Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 304 (2014) 97–110

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

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

Irina Măriuca Asăvoae[3]

*Computer Science Faculty*
*University Alexandru Ioan Cuza*
*Iaşi, Romania*

**Abstract**

This paper presents an approach to integrating analysis and verification methods in the $\mathbb{K}$ framework. We adopt the abstract interpretation perspective where the concrete system to be analyzed/verified is mapped into a suitable abstract system, and collecting semantics is applied over the abstract system to obtain the analysis/verification method itself. As such, we present the $\mathbb{K}$ perspective of collecting semantics over $\mathbb{K}$ operational semantics for abstract systems. For a good degree of generality we consider that abstract systems are $\mathbb{K}$ specifications of (finite) pushdown systems. We give the collecting semantics as a generic set of $\mathbb{K}$ rules parametrized by the $\mathbb{K}$ specification of a finite pushdown system. Further, we describe a case study which instances collecting semantics with alias analysis. For this, the abstract system is defined as an imperative language which maintains enough pointer and flow information for alias analysis to be decidable. The $\mathbb{K}$ specification of this imperative language fits the frame of a finite pushdown system specification.

*Keywords:* abstraction, collecting semantics, pushdown systems, alias analysis

## 1 Introduction

The spark of the $\mathbb{K}$ framework [14] is the observation that computation is expressed naturally with rewriting. The source of inspiration for $\mathbb{K}$ is the Rewriting Logic Semantics project [9,19,10] which has the declared purpose of unifying algebraic denotational semantics and operational semantics. This unification is achieved by considering the two semantics as different views over the same object. Namely, denotational semantics views the rewriting logic specification of a language as a designated model, while operational semantics focuses on the execution 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. $\mathbb{K}$ definitions can be mechanically translated into rewriting logic, and in particular

[1] Contract ANCS POS CCE, O2.1.2, ID nr. 602/12516, ctr.nr 161/15.06.2010, DAK.

[2] 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]

[3] Email: mariuca.asavoae@info.uaic.ro

http://dx.doi.org/10.1016/j.entcs.2014.05.005

into Maude, to obtain program analysis tools or interpreters based on term rewriting. This capability makes $\mathbb{K}$ an executable framework, with $\mathbb{K}$-Maude its prototype implementation [15,18].

A $\mathbb{K}$ definition is an executable specification of a transition system whose computations are obtained by the execution of the $\mathbb{K}$ definition. Moreover, one can also reuse a $\mathbb{K}$ definition to enable richer executions as, for example, sets of computations. When producing the plain computations, $\mathbb{K}$ can be seen as an interpreter while, when producing sets of computations, $\mathbb{K}$ can also be used as an analyzer/verifier for the specified transition system. This is the idea of the current paper in a nutshell and we frame it under the methodology proposed by abstract interpretation.

In abstract interpretation, a particular analysis/verification method is achieved by defining collecting semantics over the examined transition system [4,5]. Namely, the transition system is first transformed into a simpler "abstract" one such that the operational semantics of the two systems "agree" on the analyzed/verified set of properties. Then, collecting semantics relies on the operational semantics of the abstract transition system and collects its computations via a forward or backward fixpoint iteration. Hence, the analysis/verification methods are a semantic reflection of the operational semantics.

### *1.1 Contributions summary*

In this paper we present an infrastructure for expressing in $\mathbb{K}$ the reflection of operational semantics into collecting semantics, and the alias analysis instantiation of this reflection. The cornerstone of this infrastructure is the choice of pushdown systems as suitable $\mathbb{K}$ definitions. The semantics reflection is nicely captured in $\mathbb{K}$ by the configuration abstraction mechanism and definitional modularity. The choice of pushdown systems as focal point for this study is justified by the generality of the notion, the already available theoretical results, and the close resemblance with $\mathbb{K}$ 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 $\mathbb{K}$ rules usually rely on a pushed down stack mechanism.

In Section 2 we present an infrastructure for the $\mathbb{K}$ specification of analysis/verification methods for pushdown systems. In more detail, in Section 2.1 we present a discussion on the $\mathbb{K}$ representation of pushdown systems. We use the $\mathbb{K}$ specification of a pushdown system as support for deriving the analysis/verification infrastructure in Section 2.2. We also argue the opportunity to consider pushdown systems and their $\mathbb{K}$ specification in Section 2.3.

In Section 3 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 [17,16] with several bisimilar semantics. In Section 3.1 we present the $\mathbb{K}$ specification of one of these semantics. In particular, this semantics exhibits algorithmic details which emphasize the versatility of $\mathbb{K}$ in the area of algorithm formulation.

- This particular semantics of *SILK* has the useful feature of producing a finite reachable state space. A benefit brought by using this semantics is decidability of the analysis/verification methods. In Section 3.2 we present alias analysis as an instantiation of the general method for analysis/verification from Section 2.2.

