PhD Thesis, University of Illinois, December 2010

A REWRITING APPROACH TO CONCURRENT PROGRAMMING LANGUAGE DESIGN AND SEMANTICS

BY

TRAIAN FLORIN SERBANUTA

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Doctoral Committee:

Associate Professor Grigore Roşu, Chair and Director of Research
Principal Researcher Thomas Ball, Microsoft Research
Assistant Professor Darko Marinov
Professor José Meseguer
Assistant Professor Madhusudan Parthasarathy

Abstract

A plethora of programming languages have been and continue to be developed to keep pace with hardware advancements and the ever more demanding requirements of software development. As these increasingly sophisticated languages need to be well understood by both programmers and implementors, precise specifications are increasingly required. Moreover, the safety and adequacy with respect to requirements of programs written in these languages needs to be tested, analyzed, and, if possible, proved. This dissertation proposes a rigorous approach to define programming languages based on rewriting, which allows to easily design and test language extensions, and to specify and analyze safety and adequacy of program executions.

To this aim, this dissertation describes the $\mathbb{K}$ framework, an executable semantic framework inspired from rewriting logic but specialized and optimized for programming languages. The $\mathbb{K}$ framework consists of three components: (1) a language definitional technique; (2) a specialized notation; and (3) a resource-sharing concurrent rewriting semantics. The language definitional technique is a rewriting technique built upon the lessons learned from capturing and studying existing operational semantics frameworks within rewriting logic, and upon attempts to combine their strengths while avoiding their limitations. The specialized notation makes the technical details of the technique transparent to the language designer, and enhances modularity, by allowing the designer to specify the minimal context needed for a semantic rule. Finally, the resource-sharing concurrent semantics relies on the particular form of the semantic rules to enhance concurrency, by allowing overlapping rule instances (e.g., two threads writing in different locations in the store, which overlap on the store entity) to apply concurrently as long as they only overlap on the parts they do not change.

The main contributions of the dissertation are: (1) a uniform recasting of the major existing operational semantics techniques within rewriting logic; (2) an overview description of the $\mathbb{K}$ framework and how it can be used to define, extend and analyze programming languages; (3) a semantics for $\mathbb{K}$ concurrent rewriting obtained through an embedding in graph rewriting; and (4) a description of the K-Maude tool, a tool for defining programming languages using the $\mathbb{K}$ technique on top of the Maude rewriting language.

προς δόξαν Θϵού

Acknowledgments

Concerning my pre-PhD formation, I would like to thank my grandfather Gheorghe and my mid- and high-school mathematics professor Cornel Noană for instilling and sustaining my love for mathematics, to my mother Olivia and my brother Virgil for instilling and sustaining my love for programming, and finally to my College and Masters professors Virgil Emil Căzănescu, Alexandru Mateescu, and Răzvan Diaconescu, for showing me that computer science is more than programming, and for guiding my first research steps.

My entire PhD research, including the results presented in this dissertation would not have been possible without my advisor Grigore Roşu. I would like to especially thank him for providing countless research ideas, as well as help and guidance through the darkness of “nothing works”; his inspiration and his readiness in working together for overcoming any research obstacle have proved instrumental in my formation. In particular, his introduction of $\mathbb{K}$ in 2003 as a rewriting technique for defining programming languages, and the fact that he continuously used $\mathbb{K}$ in teaching programming language design at UIUC ever since, played a very important role in the development of the $\mathbb{K}$ framework, the K-Maude tool, and of this dissertation itself.

I would also like to thank the rest of my research committee, made up of Darko Marinov, José Meseguer, Madhusudan Parthasarathy, and Tom Ball, for providing valuable feedback which helped in presenting my research in a better light and strengthen the value of this dissertation, and for their insightful suggestions for potential future work related to the presented research; in particular, I would like to thank José Meseguer for his vision over the rewriting logic semantics project and for his direct collaboration in developing the initial stages of this dissertation.

My research life at UIUC would have been a lot more stern without all my previous and current colleagues within the Formal Systems and Formal Methods and Declarative Languages groups, among which Marcelo d'Amorim, Feng Chen, Chucky Ellison, Michael Ilseman, Mark Hills, Dongyun Jin, Mike Katelman, Choonghwan Lee, Patrick Meredith, Andrei Popescu, Ralf Sasse, and Andrei Ştefănescu were always ready to offer a good advice, to argue about everything, or simply to help procrastinating before deadlines. Thank you guys!

A special thanks is due to my family from both sides of the ocean, including, but not limited to my parents Mircea and Olivia, my brother Virgil and his wife Gabriela, and to my parents-in-law Ioan and Aurelia, for always loving me and trusting in my potential; in particular, to my wife Claudia, for going through all this process together, for growing together, celebrating together, and also suffering together at times, and to my girls, Cezara Maria and Teodora, for bringing light on dark days and constantly reminding me that there are far more important things in life than research.

Thanks is also due to my non-research related friends from this “new world”, among which Bogdan, Cristi, Francisco, Matei, Nick, and Thyago, for helping me face the “cultural shock”, and for being good friends, whether in need or in celebrating the happy moments of our lives.

Finally, I would like to thank the entire Three Hierarchs Greek Orthodox Church community, for being a home away from home for us, and in particular to Fr. George Pyle, and Fr. Michael Condos, for their continuous spiritual support and counsel during all this time.

The research in this dissertation has been supported in part by NSF grants CCF-0916893, CNS-0720512, CNS-0509321, CCF-0448501, by NASA contract NNL08AA23C, by a Samsung SAIT grant, and by several Microsoft gifts.

Table of Contents

Chapter 1

Introduction

This dissertation shows that rewriting is a natural environment to formally define the semantics of real-life concurrent programming languages and to test and analyze programs written in those languages.

1.1 Motivation

Computers are becoming an integral part of our life. Besides the increase in personal computer usage, there are now programable devices embedded in all aspects of modern life, from consumer electronics to life critical systems such as transportation and medical devices. Moreover, the internet and the information age has brought a whole world of previously remote services at only a click distance.

All these services, be they provided by embedded systems or by personal or remote computers, share a common characteristic: they result from programming their host using specialized languages. These languages can vary to a high degree in their level of abstraction from machine description languages such as Verilog and VHDL, whose instructions are very close to the hardware circuits they describe, to high level imperative, object oriented and/or functional languages, such as Java, C#, or OCaml. To address the continuous increase in difficulty of software specifications, these language have to evolve and new domain specific languages and methodologies must be designed and deployed.

At the same time, paradigm changes in hardware design must be accompanied by similar changes in language design. Moore's law [119] states that the number of transistors on an integrated circuit chip doubles about every two years. However, while this trend is expected to continue for the next 10-20 years (until its physical limits are attained), the implication that doubling the number of transistors doubles serial computing power no longer holds true. The move to multi-core architectures is therefore a must: doubling the number of cores on a chip may actually still double performance, as long as there is enough parallelism available.

To be able to take advantage of multi-core hardware, a paradigm shift is needed in programming language design, requiring that concurrency with sharing of resources becomes the norm rather than the exception. As an evidence of this trend, the design of popular programming languages is currently being revised to enhance their support for concurrency. In the case of Java, for example, the standard libraries were enhanced with advanced support for synchronization and a new Java memory model [97] was proposed to allow more realistic implementations of the JVM under current hardware architectures. At the same time, the new (in-progress) C1X standard [86] for the C language explicitly includes multithreading and advanced synchronization primitives in its library specification.

However, the need to make languages flexible enough to take advantage of concurrency, while at the same time allowing for compiler optimizations, requires allowing under-specification and non-determinism in the language. This poses significant challenges for testing and proving that specifications or implementations of languages conform with their intended semantics. Indeed, the complexity and the lack of adequate tools and techniques for throughly analyzing newly proposed definitions often leads to subtle errors which take time to be discovered. For example, the proposed specification for the x86-CC memory model [151] was shown unsound w.r.t. the actual hardware behavior [128], and the new Java memory model was shown unsound for some of the compiler optimizations it was supposed to allow [163].

Whatever an (existing or future) programming language might be, it requires a *specification* defining the language so that users and implementors can agree on the meaning of programs written in that language. This usually amounts to describing the language syntax, i.e., the constructs allowed by the language, and its semantics, i.e., the intended behavior of the language constructs. However, while there seems to be a consensus that syntax needs to be formally specified, there is more than one way in which semantics of languages are specified, ranging from using natural language (usually in the form of a reference manual), to specifying an interpreter (i.e., a standard implementation) or a compiler for the language into another language, to formally describing the language in a mathematical language.

Each of the approaches above has its own advantages. For example, natural language descriptions are more accessible, allowing virtually anyone to read the specification and to believe they understood its meaning. Interpreters and compilers allow one to gain knowledge about the language by testing it and developing usage patterns. Finally, a formal description of languages allows one to state and prove precise statements about the language and its programs.

However, each of the described approaches has non-negligible limitations. The natural language descriptions are often incomplete and sometimes even contradictory; to obtain consensus on their actual meaning requires lawyer skills in addition to computer scientist skills. Interpreters and compilers require familiarity with the target language and pose coverage problems, as corner cases are usually harder to capture in testing; moreover, any extensions to the language, or the development of analysis tools usually requires a major, if not complete rewrite of the interpreter/compiler. Formal descriptions are confined to the power of expression of their specification language. As specification languages often focus on making common programming language features simpler to express, they often make it hard, or even impossible, to satisfactorily capture advanced programming language features such as control change and multi-threaded synchronization. Moreover, each formal specification language seems to be appropriate to a single analysis area: SOS for dynamic semantics, natural semantics for type systems, reduction semantics with evaluation contexts for proving type soundness, axiomatic semantics for program verification, and so on.

The aim of our research is to show that one can keep the balance between simplicity and the power of expressivity and analysis without compromising the formal aspect. This dissertation shows how executable formal definitions can help with addressing the issues formulated above, first by providing a mathematical environment to design programming languages, then by allowing one to easily experiment with changes to those definitions by effectively testing them, and finally, by allowing to explore and abstract nondeterministic executions for analysis purposes.

1.2 Summary of Contributions

The research described in this dissertation makes the following key contributions:

1. Corroborates existing evidence, and brings further evidence that rewriting in general, and rewriting logic in particular, is a suitable environment for semantically defining programing languages and for executing, exploring, and analyzing the executions of programs written in those languages. Specifically, rewriting logic is re-affirmed as a powerful meta-logical framework for programming languages, able to capture existing language definitional frameworks with minimal representational distance, and offering them executability and powerful tool support.

2. Advances the rewriting-based $\mathbb{K}$ framework as a candidate for an ideal language definitional framework by exemplifying its power of abstraction and expression in conjunction with its natural support for concurrency, its modularity, flexibility, and simplicity.

3. Shows that $\mathbb{K}$ definitions can be easily turned into runtime verification tools which can be used to check properties such as memory safety and datarace freeness, or to test and explore behaviors of relaxed memory models such as the x86-TSO [128].

4. Endows $\mathbb{K}$ with a true concurrency with resource sharing semantics which allows to capture within the framework the intended granularity for multi-threaded concurrency, in addition to the parallelism already available through the deduction rules of rewriting logic.

5. Describes the theory and implementation of K-Maude, a tool mechanizing a representation of $\mathbb{K}$ in rewriting logic, which brings $\mathbb{K}$ definitions to life,

and which has been used to give semantics to real programming languages and to derive analysis tools out of programming language definitions.

1.3 Rewriting Logic Semantics

The research presented in this dissertation is part of the more general rewriting logic semantics project [109, 110, 111], an international collaborative project whose goal is to advance the use of rewriting logic for defining programming languages and for analyzing programs written in them.

Rewriting is an intuitive and simple mathematical paradigm which specifies the evolution of a system by matching and replacing parts of the system state according to *rewrite rules*. Besides being formal, it is also executable, by simply repeating the process of rewriting the state. Additionally, an initial state together with a set of rules defines both an execution of the system (if remembering only one state at a time), or a transition system which can be explored and model checked for safety properties. A formal and executable definition of a language brings an additional potential benefit: it can be used to (semi-)automate the verification of programs written in that language by using direct or symbolic execution or simulation of the program's execution through the definition of the language to discharge (some of) the proof obligations.

Rewriting logic [103] defines a formal logic for rewriting which, in addition to rules, allows equations to define equivalences between the states of the system. The benefits of using rewriting logic in defining the behavior of systems are multiple. First, because rewriting logic is rooted in rewriting, one directly gains executability, and thus the ability of directly using definitions as interpreters. Second, the concurrency in rewriting is the norm rather than the exception, which allows capturing the intended granularity of concurrency directly in the definition, rather than relying on subsequent abstractions. Furthermore, by encoding the deterministic rules of a rewrite system as equations through equational abstractions [112], the state-space of transition systems is drastically collapsed, thus making its exploration approachable, and even effective, and opening the door for system analysis tools derived *directly* from definitions. The Maude rewrite system [34] offers a suite of powerful tools for rewrite theories, comprising, among others, an execution debugger, and the possibility of tracing executions, state-space exploration, an explicit-state LTL model checker, and an inductive theorem prover. For example, model checking Java programs in Maude using a definition of Java, following the $\mathbb{K}$ technique presented in this dissertation, was shown to compare favorably [57] with Java PathFinder, the state-of-art explicit-state model checker for Java [72].

Here are some generic guidelines to be followed when defining the semantics of a programming language within rewriting logic. First, one needs to represent the state of a running program as a configuration term. Next, the execution is described using rewrite rules and equations: equations are used to express structural changes and irrelevant execution steps, while rewrite rules are used to express relevant computational steps, which become transitions between states. With this, the execution becomes a sequence of transitions between equivalence classes of states, and the state-space of executions becomes a transition system which can be explored and model checked.

While these guidelines are good and were shown to be applicable in practice for defining a number of paradigmatic languages like Petri-nets, Actors, CCS, and the π-calculus [103, 104, 174], when approaching complex programming languages one needs more specific methodologies to organize and direct this process.

Chapter 3 re-affirms rewriting logic as an "ecumenical" logic for representing operational programming language definitional styles, by showing in a uniform way that most relevant existing techniques and styles, namely big-step (or natural) semantics [89], SOS [136], Modular SOS [123], reduction semantics (with evaluation contexts) [180], continuation-based semantics [60], and the Cham [15], can all be captured in rewriting logic with relatively zero-representational distance.

An important consequence of this embedding is that it allows programming language designers to get the best of both worlds: they can use their favorite programming language definitional framework notation and technique in designing a language, and then, through the corresponding representation of the definition as a rewrite theory, these definitions can be executed, explored and analyzed.

Being able to execute and analyze programs using their formal definitions, and thus adding a quite practical dimension to the relevance of a language definition, it is natural to ask the question

> Can we use existing techniques to define and analyze *real* programming languages?

