateCP}\; H \;\texttt{with}\; CP} \cdots \rangle_{\mathsf{kAbs}}$ [structural]

RULE $\langle L \rangle_{\mathsf{L}} \; \langle 1 \rangle_{\mathsf{id}} \; \langle \underbrace{\texttt{updatedCP}(H)}_{\texttt{update}\; H \;\texttt{with}\; L \mapsto \bot} \cdots \rangle_{\mathsf{kAbs}}$ [structural]

RULE $\langle \underbrace{p}_{B \curvearrowright \texttt{restore}(H)} \cdots \rangle_{\mathsf{k}} \langle \underbrace{H}_{H'} \rangle_{\mathsf{heap}} \; \langle \cdots \;\; p \mapsto B \;\; \cdots \rangle_{\mathsf{prcs}}$
$\langle \underbrace{\texttt{processedCall}(H')}_{\bullet} \cdots \rangle_{\mathsf{kAbs}}$

[transition]

RULE $\langle \texttt{restore}(H') \;\cdots\rangle_{\mathsf{k}} \langle H \rangle_{\mathsf{heap}} \; \langle \underbrace{0}_{\texttt{update}\; H \;\texttt{with}\; H'} \cdots \rangle_{\mathsf{kAbs}}$

[structural]

RULE $\langle \texttt{restore}(H') \;\cdots\rangle_{\mathsf{k}} \langle H \rangle_{\mathsf{heap}} \; \langle L \rangle_{\mathsf{L}} \; \langle G \rangle_{\mathsf{G}} \; \langle F \rangle_{\mathsf{F}}$
$\langle 1 \rangle_{\mathsf{id}} \; \langle \underbrace{\bullet}_{\texttt{processingRet}(H,H',L,G,F)} \cdots \rangle_{\mathsf{kAbs}}$

[structural]

RULE $\langle \underbrace{\texttt{restore}(\_)}_{\bullet} \cdots \rangle_{\mathsf{k}} \langle \underbrace{H}_{H'} \rangle_{\mathsf{heap}} \langle \underbrace{\texttt{processedRet}(H')}_{\bullet} \cdots \rangle_{\mathsf{kAbs}}$

[transition]

Figure 6.2: $\mathbb{K}$-rules for the procedure's call and return in Shylock

```
    l2:=new; l2.f1:=g1; l1:=g2.f2; g3.f1:=g2; l2.f2:=g3;
    p1
  p1 ::
    // setting the scene for conflicting set
    g1:=new; g1.f1:=g2; l1:=g1.f1; g1.f2:=l1
}
```

The following configurations are obtained after the execution of each statement in `pgmTest`. Please follow the STEP number to observe their ordering in computation.

`Init` // STEP 1 (program load)

$\Rrightarrow$ `main ::` // STEP 2 (main procedure call)

$\langle$ `g1:=new` $\curvearrowright$ `g2:=new` $\curvearrowright$ `g2.f1:=g1` $\curvearrowright$ `g1.f1:=g2` $\curvearrowright$ `g3:=g1.f1` $\curvearrowright$ `g3:=new`
$\curvearrowright$ `g3.f1:=g1` $\curvearrowright$ `g3:=new` $\curvearrowright$ `g2:=new` $\curvearrowright$ `g2.f1:=g2` $\curvearrowright$ `g2.f2:=g1`
$\curvearrowright$ `l2:=new` $\curvearrowright$ `l2.f1:=g1` $\curvearrowright$ `l1:=g2.f2` $\curvearrowright$ `g3.f1:=g2` $\curvearrowright$ `l2.f2:=g3` $\curvearrowright$ `p1`
$\curvearrowright$ `restore`$(\langle\langle$ `f1` $\mapsto$ `wkbag`$(\langle\bot\mapsto\bot\rangle_{\mathsf{hf}})\rangle_{\mathsf{fld}}\langle$ `f2` $\mapsto$ `wkbag`$(\langle\bot\mapsto\bot\rangle_{\mathsf{hf}})\rangle_{\mathsf{fld}}\rangle_{\mathsf{h}}$
$\langle$ `g1`$\mapsto\bot$ `g2`$\mapsto\bot$ `g3`$\mapsto\bot$ `l1`$\mapsto\bot$ `l2`$\mapsto\bot$ `l3`$\mapsto\bot\rangle_{\mathsf{var}})$
$\rangle_{\mathsf{k}}$

$\langle\langle\langle$ `f1` $\mapsto$ `wkbag`$(\langle\bot\mapsto\bot\rangle_{\mathsf{hf}})\rangle_{\mathsf{fld}}$
$\langle$ `f2` $\mapsto$ `wkbag`$(\langle\bot\mapsto\bot\rangle_{\mathsf{hf}})\rangle_{\mathsf{fld}}$
$\rangle_{\mathsf{h}}$
$\langle$ `g1`$\mapsto\bot$ `g2`$\mapsto\bot$ `g3`$\mapsto\bot$ `l1`$\mapsto\bot$ `l2`$\mapsto\bot$ `l3`$\mapsto\bot\rangle_{\mathsf{var}}$
$\rangle_{\mathsf{heap}}$

$\Rrightarrow$ `g1:=new`// STEP 3 (object creation)

$\langle \langle \langle \texttt{f1} \mapsto \texttt{wkbag}(\langle 0 \mapsto \bot\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\langle \texttt{f2} \mapsto \texttt{wkbag}(\langle 0 \mapsto \bot\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\rangle_{\mathsf{h}}$

$\langle \texttt{g1} \mapsto 0\ \texttt{g2} \mapsto \bot\ \texttt{g3} \mapsto \bot\ \texttt{l1} \mapsto \bot\ \texttt{l2} \mapsto \bot\ \texttt{l3} \mapsto \bot \rangle_{\mathsf{var}}$

$\rangle_{\mathsf{heap}}$

$\Rrightarrow$ `g2:=new` // STEP 4 (object creation)

$\langle \langle \langle \texttt{f1} \mapsto \texttt{wkbag}(\langle 1 \mapsto \bot\ 0 \mapsto \bot\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\langle \texttt{f2} \mapsto \texttt{wkbag}(\langle 1 \mapsto \bot\ 0 \mapsto \bot\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\rangle_{\mathsf{h}}$

$\langle \texttt{g1} \mapsto 0\ \texttt{g2} \mapsto 1\ \texttt{g3} \mapsto \bot\ \texttt{l1} \mapsto \bot\ \texttt{l2} \mapsto \bot\ \texttt{l3} \mapsto \bot \rangle_{\mathsf{var}}$

$\rangle_{\mathsf{heap}}$

$\Rrightarrow$ `g2.f1:=g1` // STEP 5 (assignment field left)

$\langle \langle \langle \texttt{f1} \mapsto \texttt{wkbag}(\langle 1 \mapsto 0\ 0 \mapsto \bot\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\langle \texttt{f2} \mapsto \texttt{wkbag}(\langle 1 \mapsto \bot\ 0 \mapsto \bot\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\rangle_{\mathsf{h}}$

$\langle \texttt{g1} \mapsto 0\ \texttt{g2} \mapsto 1\ \texttt{g3} \mapsto \bot\ \texttt{l1} \mapsto \bot\ \texttt{l2} \mapsto \bot\ \texttt{l3} \mapsto \bot \rangle_{\mathsf{var}}$

$\rangle_{\mathsf{heap}}$

$\Rrightarrow$ `g1.f1:=g2` // STEP 6 (assignment field left)

$\langle \langle \langle \texttt{f1} \mapsto \texttt{wkbag}(\langle 1 \mapsto 0\ 0 \mapsto 1\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\langle \texttt{f2} \mapsto \texttt{wkbag}(\langle 1 \mapsto \bot\ 0 \mapsto \bot\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\rangle_{\mathsf{h}}$

$\langle \texttt{g1} \mapsto 0\ \texttt{g2} \mapsto 1\ \texttt{g3} \mapsto \bot\ \texttt{l1} \mapsto \bot\ \texttt{l2} \mapsto \bot\ \texttt{l3} \mapsto \bot \rangle_{\mathsf{var}}$

$\rangle_{\mathsf{heap}}$

$\Rrightarrow$ `g3:=g1.f1` // STEP 7 (assignment field right)

$\langle \langle \langle \texttt{f1} \mapsto \texttt{wkbag}(\langle 1 \mapsto 0\ 0 \mapsto 1\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\langle \texttt{f2} \mapsto \texttt{wkbag}(\langle 1 \mapsto \bot\ 0 \mapsto \bot\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\rangle_{\mathsf{h}}$

$\langle \texttt{g1} \mapsto 0\ \texttt{g2} \mapsto 1\ \texttt{g3} \mapsto 1\ \texttt{l1} \mapsto \bot\ \texttt{l2} \mapsto \bot\ \texttt{l3} \mapsto \bot \rangle_{\mathsf{var}}$

$\rangle_{\mathsf{heap}}$

$\Rrightarrow$ `g3:=new` // STEP 8 (object creation)

$\langle \langle \langle \texttt{f1} \mapsto \texttt{wkbag}(\langle 2 \mapsto \bot\ 1 \mapsto 0\ 0 \mapsto 1\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\langle \texttt{f2} \mapsto \texttt{wkbag}(\langle 2 \mapsto \bot\ 1 \mapsto \bot\ 0 \mapsto \bot\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\rangle_{\mathsf{h}}$

$\langle \texttt{g1} \mapsto 0\ \texttt{g2} \mapsto 1\ \texttt{g3} \mapsto 2\ \texttt{l1} \mapsto \bot\ \texttt{l2} \mapsto \bot\ \texttt{l3} \mapsto \bot \rangle_{\mathsf{var}}$

$\rangle_{\mathsf{heap}}$

$\Rrightarrow$ `g3.f1:=g1` // STEP 9 (assignment field left)

$\langle \langle \langle \texttt{f1} \mapsto \texttt{wkbag}(\langle 2 \mapsto 0\ 1 \mapsto 0\ 0 \mapsto 1\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\langle \texttt{f2} \mapsto \texttt{wkbag}(\langle 2 \mapsto \bot\ 1 \mapsto \bot\ 0 \mapsto \bot\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\rangle_{\mathsf{h}}$