The current work started with the task of giving a faithful $\mathbb{K}$ specification of the "on paper" semantics provided in [17,16], $\mathbb{K}$ specification which we call *SILK* (Section 3.1). The authors of [17,16] give the semantics as a pushdown system specification. Consequently, a methodological view of the work on *SILK* via pushdown systems comes in naturally (Section 2.1). Moreover, due to its continuation-based feature, $\mathbb{K}$ proves to be a suitable specification environment for pushdown systems (Section 2.3). The view of *SILK* as an abstraction for alias analysis is the actual novelty in this work. We present this as methodology for pushdown systems, in Section 2.2, and as instantiation of the methodology, in Section 3.2.

### *1.2 Related work*

The core of the current work is the semantics study given in [17,16] and the ideas promoted in the discussions with the authors of [17,16,2]. We take this opportunity to extend our thanks for the enriching collaboration on [2] where *SILK* is extended with fields and the programs are verified, using the Maude LTL Model Checker [6], via bounded model checking for a regular language of heap properties. Note that the mere syntactic extension involves a drastic change in the semantics. Hence *SILK* is semantically different from Shylock, the abstract language presented in [2].

The program analysis aspects approached here are a refinement of the idea presented in [1]. Earlier work on program analysis in $\mathbb{K}$ is presented in [8] 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 [11,12,13,20]. The main difference in the current approach is that we propose an abstract semantics which is decoupled from the actual code, in the style of abstract interpretation.

The champion semantics for the $\mathbb{K}$ framework is the specification of C described in [7]. 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 [6]. In this context, regarding alias analysis, we bring as witness the work in [21] where C is abstracted into a context free, flow insensitive language of equalities and alias analysis is mapped into reachability. We provide a similar approach in *SILK*, the only difference being that the alias analysis is interprocedural and flow sensitive.

## 2 Foundations

A $\mathbb{K}$ definition specifies some class of transition systems by means of a configuration and rules applied over the configuration. However, when reasoning about these transition systems for the purpose of analysis/verification, it is often the case that

one needs to restrict this class to *finite* transition systems. In Section 2.1 we propose a $\mathbb{K}$ perspective over a restricted class of potentially infinite transition systems, given by finite pushdown systems. The existence of decidability results on reasoning about finite pushdown systems is a further incentive for their integration in $\mathbb{K}$. We exploit these results in Section 2.2. In Section 2.3 we emphasize that $\mathbb{K}$ definitional style induces a natural resemblance between $\mathbb{K}$ specifications of languages and pushdown systems.

### *2.1 $\mathbb{K}$-specification of the pushdown systems*

We present next a general technique for specifying pushdown systems in $\mathbb{K}$, by means of abstraction. First we establish a frame for the $\mathbb{K}$ specification of finite pushdown systems. Then we describe how to transform an infinite pushdown system into a $\mathbb{K}$ specification of a finite pushdown system. This transformation is given as an abstraction.

Recall that a pushdown system is a quadruple $\mathcal{P} = (\Delta, \Sigma, \hookrightarrow, c_0)$ where $\Delta$ is a set of *control locations*, $\Sigma$ is a *stack alphabet*, and $\hookrightarrow$ is a subset of $(\Delta \times \Sigma) \times (\Delta \times \Sigma^*)$ representing the set of *rules*. A *configuration* is a pair $(\delta, \Gamma)$ where $\delta \in \Delta$ and $\Gamma \in \Sigma^*$. The set of all configurations is denoted as $Conf(\mathcal{P})$ and $c_0 \in Conf(\mathcal{P})$ is the initial configuration. A pushdown system is said to be *finite* when the sets $\Delta, \Sigma, \hookrightarrow$ are finite.

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

In order to define a pushdown system $\mathcal{P}$ in $\mathbb{K}$ we first assume we have an algebraic representation for $\Delta$ and $\Sigma$. The $\mathbb{K}$ configuration describes the structure of $Conf(\mathcal{P})$ as a nested bag of labeled cells, namely $\langle\langle\Delta\rangle_{\mathsf{ctrl}}\langle K\rangle_{\mathsf{k}}\rangle_{\mathcal{P}}$. The continuation-based feature of $\mathbb{K}$ is introduced by the special cell $\langle\rangle_{\mathsf{k}}$ which contains a list of computation tasks of a special sort $K$. In the $\mathbb{K}$ definition for $\mathcal{P}$, we specify that $\Sigma$ is a subsort of $K$ by the $\mathbb{K}$ syntactic command $K ::= \Sigma$. When $\Gamma = \gamma_1..\gamma_n \in \Sigma^*$ 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 computation tasks.