By expressing all the above specified frameworks together in rewriting logic this question can be asked in a unified way. Unfortunately, each of the definitional frameworks mentioned above yields a negative answer to the question, although their combined strength would seem able to answer it positively. Varying from total lack of support to full support, on average, existing frameworks have problems with: defining control-intensive features (except for evaluation contexts), being modular when new features are introduced (except for Modular SOS), and in giving a semantics where concurrency can be captured directly by the transition system, that is, true concurrency (except for the Cham), in opposition with obtaining an interleaving semantics and then obtaining the real concurrency as something external (through equivalences), or even almost complete lack of support for concurrency as in big-step semantics which can only capture the set of all possible results of computation. There are also some less important concerns, such as the need to write rules in the framework to specify the order of evaluation (except for the evaluation context approach) instead of focusing on the actual semantics of the construct, or regarding the efficiency of

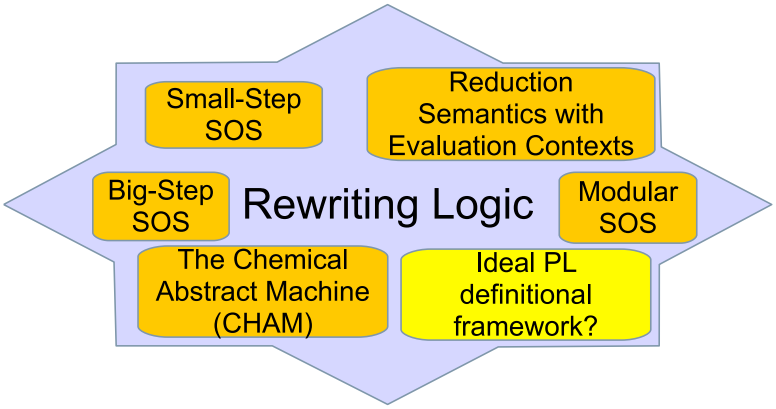

Small-Step SOS
Reduction Semantics with Evaluation Contexts
Big-Step SOS
Rewriting Logic
Modular SOS
The Chemical Abstract Machine (CHAM)
Ideal PL definitional framework?

Figure 1.1: Rewriting logic as a meta-logical semantic framework for defining programming languages

using the definition as interpreters in RWL, for which only big-step interpreters and continuation-based interpreters seem to give positive answers.

Nevertheless, the power of rewriting logic to capture all these definitional styles suggests that it might also be amenable for combining all their strengths. The remainder of the dissertation was motivated by the goal of finding "the ideal" programming language definitional framework based on rewriting logic which, based on the findings mentioned above, should be:

- At least as expressive as reduction semantics with evaluation contexts;
- At least as modular as modular SOS; and
- At least as concurrent as the Cham.

This dissertation shows that the $\mathbb{K}$ framework is a strong candidate for such an ideal programming language definitional framework.

1.4 The $\mathbb{K}$ Framework

Introduced by Roșu [142] in 2003 for teaching a programming languages class, and continuously refined and developed ever since, the $\mathbb{K}$ framework is a programming language definitional framework based on rewriting logic which brings together the strengths of existing frameworks (expressivity, modularity, concurrency, and simplicity) while avoiding their weaknesses. The $\mathbb{K}$ framework consists of the $\mathbb{K}$ technique, which can be, and has already been used to define real-life programming languages, such as Java, C, and Scheme, and program analysis tools (see Section 1.7 and the references there), and $\mathbb{K}$ rewriting, a rewriting semantics which allows more concurrency for $\mathbb{K}$ definitions than their direct representations as rewrite theories would allow. Nevertheless, the $\mathbb{K}$ framework is itself representable in rewriting logic, and this representation has been automated in the K-Maude tool for execution, testing and analysis purposes.

Figure 1.2: The full definition of the KernelC language: left—annotated syntax and desugaring rules; middle and right—initial configuration and semantic rules.

In a nutshell, the $\mathbb{K}$ framework relies on computations, configurations, and $\mathbb{K}$ rules in giving semantics to programming language constructs. *Computations*, which gave the name of the framework, are lists of tasks, including syntax; they have the role of capturing the sequential fragment of programming languages. *Configurations* of running programs are represented in $\mathbb{K}$ as bags of nested cells, with a great potential for concurrency and modularity. $\mathbb{K}$ *rules* distinguish themselves by specifying only what is needed from a configuration, and by clearly identifying what changes, and thus, being more concise, more modular, and more concurrent than regular rewrite rules.

To exemplify the $\mathbb{K}$ framework we will use here parts of the $\mathbb{K}$ definition of KernelC, which is completely defined and used in Chapter 5 for analyzing (concurrent) C programs for properties such as memory safety or datarace and deadlock freeness. KernelC defines a nontrivial subset of the C language containing functions, memory allocation, pointers and pointer arithmetic, and input/output primitives. It is expressive enough to be able to write C functions as the one below (which can be used for copying zero-terminated arrays):

```
void arrCpy(int * a, int * b) {
  while (* a ++ = * b ++) {}
}
```

Without modifying anything but the configuration, the language is extended with the following concurrency constructs: thread creation, lock-based synchronization and thread join.

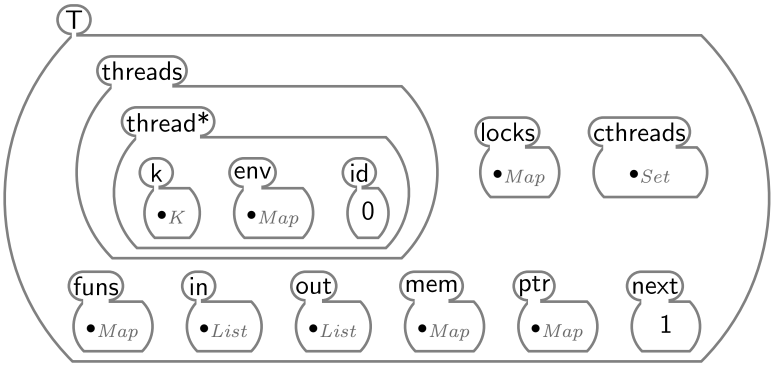


Figure 1.3: The initial configuration for executing KernelC programs.

Figure 1.2 presents the complete definition (scaled down to fit within the page size) of KernelC, including concurrency. Without expecting the reader to actually read the definition as is (we will return to parts of it here, and it will presented in detail in Chapter 5), note that the left column contains the BNF syntax and some syntax desugaring functions, while the middle and the right columns contain the description of the initial configuration for running KernelC programs, and the semantic rules. One can notice that the semantics takes no more than twice as much space as the syntax; moreover, since the graphical form of the $\mathbb{K}$ rules makes them expand vertically (thus usually taking two lines), we actually have about one rule for each language construct, with the exception of constructs like the conditional which requires two.

Configurations. The initial running configuration of KernelC is presented in Figure 1.3. The configuration is a nested multiset of labeled cells, in which each cell can contain either a list, a set, a bag, or a computation. The initial KernelC configuration consists of a top cell, labeled “T”, holding a bag of cells, among which a map cell, labeled “mem”, to map locations to values, a list cell, labeled “in”, to hold input values, and a bag cell, labeled “threads”, which can hold any number of “thread” cells (signaled by the star “$*$” attached to the label of the cell). The thread cell is itself a bag of cells, among which the “k” cell holds a computation structure, which plays the role of directing the execution.

Syntax and Computations. Computations extend syntax with a task sequentialization operation, “$\curvearrowright$”. The basic unit of computation is a task, which can either be a fragment of syntax, maybe with holes in it, or a semantic task, such as the recovery of an environment. Most of the manipulation of the computation is abstracted away from the language designer via intuitive PL syntax annotations like strictness constraints which, when declaring the syntax of a construct also specify the order of evaluation for its arguments. Similar decompositions of computations happen in abstract machines by means of stacks [59, 60], and also in the refocusing techniques for implementing reduction semantics with evaluation contexts [41]. However, what is different here is that $\mathbb{K}$ achieves the same thing *formally*, by means of rules (there are heating/cooling rules behind the strictness annotations, as explained below), not as an implementation means, which is what the others do.

The $\mathbb{K}$ BNF syntax specified below suffices to parse the program fragment `t = * x;  * x = * y;  * y = t;` specifying a sequence of statements for swapping the values at two locations in the memory syntax:

$$\begin{array}{rll}
\mathit{Exp} ::= & \mathit{Id} & \\
 & \mid \texttt{*}\ \mathit{Exp} & [\text{strict}] \\
 & \mid \mathit{Exp}\ \texttt{=}\ \mathit{Exp} & [\text{strict}(2)] \\
\mathit{Stmt} ::= & \mathit{Exp}\ \texttt{;} & [\text{strict}] \\
 & \mid \mathit{Stmt}\ \ \mathit{Stmt} & [\text{seqstrict}]
\end{array}$$

The strictness annotations add semantic information to the syntax by specifying the order of evaluation of arguments for each construct. The heating/cooling rules automatically generated for the strictness annotations above are:

$$\begin{array}{rcl}
\texttt{*}\ \mathit{ERed} & \rightleftharpoons & \mathit{ERed} \curvearrowright \texttt{*}\ \square \\
\mathit{E}\ \texttt{=}\ \mathit{ERed} & \rightleftharpoons & \mathit{ERed} \curvearrowright \mathit{E}\ \texttt{=}\ \square \\
\mathit{ERed}\ \texttt{;} & \rightleftharpoons & \mathit{ERed} \curvearrowright \square\ \texttt{;} \\
\mathit{SRed}\ \ \mathit{S} & \rightleftharpoons & \mathit{SRed} \curvearrowright \square\ \ \mathit{S} \\
\mathit{Val}\ \ \mathit{SRed} & \rightleftharpoons & \mathit{SRed} \curvearrowright \mathit{Val}\ \ \square
\end{array}$$

The heating/cooling rules specify that the arguments mentioned in the strictness constraint can be taken out for evaluation at any time and plug back in their original context. Note that the statement composition generates two such rules (as, by default, strictness applies to each argument); however, since the constraint specifies *sequential strictness*, the second statement can be evaluated only once the first statement was completely evaluated (specified by the *Val* variable which should match a value) and its side effects were propagated.

By successively applying these heating rules (from bottom towards top) on the statement sequence above we obtain the following computations:

$$\begin{array}{r}
\texttt{t = * x; * x = * y; * y = t;} \rightharpoonup \\
\texttt{t = * x;} \curvearrowright \square\ \texttt{* x = * y; * y = t;} \rightharpoonup \\
\texttt{t = * x} \curvearrowright \square\texttt{;} \curvearrowright \square\ \texttt{* x = * y; * y = t;} \rightharpoonup \\
\texttt{* x} \curvearrowright \texttt{t =}\,\square \curvearrowright \square\texttt{;} \curvearrowright \square\ \texttt{* x = * y; * y = t;} \rightharpoonup \\
\texttt{x} \curvearrowright \texttt{*}\,\square \curvearrowright \texttt{t =}\,\square \curvearrowright \square\texttt{;} \curvearrowright \square\ \texttt{* x = * y; * y = t;}
\end{array}$$

To begin, because statement composition is declared sequentially strict, the left statement must be evaluated first. The strictness rule will pull the statement out for evaluation, and leave a hole in its place. Now an expression statement construct is at the top and, being strict, it requires that the assignment be puled out. Next, the assignment construct being strict in the second argument, its right hand side must be pulled out for evaluation. Finally, the dereferencing construct is strict, and the heating rule will pull out the identifier x. Thus, through the strictness rules, we have obtained the order of evaluation as a sequence of tasks.

$\mathbb{K}$ **rules.** Consider the following "swap" function for swapping the values at the locations pointed to by the arguments:

```
void swap(int * x, int * y){
  int t = * x;  * x = * y;  * y = t;
}
```

Assume we are in the middle of a call to "swap" with arguments "a" and "b" (which are mapped to memory locations 4 and 5, respectively), and assume that all statements except the last one have already been executed, and that in that statement variable y has already been evaluated to location 5. The top configuration in Figure 1.4 represents a running configuration corresponding to this situation. By the strictness rules, we know that the next task to be executed is evaluating t.

Figure 1.4 shows how the $\mathbb{K}$ rule for reading the value of a local variable from the environment can be derived directly from the running configuration having the evaluation of a local variable as the next task to be executed. First, through a process named *cell comprehension* we focus only on the parts of the cells which are relevant for this rule. At the same time, we can declare our intention to replace t by its value in the environment (which is 1) by underlining the part that needs to change and writing its replacement under the line, through what we call *in-place rewriting*. Finally, through the process of *configuration abstraction*, only the relevant cells are kept, and through *generalization* we replace the concrete instances of identifiers and constants with variables of corresponding types. Jagged edges are used to specify that there could be more items in the cell (in the corresponding side) in addition to what is explicitly specified.

The thus obtained $\mathbb{K}$ rule succinctly describes the intuitive semantics for reading a local variable: if a local variable X is the next thing to be evaluated and if X is mapped to a value V in the environment, then replace that occurence of X by V. Moreover, it does that by only specifying what is needed from the configuration, which is essential for obtaining modular definitions, and by precisely identifying what changes, which significantly enhances the concurrency.