$\langle \texttt{g1} \mapsto 0\ \texttt{g2} \mapsto 1\ \texttt{g3} \mapsto 2\ \texttt{l1} \mapsto \bot\ \texttt{l2} \mapsto \bot\ \texttt{l3} \mapsto \bot \rangle_{\mathsf{var}}$

$\rangle_{\mathsf{heap}}$

$\Rrightarrow$ `g3:=new` // STEP 10 (object creation)

$\langle \langle \langle \texttt{f1} \mapsto \texttt{wkbag}(\langle 3 \mapsto \bot\ 2 \mapsto 0\ 1 \mapsto 0\ 0 \mapsto 1\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\langle \texttt{f2} \mapsto \texttt{wkbag}(\langle 3 \mapsto \bot\ 2 \mapsto \bot\ 1 \mapsto \bot\ 0 \mapsto \bot\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\rangle_{\mathsf{h}}$

$\langle \texttt{g1} \mapsto 0\ \texttt{g2} \mapsto 1\ \texttt{g3} \mapsto 3\ \texttt{l1} \mapsto \bot\ \texttt{l2} \mapsto \bot\ \texttt{l3} \mapsto \bot \rangle_{\mathsf{var}}$

$\rangle_{\mathsf{heap}}$

$\Rrightarrow$ `g2:=new` // STEP 11 (object creation)

$\langle \langle \langle \texttt{f1} \mapsto \texttt{wkbag}(\langle 3 \mapsto \bot\ 2 \mapsto \bot\ 1 \mapsto 0\ 0 \mapsto 1\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\langle \texttt{f2} \mapsto \texttt{wkbag}(\langle 3 \mapsto \bot\ 2 \mapsto \bot\ 1 \mapsto \bot\ 0 \mapsto \bot\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\rangle_{\mathsf{h}}$

$\langle \texttt{g1} \mapsto 0\ \texttt{g2} \mapsto 2\ \texttt{g3} \mapsto 3\ \texttt{l1} \mapsto \bot\ \texttt{l2} \mapsto \bot\ \texttt{l3} \mapsto \bot \rangle_{\mathsf{var}}$

$\rangle_{\mathsf{heap}}$

$\Rrightarrow$ `g2.f1:=g2` // STEP 12 (assignment field left)

$\langle \langle \langle \texttt{f1} \mapsto \texttt{wkbag}(\langle 3 \mapsto \bot\ 2 \mapsto 2\ 1 \mapsto 0\ 0 \mapsto 1\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\langle \texttt{f2} \mapsto \texttt{wkbag}(\langle 3 \mapsto \bot\ 2 \mapsto \bot\ 1 \mapsto \bot\ 0 \mapsto \bot\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\rangle_{\mathsf{h}}$

$\langle \texttt{g1} \mapsto 0\ \texttt{g2} \mapsto 2\ \texttt{g3} \mapsto 3\ \texttt{l1} \mapsto \bot\ \texttt{l2} \mapsto \bot\ \texttt{l3} \mapsto \bot \rangle_{\mathsf{var}}$

$\rangle_{\mathsf{heap}}$

$\Rrightarrow$ `g2.f2:=g1` // STEP 13 (assignment field left)

$\langle \langle \langle \texttt{f1} \mapsto \texttt{wkbag}(\langle 3 \mapsto \bot\ 2 \mapsto 2\ 1 \mapsto 0\ 0 \mapsto 1\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\langle \texttt{f2} \mapsto \texttt{wkbag}(\langle 3 \mapsto \bot\ 2 \mapsto 0\ 1 \mapsto \bot\ 0 \mapsto \bot\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\rangle_{\mathsf{h}}$

$\langle \texttt{g1} \mapsto 0\ \texttt{g2} \mapsto 2\ \texttt{g3} \mapsto 3\ \texttt{l1} \mapsto \bot\ \texttt{l2} \mapsto \bot\ \texttt{l3} \mapsto \bot \rangle_{\mathsf{var}}$

$\rangle_{\mathsf{heap}}$

$\Rrightarrow$ `l2:=new` // STEP 14 (object creation)

$\langle \langle \langle \texttt{f1} \mapsto \texttt{wkbag}(\langle 4 \mapsto \bot\ 3 \mapsto \bot\ 2 \mapsto 2\ 1 \mapsto 0\ 0 \mapsto 1\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\langle \texttt{f2} \mapsto \texttt{wkbag}(\langle 4 \mapsto \bot\ 3 \mapsto \bot\ 2 \mapsto 0\ 1 \mapsto \bot\ 0 \mapsto \bot\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\rangle_{\mathsf{h}}$

$\langle \texttt{g1} \mapsto 0\ \texttt{g2} \mapsto 2\ \texttt{g3} \mapsto 3\ \texttt{l1} \mapsto \bot\ \texttt{l2} \mapsto 4\ \texttt{l3} \mapsto \bot \rangle_{\mathsf{var}}$

$\rangle_{\mathsf{heap}}$

$\Rrightarrow$ `l2.f1:=g1` // STEP 15 (assignment field left)

$\langle \langle \langle \texttt{f1} \mapsto \texttt{wkbag}(\langle 4 \mapsto 0\ 3 \mapsto \bot\ 2 \mapsto 2\ 1 \mapsto 0\ 0 \mapsto 1\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\langle \texttt{f2} \mapsto \texttt{wkbag}(\langle 4 \mapsto \bot\ 3 \mapsto \bot\ 2 \mapsto 0\ 1 \mapsto \bot\ 0 \mapsto \bot\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\rangle_{\mathsf{h}}$

$\langle \texttt{g1} \mapsto 0\ \texttt{g2} \mapsto 2\ \texttt{g3} \mapsto 3\ \texttt{l1} \mapsto \bot\ \texttt{l2} \mapsto 4\ \texttt{l3} \mapsto \bot \rangle_{\mathsf{var}}$

$\rangle_{\mathsf{heap}}$

$\Rrightarrow$ `l1:=g2.f2` // STEP 16 (assignment field right)

$\langle \langle \langle \texttt{f1} \mapsto \texttt{wkbag}(\langle 4 \mapsto 0\ 3 \mapsto \bot\ 2 \mapsto 2\ 1 \mapsto 0\ 0 \mapsto 1\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\langle \texttt{f2} \mapsto \texttt{wkbag}(\langle 4 \mapsto \bot\ 3 \mapsto \bot\ 2 \mapsto 0\ 1 \mapsto \bot\ 0 \mapsto \bot\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\rangle_{\mathsf{h}}$

$\langle \texttt{g1} \mapsto 0\ \texttt{g2} \mapsto 2\ \texttt{g3} \mapsto 3\ \texttt{l1} \mapsto 0\ \texttt{l2} \mapsto 4\ \texttt{l3} \mapsto \bot \rangle_{\mathsf{var}}$

$\rangle_{\mathsf{heap}}$

$\Rrightarrow$ `g3.f1:=g2` // STEP 17 (assignment field left)

$\langle \langle \langle \texttt{f1} \mapsto \texttt{wkbag}(\langle 4 \mapsto 0\ 3 \mapsto 2\ 2 \mapsto 2\ 1 \mapsto 0\ 0 \mapsto 1\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\langle \texttt{f2} \mapsto \texttt{wkbag}(\langle 4 \mapsto \bot\ 3 \mapsto \bot\ 2 \mapsto 0\ 1 \mapsto \bot\ 0 \mapsto \bot\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\rangle_{\mathsf{h}}$

$\langle \texttt{g1} \mapsto 0\ \texttt{g2} \mapsto 2\ \texttt{g3} \mapsto 3\ \texttt{l1} \mapsto 0\ \texttt{l2} \mapsto 4\ \texttt{l3} \mapsto \bot \rangle_{\mathsf{var}}$

$\rangle_{\mathsf{heap}}$

$\Rrightarrow$ `l2.f2:=g3` // STEP 18 (assignment field left)

$\langle \langle \langle \texttt{f1} \mapsto \texttt{wkbag}(\langle 4 \mapsto 0\ 3 \mapsto 2\ 2 \mapsto 2\ 1 \mapsto 0\ 0 \mapsto 1\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\langle \texttt{f2} \mapsto \texttt{wkbag}(\langle 4 \mapsto 3\ 3 \mapsto \bot\ 2 \mapsto 0\ 1 \mapsto \bot\ 0 \mapsto \bot\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\rangle_{\mathsf{h}}$

$\langle \texttt{g1} \mapsto 0\ \texttt{g2} \mapsto 2\ \texttt{g3} \mapsto 3\ \texttt{l1} \mapsto 0\ \texttt{l2} \mapsto 4\ \texttt{l3} \mapsto \bot \rangle_{\mathsf{var}}$

$\rangle_{\mathsf{heap}}$

$\Rrightarrow$ `p1` // STEP 24 (p1 procedure return)

$\langle \langle \langle \texttt{f1} \mapsto \texttt{wkbag}(\langle 5{\mapsto}2\ 4{\mapsto}0\ 3{\mapsto}2\ 2{\mapsto}2\ 1{\mapsto}0\ 0{\mapsto}1\ \bot{\mapsto}\bot \rangle_{\mathsf{hf}})\rangle_{\mathsf{fld}}$

$\langle \texttt{f2} \mapsto \texttt{wkbag}(\langle 5{\mapsto}2\ 4{\mapsto}3\ 3{\mapsto}\bot\ 2{\mapsto}0\ 1{\mapsto}\bot\ 0{\mapsto}\bot\ \bot{\mapsto}\bot \rangle_{\mathsf{hf}})\rangle_{\mathsf{fld}}$

$\rangle_{\mathsf{h}}$

$\langle \texttt{g1}{\mapsto}5\ \texttt{g2}{\mapsto}2\ \texttt{g3}{\mapsto}3\ \texttt{l1}{\mapsto}0\ \texttt{l2}{\mapsto}4\ \texttt{l3}{\mapsto}\bot \rangle_{\mathsf{var}}$

$\rangle_{\mathsf{heap}}$

$\Rrightarrow$// STEP 25 (main procedure return)