The rules of a finite pushdown system become $\mathbb{K}$ computational rules as follows:

for any $(\delta, \gamma) \hookrightarrow (\delta', \Gamma)$ we have the $\mathbb{K}$ rule $\langle \underset{\delta'}{\delta} \rangle_{\mathsf{ctrl}} \langle \underset{\Gamma}{\gamma} \cdots \rangle_{\mathsf{k}}$

A few 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 that cell. For example, the ellipses in $\langle\rangle_{\mathsf{k}}$ make $\gamma$ the top of the continuation stack while the rest of the stack is encoded by $\cdots$. Note that in the above rule, $\delta$ represents the whole content of the $\mathsf{ctrl}$ cell. The local rewrites (specified with a bi-dimentional notation) trigger the local changes made to the configuration. As such, the rewrite in the $\mathsf{k}$ cell represents the "pop" of $\gamma$ followed

by the "push" of $\Gamma$ in the stack. The equivalent rewrite rule (available also in the $\mathbb{K}$ notation) is $\langle\delta\rangle_{\mathsf{ctrl}}\,\langle\gamma \curvearrowright \Gamma'\rangle_{\mathsf{k}} \Rightarrow \langle\delta'\rangle_{\mathsf{ctrl}}\,\langle\Gamma \curvearrowright \Gamma'\rangle_{\mathsf{k}}$ i.e., the corresponding subset of transitions in the transition system $\mathcal{T}_{\mathcal{P}}$.

The transition relation $\hookrightarrow$ is specified as a finite set of $\mathbb{K}$ rules. For defining the $\mathbb{K}$ rules, we identify finite sets of patterns $\breve{\Delta} \subseteq 2^{\Delta}, \breve{\Sigma} \subseteq 2^{\Sigma}$. Being in a rewriting environment, these patterns are naturally described using variables. We denote these patterns as $\breve{\delta} \in \breve{\Delta}$, primed or indexed, $\breve{\gamma} \in \breve{\Sigma}$, primed or indexed. Also $\breve{\Gamma} = \gamma_0 \breve{\cdot} \ldots \breve{\cdot} \gamma_n$, $n \in \mathbb{N}$, has the property $\gamma_0 \breve{\cdot} \ldots \breve{\cdot} \gamma_n = \breve{\gamma}_0 \breve{\cdot} \ldots \breve{\cdot} \breve{\gamma}_n$.

The $\mathbb{K}$ rules for the relation $\hookrightarrow$ of an infinite pushdown system are defined such that:

for any $(\delta, \gamma) \hookrightarrow (\delta', \Gamma)$ there is a rule $\langle \frac{\breve{\delta}}{\breve{\delta}'} \rangle_{\mathsf{ctrl}} \langle \frac{\breve{\gamma}}{\breve{\Gamma}} \cdots \rangle_{\mathsf{k}}$ such that

$\delta$ matches $\breve{\delta}$ and $\gamma$ matches $\breve{\gamma}$, $\delta'$ matches $\breve{\delta}'$ and $\Gamma$ matches $\breve{\Gamma}$

We associate to each of the patterns in $\breve{\Delta}$ and $\breve{\Sigma}$ a unique constant, and we denote $k\mathcal{P}$ the $\mathbb{K}$ specification of the finite pushdown system given by the configuration $\langle\langle\breve{\Delta}\rangle_{\mathsf{ctrl}}\langle\breve{\Sigma}\rangle_{\mathsf{k}}\rangle_{\mathcal{P}}$ and the above rules. Note that $k\mathcal{P}$ is an abstraction of $\mathcal{P}$. The coarsest choice of such an abstraction is given by the finite sets of patterns $\breve{\Delta} = \{\emptyset, \Delta\}$ and $\breve{\Sigma} = \{\emptyset, \Sigma\}$.

### *2.2 $\mathbb{K}$-specification of the analysis/verification for pushdown systems*

We describe a generic approach for the analysis/verification problem $\mathcal{P} \models \varphi$, where $\varphi$ is a property of interest for $\mathcal{P}$. We use the setting founded in abstract interpretation where $\mathcal{T} \models \varphi$ is defined as *collecting semantics* [4], with $\mathcal{T}$ a transition system. Namely, the "concrete" system $\mathcal{T}$ is soundly transformed into an "abstract" system $\mathcal{A}$ and the property $\varphi$ is verified by *collecting* the executions of $\mathcal{A}$. Likewise, we aim to execute exhaustively $k\mathcal{P}$ and to collect information of interest along this execution.

Note that we can infer $\mathcal{P} \models \varphi$ from the collecting execution of $k\mathcal{P}$ only if $k\mathcal{P}$ is a sound abstraction of $\mathcal{P}$ wrt the property of interest $\varphi$. However, this aspect is outside the scope of the current paper.