Modularity. As mentioned above, the configuration abstraction process is instrumental in achieving the desired modularity goal for the $\mathbb{K}$ framework. Relying on the initial configuration being specified by the designer, and the fact that usually the structure of such a configuration does not change during the execution of a program, the $\mathbb{K}$ rules are essentially invariant under change of structure. This effectively means that the same rule can be re-used in different

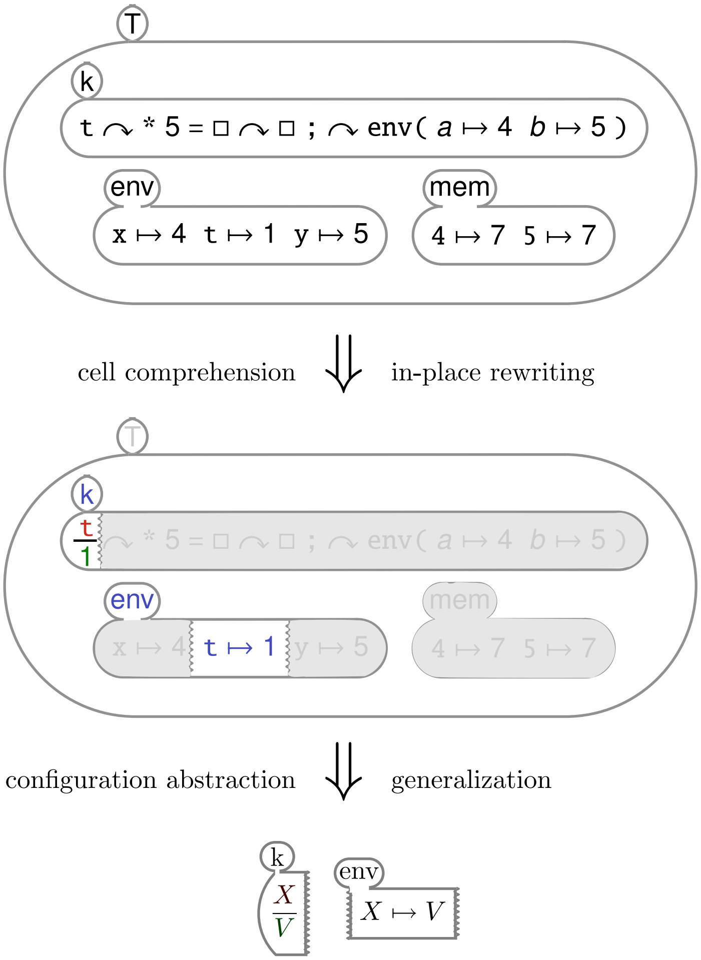
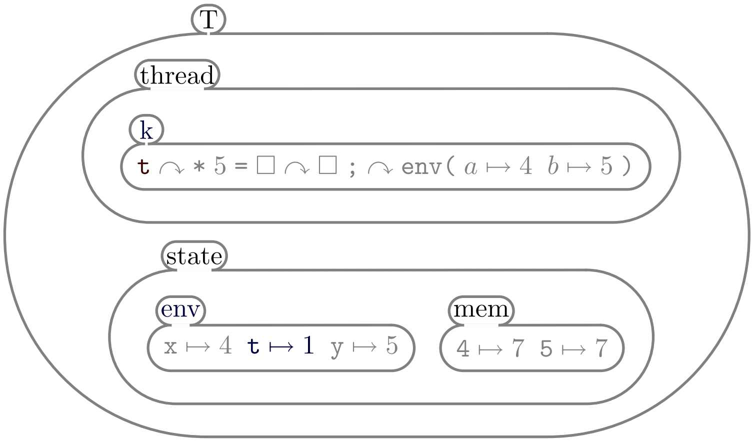


Figure 1.4: From running configurations to $\mathbb{K}$ rules

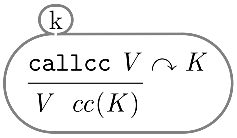
definitions as long as the required cells are present, regardless of the additional context, which can be automatically inferred from the initial configuration. As an example, the same 𝕂 rule for reading the value of a local variable can be used with a configuration as the one specified above, with the full KernelC configuration, and even with a configuration in which the computation and the environment cells are in separate parts of the configuration like the one presented below:



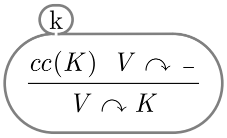
Expressivity. The particular structure of the 𝕂 computations, including the fact that the current task is always at the top of the computation, greatly enhances the expressivity of the 𝕂 framework. The next paragraphs exemplify the easiness of using 𝕂 to define constructs which are known to be hard to define using other frameworks, thus arguing that the 𝕂 framework reached the expressivity goal for an ideal definitional framework.

One first such example is the control intensive call with current continuation (`call/cc`), which is present in several functional languages (like Scheme), and to some extent in some imperative programming languages (such as the `longjmp` construct in C). Call/cc is known to be hard to capture in most frameworks (except for reduction semantics with evaluation contexts) due to their lack of access to the execution context, which is being captured at a meta-level by the logical context, and is thus not observable in the framework. Having the entire remainder of computation always following the current redex allows 𝕂 to capture this construct in a simple and succinct manner by the following two rules:

Passing computation as value:

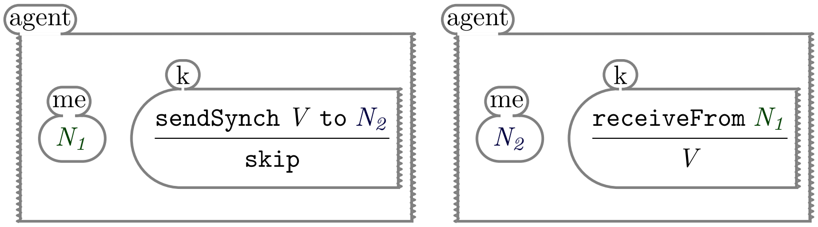


Applying computation as function:



The first rule wraps the current remainder of the computation as a value and passes it to the argument of "`callcc`", which is supposed to evaluate to a function. If during the evaluation of that function call the continuation value is applied to some value, then the remainder of the computation at that time is replaced by the saved computation to which the passed value is prefixed (as the result of the original `callcc` expression).

Another feature which is hard to represent in other frameworks is handling multiple tasks at the same time, as when defining synchronous communication, for example. Although SOS-based frameworks can capture specific versions of this feature for languages like CCS or the π-caculus, they can only do it there because the communication commands are always at the top of their processes. $\mathbb{K}$ computation's structure is again instrumental here, as it allows to easily match two redexes at the same time, as shown by the following rule, defining synchronous message passing in a multi-agent environment:



Reading the rule one can easily get the intended semantics: if one agent, identified by N_1 attempts to send a message to an agent identified by N_2, and if that agent is waiting to receive a message from N_1, then the send construct is dissolved in the first agent, while the receive expression is replaced by the value being sent within the receiver agent.

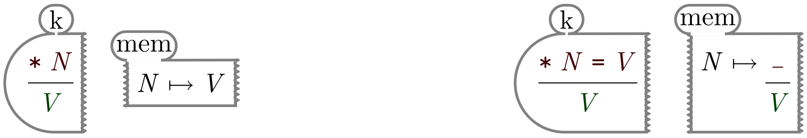
We cannot conclude the survey on the expressivity of the $\mathbb{K}$ framework without mentioning its reflective capabilities. Based on the fact that $\mathbb{K}$ regards all computational tasks as being abstract syntax trees, all language constructs become labels applied to other ASTs; for example the expression $a+3$ is represented in $\mathbb{K}$ as $\_+\_(a(\bullet_{List\{K\}}) \,\mathbf{,}\, 3(\bullet_{List\{K\}}))$. This abstract view of syntax allows reducing the computation-creating constructs to the following minimal core:

$$\begin{aligned}&K ::= KLabel(List\{K\}) \quad \mid \bullet_K \quad \mid K \curvearrowright K\\ &List\{K\} ::= K \quad \mid \bullet_{List\{K\}} \quad \mid List\{K\} \,\mathbf{,}\, List\{K\}\end{aligned}$$

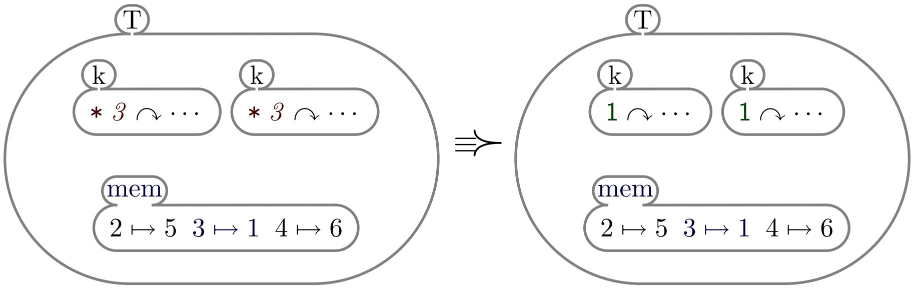
This approach allows a great power of reflection through generic AST manipulation rules. We won't go into details here, but in Section 4.3.7 we show how one can use $\mathbb{K}$ to define a generic AST visitor pattern and then how to use that to define powerful reflective features such as code generation or a binder-independent substitution.

Concurrency. Another feature that makes $\mathbb{K}$ appropriate for defining programming languages is its natural way of capturing concurrency. Besides being truely concurrent as the chemical abstract machine, and thus matching the goals set above for an ideal framework, 𝕂 rules also allow capturing concurrency with sharing of resources.

Let us exemplify this concurrency power. The two rules specified below are the KernelC rules for accessing/updating the value at a memory location:

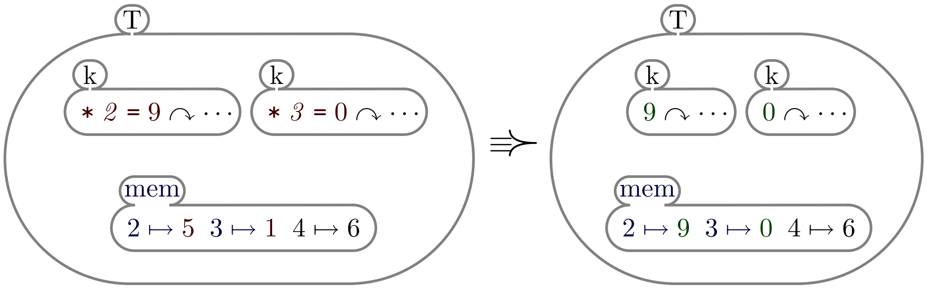


Consider first a configuration where two threads, symbolized by two computation cells, are both ready to read the value of the same memory location:



As the semantics of the 𝕂 rules specify that the parts of the configuration which are only read by the rule can be shared by concurrent applications (here the memory cell construct and the mapping of location 3 to value 1), the read rule can match for both threads and they can both advance one step concurrently.

A similar thing happens for concurrent updates. As long as the threads attempt to update distinct locations, the update rules can match at the same time and the threads can advance concurrently:



Moreover, by disallowing rule instances to overlap on the parts they change, the 𝕂 semantics enforces sequentialization of dataraces, as shown in Figure 1.5.

1.5 Formally Capturing the Concurrency of 𝕂

To formally define the intuition for 𝕂 concurrent rewriting given in the previous section, we have used the resemblance between 𝕂 rules and graph rewriting rules as well as the existing similar concurrency results for graph rewriting and

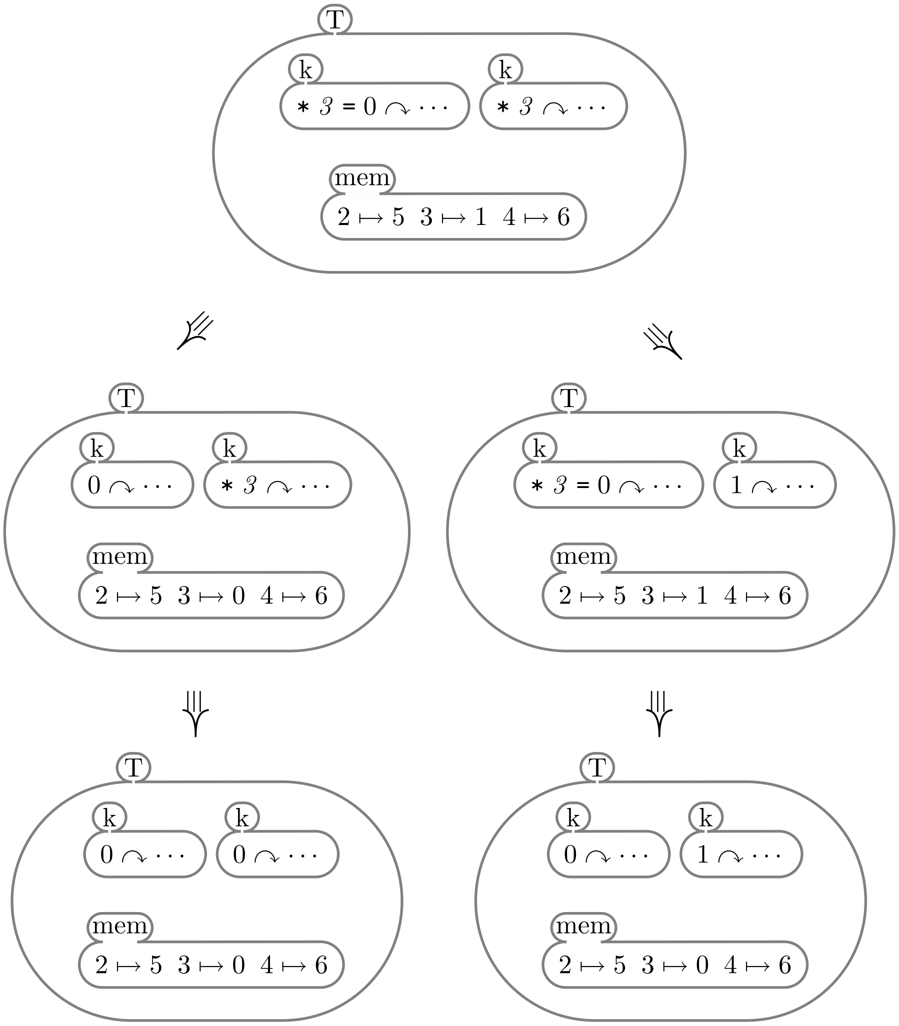


Figure 1.5: Dataraces are forced to sequentialize.

previous approaches on term graph rewriting. The result was a new formalism for term graph rewriting which captures the intended concurrency of $\mathbb{K}$ rewriting, while remaining sound and complete w.r.t. standard rewriting.

Any $\mathbb{K}$ rule can be straight-forwardly represented as a rewrite rule by simply forgetting that a part of it can be shared with other rules. For example, the direct representation of $\mathbb{K}$ rule ρ: $h(\underset{g(x,x)}{\underline{\quad x \quad}}, \_, \underset{0}{\underline{1}})$ as a rewrite rule is:

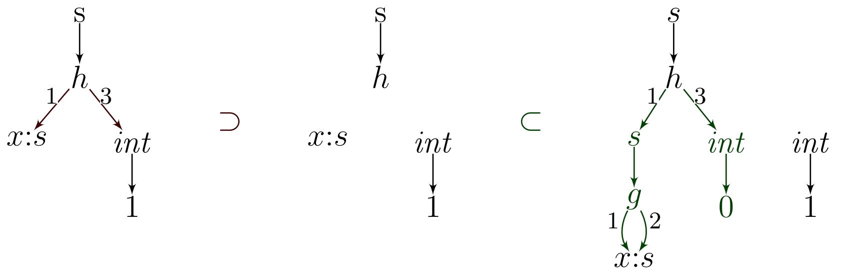
$$K2R(\rho): \; h(x, y, 1) \to h(g(x, x), y, 0),$$

in which a fresh variable is first added for each of the anonymous variables, and then the matching pattern is replicated on the right hand side with the underlined parts being replaced by their corresponding replacements.

The limitation of this representation is that the potential for concurrency with sharing of resources is compromised by the translation, as concurrent applications of rewrite rules are only allowed if the rules do not overlap.