$\langle \langle \langle \texttt{f1} \mapsto \texttt{wkbag}(\langle 5{\mapsto}2\ 4{\mapsto}0\ 3{\mapsto}2\ 2{\mapsto}2\ 1{\mapsto}0\ 0{\mapsto}1\ \bot{\mapsto}\bot \rangle_{\mathsf{hf}})\rangle_{\mathsf{fld}}$

$\langle \texttt{f2} \mapsto \texttt{wkbag}(\langle 5{\mapsto}2\ 4{\mapsto}3\ 3{\mapsto}\bot\ 2{\mapsto}0\ 1{\mapsto}\bot\ 0{\mapsto}\bot\ \bot{\mapsto}\bot \rangle_{\mathsf{hf}})\rangle_{\mathsf{fld}}$

$\rangle_{\mathsf{h}}$

$\langle \texttt{g1}{\mapsto}5\ \texttt{g2}{\mapsto}2\ \texttt{g3}{\mapsto}3\ \texttt{l1}{\mapsto}\bot\ \texttt{l2}{\mapsto}\bot\ \texttt{l3}{\mapsto}\bot \rangle_{\mathsf{var}}$

$\rangle_{\mathsf{heap}}$

$\Rrightarrow$ p1 ::// STEP 19 (procedure call)

$\langle \langle \langle \texttt{CutPoints} \mapsto \texttt{wkbag}(\langle \texttt{cp}_1{\mapsto}3\ \texttt{cp}_2{\mapsto}0 \rangle_{\mathsf{hf}})\rangle_{\mathsf{fld}}$

$\langle \texttt{f1} \mapsto \texttt{wkbag}(\langle 4{\mapsto}0\ 3{\mapsto}2\ 2{\mapsto}2\ 1{\mapsto}0\ 0{\mapsto}1\ \bot{\mapsto}\bot \rangle_{\mathsf{hf}})\rangle_{\mathsf{fld}}$

$\langle \texttt{f2} \mapsto \texttt{wkbag}(\langle 4{\mapsto}3\ 3{\mapsto}\bot\ 2{\mapsto}0\ 1{\mapsto}\bot\ 0{\mapsto}\bot\ \bot{\mapsto}\bot \rangle_{\mathsf{hf}})\rangle_{\mathsf{fld}}$

$\rangle_{\mathsf{h}}$

$\langle \texttt{g1}{\mapsto}0\ \texttt{g2}{\mapsto}2\ \texttt{g3}{\mapsto}3\ \texttt{l1}{\mapsto}\bot\ \texttt{l2}{\mapsto}\bot\ \texttt{l3}{\mapsto}\bot \rangle_{\mathsf{var}}$

$\rangle_{\mathsf{heap}}$

$\Rrightarrow$ g1:=new// STEP 20 (object creation)

$\langle \langle \langle \texttt{CutPoints} \mapsto \texttt{wkbag}(\langle \texttt{cp}_1{\mapsto}3\ \texttt{cp}_2{\mapsto}0 \rangle_{\mathsf{hf}})\rangle_{\mathsf{fld}}$

$\langle \texttt{f1} \mapsto \texttt{wkbag}(\langle 4{\mapsto}\bot\ 3{\mapsto}2\ 2{\mapsto}2\ 1{\mapsto}0\ 0{\mapsto}1\ \bot{\mapsto}\bot \rangle_{\mathsf{hf}})\rangle_{\mathsf{fld}}$

$\langle \texttt{f2} \mapsto \texttt{wkbag}(\langle 4{\mapsto}\bot\ 3{\mapsto}\bot\ 2{\mapsto}0\ 1{\mapsto}\bot\ 0{\mapsto}\bot\ \bot{\mapsto}\bot \rangle_{\mathsf{hf}})\rangle_{\mathsf{fld}}$

$\rangle_{\mathsf{h}}$

$\langle \texttt{g1}{\mapsto}4\ \texttt{g2}{\mapsto}2\ \texttt{g3}{\mapsto}3\ \texttt{l1}{\mapsto}\bot\ \texttt{l2}{\mapsto}\bot\ \texttt{l3}{\mapsto}\bot \rangle_{\mathsf{var}}$

$\rangle_{\mathsf{heap}}$

$\Rrightarrow$ g1.f1:=g2// STEP 21 (assignment field left)

$\langle \langle \langle \texttt{CutPoints} \mapsto \texttt{wkbag}(\langle \texttt{cp}_1 \mapsto 3\ \texttt{cp}_2 \mapsto 0 \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\langle \texttt{f1} \mapsto \texttt{wkbag}(\langle 4 \mapsto 2\ 3 \mapsto 2\ 2 \mapsto 2\ 1 \mapsto 0\ 0 \mapsto 1\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\langle \texttt{f2} \mapsto \texttt{wkbag}(\langle 4 \mapsto \bot\ 3 \mapsto \bot\ 2 \mapsto 0\ 1 \mapsto \bot\ 0 \mapsto \bot\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\rangle_{\mathsf{h}}$

$\langle \texttt{g1} \mapsto 4\ \texttt{g2} \mapsto 2\ \texttt{g3} \mapsto 3\ \texttt{l1} \mapsto \bot\ \texttt{l2} \mapsto \bot\ \texttt{l3} \mapsto \bot \rangle_{\mathsf{var}}$

$\rangle_{\mathsf{heap}}$

$\Rrightarrow$ `l1:=g1.f1`// STEP 22 (assignment field right)

$\langle \langle \langle \texttt{CutPoints} \mapsto \texttt{wkbag}(\langle \texttt{cp}_1 \mapsto 3\ \texttt{cp}_2 \mapsto 0 \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\langle \texttt{f1} \mapsto \texttt{wkbag}(\langle 4 \mapsto 2\ 3 \mapsto 2\ 2 \mapsto 2\ 1 \mapsto 0\ 0 \mapsto 1\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\langle \texttt{f2} \mapsto \texttt{wkbag}(\langle 4 \mapsto \bot\ 3 \mapsto \bot\ 2 \mapsto 0\ 1 \mapsto \bot\ 0 \mapsto \bot\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\rangle_{\mathsf{h}}$

$\langle \texttt{g1} \mapsto 4\ \texttt{g2} \mapsto 2\ \texttt{g3} \mapsto 3\ \texttt{l1} \mapsto 2\ \texttt{l2} \mapsto \bot\ \texttt{l3} \mapsto \bot \rangle_{\mathsf{var}}$

$\rangle_{\mathsf{heap}}$

$\Rrightarrow$ `g1.f2:=l1`// STEP 23 (assignment field right)

$\langle \langle \langle \texttt{CutPoints} \mapsto \texttt{wkbag}(\langle \texttt{cp}_1 \mapsto 3\ \texttt{cp}_2 \mapsto 0 \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\langle \texttt{f1} \mapsto \texttt{wkbag}(\langle 4 \mapsto 2\ 3 \mapsto 2\ 2 \mapsto 2\ 1 \mapsto 0\ 0 \mapsto 1\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\langle \texttt{f2} \mapsto \texttt{wkbag}(\langle 4 \mapsto 2\ 3 \mapsto \bot\ 2 \mapsto 0\ 1 \mapsto \bot\ 0 \mapsto \bot\ \bot \mapsto \bot \rangle_{\mathsf{hf}}) \rangle_{\mathsf{fld}}$

$\rangle_{\mathsf{h}}$

$\langle \texttt{g1} \mapsto 4\ \texttt{g2} \mapsto 2\ \texttt{g3} \mapsto 3\ \texttt{l1} \mapsto 2\ \texttt{l2} \mapsto \bot\ \texttt{l3} \mapsto \bot \rangle_{\mathsf{var}}$

$\rangle_{\mathsf{heap}}$

6.2 Model checking Shylock programs

"Tell me where is fancy bred,

Or in the heart, or in the head?"

— William Shakespeare, The Merchant of Venice.

In this section we discuss opportunities and limitations of shape analysis as model checking for Shylock. We first revise why using a Maude's model checker imposes artificial limitations for Shylock, then we propose a $\mathbb{K}$ specification for state invariant model checking.

We employed the $\mathbb{K}$ specification of Shylock to derive verification capabilities provided for free by the Maude LTL model checker. The atomic properties, defined as regular expressions for the heap structure exploration, were specified directly in Maude using the circularity principle [40, 87, 13] to handle the eventual cycles in the heap.

Maude's model checker provides a substantial expressivity w.r.t. atomic properties in comparison with other model checking tools. This feature justifies the appropriateness of using $\mathbb{K}$-Maude for Shylock in order to consequently model check Shylock programs. However, due to the prerequisites imposed by the Maude's model checker, we could successfully verify only a restricted class of Shylock programs, namely the non-recursive ones. This is because, recursion in Shylock increases the stack size infinitely, hence the number of reachable configurations is infinite. In these cases, the Maude LTL model checker fails to produce a result.Consequently, if we are to use Shylock to its full potential, we need to attempt either patching this approach or choosing other solution(s).

A patch that one could approach is applying an additional abstraction on Shylock's k cell, i.e., the stack of the pushdown system, in order to have a finite set of reachable configurations, even for programs with recursion. A different solution is specifying a model checking algorithm directly in $\mathbb{K}$ and use it for the abstract semantics of Shylock

programs. We present here the second option, by describing how a state of the art algorithm for model checking pushdown systems is adapted and translated into a $\mathbb{K}$-specification.

One of the major problems in model checking programs which manipulate dynamic structures, such as linked lists, is that it is not possible to bound a priori the state space of the possible computations. This is due to the fact that programs may manipulate the heap by dynamically allocating an unbounded number of new objects and by updating reference fields (pointers). The combination of these two features implies that the state space is infinite, and model checking as well as reachability is in general undecidable. Consequently for model checking purposes we need to impose some suitable bounds on the model of the program. A natural bound for model checking programs without necessarily restricting their capability of allocating an unbounded number of objects is to impose constraints on the size of the *visible* heap [15]. The visible heap consists of those objects which are reachable from the variables in the scope of the currently executed procedure. Such a bound still allows for storage of an unbounded number of objects onto the call-stack, using local variables.

Under the assumption of a bounded heap the $\mathbb{K}$ specification for Shylock programs is bisimilar with a finite state pushdown system and compiles in Maude into a rewriting system. Obviously, in the presence of recursive procedures, the stack in the pushdown system grows unboundedly and, even if the abstraction ensures a finite state space, the equivalent transition system is infinite and so is the associated rewriting system. To overcome this drawback we apply collecting semantics over the current abstraction to specify model checking algorithms for pushdown systems.