Based on the results in [3], if there exists an infinite path in a finite pushdown system, 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 head) as $c_0 \xrightarrow{*} (\delta, \gamma Y) \xrightarrow{w} (\delta, \gamma X^r Y)$, where $X^r$ is a new term from $\Sigma^*$ such that $(\delta, \gamma X^r Y) = (\delta, \gamma XY)$.

**Example 2.1** We consider $\mathcal{P}_0 = (\{x, y\}, \{a, b, c\}, \hookrightarrow_0, (x, abc))$ a pushdown system,

where the relation $\hookrightarrow_0$ is defined by the set:

$$\{(x,a) \hookrightarrow_0 (y,a), (y,a) \hookrightarrow_0 (x,bca), (x,b) \hookrightarrow_0 (x,\epsilon), (x,c) \hookrightarrow_0 (y,\epsilon)\}$$

The only computation in $\mathcal{P}_0$ is:

$$(x,abc) \xrightarrow{a} (y,abc) \xrightarrow{a} (x,bcaabc) \xrightarrow{b} (x,caabc) \xrightarrow{c}$$
$$(y,aabc) \xrightarrow{a} (x,bcaaabc) \xrightarrow{b} (x,caaabc) \xrightarrow{c} (y,aaabc) \dots$$

Hence, we identify the lasso for $\delta$ as $y$, $\gamma$ as $a$, $X$ as $a$, $Y$ as $bc$, and $w$ as $abc$. This computation can be characterized by its finite prefix:

$$(x,abc) \xrightarrow{a} (y,abc) \xrightarrow{a} (x,bcaabc) \xrightarrow{b} (x,caabc) \xrightarrow{c} (y,aa^rbc)$$

We aim to develop a technique for reusing the $\mathbb{K}$-definition of a pushdown system to obtain the collecting semantics, using a forward fixpoint iteration. The configuration in collecting semantics for $k\mathcal{P}$, denoted $\breve{k\mathcal{P}}$, is defined as:

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

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 contains the current execution level, while in traces′ we construct the next level. The breadth-first strategy is tantamount to imposing fairness in the application of rewrite rules. As usual, we need some fairness condition to ensure the monotonicity of the fixpoint iteration.

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

for each $\breve{\gamma} \in \breve{\Sigma}$, $\breve{\delta} \in \breve{\Delta}$, consider all the rules $\langle \underbrace{\breve{\delta}}_{\breve{\delta}_i} \rangle_{\mathsf{ctrl}} \langle \underbrace{\breve{\gamma}}_{\breve{\Gamma}_i} \; \cdots \rangle_{\mathsf{k}}$, $0 \leq i \leq n$, where

$n \in \mathbb{N}$ is the nondeterminism factor of $\langle \breve{\delta} \rangle_{\mathsf{ctrl}} \langle \breve{\gamma} \; \cdots \rangle_{\mathsf{k}}$.

We denote by $\mathtt{post}(\langle \breve{\delta} \rangle_{\mathsf{ctrl}} \langle \breve{\gamma} \curvearrowright \breve{\Gamma}' \rangle_{\mathsf{k}})$ the set $\{\langle \breve{\delta}_i \rangle_{\mathsf{ctrl}} \langle \breve{\gamma}_i \curvearrowright \breve{\Gamma}' \rangle_{\mathsf{k}} \mid 0 \leq i \leq n\}$. We use the `post` operator in the $\mathbb{K}$ rules from Fig. 1, where the relation $\ll \subseteq Bag \times Bag$ is well-founded over the contents of the collect cell. Note that $\mathtt{post}(\langle \breve{\delta} \rangle_{\mathsf{ctrl}} \; \langle \breve{\gamma} \rangle_{\mathsf{k}})$ can be obtained with a search command as "`search` : $\langle \langle \breve{\delta} \rangle_{\mathsf{ctrl}} \; \langle \breve{\gamma} \rangle_{\mathsf{k}} \rangle_{\mathcal{P}}$ `=>1` $\langle \langle D : \breve{\Delta} \rangle_{\mathsf{ctrl}} \; \langle K : K \rangle_{\mathsf{k}} \rangle_{\mathcal{P}}$".

The collecting rules in Fig. 1 simulate a shared memory model, where the cell collect is the shared memory. The content of collect cell is customized to maintain the desired outcome of the analysis/verification method. The first rule encodes the exhaustive step of execution of $k\mathcal{P}$ by consuming a current computation trace from traces and producing new computation traces in traces′ cell, provided that this step increases the contents of collect cell. The second collecting rule 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 content of the collect cell, because its structure and update operation depend on the targeted

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

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

$$\text{when}\quad Bag \ll \mathtt{update}(\ Bag, \langle\breve{\delta}\rangle_{\mathsf{ctrl}}\ \langle\breve{\gamma}\curvearrowright\breve{\Gamma}'\rangle_{\mathsf{k}}\ )$$

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

$$\text{when}\quad Bag \not\ll \mathtt{update}(\ Bag, \langle\breve{\delta}\rangle_{\mathsf{ctrl}}\ \langle\breve{\gamma}\curvearrowright\breve{\Gamma}'\rangle_{\mathsf{k}}\ )$$

Fig. 1. The rules for the $\mathbb{K}$ specification of the collecting semantics over $k\mathcal{P}$.

analysis/verification method.