Noticing that the read-only pattern of a $\mathbb{K}$ rule is similar with the interface graph of a graph rewriting rule, we have formalized $\mathbb{K}$ rewriting through the help of graph rewriting, adapting existing representations of terms and rules as graph and graph rewrite rules (mentioned in Section 2.3) to maximize their potential for concurrent application. Without going into details here (the formalization and results expand over the entire Chapter 6), let us exemplify here the representation of $\mathbb{K}$ rules as graph rewrite rules and sketch the main result.

The graph rewrite rule corresponding to the $\mathbb{K}$ rule ρ above is $K2G(\rho)$:



The graph on the left represents the matching pattern, in which *operation nodes*, such as the one labeled by h link a node corresponding to their result sort (here s) to other *sort nodes* corresponding to the arguments of the operation. To preserve the order of arguments, the edges leaving an operation node are labeled with the position of the corresponding arguments. Constants are operation nodes with no outgoing edge, while variables are identified with sort nodes, with the observation that anonymous variables do not even need to be mentioned—in our example, there is no outgoing edge from operation node h labeled with 2, as that position corresponds to an anonymous variable. The middle graph, also called the *interface graph* specifies the shared part of the rule. In order to maximize sharing, note that every node from the matching pattern and all edges except for the outgoing edges of the operation node h are being shared. The right graph adds edges to replace the removed ones, and maybe additional nodes and edges, re-linking the nodes of the interface graph.

Applying a graph rule is done by first matching the left graph against the graph to be rewritten, then removing the part which is not shared in the interface, and finally using the right graph to re-link the disconnected parts of the graph. During this process, some nodes and edges might become unreachable from the root of the term such as, for example the graph corresponding to 1 in our example, with the intuition that these parts can be garbage-collected.

Classical results in the algebraic theory of graph rewriting (reviewed in Section 2.2) ensure that concurrent applications of graph rules are possible and also serializable if they only overlap on the parts matched by their interface graphs. Since this is the kind of concurrency we intended for $\mathbb{K}$ rules, we can now define the $\mathbb{K}$ rewriting relation on terms as follows: term t rewrites in one step to term t' if its graph representation as a tree rewrites in one concurrent step to a graph corresponding to term t'.

Our main result related to the concurrency of $\mathbb{K}$, Theorem 7, shows that $\mathbb{K}$-rewriting is *sound and complete* w.r.t. standard term rewriting using the direct representation of $\mathbb{K}$ rules as rewrite rules, and that the concurrent application of $\mathbb{K}$ rules is *serializable*. Soundness means that applying one $\mathbb{K}$ rule can be simulated by applying its corresponding direct representation as a rewrite rule. Completeness means the converse, i.e., that one application of the direct representation of a $\mathbb{K}$ rule can be simulated by applying the $\mathbb{K}$ rule directly. Finally, the serialization result ensures that applying multiple $\mathbb{K}$ rules in parallel can be simulated by applying them one by one, maybe multiple times, whence, if one does not care about the amount of progress achievable in one step, the direct rewriting logic representation can simulate $\mathbb{K}$ rewriting for all practical purposes.

1.6 K-Maude

Since its semantics is given through an embedding into graph rewriting, $\mathbb{K}$ rewriting can be faithfully captured in rewriting logic, using the representation of graph rewriting in rewriting logic as theories over concurrent objects [104]. This representation of $\mathbb{K}$ rewriting, described in Section 6.5, allows to study within rewriting logic the amount of concurrency allowed by $\mathbb{K}$ definitions. However, as this representation is non-executable, and since (as seen above) $\mathbb{K}$ rewriting can be simulated using the direct representation of $\mathbb{K}$ rules as rewrite rules for most practical purposes, the current implementation of the $\mathbb{K}$ framework, K-Maude, uses the direct representation of $\mathbb{K}$ in rewriting logic for executing and analyzing $\mathbb{K}$ definitions. As previously mentioned, this direct representation is easy to achieve, by simply discarding the read-only information of $\mathbb{K}$ rules. By being executable, it gains full access to all the exploration and verification tools for rewriting logic provided by the Maude rewrite engine [34]. Also, although it loses the one-step concurrency (which is now being simulated by multiple rewrite steps), it is nevertheless sound for all other analysis purposes.

Flattening 𝕂 rules to rewrite rules is the main task of the K-Maude compiler, comprising 16 of the 22 stages of the tool, along with generating heating/cooling rules from strictness annotations and some interface-related stages. Among these 16 stages, 6 stages are used for transformation and inference processes like configuration concretization (based on an initial configuration), replacing anonymous variables with named ones, or reducing in-place rewriting to regular rewriting as shown above. The remainder 10 stages deal with the flattening of the 𝕂 syntax into the AST form. To give an example, the ASCII representation in the K-Maude tool for the KernelC rule for reading the value of a memory location presented above is

$$\mathbf{rule}\ [\mathrm{deref}]{:}\ \ \langle \mathsf{k}\rangle\ *\mathrm{N{:}Nat} \Rightarrow \mathrm{V{:}Val}\ \langle \_/\mathsf{k}\rangle\ \langle \mathsf{mem}\_\rangle\ \mathrm{N} \mapsto \mathrm{V}\ \langle \_/\mathsf{mem}\rangle$$

while the rewrite rule generated from it which is used to execute programs is

$$\begin{array}{l}
\mathrm{rl}\ \ [\mathrm{deref}]{:}\\
\quad \langle\ \mathsf{threads}\ \rangle\ ?10{:}\mathit{Bag}\\
\quad\ \ \langle\ \mathsf{thread}\ \rangle\ ?11{:}\mathit{Bag}\\
\quad\quad \langle\ \mathsf{k}\ \rangle\ \text{'}{*}\_(\mathit{Int}\ \mathrm{N{:}Nat}(.\mathit{List}\{\mathit{K}\}))\curvearrowright\ ?12{:}\mathit{K}\ \langle/\ \mathsf{k}\ \rangle\ \ \langle/\ \mathsf{thread}\ \rangle\ \ \langle/\ \mathsf{threads}\ \rangle\\
\quad \langle\ \mathsf{mem}\ \rangle ?9{:}\mathit{Map}\ \mathit{Int}\ \mathrm{N{:}Nat}(.\mathit{List}\{\mathit{K}\})\ \mapsto \mathrm{V}{:}\mathit{KResult}\ \langle/\ \mathsf{mem}\ \rangle\\
\Rightarrow\\
\quad \langle\ \mathsf{threads}\ \rangle\ ?10{:}\mathit{Bag}\\
\quad\ \ \langle\ \mathsf{thread}\ \rangle\ ?11{:}\mathit{Bag}\\
\quad\quad \langle\ \mathsf{k}\ \rangle\ \mathrm{V}{:}\mathit{KResult}\curvearrowright\ ?12{:}\mathit{K}\ \langle/\ \mathsf{k}\ \rangle\ \ \langle/\ \mathsf{thread}\ \rangle\ \ \langle/\ \mathsf{threads}\ \rangle\\
\quad \langle\ \mathsf{mem}\ \rangle ?9{:}\mathit{Map}\ \mathit{Int}\ \mathrm{N{:}Nat}(.\mathit{List}\{\mathit{K}\})\ \mapsto \mathrm{V}{:}\mathit{KResult}\ \langle/\ \mathsf{mem}\ \rangle\\
.
\end{array}$$

To allow the visualization of definitions, K-Maude provides a 𝕂-to-LaTeX compiler which translates the ASCII 𝕂 into a highly customizable LaTeX code which can be used to typeset the definitions for presentation and documentation purposes. Currently the tool offers two LaTeX styles which can turn the generated code into rules as the ones presented in this introduction, which rely on their graphical form to rapidly communicate the intuition behind the rules, or into more mathematical-looking rules as the ones used in the remainder of this dissertation; for example, the mathematical way to typeset the rule for reading a location from the memory is

$$\langle \underset{V}{\underline{*\,N}}\ \ \cdots\rangle_{\mathsf{k}}\ \ \langle\cdots\ \ N \mapsto V\ \ \cdots\rangle_{\mathsf{mem}}$$

1.7 Programming Language Definitions and Tools in K-Maude and 𝕂

K-Maude gives 𝕂 definitions access to the exploration, analysis and proving tools available for rewriting logic, allowing programs written in the defined programming languages to be executed, traced, explored, and model checked. In Chapter 5 we show how, with minor alterations to a definition, one can obtain runtime verification tools for properties such as memory safety or datarace and deadlock freeness, or how easily is to experiment with different memory models (i.e., a sequentially consistent one, and one based on x86-TSO [128]). Although presented in this dissertation on KernelC, these techniques have already been applied or are in the process of being a applied on the full definition of C in K-Maude developed by Chucky Ellison [51].

In addition to these runtime analysis efforts, K-Maude is being used for tools like type checkers (as the one presented in Section 4.1.3) and type inferencers [52], and in the development (by Elena Naum and Andrei Ștefănescu) of a new program verification tool using Hoare-like assertions based on matching logic [141], or a model checking tool (by Irina and Mihai Asavoae [8]) based on the CEGAR cycle.

Regarding the definitional efforts using K-Maude, besides the C definition mentioned above, K-Maude was used by Patrick Meredith to formalize the $\mathbb{K}$ definition of Scheme [100]. Among the started but not yet finished projects, we should mention the definition of X10 (by Milos Gligoric and Andrei Ștefănescu), the definition of Haskell (by Michael Ilseman and David Lazar), and that of Javascript (by Maurice Rabb). All these definitions and analysis tools can be found on the K-Maude website [88].

Other language definitions and analysis tools developed using the $\mathbb{K}$ technique before the development of the K-Maude tool include Feng Chen's definition of Java [57], Patrick Meredith's and Michael Katelman's definition of Verilog [101], and Mark Hills' static policy checker for C [81].

1.8 Overall Guide to the Dissertation

Chapter 2 provides a brief introduction to background information on rewriting logic, which is required for understanding the remainder of the dissertation, and on graph rewriting, which is required for understanding Chapter 6.

Chapter 3 offers a uniform presentation of the most popular formalisms for defining the operational semantics of programming languages: natural semantics, structural operational semantics (SOS), modular SOS, reduction semantics with evaluation contexts, and the chemical abstract machine. Each of them is shown to be representable into rewriting logic, and this representation is exemplified by using it to define the same simple imperative language. Moreover, the frameworks are compared against each other, and their strengths and weaknesses are analyzed and presented.

Chapter 4 gives an overview of the $\mathbb{K}$ framework. It begins with an example-based intuitive description of the $\mathbb{K}$ framework, showing how it can be used to define both dynamic and static semantics for languages and how to modularly extend existing language definitions with new features. The $\mathbb{K}$ technique is then detailed, explaining essentially how rewriting can be used to define programming language semantics by means of nested-cell configurations, computations, and ($\mathbb{K}$) rewrite rules. The chapter concludes by presenting the definition of the Agent language, which starts as a language of expressions and then iteratively adds functional, imperative, control changing, multithreading, multi-agent, and code generation features. This complex definition aims on challenging existing language definitional frameworks, while highlighting the power of expression and the modularity of the $\mathbb{K}$ framework.

Chapter 5 shows how one can easily transform $\mathbb{K}$ definitions of programming languages into runtime verification tools. The chapter starts with the definition of KernelC, a subset of the C programming language containing functions, memory allocation, pointer arithmetic, and input/output, which can be used to execute and test real C programs. Next, by performing some minor alterations to the definition, one obtains a runtime analysis tool for memory safety. Then, KernelC is extended with threads and synchronization constructs, and two concurrent semantics are derived from its sequential semantics. The first semantics, defining a sequentially consistent memory model can be easily transformed into a runtime verification tool for checking datarace and deadlock freeness, or into a monitoring tool for collecting the trace of events of interest obtained upon executing a program. The second semantics defines a relaxed memory model based on the x86-TSO [128] memory model. By exploring the executions of an implementation of Peterson's mutual exclusion algorithm [132] for both definitions, it is shown that the algorithm guarantees mutual exclusion for the sequentially consistent model, but cannot guarantee it for the relaxed model.

Chapter 6 formally defines a truly concurrent semantics for the $\mathbb{K}$ framework which allows $\mathbb{K}$ definitions to capture concurrency with sharing of data as expressible in current multithreaded languages. The resemblance between $\mathbb{K}$ rules and graph rewrite rules is used to define $\mathbb{K}$ concurrent rewriting through an embedding of $\mathbb{K}$ into a novel term-graph rewriting formalism, named $\mathbb{K}$ graph rewriting. Under reasonable restrictions it is shown that $\mathbb{K}$ graph rewriting allows a higher degree of concurrency than the existing term graph rewriting formalisms, while it remains serializable and sound w.r.t rewriting. $\mathbb{K}$ rewriting is then defined as the projection of $\mathbb{K}$ graph rewriting on terms and is shown serializable, as well as sound and complete w.r.t. standard term rewriting. Finally, it is shown how the concurrency available through $\mathbb{K}$ rewriting can be theoretically captured within rewriting logic, by using rewriting logic's ability of capturing graph rewriting.

Chapter 7 describes the K-Maude tool, a Maude implementation of the direct representation of $\mathbb{K}$ in rewriting logic. It builds upon the $\mathbb{K}$ notation and technique presented in Chapter 4 and provides a concrete ASCII representation for them, as well as a concrete compiler onto the Maude rewrite engine. The chapter starts with presenting the ASCII representation of the $\mathbb{K}$ mathematical notation which is used in writing $\mathbb{K}$ definitions using the K-Maude tool. Then K-Maude is explained from an user point of view, insisting on its features and on usability. Then the two translations provided by the tool are explained, one implementing the direct representation of $\mathbb{K}$ into rewriting logic as executable rewrite theories, and the other transforming the definitions into LaTeX code.

Finally, Chapter 8 discusses relevant related work, while Chapter 9 presents the conclusions of the dissertation and opportunities for future research.

1.9 Relationship with Previous Work

This dissertation is part of the rewriting logic semantics (RLS) project, proposed by Meseguer and Roșu [109, 110, 111] as an ongoing effort towards developing a tool-supported computational logical framework for modular programming language design, semantics, formal analysis and implementation based on rewriting logic [103], with an emphasis on the ability of rewriting to naturally capture concurrency.

The $\mathbb{K}$ framework itself is based on the experience gained from trying to define complex programming language features and observing how other techniques coped with the same features. In particular, some features of the $\mathbb{K}$ framework are inspired, or closely related with successful features of other definitional frameworks. The representation of the computation as a list of tasks has evolved from the representation of computations in continuations-based interpreters [60]. The idea of cells and structural rules resembles the cells and heating/cooling rules of the Chemical abstract machine [15]. The ideas of only representing what is needed in a configuration, relying on details being completed once all pieces are put together, is related to the Modular SOS [123] approach, as well as to the monad transformers approach [118] used for the Semantic Lego project [53]. Evaluation contexts are the same concept used in reduction semantics [180].