We give next some keynotes on Algorithm 4.

In the Algorithm 4, the “trans” variable is a queue containing the transitions to be processed. Along the execution of the Algorithm 4 the transitions of the A_{post^*}

Algorithm 4: The algorithm for obtaining A_{post*}, and the set of reachable configurations of finite a pushdown system, adapted from [95] by Marcello Bonsangue and Jurriaan Rot for specifications of finite pushdown system semantics with the restriction that a rule application can increase the stack size by *at most* one.

Input: a concrete configuration $< h, p >$, where h is a heap, and p is a procedure name and a heap formula ϕ.

```
1  if x_0 ⊭ φ then return false;
2  trans := [(x_0, γ_0, f)];
3  rel := ∅;
4   while trans = [(x, γ, y)]| trans' do
5    trans := trans';
6    if (x, γ, y) ∉ rel then
7      rel := rel ∪{(x, γ, y)};
8      if γ ≠ ε then
9        for all z such that < x, γ > ⟶ < z, ε > is derivable from semantics do
10         if z ⊭ φ then return false;
11         trans := trans |[(z, ε, y)];
12       for all z such that < x, γ > ⟶ < z, γ_1 > is derivable from semantics do
13         if z ⊭ φ then return false;
14         trans := trans |[(z, γ_1, y)];
15       for all z such that < x, γ > ⟶ < z, γ_1 γ_2 > is derivable from semantics do
16         if z ⊭ φ then return false;
17         trans := trans |[(z, γ_1, y_{z,γ_1})];
18         rel := rel ∪{(y_{z,γ_1}, γ_2, y)});
19         for all (u, ε, y_{z,γ_1}) ∈ rel do
20           trans := trans ∪{((u, γ_2, y))};
21     else
22       for all (y, γ_1, z) ∈ rel do
23         trans := trans ∪{((x, γ_1, z))};
24 od;
25 return true
```

automaton are incrementally deposited in the "rel" variable. By

$$< x, \gamma > \longrightarrow < z, \varepsilon > \text{ is derivable from semantics}$$

we mean this concrete rule matches a rule in a pushdown system specification, e.g., Shylock.

The outermost **while** is executed until the end, i.e., until "trans" is empty, only if all states satisfy the control state formula ϕ. Thus termination is guaranteed with heap-bounded model checking of the form

$$\models_k \Box\phi$$

meaning $\models \Box\phi \wedge le(k)$, where $le(k)$ is verifies if the size of the reachable heap is smaller than k.

However, note that Algorithm 4 assumes that the pushdown system specification has only rules which push on the stack *a*t most two elements. This makes impossible the application of Algorithm 4 for Shylock, where the stack size can be increased with any number of stack symbols in the rule for procedure call . The solution we propose in Algorithm 5 adapts Algorithm 4 to the general case when the stack can grow arbitrarily. This solution induces a localized modification of the lines 15-20 in the Algorithm 5.

6.2.1 Model checking pushdown systems specifications in $\mathbb{K}$

Here we give in Figure 6.3 a $\mathbb{K}$-specification called $kA_{post^*}(\phi, k\mathscr{P})$ a parametric $\mathbb{K}$ specification, where the two parameters are ϕ a state invariant and $k\mathscr{P}$, the $\mathbb{K}$ specification of a pushdown system. We show that $kA_{post^*}(\phi, k\mathscr{P})$ is behaviorally equivalent with the Algorithm 5. We describe $kA_{post^*}(\phi, k\mathscr{P})$ along justifying the behavioral equivalence with Algorithm 5.

Algorithm 5: The algorithm for obtaining A_{post*}, and the set of reachable configurations of finite a pushdown system, modified for the general case.

Input: a concrete configuration $< h, p >$, where h is a heap, and p is a procedure name and a heap formula ϕ.

1 **if** $h \not\models \phi$ **then return** false;
2 trans := $[(h, p, f)]$;
3 rel := $\emptyset$;
4 **while** trans = $[(x, \gamma, y)]|$ trans' **do**
5 trans := trans';
6 **if** $(x, \gamma, y) \notin$ rel **then**
7 rel := rel $\cup\{(x, \gamma, y)\}$;
8 **if** $\gamma \neq \varepsilon$ **then**
9 **for all** z such that $< x, \gamma > \longrightarrow < z, \varepsilon >$ is derivable from semantics **do**
10 **if** $z \not\models \phi$ **then return** false;
11 trans := trans $|[(z, \varepsilon, y)]$;
12 **for all** z such that $< x, \gamma > \longrightarrow < z, \gamma_1 >$ is derivable from semantics **do**
13 **if** $z \not\models \phi$ **then return** false;
14 trans := trans $|[(z, \gamma_1, y)]$;
15 **for all** z such that $< x, \gamma > \longrightarrow < z, \gamma_1..\gamma_n >$ is derivable from semantics s.t. $n \geq 2$ **do**
16 **if** $z \not\models \phi$ **then return** false;
17 trans := trans $|[(z, \gamma_1, y_{z,\gamma_1})]$;
18 rel := rel $\cup\{(y_{z,\delta(r,i)}, \gamma_{i+2}, y_{z,\delta(r,i+1)}) \mid 0 \leq i \leq n-2\}$;
where r denotes $< x, \gamma > \longrightarrow < z, \gamma_1..\gamma_n >$
and $\delta(r,i), 1 \leq i \leq n-2$ are new symbols
and $y_{z,\delta(r,0)} = y_{z,\gamma_1}$ and $y_{z,\delta(r,n-1)} = y$
19 **for all** $(u, \varepsilon, y_{z,\delta(r,i)}) \in rel, 0 \leq i \leq n-2$ **do**
20 trans := trans $\cup\{(u, \gamma_{i+2}, y_{z,\delta(r,i+1)}) \mid 0 \leq i \leq n-2\}$;
21 **else**
22 **for all** $(y, \gamma_1, z) \in rel$ **do**
23 trans := trans $\cup\{(x, \gamma_1, z)\}$;
24 **od**;
25 **return** true

RULE1 [**if** $(x,\gamma,y) \notin rel$ **else**] :

$\langle\cdot\rangle_{\mathsf{traces}}\langle\cdot\rangle_{\mathsf{traces}'}\langle\cdots\ \langle\cdots\ \underset{\cdot}{x \overset{\gamma}{\rightsquigarrow} y}\ \cdots\rangle_{\mathsf{trans}}\langle\cdots\ x \overset{\gamma}{\rightsquigarrow} y\ \cdots\rangle_{\mathsf{rel}}\langle\cdot\rangle_{\mathsf{memento}}\ \cdots\rangle_{\mathsf{collect}}$

RULE2 [**if** $(x,\gamma,y) \notin rel$ **then**...**if** $\gamma \neq \varepsilon$ **else**] :

$\langle\cdot\rangle_{\mathsf{traces}}\langle\cdot\rangle_{\mathsf{traces}'}\langle\cdots\ \langle\cdots\ \underset{(x \rightsquigarrow Rel[y \overset{-}{\rightsquigarrow}])}{x \overset{\varepsilon}{\rightsquigarrow} y}\ \cdots\rangle_{\mathsf{trans}}\langle\ \underset{x \overset{\varepsilon}{\rightsquigarrow} y}{\cdot}\ Rel\rangle_{\mathsf{rel}}\langle\cdot\rangle_{\mathsf{memento}}\ \cdots\rangle_{\mathsf{collect}}$

when $x \overset{\varepsilon}{\rightsquigarrow} y \notin Rel$

RULE3 [**if** $(x,\gamma,y) \notin rel$ **then**...**if** $\gamma \neq \varepsilon$ **then**] :

$\langle\ \underset{\langle\langle x\rangle_{\mathsf{ctrl}}\langle\gamma\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}}{\cdot}\ \rangle_{\mathsf{traces}}\langle\cdot\rangle_{\mathsf{traces}'}\langle\cdots\ \langle\cdots\ \underset{\cdot}{x \overset{\gamma}{\rightsquigarrow} y}\ \cdots\rangle_{\mathsf{trans}}\langle\ \underset{x \overset{\gamma}{\rightsquigarrow} y}{\cdot}\ Rel\rangle_{\mathsf{rel}}\langle\ \underset{x \overset{\gamma}{\rightsquigarrow} y}{\cdot}\ \rangle_{\mathsf{memento}}\ \cdots\rangle_{\mathsf{collect}}$

when $x \overset{\gamma}{\rightsquigarrow} y \notin Rel\ and_{Bool}\ \gamma \neq \varepsilon$

RULE4 [**for all** z such that $\langle x,\gamma\rangle \longrightarrow \langle z,\varepsilon|\gamma|\gamma_1..\gamma_n\rangle$ is derivable from semantics **do**] :

$\langle\cdot\rangle_{\mathsf{traces}}\langle\underset{\cdot}{\langle\langle z\rangle_{\mathsf{ctrl}}\ \cdots\rangle_{\mathsf{trace}}}\rangle_{\mathsf{traces}'}\langle\langle\ \underset{\cdot}{-}\ \rangle_{\mathsf{trans}}\langle\ \underset{\cdot}{-}\ \rangle_{\mathsf{rel}}\langle\ \underset{\cdot}{-}\ \rangle_{\mathsf{memento}}\langle\phi\rangle_{\mathsf{formula}}\langle\ \underset{false}{true}\ \rangle_{\mathsf{return}}\rangle_{\mathsf{collect}}$

when $z \not\models \phi$

RULE5 [**for all** z such that $\langle x,\gamma\rangle \longrightarrow \langle z,\varepsilon|\gamma\rangle$ is derivable from semantics **do**] :