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

$$\text{RULE}\quad \langle \underbrace{\cdot}_{\mathtt{bag2set}(\mathit{NextLevel})} \rangle_{\mathsf{traces}} \langle \underbrace{\mathit{NextLevel}}_{\cdot}\rangle_{\mathsf{traces}'} \quad \text{when } \mathit{NextLevel} \neq \cdot$$

where the operator `bag2set` eliminates duplicated elements from *NextLevel*.

The termination of the "exhaustive" execution of $\mathcal{P}$ designed in $\breve{k\mathcal{P}}$ is ensured by the well-foundedness of the relation $\ll$ 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 collect is basically formed by various representations or abstractions of the computations prefixes displayed by the exhaustive breadth-first execution of $k\mathcal{P}$. 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 the loop of a lasso.

With this design for the content of the collect cell, the relation $\ll$ 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 $k\mathcal{P}$ are either finite, or reduced to finite prefixes.

### *2.3 Relating pushdown systems and programming language semantics*

We now argue 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 instanced for some program. At this point pushdown systems show theoretical relevance, when 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 of programming languages defined in $\mathbb{K}$.

Consider the $\mathbb{K}$ specification $\mathcal{S}$ for the semantics of a language and a program $P$ in this programming language. According to the methodology provided in [20] for designing $\mathcal{S}$ , 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 $\mathcal{S}$ is tantamount to the specification of pushdown systems produced by the syntactic part of $\mathcal{S}$ as stack language and all the cells, besides k, as control location. Note, however, that there are restrictions as, for example, the matching in the k cell has to be made always at the top.

The finiteness of the pushdown system produced for a program $P$ with a specification $\mathcal{S}$ is, nonetheless, worth discussing from the point of view of the analysis/verification methods. 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.

Moreover, [5] shows that the abstraction meta-operator induces a transformation on the syntactic elements as well (i.e., the stack language). This transformation is quite natural, taking into account that some syntactic elements target parts of the control location which are abstracted away. For example, say that some branch conditions depend on current values of some variables but the abstraction eliminates the actual values of the variables. Then the branch condition is usually abstracted into nondeterministic choice and its semantics is replaced with the semantics of the abstract syntactic element. Hence, an abstraction meta-operator has to transform, as well, the rules in $\mathcal{S}$ into equivalent (abstract) rules from the abstract semantics.

## 3 A simple imperative language with object creation

In this section we describe *SILK* - the $\mathbb{K}$ specification of a simple block-structured programming language which supports object creation, global variables, static scoping and recursive procedures with local variables. The language is introduced and studied in [17,16] as a pushdown system specification. Here we present a faithful $\mathbb{K}$ representation of the *SILK* syntax, in Fig. 2, and semantics, in Section 3.1. This semantics is called "abstract" in [17,16] and is proved to produce *finite* pushdown systems. Hence we can apply collecting semantics over this abstract semantics. In Section 3.2 we exemplify collecting semantics with alias analysis. Note that we choose alias analysis because the control state of *SILK* focuses on maintaining the so called "object identities equivalence classes", i.e., alias classes.

A *SILK* program consists of a finite set of procedures acting on some global and

$Pgm$ ::= `gvars:` $Ids$ `lvars:` $Ids$ { $Procs$ }
$Ids$ ::= $List${$\#Id$,"," }
$Procs$ ::= $ProcId$ `::` $B$ | $Procs$ $Procs$
$ProcId$ ::= $\#Id$
$VExp$ ::= $\#Id$
$BExp$ ::= $\#Id$ `=` $\#Id$ | $\#Id$ `/=` $\#Id$
$B$ ::= $VExp$ `:=` $VExp$ | $VExp$ `:= new` | $B$ `;` $B$ | `[` $BExp$ `]` $B$ | $B$ `+` $B$ | $ProcId$
$IntBot$ ::= $\#Int$ | $\bot$

Fig. 2. The *SILK* syntax

local set of variables. *SILK* is statically scoped. Upon a procedure's call, the body of the procedure is executed with the same global variables and a fresh instantiation of the local variables. Upon a procedure's return, the changes to the global variables are preserved, while the local variables from the procedure's call point are restored. The assignment statement $x$ `:=` $y$ assigns the identity stored in $y$ (if any) to $x$, while 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 $B_2$.