This and other related work are extensively addressed in Chapter 8.

1.10 Related Publications and Collaborators

This section provides a quick overview of the publications based on the research presented in this dissertation, and gives attribution to fellow researchers with whom the presented results were obtained. As all the work in this dissertation was done in collaboration with Grigore Roșu, he will only be mentioned here, with the understanding that he is an implicit co-author to all the publications listed below.

Rewriting Logic Semantics. The research in Chapter 3 was done in collaboration with José Meseguer and was published as *A Rewriting Logic Approach to Operational Semantics* [160] in the Journal of Information and Computation.

$\mathbb{K}$. The overview of the $\mathbb{K}$ framework (Chapter 4), as well as the intuition and rationale for $\mathbb{K}$ concurrent rewriting (Section 6.2) were published as *An Overview of the K Semantic Framework* [146] in the Journal of Logic and Algebraic Programming.

Runtime analysis using $\mathbb{K}$. Chapter 5 presents and combines ideas and constructions from two sources. The work on runtime verification of memory safety (for a smaller fragment of C) using $\mathbb{K}$ was included in the article *Runtime Verification of C Memory Safety* [149] co-authored with Worlfram Schulte, which was presented at Runtime Verification 2009. The work on exploring and analyzing concurrent executions is based on the author's previous work with Feng Chen on predictive runtime analysis, and recasts in $\mathbb{K}$ ideas from the paper *jPredictor: A Predictive Runtime Analysis Tool for Java* [32] which was presented at the 2008 International Conference on Software Engineering.

K-Maude. The research presented in Chapter 7 was included in *K-Maude: A Rewriting Based Tool for Semantics of Programming Languages* [159] which was presented together with a K-Maude demonstration at the 2010 Workshop on Rewriting Logic and its Applications. The K-Maude tool itself [88] is a collaborative project involving researchers from Grigore Roșu's UIUC group and Dorel Lucanu's University of Iași group which have contributed over the time to the development of the tool through many suggestions, examples, and feature requests. Among them, the author would like to particularly acknowledge Andrei Arusoaie and Michael Ilseman for the development and maintenance of the external user interface, and Chucky Ellison, whose contributions to developing languages using the K-Maude tool (including a complete definition of C programming language [51]) have led to numerous suggestions for improvements in terms of both new features and design thus making the tool better and more stable.

Chapter 2

Background

This chapter revisits the theoretical background of this dissertation. Since $\mathbb{K}$ is a *rewriting* framework, we present below two frameworks formalizing two types of rewriting which will be assumed known for the subsequent development of the dissertation. These are rewriting logic, presented in Section 2.1, and graph rewriting, presented in its so called double-pushout algebraic formalization in Section 2.2. The aim of $\mathbb{K}$ is to achieve the concurrency with sharing of resources made manifest in graph rewriting, while maintaining the simplicity of a first order representation of terms, as well as benefiting from the generic techniques and tools available for rewriting logic.

2.1 Rewriting Logic

Rewriting is a crucial paradigm in algebraic specifications, since it provides a natural means for executing equational specifications. Many specification languages, including CafeOBJ [42], ELAN [16], Maude [34], OBJ [65], ASF/SDF [171], provide conditional rewrite engines to execute and reason about specifications.

A rewrite system over a term universe consists of rewrite rules, which can be locally matched and applied at different positions in a term to gradually transform it. Due to the locality of rewrite rules, it follows naturally that multiple rules can apply in parallel. Rewriting logic exploits this, by allowing sideways and nested (i.e. rewriting-under-variables) parallelism in rule application.

Rewriting logic was introduced by Meseguer [102, 103] as a unified model for concurrency, capturing the concurrency of standard term rewriting as logical deduction. Rewriting logic is a computational logic that can be efficiently implemented and that has good properties as a general and flexible *logical and semantic framework*, in which a wide range of logics and models of computation can be faithfully represented [99].

Two key points to explain are: (i) how rewriting logic combines equational logic and traditional term rewriting; and (ii) what the intuitive meaning of a rewrite theory is all about. A *rewrite theory* is a triple $\mathcal{R} = (\Sigma, E, R)$ with Σ a signature of function symbols, E a set of (possibly conditional) Σ-equations, and R a set of Σ-rewrite rules which in general may be conditional, with conditions involving both equations and rewrites. That is, a rule in R can have the general form

$$(\forall X)\ t \longrightarrow t' \ \text{ if } \ (\bigwedge_i u_i = u'_i) \wedge (\bigwedge_j w_j \longrightarrow w'_j)$$

Alternatively, such a conditional rule could be displayed with an inference-rule-like notation as

$$\frac{(\bigwedge_i u_i = u'_i) \wedge (\bigwedge_j w_j \longrightarrow w'_j)}{t \longrightarrow t'}$$

Therefore, the logic's atomic sentences are of two kinds: equations, and rewrite rules. Equational theories and traditional term rewriting systems then appear as special cases. An equational theory (Σ, E) can be faithfully represented as the rewrite theory $(\Sigma, E, \emptyset)$; and a term rewriting system (Σ, R) can likewise be faithfully represented as the rewrite theory $(\Sigma, \emptyset, R)$.

Of course, if the equations of an equational theory (Σ, E) are *confluent*, there is another useful representation, namely, as the rewrite theory $(\Sigma, \emptyset, \overrightarrow{E})$, where $\overrightarrow{E}$ are the rewrite rules obtained by orienting the equations E as rules from left to right. This representation is at the basis of much work in term rewriting, but by implicitly suggesting that rewrite rules are just an efficient technique for equational reasoning it can blind us to the fact that rewrite rules can have a much more general *non-equational* semantics. This is the whole *raison d'être* of rewriting logic. In rewriting logic a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ axiomatizes a *concurrent system*, whose states are elements of the algebraic data type axiomatized by (Σ, E), that is, they are E-equivalence classes of ground Σ-terms, and whose *atomic transitions* are specified by the rules R. The inference system of rewriting logic described below then allows us to derive as *proofs* all the possible *concurrent computations* of the system axiomatized by $\mathcal{R}$, that is, concurrent computation and rewriting logic deduction *coincide*.

2.1.1 Rewriting Logic Deduction

The inference rules below assume a typed setting, in which (Σ, E) is a *membership equational theory* [105] having sorts (denoted s, s', s'', etc.), subsort inclusions, and kinds (denoted k, k', k'', etc.), which gather together connected components of sorts. Kinds allow error terms like 3/0, which has a kind but no sort. Similar inference rules can be given for untyped or simply typed (many-sorted) versions of the logic. Given $\mathcal{R} = (\Sigma, E, R)$, the sentences that $\mathcal{R}$ proves are universally quantified rewrites of the form $(\forall X)\ t \longrightarrow t'$, with $t, t' \in T_\Sigma(X)_k$, for some kind k, which are obtained by finite application of the following *rules of deduction*:

- **Reflexivity**. For each $t \in T_\Sigma(X)$, $\dfrac{}{(\forall X)\ t \longrightarrow t}$
- **Equality**. $\dfrac{(\forall X)\ u \longrightarrow v \quad E \vdash (\forall X) u = u' \quad E \vdash (\forall X) v = v'}{(\forall X)\ u' \longrightarrow v'}$

- **Congruence**. For each $f : s_1 \dots s_n \longrightarrow s$ in Σ, with $t_i \in T_\Sigma(X)_{s_i}$, $1 \leq i \leq n$, and with $t'_{j_l} \in T_\Sigma(X)_{s_{j_l}}$, $1 \leq l \leq m$,

$$\frac{(\forall X)\ t_{j_1} \longrightarrow t'_{j_1} \quad \dots \quad (\forall X)\ t_{j_m} \longrightarrow t'_{j_m}}{(\forall X)\ f(t_1, \dots, t_{j_1}, \dots, t_{j_m}, \dots, t_n) \longrightarrow f(t_1, \dots, t'_{j_1}, \dots, t'_{j_m}, \dots, t_n)}$$

- **Replacement**. For each $\theta : X \longrightarrow T_\Sigma(Y)$ and for each rule in R of the form

$$(\forall X)\ t \longrightarrow t' \ \text{ if } \ (\bigwedge_i u_i = u'_i) \wedge (\bigwedge_j w_j \longrightarrow w'_j),$$

$$\frac{(\bigwedge_x (\forall Y)\ \theta(x) \longrightarrow \theta'(x)) \wedge (\bigwedge_i (\forall Y)\ \theta(u_i) = \theta(u'_i)) \wedge (\bigwedge_j (\forall Y)\ \theta(w_j) \longrightarrow \theta(w'_j))}{(\forall Y)\ \theta(t) \longrightarrow \theta'(t')}$$

  where θ' is the new substitution obtained from the original substitution θ by some possibly complex rewriting of each $\theta(x)$ to some $\theta'(x)$ for each $x \in X$.

- **Transitivity**

$$\frac{(\forall X)\ t_1 \longrightarrow t_2 \qquad (\forall X)\ t_2 \longrightarrow t_3}{(\forall X)\ t_1 \longrightarrow t_3}$$

We can visualize the above inference rules as in Figure 2.1.

The notation $\mathcal{R} \vdash t \longrightarrow t'$ states that the sequent $t \longrightarrow t'$ is *provable* in the theory $\mathcal{R}$ using the above inference rules. Intuitively, we should think of the inference rules as different ways of *constructing* all the (finitary) *concurrent computations* of the concurrent system specified by $\mathcal{R}$. The "**Reflexivity**" rule says that for any state t there is an *idle transition* in which nothing changes. The "**Equality**" rule specifies that the states are in fact equivalence classes modulo the equations E. The "**Congruence**" rule is a very general form of "sideways parallelism," so that each operator f can be seen as a *parallel state constructor*, allowing its arguments to evolve in parallel. The "**Replacement**" rule supports a different form of parallelism, which could be called "parallelism under one's feet," since besides rewriting an instance of a rule's left-hand side to the corresponding right-hand side instance, the state fragments in the substitution of the rule's variables can also be rewritten. Finally, the "**Transitivity**" rule allows us to build longer concurrent computations by composing them sequentially.

A somewhat more general version of rewriting logic [24] allows rewrite theories of the form $\mathcal{R} = (\Sigma, E \cup A, R, \phi)$, where the additional component ϕ is a function assigning to each function symbol $f \in \Sigma$ with n arguments a subset $\phi(f) \subseteq \{1, \dots, n\}$ of those argument positions that are *frozen*, that is, positions under which rewriting is forbidden. The above inference rules can then be slightly generalized. Specifically, the **Congruence** rule is restricted to non-frozen positions $\{j_1, \dots, j_m\}$, and the substitution θ' in the **Replacement** rule should

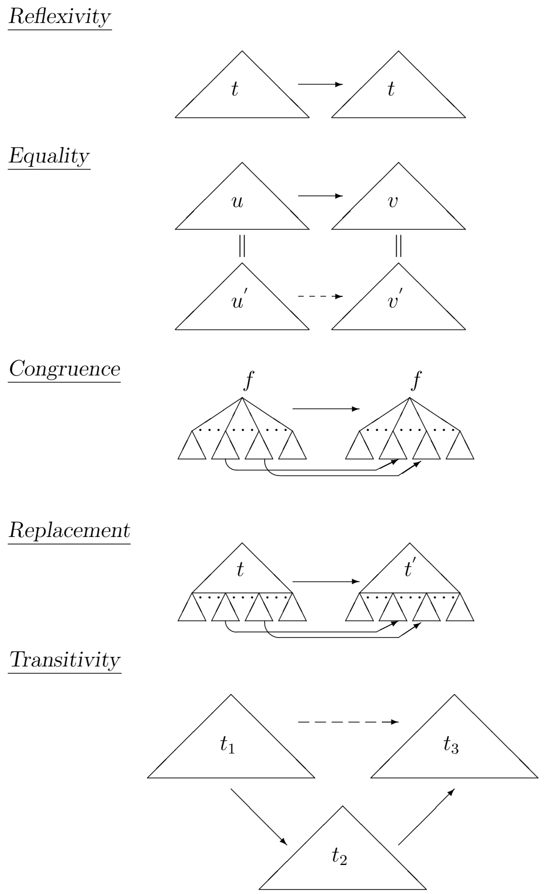


Figure 2.1: Visual representation of rewriting logic deduction.

only differ from θ for variables x in non-frozen positions. The generalized form $\mathcal{R} = (\Sigma, E \cup A, R, \phi)$, makes possible a more expressive control of the possibility of rewriting under contexts already supported by the **Congruence** rule; that is, it endows rewrite theories with flexible context-sensitive rewriting capabilities.[1]

Note that, in general, a proof $\mathcal{R} \vdash t \longrightarrow t'$ does not represent an *atomic* step, but can represent a complex concurrent computation. In some of the mathematical proofs that we will give to relate different operational semantics definitions, it will be easier to work with a "one step" rewrite relation $\rightarrow^1$, defined on ground terms. This relation is just the special case in which: (i) **Transitivity** is excluded; (ii) $m = 1$ in the **Congruence** rule (only one rewrite below); and (iii) **Replacement** is restricted, so that no rewriting of the substitution θ to θ' is allowed; and (iv) there is exactly *one* application of **Replacement**. The relation $\rightarrow^{\leq 1}$ is defined by allowing either one or no applications of **Replacement** in the last condition. Similarly, one can define relations $\rightarrow^n$ (or $\rightarrow^{\leq n}$) by controlling the number of applications of the **Transitivity** rule. However, it should be noted that rewriting logic *does not* have a builtin "one-step" rewrite relation, that

[1] We will not consider this general version. The interested reader is referred to [24].

being the reason for which we need a methodology to encode "one step"-based formalisms such as SOS semantics. The "one-step" relation we define above is only at the deduction level and is introduced solely to help our proofs.

The whole point of RLS is then to define the semantics of a programming language $\mathcal{L}$ as a rewrite theory $\mathcal{R}_{\mathcal{L}}$. RLS uses the fact that rewriting logic deduction is performed *modulo* the equations in $\mathcal{R}_{\mathcal{L}}$ to faithfully capture the desired granularity of a language's computations. This is achieved by making rewriting rules all intended computational steps, while using equations for convenient equivalent structural transformations of the state, or auxiliary "infrastructure" computations, which should not be regarded as computation steps. Note that this does not preclude performing also equational simplification with equations. That is, the set E of equations in a rewrite theory can often be fruitfully decomposed as a disjoint union $E = E_0 \cup A$, where A is a set of *structural axioms*, such as associativity, commutativity and identity of some function symbols, and E_0 is a set of equations that are confluent and terminating *modulo* the axioms A. A rewrite engine supporting rewriting modulo A will then execute both the equations E_0 and the rules R modulo A by rewriting. Under a condition called *coherence* [176], this form of execution then provides a complete inference system for the given rewrite theory (Σ, E, R). However, both conceptually and operationally, the execution of rules R and equations E_0 must be separated. Conceptually, what we are rewriting with R are E-equivalence classes, so that the E_0-steps become *invisible*. Operationally, the execution of rules R and equations E_0 must be kept separate for soundness reasons. This is particularly apparent in the case of executing *conditional* equations and rules: for a conditional equation it would be *unsound* to use rules in R to evaluate its condition; and for a conditional rule it would likewise be unsound to use rules in R to evaluate the *equational* part of its condition.