$\langle\cdot\rangle_{\mathsf{traces}}\langle\cdots\ \underset{\cdot}{\langle\langle z\rangle_{\mathsf{ctrl}}\langle\Gamma'\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}}\ \cdots\rangle_{\mathsf{traces}'}\langle\cdots\ \langle\cdots\ \underset{z \overset{\Gamma'}{\rightsquigarrow} y}{\cdot}\ \cdots\rangle_{\mathsf{trans}}\langle x \overset{\gamma}{\rightsquigarrow} y\rangle_{\mathsf{memento}}\langle\phi\rangle_{\mathsf{formula}}\ \cdots\rangle_{\mathsf{collect}}$

when $|\Gamma'| \leq 1\ and_{Bool}\ z \models \phi$

RULE6 [**for all** z such that $\langle x,\gamma\rangle \longrightarrow \langle z,\gamma_1..\gamma_n\rangle$ is derivable from semantics **do**] :

$\langle\cdot\rangle_{\mathsf{traces}}\langle\cdots\ \underset{\cdot}{\langle\langle z\rangle_{\mathsf{ctrl}}\langle\gamma' \curvearrowright \Gamma'\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}}\ \cdots\rangle_{\mathsf{traces}'}$

$\langle\cdots\ \underset{z \overset{\gamma'}{\rightsquigarrow} new(z,\gamma')\ (Rel[\overset{\varepsilon}{\rightsquigarrow} new(z,\gamma'), news(x,\gamma,z,\gamma',\Gamma')] \overset{\Gamma'}{\rightsquigarrow} news(x,\gamma,z,\gamma',\Gamma'),y)}{\cdot}\ \cdots\rangle_{\mathsf{trans}}$

$\langle Rel\ \underset{new(z,\gamma'),news(x,\gamma,z,\gamma',\Gamma') \overset{\Gamma'}{\rightsquigarrow} news(x,\gamma,z,\gamma',\Gamma'),y}{\cdot}\ \rangle_{\mathsf{rel}}\langle x \overset{\gamma}{\rightsquigarrow} y\rangle_{\mathsf{memento}}\langle\phi\rangle_{\mathsf{formula}}$

when $|\Gamma'| \geq 1\ and_{Bool}\ z \models \phi$

RULE7 [**for all** z such that $\langle x,\gamma\rangle \longrightarrow \langle z,\varepsilon|\gamma|\gamma_1..\gamma_n\rangle$ is derivable from semantics **do**] :

$\langle\cdot\rangle_{\mathsf{traces}}\langle\cdot\rangle_{\mathsf{traces}'}\langle\cdots\ \underset{\cdot}{\langle x \overset{\gamma}{\rightsquigarrow} y\rangle_{\mathsf{memento}}}\ \cdots\rangle_{\mathsf{collect}}$

RULE$\mathscr{P}$ [**for all** z such that $\langle x,\gamma\rangle \longrightarrow \langle z,\Gamma\rangle$ is derivable from semantics **do**] :

$\langle\cdots\ \underset{\cdot}{\langle\langle x\rangle_{\mathsf{ctrl}}\langle\gamma \curvearrowright \Gamma\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}}\ \cdots\rangle_{\mathsf{traces}}\langle\cdots\ \underset{\langle\langle z_0\rangle_{\mathsf{ctrl}}\langle\Gamma_0 \curvearrowright \Gamma\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}..\langle\langle z_n\rangle_{\mathsf{ctrl}}\langle\Gamma_n \curvearrowright \Gamma\rangle_{\mathsf{k}}\rangle_{\mathsf{trace}}}{\cdot}\ \cdots\rangle_{\mathsf{traces}'}$

Figure 6.3: $kA_{post^*}(\phi, k\mathscr{P})$

Proposition 4. *The $\mathbb{K}$ specification of $kA_{post^*}(true, k\mathscr{P})$ in Figure 6.3 defines the generation of the A_{post^*} automaton for pushdown systems $\mathbb{K}$ specifications $k\mathscr{P}$.*

Proof. We prove this by showing the behavioral equivalence between $kA_{post^*}(\phi, k\mathscr{P})$ and Algorithm 5.

The **while** loop in Algorithm 5 is maintained in A_{post^*} by the application of rewriting, until the term reaches the normal form, i.e. no other rule can be applied. This is ensured by the fact that from the initial configuration

$$Init \equiv \langle\cdot\rangle_{\mathsf{traces}}\langle\cdot\rangle_{\mathsf{traces}'}\langle\langle x_0 \overset{\gamma_0}{\rightsquigarrow} f\rangle_{\mathsf{trans}}\langle\cdot\rangle_{\mathsf{rel}}\langle\cdot\rangle_{\mathsf{memento}}\langle\phi\rangle_{\mathsf{formula}}\langle true\rangle_{\mathsf{return}}\rangle_{\mathsf{collect}}$$

the rules keep applying, as long as trans-cell is nonempty. Note that we assume, without loss of generality, that the initial stack has one symbol on it.

We assume that the rewrite rules are applied at-random, so we need to direct/pipeline the flow of their application via matching and conditions. The notation RULEi $[label]$ in the beginning of each rule hints, via $[label]$, towards which part of the Algorithm 5 that rule is handling. In the followings we discuss each rule and justify its connection with code fragments in Algorithm 5.

- The last rule, RULE$\mathscr{P}$, performs the exhaustive unfolding for a particular configuration in cell trace. We use this rule in order to have a parametric definition of the kA_{post^*} specification, where one of the parameters is the $\mathbb{K}$ semantics providing the pushdown system specification. The other parameter is the specification of the language defining the state invariant properties which are to be verified on the produced pushdown system.

  Pipeline:

- Firstly, when a semantic derivation is processed by kA_{post^*}, we enforce both cells $\mathsf{traces}, \mathsf{traces}'$ to be empty (using the matching $\langle\cdot\rangle_{\mathsf{traces}}\langle\cdot\rangle_{\mathsf{traces}'}$). This happens

for RULE1 and RULE2 because the respective portions in the Algorithm 5 do not need transitions to update the variables "trans" and "rel";

- The other cases, namely when transitions are used for updating "trans" and "rel", are triggered in RULE3 by placing the desired configuration in the cell traces, while the cell traces$'$ is empty. At this point, since all the other rules match on either traces empty, or traces$'$ nonempty, only RULE$\mathscr{P}$ can be applied. This rule uses the populate traces$'$ with all the next configurations, using directly the semantics.

- After the application of RULE$\mathscr{P}$, only one of the rules RULE4, 5, 6 can be applied because these rules are the only ones in $kAlg_{post^*}$ matching an empty traces cell and a nonempty traces$'$ cell.

- Among the rules 4,5,6 the differentiation is made via conditions as follows:

  + RULE4 handles all the cases when the new configuration has a control location z which doesn't verify the state invariant ϕ (i.e., lines $10, 13, 16$). In this case we close the pipeline and the algorithm by emptying all the cells traces, traces, trans. Note that all the rules handling the **while** loop match on at least a nonempty cell traces, traces or trans, with a pivot in a nonempty-trans.

  + Rules 5 and 6 are applied disjunctively of rule 4 because both have the condition $z \models \phi$. Next we describe these two rules:

    ++ RULE5 handles the case when the semantic rule in $\mathscr{P}$ which matches the current $< x, \gamma >$ does not increase the stack. This case is associated with the lines $9+11, 12+14$.

    ++ RULE6 handles the case when the semantic rule in $\mathscr{P}$ which matches the current $< x, \gamma >$ increases the stack size, associate with the lines

$15+17-20$ in Algorithm 5.

+ Both rules 4 and 5 use the memento cell which is filled upon pipeline initialization, in RULE3.

+ the most intricate rule is rule 6, because it handles a **for all** piece of code, i.e., lines 17-20 in Algorithm 5. This part is made by matching the entire content of cell rel with *Rel*, and using the projection operator

$$Rel[\overset{\gamma}{\rightsquigarrow} z_1,..,z_n] := \{u \mid (u,\gamma,z_1) \in Rel\},..,\{u \mid (u,\gamma,z_n) \in Rel\}$$

where $z_1,..,z_n$ in the left hand-side is a list of z-symbols, while in the right hand-side we have a list of sets.

+ Note that rules 4,5,6 match on a nonempty traces′-cell and an empty traces, and no other rule matches alike.

• RULE7 closes the pipeline when the traces′ cell becomes empty (the rules 4, 5, 6 keep consuming from it), by making the memento cell empty. Note that the memento cell was filled in the beginning of the pipeline, in rule3, and marks the end of the pipeline.

□

6.2.2 Shape analysis via model checking Shylock

To this end, we use atomic propositions defined as regular expressions in what is basically a Kleene algebra with tests [56]. Namely, the global and local variables of a program are used as *nominals*, whereas the fields constitute the set of basic actions. We give in Figure 6.4 the definition of these regular expressions as defined in [90].

The $\mathbb{K}$ specification of *Rite* is based on the circularity principle [40, 87, 13]. We employ *Rite* with the $kA_{post^*}(\phi, k\mathscr{P})$ invariant model checker for verifying heap-structure

Rite is the smallest set defined by the following grammar:

$$r ::= \varepsilon \mid x \mid \neg x \mid f \mid r.r \mid r + r \mid r^*$$

where x ranges over variable names (to be used as tests) and f over field names (to be used as actions).
We define a transition relation $n \xrightarrow{r}_H m$ between objects of a heap H as the least relation such that:

$$\begin{array}{ll}
n \xrightarrow{\varepsilon}_H n & \\
n \xrightarrow{x}_H n & \text{if } H(x) = n \\
n \xrightarrow{\neg x}_H n & \text{if } H(x) \neq n \\
n \xrightarrow{f}_H m & \text{if } H(f)(n) = m \\
n \xrightarrow{r_1 + r_2}_H m & \text{if } n \xrightarrow{r_1} m \text{ or } n \xrightarrow{r_2} m \\
n \xrightarrow{r_1 . r_2}_H m & \text{if exists an object } n' \text{ such that } n \xrightarrow{r_1}_H n' \text{ and } n' \xrightarrow{r_2}_H m \\
n \xrightarrow{r^*}_H m & \text{if either } n = m \text{ or there exists an object } n' \text{ such that} \\
 & \quad n \xrightarrow{r}_H n' \text{ and } n' \xrightarrow{r^*}_H m
\end{array}$$

Further we introduce the following *modal* interpretation of regular expressions:

$H \models r$ *iff* for each reachable object n in H there exists m such that $n \xrightarrow{r}_H m$.