### *3.1 SILK abstract semantics*

In this section we describe the $\mathbb{K}$ specification for the *SILK* abstract semantics, i.e., the semantics which defines finite pushdown systems for alias analysis.

For decidability reasons, the collecting semantics has to work with a finite state model. In particular, the abstract semantics presented in [16,17] uses a memory allocation protocol with *abstract* memory addresses. As such, the *SILK* state associates to each pointer variable some *abstract memory address* from the set $\{\bot\} \cup 1..2|V_g| + |V_l|$, where $|V_g|$ and $|V_l|$ represent the number of global and local variables, respectively, and $\bot$ is associated to undefined objects. Consequently, the state space of *SILK* programs is finite because the programs have a finite number of pointer variables, either global or local.

The abstraction modifies the statements concerning the memory allocation, namely object creation, procedure's 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. 3. As such, the abstract semantics for each statement is specified in two stages: `ping` and `ped`. The `ping` stage is implemented by a structural rule which pushes in the kAbs cell the p*rocess*`ing` of the next abstract state. The `ped` stage recognizes the fact of having received the next p*rocess*`ed` 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

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

[structural]

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

[transition]

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

[structural]

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

[transition]

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

[structural]

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

[transition]

Fig. 3. The `ping-ped` abstraction mechanism.

abstraction is carried on an enhanced signature $\Sigma \cup \Sigma'$, where $\Sigma$ is the signature of the specification for the "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 $\Sigma'$.

The procedure's 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 [16,17] as an iterative algorithm which reads as:

- let $\sigma$ be the current state and $\sigma'$ be the state from the procedure's 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

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

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 $\neg_{Bool}\ K\ \texttt{in}\ S\ \texttt{[}\ G\ \ GGs\ \texttt{-}_{Set}\ Gi\ \ Gs\ \texttt{]}$

[step3. structural]

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

$\langle \underset{Si\ \texttt{[ nextFreeValue(}\ S\ \texttt{(}\ Gi\ \texttt{) , | values}\ S\ \texttt{| , }\ S\ \texttt{) / }\ Gi\ \texttt{]}}{Si} \rangle_{\mathsf{sigmai}}$ when

$\neg_{Bool}\ K\ \texttt{in}\ S\ \texttt{[ frzSet(}\ GGs\ \texttt{)]} \wedge_{Bool} \neg_{Bool}\ K\ \texttt{in}\ S\ \texttt{[}\ GGs\ \texttt{-}_{Set}\ Gi\ \ Gs\ \texttt{]}$
$\wedge_{Bool}\ K \neq_{Bool} \bot$

[steps4-5. structural]

Fig. 4. The ping-ped structural rules for the abstract procedure's return statement.

4. if in $\sigma_{i-1}$ all indices except $\bot$ 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 $\sigma_{i-1}$.

In Fig. 4 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 $\sigma, \sigma'$ and $\sigma_i$, respectively. The iteration is maintained in 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 global variables, and the freeze variables $g'$ are represented as $\texttt{frz(}\ g\ \texttt{)}$.

Several considerations regarding the algorithm in Fig. 4: 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.

### *3.2 Alias analysis for SILK*

Having the abstract semantics for *SILK*, we can present the alias analysis as an instantiation of the collecting semantics introduced in Section 2.2. Namely, we describe the content of the collect cell and the relation $\ll$.

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 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 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 any more 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 associative matching as follows:

- $\ll$ is false, i.e., the currently considered computation trace $\langle\langle P \curvearrowright X \curvearrowright X \curvearrowright Y\rangle_{\mathsf{k}}\langle S\rangle_{\mathsf{state}}\rangle_{\mathsf{trace}}$ 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. 1 is instanced as:

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

- $\ll$ is true when the computation trace is not a repetition of any of the previously collected heads. Hence, the first rule in Fig. 1 reads as:

  RULE $\dfrac{\langle\langle P \curvearrowright K\rangle_{\mathsf{k}}\langle S\rangle_{\mathsf{state}}\rangle_{\mathsf{trace}}}{\cdot}\ \dfrac{\cdot}{\langle \mathtt{post(}\ \langle P \curvearrowright K\rangle_{\mathsf{k}}\langle S\rangle_{\mathsf{state}}\ \mathtt{)}\rangle_{\mathsf{trace}}}$

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

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

Note that, for efficiency reasons, we apply the $\not\in^{\mathtt{rep}}$ 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.