There are many systems that either specifically implement term rewriting efficiently, so-called as *rewrite engines*, or support term rewriting as part of a more complex functionality. Any of these systems can be used as an underlying platform for execution and analysis of programming languages defined using the techniques described in this dissertation. Without attempting to be exhaustive, we here only mention (alphabetically) some engines that we are more familiar with, noting that many functional languages and theorem provers provide support for term rewriting as well: ASF+SDF [171], CafeOBJ [42], Elan [16], Maude [36], OBJ [65], and Stratego [177]. Some of these engines can achieve remarkable speeds on today's machines, in the order of tens of millions of rewrite steps per second.

2.2 Graph Rewriting

Graph grammars and graph transformations were introduced in the early seventies in an effort to generalize the concept of Chomsky grammars from strings to graphs. The main idea was to generalize the concatenation of strings to a gluing construction for graphs. One of the most successful approaches in this direction was the algebraic approach [46], which defined a graph rewriting step by two gluing constructions, which in turn, could be algebraically expressed as a double pushout in the category of graphs. Although initially developed using the intuition of grammars and focusing on their generative power, graph grammars were soon adopted for computation purposes, which were called initially graph transformations, and later graph rewriting, by their analogy to term rewriting. Nowadays, graph grammars and graph transformations are studied and applied in many fields of computers science (e.g., data and control flow diagrams, entity-relationship and UML diagrams, Petri nets) for their ability to explain complex situations at an intuitive level [47]. We refer the interested reader to Corradini et al. [39] for a survey of the graph rewriting concepts used in this dissertation, or to the more recent monography of Ehrig et al. [47] for a comprehensive treatment of the algebraic approach to graph rewriting.

A graph consists of a set of (labeled) vertices V and a set of (labeled) edges E linking the vertices in V. Assuming fixed sets $\mathcal{L}_V$ and $\mathcal{L}_E$ for node and for edge labels, respectively, a *graph G over labels* $(\mathcal{L}_V, \mathcal{L}_E)$ is a tuple $G = \langle V, E, \mathrm{source}, \mathrm{target}, \mathrm{lv}, \mathrm{le}\rangle$, where V is the set of *vertices* (or *nodes*), E is a set of *edges*, $\mathrm{source}, \mathrm{target} : E \to V$ are the *source* and the *target* functions, and $\mathrm{lv} : V \to \mathcal{L}_V$ and $\mathrm{le} : E \to \mathcal{L}_E$ are the node and the edge *labeling functions*, respectively. We will use V_G, E_G, source_G, $\ldots$, to refer to the corresponding components of the tuple describing a graph G. A *graph morphism* $f : G \to G'$ is a pair $f = \langle f_V : V_G \to V_{G'}, f_E : E_G \to E_{G'}\rangle$ of functions preserving sources, targets, and labels. Let $\mathbf{Graph}(\mathcal{L}_\mathbf{V}, \mathcal{L}_\mathbf{E})$ denote the category of graphs over labels $(\mathcal{L}_V, \mathcal{L}_E)$. Given graph G, let $\prec_G \subseteq V \times V$ be its *path relation*: $v_1 \prec_G v_2$ iff there is a path from v_1 to v_2 in G. G is *cyclic* iff there is some $v \in V_G$ s.t. $v \prec_G v$. Given $v \in V_G$, let $G{\restriction_v}$ be the subgraph of G (forwardly) reachable from v.

A *graph rewrite rule* $p : (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$, where p is its name, is a pair of graph morphisms $l : K \to L$ and $r : K \to R$, where l is injective. The graphs L, K, and R are called the *left-hand-side* (lhs), the *interface*, and the *right-hand-side* (rhs) of p, respectively.

The graph rewriting process can be intuitively described as follows. Given a graph G and a match of L into G satisfying some *gluing conditions* (discussed below), we can rewrite G into a graph H in two steps: (1) delete from G the part from L that does not belong to K, obtaining a context C—the gluing conditions ensure that C is still a graph; (2) embed R into C by gluing it along the instance of K in C.

Consider the example in Figure 2.2. Assume we are working in a graph category where there is only one label for nodes and two labels for edges, which are represented graphically as full edges or dotted edges. Moreover, suppose we want to represent undirected graphs, so each edge in the graphical representation represents actually two edges in opposite directions (suggested by the double arrowhead). The intuition for the dotted edges is that they represent edges

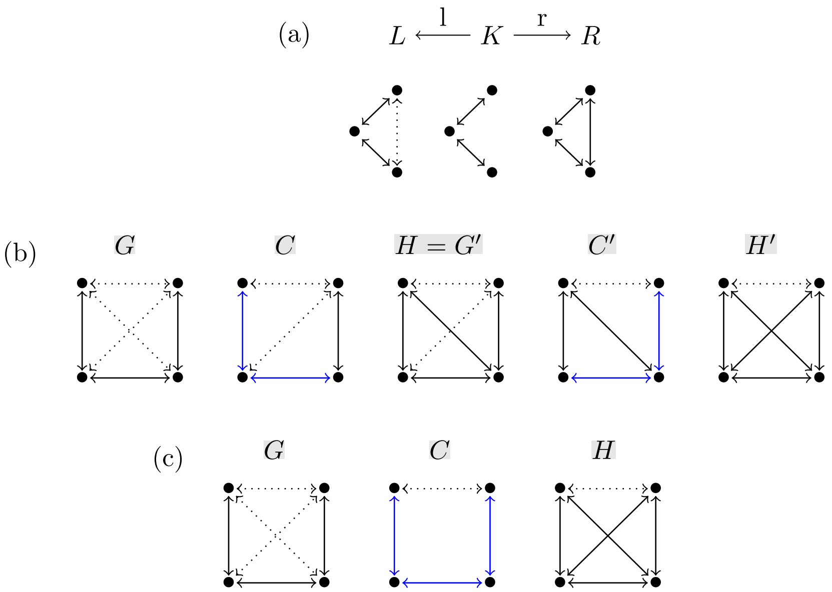


Figure 2.2: Illustration of graph rules and graph transformations: (a) A graph rule for expressing transitivity; (b) a possible sequential application; (c) and a parallel application combining the two applications above.

missing from the graph. Therefore, the graphs we work with will be full graphs, with their edges being split between full and dotted ones. In this context, the rule in Figure 2.2(a) specifies the closure of the edge relation, that is, if there is a (full) edge from node v_1 to node v_2 and one from node v_2 to node v_3, and the edge from v_1 to v_3 is dotted (i.e., is missing), then there that edge should become full as well. Two steps of applying the rule, each with their intermediate context are represented in Figure 2.2(b).

Formally, given a graph G, a graph rule $p : (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$, and a *match* $m : L \rightarrow G$, a *direct derivation from* G *to* H *using* p *(based on* m*)* exists iff the diagram below can be constructed,

$$\begin{tikzcd}
\mathrm{L} \arrow[d, "m"] & \mathrm{K} \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "\overline{m}"] & \mathrm{R} \arrow[d, "m^*"] \\
\mathrm{G} & \mathrm{C} \arrow[l, "l^*"'] \arrow[r, "r^*"] & \mathrm{H}
\end{tikzcd}$$

where both squares are pushouts in the category of graphs. In this case, C is called the *context* graph, and we write $G \overset{p,m}{\Longrightarrow} H$ or $G \overset{p}{\Rightarrow} H$. As usual with pushouts, whenever l or r is an inclusion, the corresponding l^* or r^* can be chosen to also be an inclusion.

A direct derivation $G \overset{p,m}{\Longrightarrow} H$ exists iff the following *gluing conditions* hold [44]: *(Dangling condition)* no edge in $E_G \setminus m_E(E_L)$ is incident to any node in $m_V(V_L \setminus l_V(V_K))$; and *(Identification condition)* there are no $x, y \in V_L \cup E_L$ with $x \neq y$, $m(x) = m(y)$ and $x, y \notin l(V_K \cup E_K)$. If it exists, H is unique up to graph isomorphism. The gluing conditions say that whenever a transformation deletes a node, it should also delete all its edges (dangling condition), and that a match is only allowed to identify elements coming from K (identification condition).

Given a family of graph-rewrite rules $p_i : (L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i)$, $i = \overline{1,n}$, not necessarily distinct, their *composed graph-rewrite rule*, denoted as $p_1 + \cdots + p_n$, is a rule $p : (L \xleftarrow{l} K \xrightarrow{r} R)$ where L, K, and R are the direct sums of the corresponding components from $(p_i)_{i=\overline{1,n}}$ and, similarly, l and r are the canonical morphisms induced by $(l_i)_{i=\overline{1,n}}$ and $(r_i)_{i=\overline{1,n}}$, respectively. Given a graph G, matches $(m_i : L_i \to G)_{i=\overline{1,n}}$ induce a combined match $m : L \to G$ defined as the unique arrow amalgamating all particular matches from the universality property of the direct sum. Matches $(m_i : L_i \to G)_{i=\overline{1,n}}$ have the *parallel independence* property iff for all $1 \leq i < j < n$, $m_i(L_i) \cap m_j(L_j) \subseteq m_i(K_i) \cap m_j(K_j)$. If $(m_i : L_i \to G)_{i=\overline{1,n}}$ have the parallel independence property and each m_i satisfies the gluing conditions for rule p_i, then the combined match m satisfies the gluing conditions for the composed rule $p_1 + \cdots + p_n$, and thus there exists a graph H such that $G \xRightarrow{p_1+\cdots+p_n,m} H$. Moreover, this derivation is serializable, i.e., $G \xRightarrow{p_1+\cdots+p_{n-1},m'} H_{n-1} \xRightarrow{p_n} H$, where m' is the composition of $(m_i)_{i=\overline{1,n-1}}$ [69, Theorem 7.3] (recasting prior results [45, 93]).

2.3 Term Graph Rewriting

Term graph rewriting is a research direction within the larger graph-rewriting context, concerned with using graph rewriting techniques in defining and efficiently implementing term rewriting. The rationale behind this line of work is generally to be able to give a more efficient implementation of functional programming languages in the form of lambda calculus or term rewrite systems: identical subterms are shared using pointers [12]. To this aim, there have been proposed (and proven sound) several approaches [12, 137, 94] to encoding terms as graphs (or hypergraphs) and rewrite rules as graph transformation rules.

The term-graph rewriting approach we build upon [68, 83] uses (directed) hypergraphs, called *jungles* to encode terms and rules. A *hypergraph* $G = (V, E, \mathrm{source}, \mathrm{target}, \mathrm{lv}, \mathrm{le})$ over labels $(\mathcal{L}_V, \mathcal{L}_E)$ has basically the same structure as a graph; however, each edge is allowed to have (ordered) multiple sources and targets, that is, the source and target mappings now have as range V^*, the set of strings over V. For a hypergraph G, $\mathit{indegree}_G(v)/\mathit{outdegree}_G(v)$ denote the number of occurrences of a node v in the target/source strings of all edges in G.

Given a signature $\Sigma = (S, F)$, a *jungle* is a hypergraph over (S, F) satisfying that: (1) each edge is compatible with its arity, i.e., for each $e \in E$ such that $\mathrm{le}(e) = f : s_1 \ldots s_k \to s$, it must be that $\mathrm{lv}^*(\mathrm{source}(e)) = s$ and $\mathrm{lv}^*(\mathrm{target}(e)) = s_1 \ldots s_k$; (2) $\mathit{outdegree}(v) \leq 1$ for any $v \in V$, that is, each node can be the source of at most one edge; and (3) G is acyclic.

A jungle represents a term as an acyclic hypergraph whose nodes are labeled by sort names, and whose edges are labeled by names of operations in the signature; Figure 2.3 (graph G) depicts the jungle representation of term $h(f(a), 0, 1)$.

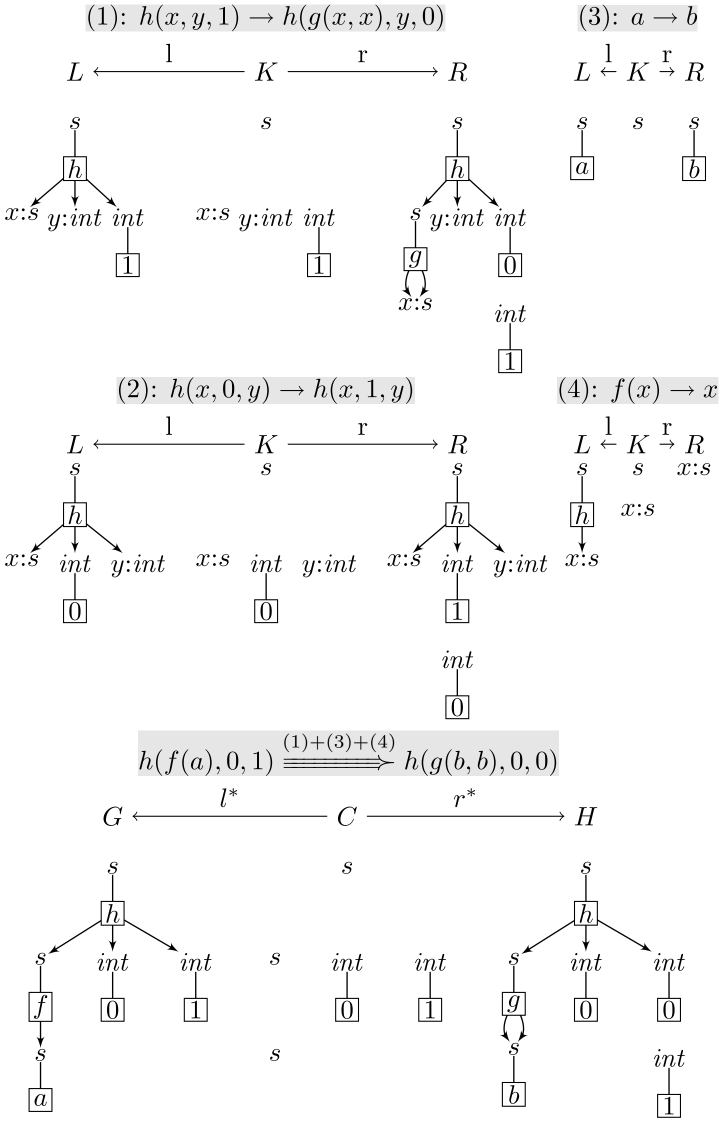

(1): $h(x, y, 1) \to h(g(x, x), y, 0)$
$L \xleftarrow{l} K \xrightarrow{r} R$
(3): $a \to b$
$L \xleftarrow{l} K \xrightarrow{r} R$
(2): $h(x, 0, y) \to h(x, 1, y)$
$L \xleftarrow{l} K \xrightarrow{r} R$
(4): $f(x) \to x$
$L \xleftarrow{l} K \xrightarrow{r} R$
$h(f(a), 0, 1) \xRightarrow{(1)+(3)+(4)} h(g(b, b), 0, 0)$
$G \xleftarrow{l^*} C \xrightarrow{r^*} H$
s
h
x:s
y:int
int
g
f
a
b
0
1

Figure 2.3: Jungle representations for a couple of rewrite rules and a possible concurrent step.