Figure 6.4: The shape properties for the heap as defined in [90]

properties for Shylock programs (i.e., $\phi \in$ *Rite* and $k\mathscr{P}$ is Shylock's $\mathbb{K}$-specification).

Example 12. *The following Shylock program is the basic example we use to exemplify Shylock .*

```
macro pgmExample0 =

gvars: g

{

  main :: p0

  p0 :: g:=new;  p0

}
```

The pushdown automaton associated to `pgmExample0` *Shylock program has the following (ground) rules:*

$(\mathtt{g} : \bot,\ \mathtt{main}) \hookrightarrow (\mathtt{g} : \bot,\ \mathtt{p0};\mathtt{restore}(\mathtt{g} : \bot))$

$(\mathtt{g} : \bot,\ \mathtt{p0}) \hookrightarrow (\mathtt{g} : \bot,\ \mathtt{g} := \mathtt{new};\mathtt{p0};\mathtt{restore}(\mathtt{g} : \bot))$

$(\mathtt{g} : \bot,\ \mathtt{g} := \mathtt{new}) \hookrightarrow (\mathtt{g} : 0,\ \varepsilon)$

$(\mathtt{g} : 0,\ \mathtt{p0}) \hookrightarrow (\mathtt{g} : 0,\ \mathtt{g} := \mathtt{new};\mathtt{p0};\mathtt{restore}(\mathtt{g} : 0))$

$(\mathtt{g} : 0,\ \mathtt{g} := \mathtt{new}) \hookrightarrow (\mathtt{g} : 1,\ \varepsilon)$

$(\mathtt{g} : 1,\ \mathtt{p0}) \hookrightarrow (\mathtt{g} : 1,\ \mathtt{g} := \mathtt{new};\mathtt{p0};\mathtt{restore}(\mathtt{g} : 1))$

$(\mathtt{g} : 1,\ \mathtt{g} := \mathtt{new}) \hookrightarrow (\mathtt{g} : 0,\ \varepsilon)$

In Figure 6.5 we give the reachability automaton produced by kA_{post^*} *for this example with the default property* $\phi = true$*. Note that we cannot obtain the pushdown system automaton by the exhaustive execution of* `pgmExample0` *with Shylock abstract semantics because the exhaustive execution is infinite due to recursive procedure* `p0`*. Hence we cannot use directly the dedicated pushdown systems model checkers as these work with the pushdown system automaton, while what we have in Shylock*[`pgmExample0`] *is a pushdown system specification.*

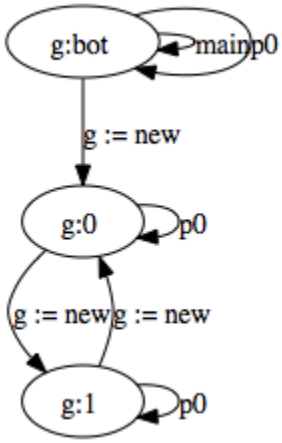


Figure 6.5: The reachability automaton produced for `pgmExample0` by kA_{post^*}

Example 13. *The following Shylock program creates an infinite linked list which starts in* `first` *object and ends with* `last` *object.*

```
macro pgmBoundedList =
  gvars: first, last
  lvars: tmp
  flds:  next
{
  main ::  last:=new;  last.next:=last;  first:=last;  p0
  p0 ::  tmp:=new;  tmp.next:=first;  first:=tmp;   (p0 + skip)
}
```

This is an example of a program which induces, on some computation path, an unbounded heap. When we apply the heap-bounded model checking specification, by instantiating ϕ with the property $le(10)$ we obtain in the collect *cell all lists with a length smaller or equal than* 10*. Furthermore, we can check the shape of these lists with the property*

$$(\neg \texttt{first} + \texttt{first.next}^{*}.\texttt{last})$$

This shape property says that either the `first` *object is not defined or that the* `last` *object is reached from the* `first` *object via the* `next` *field.*

Example 14. *The following Shylock program creates an "undecided" list which turns left or right, according to the nondeterministic choice* $\texttt{tmp.left} := \texttt{x} + \texttt{tmp.right} := \texttt{y}$.

```
macro pgmUndecidedList =
  gvars: root, leaf
  lvars: x, y, tmp, aux
  flds: left, right
{
```

```
  main :: root := new; leaf := root; p(n)
  p(n) :: tmp := new; x := new; y := new;
              (tmp.left := x + tmp.right := y);
              p(n-1);
              aux := tmp.left;
              [aux = x] (x.right := root);
              aux := tmp.right;
              [aux = y] (y.left := root);
              root := tmp
 p(0) :: skip
}
```

Running this program with $kA_{post^*}(true, \text{Shylock})$*, we obtain all the lists turning left or right. What is important to note here about Shylock is that the objects forming any of the "undecided" lists, starting with the id of the* `root`*, are always a permutation of the integers* 0..2n*. This is due to the memory reuse and the renaming scheme used by Shylock. An example of a shape invariant we verify on* `pgmUndecidedList` *is:*

$$(\neg \texttt{root} + \texttt{root}.(\texttt{left} + \texttt{right})^*.\texttt{leaf})$$

This shape property specifies that in `pgmUndecidedList` *the* `root` *variable is either undefined of, if it is defined, then from the* `root` *we can reach the* `leaf` *via repeatedly either* `left` *or* `right` *fields.*

6.3 Related work

The work in [78] introduces the notion of cut points to employ a pure abstract interpretation fixpoint for shape analysis. Here we use the analysis view promoted in [94], and

described in Chapter 1. Namely, the shape analysis is achieved in a demand driven fashion using state invariant model checking for pushdown systems with atomic properties specifying shape patterns for the reachable heaps.

In [15] a language is studied with the same features as our Shylock programs extended with a bounded form of concurrency. The work [15] uses finite graphs and graph isomorphisms to represent heaps and ensure a finite reachable state space. The results in [15] are purely theoretical while the $\mathbb{K}$ specification for Shylock and kA_{post^*} ensures executability for our work. Another difference is in the fact that instead of graph isomorphism, Shylock compares two heaps via matching which is more efficient in a rewriting environment. In future work we plan to implement a case-based comparison of Shylock with [15].

In the area of interprocedural shape analysis, in the presence of recursion, we mention two important abstract interpretation based tools: TVLA and Xisa. TVLA framework [12, 53, 111] uses three-valued structures for predicates modeling stack summaries, while Xisa [79, 16, 112] exploits the regular, inductive structure of the stack with a shape properties defined in the style of separation logic. Fundamentally we exploit the same result, i.e., the regularity of the stack, but we use this result via a state of the art model checking algorithm for pushdown systems [95]. The drawback in our case is that we define a demand-driven shape analysis, i.e., without inferring the shape invariants.

The matching logic generic reachability approach in [84, 82, 85] is probably the closest work w.r.t. the $\mathbb{K}$ model checking part of this chapter. What differs in [84, 82, 85] is that matching logic proposes a deductive verification, while our approach relies on model checking for pushdown systems. Moreover, the current status of the matching logic for reachability works under the assumption that the computation terminates, while an asset of our approach is that kA_{post^*} can handle infinite recursion. Nevertheless, automated methods for deductive verification is an prolific research area. Besides

matching logic, other tools and techniques for verifying programs manipulating the heap by deductive verification methods are presented in, e.g., [32, 59, 77].

6.4 Conclusions

In this chapter we presented the $\mathbb{K}$ implementation of an abstract programming language, called Shylock. We used Shylock as an abstraction for shape analysis by model checking shape invariants in the heap, according to the methodology presented in Chapter 3.

Shylock's $\mathbb{K}$ specification presented in Section 6.1 is faithful to the "on paper" semantics provided in [90]. The $\mathbb{K}$ specification for Shylock has the following desiderata: to contribute to the pool of languages defined in $\mathbb{K}$ with a fairly elaborated language semantics; to set the grounds for an extensive future work of semi-automatically proving the simulation between the concrete and abstract semantics; most importantly, to use the semantics for shape analysis acquired via invariant model checking.

In order to produce the shape analysis based on Shylock, in Section 6.2 we defined kA_{post^*} - a $\mathbb{K}$ specification of a state of the art algorithm for model checking pushdown system from [95]. In order to define shape properties, we used the $\mathbb{K}$ specification of *Rite* a regular expression language for shape patterns in the heap. We employed kA_{post^*} over Shylock to obtain the reachable state space and used *Rite* to obtain the desired demand driven shape analysis. The analysis we proposed is sound for programs which guarantee a bound for the reachable heap.

Chapter 7

Conclusions

Motto: *"I think and think for months and years,*
ninety-nine times, the conclusion is false.
The hundredth time I am right."
— Albert Einstein.

The main objective of the current work was the study of a methodology and some of its instantiations on how abstract interpretation can be used in $\mathbb{K}$ for program analysis and verification. Namely, in Chapter 3 we were concerned with defining the methodological perspective employed by collecting semantics over dynamic semantics of push-down system abstractions. In the subsequent chapters we presented three instantiations of the methodology with some standard results achieved by static analysis via static semantics. In doing this, we rely on [94] where static analysis is mapped into abstract model checking.

The three instantiations of the methodology are data analysis in Chapter 4, alias analysis in Chapter 5, and shape analysis in Chapter 6.

For data analysis we proposed a $\mathbb{K}$ specification for collecting semantics under predicate abstraction of a simple imperative language. The collecting semantics in this case defined state invariant model checking for sequential programming languages with tail

recursion, exemplified here by a "while" language.

We achieved alias analysis by means of collecting semantics for an abstract programming language, called *SILK*. The "on paper" semantics of this language is presented as a finite pushdown system specification in [91, 89] and we described here a faithful $\mathbb{K}$ specification. We defined the collecting semantics specification for *SILK* by an associative matching representation of the fundamental decidability result of finite pushdown systems' reachability from [14]. Our approach renders an interprocedural, flow sensitive, demand driven, may or must alias analysis.