**Example 3.1** We discuss alias analysis results for the *SILK* program in Fig. 5. Due to space restrictions we do not present in detail how the computation for alias analysis evolves. Instead, we enlist only the first few steps and explain the reasoning

used by may and must alias over the final result.

```
gvars: x lvars: y { main ::  y := new; (x := y + x := new); flag; main; y := x }
```

Fig. 5. A simple *SILK* program.

$$
\begin{array}{ll}
 & \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 \texttt{y} := \texttt{new} \curvearrowright (\texttt{x} := \texttt{y} + \texttt{x} := \texttt{new}); \texttt{flag}; \texttt{main}; \texttt{y} := \texttt{x} \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}}\ \ \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}}\ \ \ldots \\
\Rrightarrow & \langle\langle(\texttt{x} := \texttt{y} + \texttt{x} := \texttt{new}) \curvearrowright B \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}}\ \ \ldots\ \ \text{where } B \text{ is } \texttt{flag}; \texttt{main}; \texttt{y} := \texttt{x} \\
\Rrightarrow & \langle\langle \texttt{x} := \texttt{new} \curvearrowright B \curvearrowright \texttt{restore(frz(x)}\mapsto\bot\ \texttt{x}\mapsto\bot\ \texttt{y}\mapsto\bot\texttt{)}\rangle_{\mathsf{k}}\ \ \langle \texttt{frz(x)}\mapsto\bot\ \texttt{x}\mapsto\bot\ \texttt{y}\mapsto 0\rangle_{\mathsf{state}}\rangle_{\mathsf{trace}} \\
 & \langle\langle \texttt{x} := \texttt{y} \curvearrowright B \curvearrowright \texttt{restore(frz(x)}\mapsto\bot\ \texttt{x}\mapsto\bot\ \texttt{y}\mapsto\bot\texttt{)}\rangle_{\mathsf{k}}\ \ \langle \texttt{frz(x)}\mapsto\bot\ \texttt{x}\mapsto\bot\ \texttt{y}\mapsto 0\rangle_{\mathsf{state}}\rangle_{\mathsf{trace}}\ \ldots \\
\overset{*}{\Rrightarrow} & \langle\langle \texttt{flag} \curvearrowright \texttt{main}; \texttt{y} := \texttt{x} \curvearrowright \texttt{restore(frz(x)}\mapsto\bot\ \texttt{x}\mapsto\bot\ \texttt{y}\mapsto\bot\texttt{)}\rangle_{\mathsf{k}}\langle \texttt{frz(x)}\mapsto\bot\ \texttt{x}\mapsto 1\ \texttt{y}\mapsto 0\rangle_{\mathsf{state}}\rangle_{\mathsf{trace}} \\
 & \langle\langle \texttt{flag} \curvearrowright \texttt{main}; \texttt{y} := \texttt{x} \curvearrowright \texttt{restore(frz(x)}\mapsto\bot\ \texttt{x}\mapsto\bot\ \texttt{y}\mapsto\bot\texttt{)}\rangle_{\mathsf{k}}\langle \texttt{frz(x)}\mapsto\bot\ \texttt{x}\mapsto 0\ \texttt{y}\mapsto 0\rangle_{\mathsf{state}}\rangle_{\mathsf{trace}} \\
\overset{*}{\Rrightarrow} & \langle\langle \texttt{main} \curvearrowright \texttt{y} := \texttt{x} \curvearrowright \texttt{restore(frz(x)}\mapsto\bot\ \texttt{x}\mapsto\bot\ \texttt{y}\mapsto\bot\texttt{)}\rangle_{\mathsf{k}}\ \ \langle \texttt{frz(x)}\mapsto\bot\ \texttt{x}\mapsto 1\ \texttt{y}\mapsto 0\rangle_{\mathsf{state}}\rangle_{\mathsf{trace}} \\
 & \langle\langle \texttt{main} \curvearrowright \texttt{y} := \texttt{x} \curvearrowright \texttt{restore(frz(x)}\mapsto\bot\ \texttt{x}\mapsto\bot\ \texttt{y}\mapsto\bot\texttt{)}\rangle_{\mathsf{k}}\ \ \langle \texttt{frz(x)}\mapsto\bot\ \texttt{x}\mapsto 0\ \texttt{y}\mapsto 0\rangle_{\mathsf{state}}\rangle_{\mathsf{trace}} \\
 & \langle\langle \texttt{frz(x)}\mapsto\bot\ \texttt{x}\mapsto 0\ \texttt{y}\mapsto 0\rangle_{\mathsf{state}}\langle \texttt{frz(x)}\mapsto\bot\ \texttt{x}\mapsto 1\ \texttt{y}\mapsto 0\rangle_{\mathsf{state}}\rangle_{\mathsf{aliases}}\ \ \ldots\ \ \overset{*}{\Rrightarrow}\ldots
\end{array}
$$

Note that in this example we use `flag` 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 flags for each program point.

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 $\texttt{x} := \texttt{new}$ 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 $\langle \texttt{frz(x)}\mapsto a\ \texttt{x}\mapsto b\ \texttt{y}\mapsto c\rangle_{\mathsf{state}}$, while $i$ is the watermark). A query for must alias $\texttt{x} \overset{?}{=} \texttt{y}$ 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.