Constants are edges without any target. Variables are represented as nodes which are not sources of any edge. Non-linear terms are represented by identifying the nodes corresponding to the same variable. There could be multiple possible representations of the same term as a jungle, as identical subterms can be identified (or not) in the jungle representation.

Let VAR_G denote the *variables of* G; we have that $VAR_G = \{v \in V_G \mid \mathit{outdegree}_G(v) = 0\}$. The *term represented by some node* v *in a jungle* G, $\mathit{term}_G(v)$, is obtained by descending along hyperedges and collecting the hyperedge labels. Let G be a jungle. Then

$$\mathit{term}_G(v) = \begin{cases} v & \text{if } v \in VAR_G \\ \mathrm{le}(e)(\mathit{term}^*_G(\mathrm{target}(e)) & \text{otherwise, where } \{e\} = \mathrm{source}^{-1}(v) \end{cases}$$

A *root* of a jungle is a node v such that $\mathit{indegree}(v) = 0$. Let $ROOT_G$ denote the set of roots of G. Given a term t (with variables), a *variable-collapsed* tree representing t is a jungle G with a single root root_G which is obtained from the tree representing t by identifying all nodes corresponding to the same variable, that is, $\mathit{term}_G(\mathit{root}_G) = t$, and for all $v \in V$, $\mathit{indegree}(v) > 1$ implies that $v \in VAR_G$.

A term rewrite rule $\mathit{left} \to \mathit{right}$ is encoded as a *jungle evaluation rule* $L \hookleftarrow K \xrightarrow{r} R$ in the following way:

L is a variable-collapsed tree corresponding to *left*.

K is obtained from L by removing the hyperedge corresponding to the top operation of *left* (that is, $\mathrm{source}^{-1}(\mathit{root}_L)$).

R is obtained from K as follows: if *right* is a variable (i.e., the rule is collapsing, then the root_L is identified with *right*; otherwise, R is the disjoint union of K and a variable collapsing tree R' corresponding to *right*, where $\mathit{root}_{R'}$ is identified with root_L and each variable of R' is identified with its counterpart from VAR_L.

l, r $L \overset{l}{\hookleftarrow} K$ and $K \xrightarrow{r} R$ are inclusions with the exception that r maps root_L to *right* if *right* is a variable.

Jungle evaluation is done according to the DPO graph rewriting approach presented above (although here we are talking about hypergraphs, the results above carry through [137]). In particular, note that the gluing conditions are satisfied for each matching morphism m in a jungle G. The dangling condition holds because $V_K = V_L$ so there could be no dangling edge. The identification condition could be violated only if the edge e representing the top operation of l, i.e., $\{e\} = \mathrm{source}^{-1}(\mathit{root}_L)$ was identified with another edge e'. However, since the source of an edge is unique, this would lead to $m(\mathit{root}_L) = m(\mathrm{source}(e'))$, which contradicts with the fact that G (being a jungle) is acyclic. The (hyper)graph rewriting step obtained upon applying an evaluation rule to a jungle is called an *evaluation step*. Since $V_K = V_L$ we have that all the nodes of the graph to be rewritten are preserved in the context graph C (i.e., $V_C = V_G$), and thus for each $v \in V_G$, its correspondent in H is $r^*(v)$.

Among the many interesting results relating term rewriting with jungle evaluation, we will build our results on the ones presented below.

Theorem 1. *Let p be an evaluation rule for a rewrite rule ρ, and let G be a jungle.*

1. *Evaluation steps preserve jungles, i.e., if $G \overset{p}{\Rightarrow} H$ then H is a jungle;*
2. *If $G \overset{p}{\Rightarrow} H$, then for each $v \in V_G$ $term_G(v) \overset{\rho^n}{\Longrightarrow} term_H(r^*(v))$*
3. *If ρ is left-linear and $term_G(v) \overset{\rho}{\Rightarrow} t'$ for some $v \in V_G$, then there exists H such that $G \overset{p}{\Rightarrow} H$.*

Proof. (1) and (2) are proved by Hoffmann and Plump [83][Theorems 5.4 and 5.5].

(3) follows from [83][Lemma 6.2] and [138][Theorem 4.8]. □

The Induced Rewriting Relation.

Theorem 1 actually shows that jungle evaluation is both *sound and complete* for one step of term rewriting using left linear rules.

Corollary 1. *If G is a variable-collapsed tree and ρ is left-linear, then $term_G(root_G) \overset{\rho}{\Rightarrow} t'$ iff there exists a jungle G' such that $G \overset{p}{\Rightarrow} G'$ and $term_{G'}(r^*(root_G)) = t'$.*

Indeed, let $\mathcal{R}$ be a left-linear rewrite system, and let $\overline{\mathcal{R}}$ be the system containing the evaluation rules corresponding to the rewrite rules in $\mathcal{R}$. Let $\Rightarrow^{1}_{\overline{\mathcal{R}}}$ be the relation defined on Σ-terms by $t \Rightarrow^{1}_{\overline{\mathcal{R}}} t'$ iff $G \overset{p}{\Rightarrow} H$, where G is a variable-collapsed tree, $term_G(root_G) = t$, $p \in \overline{\mathcal{R}}$, and $term_H(r^*(root_G)) = t'$. Then,

Corollary 2. $\Rightarrow^{1}_{\overline{\mathcal{R}}} = \Rightarrow^{1}_{\mathcal{R}}$.

2.3.1 The Bipartite Graph Representation of Jungles

Although inspired from jungle evaluation, and relying of the results presented above, our graph rewriting approach for capturing $\mathbb{K}$ rewriting will not use hypergraph jungles, but rather an extension of their equivalent (bipartite) graph representation.

Given a hypergraph $G = (V_G, E_G, \mathrm{source}_G, \mathrm{target}_G, \mathrm{lv}_G, \mathrm{le}_G)$, a *graph representation of G over labels* $(\mathcal{L}_V, \mathcal{L}_E)$ is any graph isomorphic with the bipartite graph $G' = (V_{G'}, E_{G'}, \mathrm{source}_{G'}, \mathrm{target}_{G'}, \mathrm{lv}_{G'}, \mathrm{le}_{G'})$ over labels $(\mathcal{L}_V \cup \mathcal{L}_E, \mathit{Int})$, defined by

- $V_{G'} = V_G \cup E_G$;
- $E_{G'} = \bigcup_{e \in E_G} \{(e, i) \mid$ if $|\mathrm{source}_G(e)| < i \leq 0$ or $0 < i \leq |\mathrm{target}_G(e)|$;

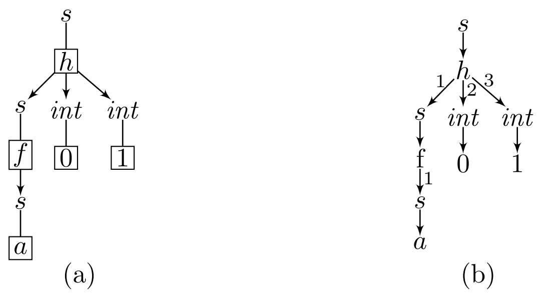


Figure 2.4: (a) A hypergraph jungle representing $h(f(a), 0, 1)$; and (b) its bipartite graph representation.

- $\mathrm{source}_{G'}((e, i)) = v$, if $i \leq 0$ and v is the $(-i + 1)$th element in $\mathrm{source}_G(e)$, and $\mathrm{source}_{G'}((e, i)) = e$, if $i > 0$;
- $\mathrm{target}_{G'}((e, i)) = e$, if $i \leq 0$, and $\mathrm{target}_{G'}((e, i)) = v$, if $i > 0$ and v is the ith element of $\mathrm{target}_G(e)$;
- $\mathrm{lv}_{G'} = \mathrm{lv}_G \cup \mathrm{le}_G$;
- $\mathrm{le}((e, i)) = i$

Conversely, to any bipartite labeled graph such that the two partitions V_1 and V_2 have labels in disjoint sets L_1 and L_2, respectively, we can associate a hypergraph over (L_1, L_2), or one over (L_2, L_1), respectively, depending whether V_1 are chosen to be nodes and V_2 edges, or the converse.

G is a *graph jungle over* Σ if it is the graph representation of some jungle over Σ. Graph jungles can be characterized as follows:

Proposition 1. *Given a signature $\Sigma = (S, F)$, a graph G over $(S \cup F, Int)$ is a graph jungle over Σ iff:*

*0. G is bipartite, partitions given by nodes with labels in S—**sort nodes**—, and F—**operation nodes**—;*

1. every operation node labeled by $f : s_1 \cdots s_n \to s$ is

   (i) the target of exactly one edge, labeled with 0 and having its source labeled with s, and

   (ii) the source of n edges having distinct labels in $\{1, \cdots, n\}$, such that $\mathrm{lv}(\mathrm{target}(e)) = s_{\mathrm{le}(e)}$ for each such edge e;

2. every sort node has at most one outward edge; and

3. G is acyclic.

For example, the bipartite graph representation of the jungle G in Figure 2.3 (associated to the term $h(f(a), 0, 1)$) is represented by the graph in Figure 2.4(b). To avoid cluttering, and since there is no danger of confusion, we choose to omit the label 0 when representing the graphs. The above definitions and results carry on, but must be adjusted to address the fact that hypergraph edges are translated into operation nodes in addition to the edges. Let us quickly revise the definitions and results.

The variables of a graph jungle are sort nodes without outward edges: $VAR_G = \{v \in V_G \mid \mathrm{lv}(v) \in S \text{ and } \mathit{outdegree}_G(v) = 0\}$. $\mathit{term}_G(v)$ is defined on(ly) on sort nodes by:

$$\mathit{term}_G(v_s) = \begin{cases} v_s, \text{ if } v_s \in VAR_G \\ \sigma(t_1, \ldots, t_n), \text{ if } \{v_e\} = \mathrm{target}(\mathrm{source}^{-1}(v_s)), \\ \quad \mathrm{le}(v_e) = \sigma : s_1 \ldots s_n \to s, \text{ and} \\ \quad t_i = \mathit{term}_G(\mathrm{target}(e)) \\ \quad \text{where } \mathrm{source}(e) = v_e \text{ and } \mathrm{le}(e) = i \end{cases}$$

The notions of root and variable-collapsing tree do not change. For the graph representation of evaluation rules, the only thing that needs to be adjusted is that if the rule is non-collapsing, then K is now obtained from L by removing the operation node linked to root of L and all its adjacent edges. Again, the gluing conditions are satisfied for any matching (graph) morphism into a graph jungle G. The argument for the identification condition carries on. The dangling condition is also satisfied by the fact that in any graph representing a jungle (including L and G), any operation node with label $\sigma : s_1 \ldots s_n \to s$ has exactly $n+1$ adjacent edges, which are all present in L. Evaluations steps being performed according to the DPO approach now in the context of concrete graphs, Theorem 1 can be recast as follows:

Theorem 2. *Let p be a graph evaluation rule for a rewrite rule ρ, and let G be a graph jungle.*

1. *Evaluation steps preserve graph jungles, i.e., if $G \overset{p}{\Rightarrow} H$ then H is a graph jungle;*
2. *If $G \overset{p}{\Rightarrow} H$, then for each $v \in V_G$ $\mathit{term}_G(v) \overset{\rho^n}{\Longrightarrow} \mathit{term}_H(r^*(v))$*
3. *If ρ is left-linear and $\mathit{term}_G(v) \overset{\rho}{\Rightarrow} t'$ for some $v \in V_G$, then there exists H such that $G \overset{p}{\Rightarrow} H$.*

2.4 Discussion

Although the theory of graph rewriting has early on shown the potential for parallelism with sharing of context, the existing term-graph rewriting approaches seem to primarily aim at efficiency: rewrite common subterms only once; while the additional potential for concurrency seems to be regarded more like a bonus. More specifically, they do not attempt to use the context-sharing information for enhancing the potential for concurrency, and thus they are not able to specify more concurrent behaviors than rewriting logic already provides. Moreover, while subterm sharing might be a wonderful idea for functional programs, rewriting with subterm sharing in the presence of non-determinism can actually miss behaviors which would have been captured in regular term rewriting. Therefore, the term-graph rewriting approaches cannot be used directly to achieve the concurrency we would like for the $\mathbb{K}$ framework. In Chapter 6 we will show how the term graph rewriting approach presented in this section can be relaxed to allow the representation of $\mathbb{K}$ rules and to capture the intended concurrency for $\mathbb{K}$ rewriting.

Chapter 3

A Rewriting Logic Approach to Operational Semantics

The purpose of this chapter is to show not only that Rewriting Logic is amenable for defining programming languages, but also that it can capture with minimal representational distance the existing major operational semantics definitional techniques. Therefore, language designers can use their favorite technique to define languages within rewriting logic, thus benefiting of the generic execution and analysis tools available for rewriting logic definitions. This chapter can be considered as a basis for the following development of the dissertation, as the $\mathbb{K}$ framework builds upon the lessons learned from representing these techniques in rewriting logic, in the attempt to develop the ideal rewriting-based language definitional framework. A particular emphasis is put on the continuation-based definitions and their representation in rewriting logic (Section 3.7), which can be seen as a precursor of the $\mathbb{K}$ technique.

An immediate advantage of defining a language as a theory in an existing logic (as opposed to defining it as a new logic in which one can derive precisely the intended computations), is that one can use the entire arsenal of techniques and tools developed for the underlying logic to obtain corresponding techniques and tools for the particular programming language defined as a theory. Since rewriting logic is a computational logical framework, “execution” of programs becomes logical deduction. That means that one can formally analyze programs or their executions directly within the semantic definition of their programming language. Moreover, generic analysis tools for rewrite logic specifications can translate into analysis tools for the defined programming languages. Maude [34] provides an execution and debugging platform, a BFS state-space exploration, and an LTL model checker [48], as well as an inductive theorem prover [33] for rewrite logic theories; these translate immediately into corresponding BFS reachability analysis, LTL model checking tools, and theorem provers for the defined languages for free. For example, these generic tools were used to derive a competitive model checker [57], and a Hoare logic verification tool [152] for the Java programming language.