The shape analysis was achieved via abstract model checking. Namely, we first presented the $\mathbb{K}$ specification of Shylock - an abstract programming language introduced in [90, 8] as "on paper" pushdown system specification. For collecting semantics we defined kA_{post^*} - the $\mathbb{K}$ specification of a state invariant model checking for pushdown systems specifications. kA_{post^*} is based on a state of the art model checker for pushdown systems introduced in [95]. This collecting semantics is employed with *Rite* - a regular language defining shape patterns for the heap, which is introduced in [90]. Hence, we give the $\mathbb{K}$ specification for a demand driven interprocedural, flow sensitive shape analysis which is sound under the assumption that the reachable heaps are bounded.

Ongoing work includes completing the predicate abstraction mechanism with counter-example guided refinement [5], extending the invariant model checking algorithm kA_{post^*} to generic LTL model checking, and updating the presented $\mathbb{K}$ specifications for the latest $\mathbb{K}$-tool. In case the $\mathbb{K}$-tool allows running performance, we can also approach benchmarking the implemented methods, as for the moment we resumed on testing the specifications on small key examples.

References

[1] M. Alba-Castro, M. Alpuente, and S. Escobar. Automatic certification of Java source code in rewriting logic. S. Leue and P. Merino, editors, *FMICS*, volume 4916 of *Lecture Notes in Computer Science*, pages 200–217. Springer, 2007.

[2] I. M. Asăvoae. Abstract semantics for alias analysis in K. *K 2011*, 2012. To appear.

[3] I. M. Asăvoae. Systematic design of abstractions in K. *WADT 2012, Technical Report TR-08/12, Universidad Complutense de Madrid, Departamento de Sistemas Informáticos y Computación*, pages 9–11, 2012. Accepted for presentation at WADT'12, `http://maude.sip.ucm.es/wadt2012/docs/WADT2012-preproceedings.pdf`.

[4] I. M. Asăvoae and M. Asăvoae. Collecting semantics under predicate abstraction in the K framework. P. C. Ölveczky, editor, *Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010, Held as a Satellite Event of ETAPS 2010, Paphos, Cyprus, March 20-21, 2010, Revised Selected Papers*, volume 6381 of *Lecture Notes in Computer Science*, pages 123–139. Springer, 2010.

[5] I. M. Asăvoae, M. Asăvoae, and D. Lucanu. Path directed symbolic execution in the k framework. T. Ida, V. Negru, T. Jebelean, D. Petcu, S. M. Watt, and D. Zaharie, editors, *12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2010, Timisoara, Romania, 23-26 September 2010*, pages 133–141. IEEE Computer Society, 2010.

[6] I. M. Asăvoae, F. de Boer, M. Bonsangue, D. Lucanu, and J. Rot. Bounded model checking of recursive programs with pointers in K. *WRLA 2012*, 2012. Accepted for presentation as work in progress at WRLA'12.

[7] I. M. Asăvoae, F. de Boer, M. Bonsangue, D. Lucanu, and J. Rot. Bounded model checking of recursive programs with pointers in K. *WADT 2012, Technical Report TR-08/12, Universidad Complutense de Madrid, Departamento de Sistemas Informáticos y Computación*, pages 12–15, 2012. Accepted for presentation at WADT'12, `http://maude.sip.ucm.es/wadt2012/docs/WADT2012-preproceedings.pdf`.

[8] I. M. Asăvoae, F. de Boer, M. Bonsangue, D. Lucanu, and J. Rot. Model checking programs with dynamic linked structures. Technical Report LIACS Technical Report 2012-02, LIACS, Universiteit Leiden, 2012.

[9] G. Barthe, G. Dufay, L. Jakubiec, B. P. Serpette, and S. M. de Sousa. A formal executable semantics of the javacard platform. D. Sands, editor, *Programming Languages and Systems, 10th European Symposium on Programming, ESOP 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2028 of *Lecture Notes in Computer Science*, pages 302–319. Springer, 2001.

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

[11] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. R. Cytron and R. Gupta, editors, *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, pages 196–207. ACM, 2003.

[12] I. Bogudlov, T. Lev-Ami, T. W. Reps, and M. Sagiv. Revamping tvla: Making parametric shape analysis competitive. W. Damm and H. Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 221–225. Springer, 2007.

[13] M. M. Bonsangue, G. Caltais, E.-I. Goriac, D. Lucanu, J. J. M. M. Rutten, and A. Silva. A decision procedure for bisimilarity of generalized regular expressions. J. Davies, L. Silva, and A. da Silva Simão, editors, *Formal Methods: Foundations and Applications - 13th Brazilian Symposium on Formal Methods, SBMF 2010, Natal, Brazil, November 8-11, 2010, Revised Selected Papers*, volume 6527 of *Lecture Notes in Computer Science*, pages 226–241. Springer, 2011.

[14] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. A. W. Mazurkiewicz and J. Winkowski, editors, *CONCUR '97: Concurrency Theory, 8th International Conference, Warsaw, Poland, July 1-4, 1997, Proceedings*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1997.

[15] A. Bouajjani, S. Fratani, and S. Qadeer. Context-bounded analysis of multithreaded programs with dynamic linked structures. W. Damm and H. Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 207–220. Springer, 2007.

[16] B.-Y. E. Chang, X. Rival, and G. C. Necula. Shape analysis with structural invariant checkers. H. R. Nielson and G. Filé, editors, *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24,*

2007, Proceedings, volume 4634 of *Lecture Notes in Computer Science*, pages 384–401. Springer, 2007.

[17] N. Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 1959.

[18] E. M. Clarke, E. A. Emerson, and J. Sifakis. Model checking: algorithmic verification and debugging. *Commun. ACM*, 52(11):74–84, 2009.

[19] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[20] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[21] P. Cousot. Abstract interpretation. `http://www.di.ens.fr/~cousot/AI/`.

[22] P. Cousot. Abstraction in abstract interpretation. `http://cs.nyu.edu/~pcousot/publications.www/slides/Cousot-Osaka-99-slides-4-1.pdf`, Nov. 1999.

[23] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1–2):47–103, 2002.

[24] P. Cousot. Abstract interpretation. MIT course 16.399, `http://web.mit.edu/16.399/www/`, Feb.–May 2005.

[25] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[26] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *J. Log. Program.*, 13(2&3):103–179, 1992.

[27] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. J. Launchbury and J. C. Mitchell, editors, *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 178–190. ACM, 2002.

[28] T. F. Şerbănuţă, A. Arusoaie, D. Lazar, C. Ellison, D. Lucanu, and G. Roşu. The K primer (version 2.5). M. Hills, editor, *K'11*, Electronic Notes in Theoretical Computer Science, to appear.

[29] T.-F. Şerbănuţă and G. Roşu. K-Maude: A rewriting based tool for semantics of programming languages. P. C. Ölveczky, editor, *Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010, Held as a Satellite Event*

of ETAPS 2010, Paphos, Cyprus, March 20-21, 2010, Revised Selected Papers, volume 6381 of *Lecture Notes in Computer Science*, pages 104–122. Springer, 2010.

[30] T. F. Şerbănuţă, G. Roşu, and A. Ştefănescu. An overview of K and matching logic. M. Hills, editor, *K'11*, Electronic Notes in Theoretical Computer Science, to appear.

[31] S. Das and D. L. Dill. Counter-example based predicate discovery in predicate abstraction. *Formal Methods in Computer-Aided Design*, volume 2517 of *LNCS*, pages 19–32. Springer-Verlag, 2002.

[32] F. S. de Boer, M. M. Bonsangue, and J. Rot. Automated verification of recursive programs with pointers. B. Gramlich, D. Miller, and U. Sattler, editors, *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, volume 7364 of *Lecture Notes in Computer Science*, pages 149–163. Springer, 2012.

[33] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. F. Gaducci and U. Montanari, editors, *Workshop on Rewriting Logic and Its Applications*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, September 2002.

[34] C. Ellison. *An Executable Formal Semantics of C with Applications*. PhD thesis, University of Illinois at Urbana-Champaign, July 2012. `http://fsl.cs.uiuc.edu/index.php/A_Formal_Semantics_of_C_with_Applications`.

[35] C. Ellison, T.-F. Şerbănuţă, and G. Roşu. A rewriting logic approach to type inference. A. Corradini and U. Montanari, editors, *Recent Trends in Algebraic Development Techniques, 19th International Workshop, WADT 2008, Pisa, Italy, June 13-16, 2008, Revised Selected Papers*, volume 5486 of *Lecture Notes in Computer Science*, pages 135–151. Springer, 2009.

[36] C. Ellison and G. Roşu. An executable formal semantics of c with applications. J. Field and M. Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 533–544. ACM, 2012.

[37] A. Farzan, F. Chen, J. Meseguer, and G. Roşu. Formal analysis of java programs in javafan. R. Alur and D. Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 501–505. Springer, 2004.

[38] R. W. Floyd. Toward interactive design of correct programs. *IFIP Congress (1)*, pages 7–10, 1971.

[39] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of programming languages (2. ed.)*. MIT Press, 2001.

[40] J. A. Goguen, K. Lin, and G. Roşu. Circular coinductive rewriting. *ASE*, pages 123–132, 2000.

[41] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. J. Edward G. Coffman, editor, *Journal of the ACM (JACM)*, volume 24, pages 68–95. ACM New York, 1977.

[42] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. *Proceedings of the 9th Conference on Computer-Aided Verification*, pages 72–83. Springer-Verlag, 1997.

[43] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, volume 74, pages 358–366. AMS, 1953.

[44] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. *SIGPLAN Notices*, 37(1):58–70, 2002.

[45] M. Hills. *A Modular Rewriting Approach to Language Design, Evolution, and Analysis*. PhD thesis, University of Illinois at Urbana-Champaign, 2009. `http://fsl.cs.uiuc.edu/~mhills/thesis/thesis.pdf`.

[46] M. Hills and G. Roşu. KOOL: An application of rewriting logic to language prototyping and analysis. *Proceedings of the 18th International Conference on Rewriting Techniques and Applications (RTA'07)*, volume 4533 of *LNCS*, pages 246–256. Springer, 2007.

[47] M. Hills and G. Roşu. A rewriting logic semantics approach to modular program analysis. C. Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 151–160, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[48] M. Hind. Pointer analysis: haven't we solved this problem yet? J. Field and G. Snelting, editors, *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'01, Snowbird, Utah, USA, June 18-19, 2001*, pages 54–61. ACM, 2001.