## 4 Conclusions and Future Work

In this paper we propose a technique for defining analysis and verification methods over $\mathbb{K}$ specifications of abstract semantics. In short, we apply collecting semantics over $\mathbb{K}$ definitions given as finite pushdown systems. We instance this technique with alias analysis for an abstract semantics of *SILK*.

We plan to apply the $\mathbb{K}$ collecting semantics technique to model checking the abstract semantics of *Shylock*, an extension of *SILK* with object fields. Furthermore, we plan to study and standardize a $\mathbb{K}$ methodology for defining abstractions à la abstract interpretation over "concrete" semantics, i.e., already defined semantics for "real" programming languages.

In conclusion, we would like to extend our thanks to the team of coauthors from [2] for creating the premises of this work, to the $\mathbb{K}$ framework and matching logic team for posing the challenge and the means not only for this work but many others, and last but not least to the anonymous reviewers for the detailed and much appreciated feedback.

# References


[1] Asăvoae, I. M., and Asăvoae, M., *Collecting Semantics under Predicate Abstraction in the K Framework,* Proc. WRLA, LNCS, **6381** (2010), 123–139.

[2] Asăvoae, I. M., de Boer, F., Bonsangue, M., Lucanu, D., and Rot, J., *Bounded Model Checking of Recursive Programs with Pointers in K,* WRLA (2012), Accepted for presentation.

[3] Bouajjani, A., Esparza, J., and Maler, O., *Reachability Analysis of Pushdown Automata: Application to Model Checking,* Proc. CONCUR, LNCS , **1243**, (1997), 135–150.

[4] Cousot, P., *Abstract Interpretation,* MIT course 16.399, (2005).

[5] Cousot, P., and Cousot, R., *Abstract Interpretation and Application to Logic Programs,* Journal of Logic Programming, **13(2–3)** (1992), 103–179.

[6] Eker, S., Meseguer, J., and Sridharanarayanan, A., *The Maude LTL Model Checker,* Proc. WRLA, ENTCS, **71** (2002), 162–187.

[7] Ellison, C., and Roşu, G., *An Executable Formal Semantics of C with Applications,* Proc. POPL, ACM, (2012), 533–544.

[8] Hills, M., and Roşu, G., *A Rewriting Logic Semantics Approach to Modular Program Analysis,* Proc. RTA, Schloss Dagstuhl, (2010), 151–160.

[9] Meseguer, J., and Roşu, G., *The Rewriting Logic Semantics Project,* Theoretical Computer Science **373** (2007), 213–237.

[10] Meseguer, J., and Roşu, G., *The Rewriting Logic Semantics Project: a Progress Report,* FCT, (2011), 1–37.

[11] Roşu, G., Ellison, C., and Schulte, W., *Matching Logic: an Alternative Approach to Hoare/Floyd Logic,* AMAST, (2010), 142–162.

[12] Roşu, G., and Ştefănescu, A., *Matching Logic Rewriting: Unifying Operational and Axiomatic Semantics in a Practical and Generic Framework,* Technical Report http://hdl.handle.net/2142/28357, University of Illinois, (2011).

[13] Roşu, G., and Ştefănescu, A., *Matching Logic: A New Program Verification Approach,* NIER ICSE'11, (2011), 868–871.

[14] Roşu, G., and Şerbănuţă, T. F., *An Overview of the K Semantic Framework,* Journal of Logic and Algebraic Programming, **79(6)** (2010), 397–434.

[15] Roşu, G., and Şerbănuţă, T. F., *K-Maude: A Rewriting Based Tool for Semantics of Programming Languages,* Proc. WRLA, LNCS, **6381** (2010), 104–122.

[16] Rot, J., *A Pushdown System Representation for Unbounded Object Creation,* Internal Report http://www.liacs.nl/assets/Masterscripties/10-06-JurriaanRot.pdf, B.Sc. Thesis, Leiden University, (2010).

[17] Rot, J., Bonsangue, M., and de Boer, F., *A Pushdown Automaton for Unbounded Object Creation,* FOVEOOS, (2010), Accepted for presentation.

[18] Şerbănuţă, T. F., Arusoaie, A., Lazăr, D., Ellison, C., Lucanu, D., and Roşu, G., *The K primer (version 2.5),* In the current volume.

[19] Şerbănuţă, T. F., Roşu, G., and Meseguer, J., *A Rewriting Approach to Operational Semantics,* Inf. Comput., **207(2)** (2009), 305–340.

[20] Şerbănuţă, T. F., Roşu, G., and Ştefănescu, A., *An Overview of K and Matching Logic,* In the current volume.

[21] Zheng, X., and Rugină, R., *Demand-driven Alias Analysis for C,* POPL, (2008), 197–208.