[49] M. Hind and A. Pioli. Which pointer analysis should i use? *ISSTA*, pages 113–123, 2000.

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

[51] M. Huth and M. D. Ryan. *Logic in computer science - modelling and reasoning about systems (2. ed.)*. Cambridge University Press, 2004.

[52] S. Jagannathan, P. Thiemann, S. Weeks, and A. K. Wright. Single and loving it: Must-alias analysis for higher-order languages. D. B. MacQueen and L. Cardelli, editors, *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, pages 329–341. ACM, 1998.

[53] B. Jeannet, A. Loginov, T. W. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. *ACM Trans. Program. Lang. Syst.*, 32(2), 2010.

[54] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice Hall international series in computer science. Prentice Hall, 1993.

[55] C. Kaner, J. Falk, and H. Q. Nguyen. *Testing computer software (2. ed.)*. Van Nostrand Reinhold, 1993.

[56] D. Kozen. Kleene algebra with tests. *ACM Trans. Program. Lang. Syst.*, 19(3):427–443, 1997.

[57] W. Landi. Undecidability of static analysis. *LOPLAS*, 1(4):323–337, 1992.

[58] D. Lucanu, T.-F. Şerbănuţă, and G. Roşu. K framework distilled. F. Duran, editor, *WRLA 2012*, volume ? of *Lecture Notes in Computer Science*, pages ?–?? Springer, 2012. To appear.

[59] P. Madhusudan and G. Parlato. The tree width of auxiliary storage. T. Ball and M. Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 283–294. ACM, 2011.

[60] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. *Electronic Notes in Theoretical Computer Science*, 4, 1996.

[61] K. L. McMillan. Applications of craig interpolants in model checking. N. Halbwachs and L. D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2005.

[62] K. L. McMillan. Lazy abstraction with interpolants. *Computer-Aided Verification*, pages 123–136, 2006.

[63] P. Meredith, M. Hills, and G. Roşu. An executable rewriting logic semantics of K-scheme. D. Dube, editor, *Proceedings of the 2007 Workshop on Scheme and Functional Programming (SCHEME'07), Technical Report DIUL-RT-0701*, pages 91–103. Laval University, 2007.

[64] P. O. Meredith, M. Katelman, J. Meseguer, and G. Roşu. A formal executable semantics of verilog. *8th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010), Grenoble, France, 26-28 July 2010*, pages 179–188. IEEE Computer Society, 2010.

[65] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[66] J. Meseguer. Twenty years of rewriting logic. P. C. Ölveczky, editor, *Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010, Held as a Satellite Event of ETAPS 2010, Paphos, Cyprus, March 20-21, 2010, Revised Selected Papers*, volume 6381 of *Lecture Notes in Computer Science*, pages 15–17. Springer, 2010.

[67] J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstraction. *Automated Deduction CADE-19*, volume 2741 of *Lecture Notes in Computer Science*, pages 2–16. Springer, 2003.

[68] J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. *Theor. Comput. Sci.*, 403(2-3):239–264, 2008.

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

[70] J. Meseguer and G. Roşu. The rewriting logic semantics project: A progress report. O. Owe, M. Steffen, and J. A. Telle, editors, *Fundamentals of Computation Theory - 18th International Symposium, FCT 2011, Oslo, Norway, August 22-25, 2011. Proceedings*, volume 6914 of *Lecture Notes in Computer Science*, pages 1–37. Springer, 2011.

[71] P. D. Mosses. Modular structural operational semantics. *J. Log. Algebr. Program.*, 60-61:195–228, 2004.

[72] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., 1999.

[73] S. Owens. A sound semantics for ocamllight. *In: Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Lecture Notes in Computer Science*, pages 1–15. Springer, 2008.

[74] M. Palomino. A predicate abstraction tool for Maude. Documentation and tool available at `http://maude.sip.ucm.es/~miguelpt/bibliography.html`.

[75] M. D. Preda, M. Christodorescu, S. Jha, and S. K. Debray. A semantics-based approach to malware detection. M. Hofmann and M. Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 377–388. ACM, 2007.

[76] J. F. Ranson, H. J. Hamilton, P. W. L. Fong, H. J. Hamilton, and P. W. L. Fong. A semantics of python in isabelle/hol, 2008.

[77] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002.

[78] N. Rinetzky, J. Bauer, T. W. Reps, S. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. J. Palsberg and M. Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 296–309. ACM, 2005.

[79] X. Rival and B.-Y. E. Chang. Calling context abstraction with shapes. T. Ball and M. Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 173–186. ACM, 2011.

[80] G. Roşu. K: A rewriting-based framework for computations – preliminary version. Technical Report Department of Computer Science UIUCDCS-R-2007-2926 and College of Engineering UILU-ENG-2007-1827, University of Illinois at Urbana-Champaign, 2007.

[81] G. Roşu and T. Şerbănuţă. An overview of the K semantic framework. *J. Log. Algebr. Program.*, 79(6):397–434, 2010.

[82] G. Roşu and A. Ştefănescu. From hoare logic to matching logic reachability. *Proceedings of the 18th International Symposium on Formal Methods (FM'12)*, Lecture Notes in Computer Science. Springer, 2012. To appear.

[83] G. Roşu and A. Ştefănescu. Matching logic: a new program verification approach. R. N. Taylor, H. Gall, and N. Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 868–871. ACM, 2011.

[84] G. Roşu and A. Ştefănescu. Checking reachability using matching logic. *Proceedings of the 27th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12)*. ACM, 2012. To appear.

[85] G. Roşu and A. Ştefănescu. Towards a unified theory of operational and axiomatic semantics. *Proceedings of the 39th International Colloquium on Automata, Languages and Programming (ICALP'12)*, volume 7392 of *Lecture Notes in Computer Science*, pages 351–363. Springer, 2012.

[86] G. Roşu, C. Ellison, and W. Schulte. Matching logic: An alternative to hoare/floyd logic. M. Johnson and D. Pavlovic, editors, *Algebraic Methodology and Software Technology - 13th International Conference, AMAST 2010,*

Lac-Beauport, QC, Canada, June 23-25, 2010. Revised Selected Papers, volume 6486 of *Lecture Notes in Computer Science*, pages 142–162. Springer, 2011.

[87] G. Roşu and D. Lucanu. Circular coinduction: A proof theoretical foundation. A. Kurz, M. Lenisa, and A. Tarlecki, editors, *Algebra and Coalgebra in Computer Science, Third International Conference, CALCO 2009, Udine, Italy, September 7-10, 2009. Proceedings*, volume 5728 of *Lecture Notes in Computer Science*, pages 127–144. Springer, 2009.

[88] G. Roşu and W. Schulte. Matching logic — extended report. Technical Report Department of Computer Science UIUCDCS-R-2009-3026, University of Illinois at Urbana-Champaign, January 2009.

[89] J. Rot. *A Pushdown System Representation for Unbounded Object Creation*. PhD thesis, Universiteit Leiden Opleiding Informatica, June 2010. `http://www.liacs.nl/assets/Masterscripties/10-06-JurriaanRot.pdf`.

[90] J. Rot, I. M. Asăvoae, F. de Boer, M. Bonsangue, and D. Lucanu. Interacting via the heap in the presence of recursion. *ICE 2012*, 2012. To appear.

[91] J. Rot, M. Bonsangue, and F. de Boer. A pushdown automaton for unbounded object creation. *FOVEOOS 2010*, 2010. Accepted for presentation as position paper/ work in progress at FOVEOOS.

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

[93] D. A. Schmidt. Internal and external logics of abstract interpretations. *Verification, Model Checking and Abstract Interpretation*, volume 4905 of *LNCS*, pages 263–278. Springer, 2008.

[94] D. A. Schmidt and B. Steffen. Program analysis *as* model checking of abstract interpretations. G. Levi, editor, *Static Analysis, 5th International Symposium, SAS '98, Pisa, Italy, September 14-16, 1998, Proceedings*, volume 1503 of *Lecture Notes in Computer Science*, pages 351–380. Springer, 1998.

[95] S. Schwoon. *Model Checking Pushdown Systems*. PhD thesis, Technische Universität München, June 2002. `http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/schwoon.pdf`.

[96] T. F. Şerbănuţă. *A Rewriting Approach to Concurrent Programming Language Design and Semantics*. PhD thesis, University of Illinois at Urbana-Champaign, December 2010. `https://www.ideals.illinois.edu/handle/2142/18252`.

[97] T. F. Şerbănuţă, G. Roşu, and J. Meseguer. A rewriting logic approach to operational semantics. *Information and Computation*, 207:305–340, 2009.

[98] D. Suwimonteerabuth, F. Berger, S. Schwoon, and J. Esparza. jmoped: A test environment for java programs. W. Damm and H. Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 164–167. Springer, 2007.

[99] Wikipedia. Abstract interpretation. `http://en.wikipedia.org/wiki/Abstract_interpretation`.

[100] Wikipedia. Formal verification. `http://en.wikipedia.org/wiki/Formal_verification`.

[101] Wikipedia. Model checking. `http://en.wikipedia.org/wiki/Model_checking`.

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

[103] WWW. Asml. `http://research.microsoft.com/en-us/projects/asml/`.

[104] WWW. Astrée. `http://www.astree.ens.fr/`.

[105] WWW. Coq. `http://coq.inria.fr/`.

[106] WWW. Crystal. `https://www.cs.cornell.edu/projects/crystal/`.

[107] WWW. Isabelle. `http://www.cl.cam.ac.uk/research/hvg/isabelle/`.

[108] WWW. Maude. `http://maude.cs.uiuc.edu/`.

[109] WWW. PVS. `http://pvs.csl.sri.com/`.

[110] WWW. SLAM. `http://research.microsoft.com/en-us/projects/slam/`.

[111] WWW. TVLA. `http://www.cs.tau.ac.il/~tvla/`.

[112] WWW. Xisa. `http://xisa.cs.colorado.edu/`.

[113] D. Yan, G. H. Xu, and A. Rountev. Demand-driven context-sensitive alias analysis for Java. M. B. Dwyer and F. Tip, editors, *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, pages 155–165. ACM, 2011.

[114] X. Zheng and R. Rugina. Demand-driven alias analysis for c. G. C. Necula and P. Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 197–208. ACM, 2008.