Any logical framework worth its salt should be evaluated in terms of its expressiveness and flexibility. Regarding expressiveness, a very pertinent question is: how does RLS express various approaches to operational semantics? In particular, how well can it express various approaches in the SOS tradition? This chapter proposes an answer to these questions. Partial answers, giving detailed comparisons with specific approaches have appeared elsewhere. For example, [99] and [175] provide comparisons with standard SOS [136]; [108] compares RLS with both standard SOS and Mosses' modular structural operational semantics (MSOS) [123]; and [103] compares RLS with chemical abstract machine (Cham) semantics [15]. However, no comprehensive comparison encompassing most approaches in the SOS tradition has been given to date. To make our ideas more concrete, in we here use a simple programming language, show how it is expressed in each different definitional style, and how that style can be faithfully captured as a rewrite theory in the RLS framework. We furthermore prove correctness theorems showing the faithfulness of the RLS representation for each style. Even though we exemplify the techniques and proofs with a simple language for concreteness' sake, the process of representing each definitional style in RLS and proving the faithfulness of the representation is completely general and mechanical, and in some cases like MSOS has already been automated [28]. The range of styles covered includes: big-step (or natural) SOS semantics; small-step SOS semantics; MSOS semantics; context-sensitive reduction semantics; continuation-based semantics; and Cham semantics.

Concerning flexibility, we show that each language definitional style can be used as a particular definitional methodology within rewriting logic. However, representing a language definitional style in rewriting logic does not make that style more flexible: the technique representing it within rewriting logic inherits the same benefits and limitations that the original definitional style had. Nevertheless, rewriting logic also captures for each framework their best features.

Challenges

Any logical framework for operational semantics of programming languages has to meet strong challenges. We list below some of the challenges that we think any such framework must meet to be successful. We do so in the form of questions from a skeptical language designer, following each question by our answer on how the RLS framework meets each challenge question.

1. **Q**: *Can you handle standard SOS?*

   A: As illustrated in Sections 3.3 and 3.4 for our example language, and also shown in [99, 175, 108] using somewhat different representations, both big-step and small-step SOS definitions can be expressed as rewrite theories in RLS. Furthermore, as illustrated in Section 3.5 for our language, and systematically explained in [108], MSOS definitions can also be faithfully captured in RLS.

2. **Q**: *Can you handle context-sensitive reduction?*

   A: There are two different questions implicit in the above question: (i) how are approaches to reduction semantics based on evaluation contexts (e.g., [180]) represented as rewrite theories? and (ii) how does RLS support context-sensitive rewriting in general? We answer subquestion (i) in Section 3.6, where we illustrate with our example language a general method to handle evaluation contexts in RLS. Regarding subquestion (ii), it is worth pointing out that, unlike standard SOS, because of its congruence rule, rewriting logic *is* context-sensitive and, furthermore, using *frozen* operator arguments, reduction can be blocked on selected arguments (see Section 2.1). Rewriting logic provides no support for matching the context in which a rewrite rule applies and to modify that context at will, which is one of the major strengths of reduction semantics with evaluation contexts. If that is what one wants to do, then one should use the technique in Section 8 instead.

3. **Q**: *Can you handle higher-order syntax?*

   A: Rewriting logic, cannot *directly* handle higher-order syntax with bindings and reasoning modulo α-conversion. However, it is well-known that higher-order syntax admits first-order representations, such as explicit substitution calculi and de Bruijn numbers, e.g., [1, 14, 167]. However, the granularity of computations is changed in these representations; for example, a single β-reduction step now requires additional rewrites to perform substitutions. In rewriting logic, because computation steps happen in equivalence classes modulo equations, the granularity of computation remains the same, because all explicit substitution steps are equational. Furthermore, using explicit substitution calculi such as CINNI [167], all this can be done automatically, keeping the original higher-order syntax not only for λ-abstraction, but also for any other name-binding operators.

4. **Q**: *What about continuations?*

   A: Continuations [59, 140] are traditionally understood as higher-order functions. Using the above-mentioned explicit calculi they can be represented in a first-order way. In Section 3.7 we present an alternative view of continuations that is intrinsically first-order in the style of, e.g., Wand [179], and prove a theorem showing that, for our language, first-order continuation semantics and context-sensitive reduction semantics are equivalent as rewrite theories in RLS. We also emphasize that in a computational logical framework, continuations are not just a means of implementing a language, but can be used to actually *define* the semantics of a language.

5. **Q**: *Can you handle concurrency?*

   A: One of the strongest points of rewriting logic is precisely that it is a logical framework for concurrency that can naturally express many different concurrency models and calculi [104, 98]. Unlike standard SOS, which forces an interleaving semantics, true concurrency is directly supported.

We illustrate this in Section 3.8, where we explain how Cham semantics is a particular style within RLS.

6. **Q**: *How expressive is the framework?*

   A: RLS is truly a framework, which does not force on the user any particular definitional style. This is in fact the main purpose of this chapter, achieved by showing how quite different definitional styles can be faithfully captured in RLS. Furthermore, as already mentioned, RLS can express a wide range of concurrent languages and calculi very naturally, without artificial encodings. Finally, real-time and probabilistic systems can likewise be naturally expressed [127, 2, 107].

7. **Q**: *Is anything lost in translation?*

   A: This is a very important question, because the worth of a logical framework does not just depend on whether something can be represented "in principle," but on *how well* it is represented. The key point is to have a very small *representational distance* between what is represented and the representation. Turing machines have a huge representational distance and are not very useful for semantic definitions exactly for that reason. Typically, RLS representations have what we call "ϵ-representational distance", that is, what is represented and its representation differ at most in inessential details. We will show that all the RLS representations for the different definitional styles we consider have this feature. In particular, we show that the original computations are represented in a one-to-one fashion. Furthermore, the good features of each style are preserved. For example, the RLS representation of MSOS is as modular as MSOS itself.

8. **Q**: *Is the framework purely operational?*

   A: Although RLS definitions are executable in a variety of systems supporting rewriting, rewriting logic itself is a complete logic with both a computational proof theory and a model-theoretic semantics. In particular, any rewrite theory has an *initial model*, which provides inductive reasoning principles to prove properties. What this means for RLS representations of programming languages is that they have *both* an operational rewriting semantics, and a mathematical model-theoretic semantics. For sequential languages, this model-theoretic semantics is an initial-algebra semantics. For concurrent languages, it is a truly concurrent initial-model semantics. In particular, this initial model has an associated Kripke structure in which temporal logic properties can be both interpreted and model-checked [106].

9. **Q**: *What about performance?*

   A: RLS as such is a *mathematical framework*, not bound to any particular rewrite engine implementation. However, because of the existence of a range of high-performance systems supporting rewriting, RLS semantic

> definitions can *directly* be used as interpreters when executed in such systems. Performance will then depend on both the system chosen and the particular definitional style used. The RLS theory might need to be slightly adapted to fit the constraints of some of the systems. In Section 3.9 we present experimental performance results for the execution of mechanically generated interpreters from RLS definitions for our example language using various systems for the different styles considered. Generally speaking, these performance figures are very encouraging and show that good performance interpreters can be directly obtained from RLS semantic definitions.

Benefits

Our skeptical language designer could still say,

> *So what? What do I need a logical framework for?*

It may be appropriate to point out that he/she is indeed free to choose, or not choose, any framework. However, using RLS brings some intrinsic benefits that might, after all, not be unimportant to him/her.

Besides the benefits already mentioned in our answers to questions in Section 3, one obvious benefit is that, since rewriting logic is a *computational* logic, and there are state-of-the-art system implementations supporting it, there is *no gap* between an RLS operational semantics definition and an implementation. This is an obvious advantage over the typical situation in which one gives semantics to a language on paper following one or more operational semantics styles, and then, to "execute" it, one implements an interpreter for the desired language following "in principle" its operational semantics, but using one's favorite programming language and specific tricks and optimizations for the implementation. This creates a nontrivial gap between the formal operational semantics of the language and its implementation.

A second, related benefit, is the possibility of *rapid prototyping* of programming language designs. That is, since language definitions can be directly executed, the language designer can experiment with various new features of a language by just defining them, eliminating the overhead of having to implement them as well in order to try them out. As experimentally shown in Section 3.9, the resulting prototypes can have reasonable performance, sometimes faster than that of well-engineered interpreters.

A broader, third benefit, of which the above two are special cases, is the availability of *generic tools* for: (i) syntax; (ii) execution; and (iii) formal analysis. The advantages of generic execution tools have been emphasized above. Regarding (i), languages such as ASF+SDF [171] and Maude [34] support user-definable syntax for RLS theories, which for language design has two benefits. First, it gives a prototype parser for the defined language essentially for free; and second, the language designer can use directly the concrete syntax of the desired language features, instead of the more common, but harder to read, abstract syntax tree (AST) representation. Regarding (iii), there is a wealth of theorem proving and model checking tools for rewriting/equational-based specifications, which can be used directly to prove properties about language definitions. The fact that these formal analysis tools are generic, should not fool one into thinking that they *must* be inefficient. For example, the LTL model checkers obtained for free in Maude from the RLS definitions of Java and the JVM compare favorably in performance with state-of-the-art Java model checkers [57, 58].

A fourth benefit comes from the availability in RLS of what we call the "abstraction dial," which can be used to reach a good balance between abstraction and computational observability in semantic definitions. The point is which *computational granularity* is appropriate. A small-step semantics opts for very fine-grained computations. But this is not necessarily the only or the best option for all purposes. The fact that an RLS theory's axioms include both equations and rewrite rules provides the useful "abstraction dial," because rewriting takes place *modulo* the equations. That is, computations performed by equations are abstracted out and become *invisible*. This has many advantages, as explained in [111]. For example, for formal analysis it can provide a huge reduction in search space for model checking purposes, which is one of the reasons why the Java model checkers described in [57, 58] perform so well. For language definition purposes, this again has many advantages. For example, in Sections 3.4 and 3.3, we use equations to define the semantic infrastructure (stores, etc.) of SOS definitions; in Section 3.6 equations are also used to hide the extraction and application of evaluation contexts, which are "meta-level" operations, carrying no computational meaning; in Section 3.7, equations are also used to decompose the evaluation tasks into their corresponding subtasks; finally, in Sections 3.5 and 3.8, equations of associativity and commutativity are used to achieve modularity of language definitions, and true concurrency in chemical-soup-like computations, respectively. The point in all these cases is always the same: to achieve the right granularity of computations.

The remainder of this chapter is organized as follows. Section 3.1 introduces a simple imperative language that will be used to discuss the various definitional styles and their RLS representations. Section 3.2 gathers some useful facts about the algebraic representation of stores. Section 3.3 addresses the big-step semantics. Section 3.4 discusses the small-step SOS, followed by Section 3.5 which discusses modular SOS. Sections 3.6 and 3.7 show how reduction semantics with evaluation contexts and continuation-based semantics can respectively be faithfully captured as RLS theories, as well as results discussing the relationships between these two interesting semantics. Section 3.8 presents the Cham semantics. Section 3.9 shows that the RLS theories corresponding to the various definitional styles provide relatively efficient interpreters to the defined languages when executed on systems that provide support for term rewriting.

$$\begin{array}{rcl}
\mathit{AExp} & ::= & \mathit{Name} \mid \mathit{Int} \mid \mathit{AExp}\ \mathtt{+}\ \mathit{AExp} \mid \mathit{AExp}\ \mathtt{-}\ \mathit{AExp} \mid \mathit{AExp}\ \mathtt{*}\ \mathit{AExp} \mid \\
 & & \mathit{AExp}\ \mathtt{/}\ \mathit{AExp} \mid \mathtt{++}\ \mathit{Name} \\
\mathit{BExp} & ::= & \mathit{Bool} \mid \mathit{AExp}\ \mathtt{<=}\ \mathit{AExp} \mid \mathit{AExp}\ \mathtt{>=}\ \mathit{AExp} \mid \mathit{AExp}\ \mathtt{==}\ \mathit{AExp} \mid \\
 & & \mathit{BExp}\ \mathtt{and}\ \mathit{BExp} \mid \mathit{BExp}\ \mathtt{or}\ \mathit{BExp} \mid \mathtt{not}\ \mathit{BExp} \\
\mathit{Stmt} & ::= & \mathtt{skip} \mid \mathit{Name}\ \mathtt{:=}\ \mathit{AExp} \mid \mathit{Stmt}\ \mathtt{;}\ \mathit{Stmt} \mid \{\mathit{Stmt}\} \mid \\
 & & \mathtt{if}\ \mathit{BExp}\ \mathtt{then}\ \mathit{Stmt}\ \mathtt{else}\ \mathit{Stmt} \mid \mathtt{while}\ \mathit{BExp}\ \mathit{Stmt} \mid \mathtt{halt}\ \mathit{AExp} \\
\mathit{Pgm} & ::= & \mathit{Stmt}\ \mathtt{.}\ \mathit{AExp}
\end{array}$$

Figure 3.1: A Small Imperative Language

3.1 A Simple Imperative Language

To illustrate the various operational semantics styles, we have chosen a small imperative language having arithmetic and boolean expressions with side effects (increment expression), short-circuited boolean operations, assignment, conditional, while loop, sequential composition, blocks and halt. The syntax of the language is depicted in Figure 3.1.

The semantics of $\mathtt{++}x$ is that of incrementing the value of x in the store and then returning the new value. The increment is done at the moment of evaluation, not after the end of the statement as in C/C++. Also, we assume short-circuit semantics for boolean operations.

This BNF syntax is entirely equivalent to an algebraic order-sorted signature having one (mixfix) operation definition per production, terminals giving the name of the operation and non-terminals the arity. For example, the production defining if-then-else can be seen as an algebraic operation

$$\mathtt{if\_then\_else\_} : \mathit{BExp} \times \mathit{Stmt} \times \mathit{Stmt} \to \mathit{Stmt}$$

We will use the following conventions for variables throughout the remainder of the chapter: $X \in \mathit{Name}$, $A \in \mathit{AExp}$, $B \in \mathit{BExp}$, $\mathit{St} \in \mathit{Stmt}$, $P \in \mathit{Pgm}$, $I \in \mathit{Int}$, $T \in \mathit{Bool} = \{\mathit{true}, \mathit{false}\}$, any of them primed or indexed.

The next sections will use this simple language and will present definitions in various operational semantics styles (big step, small step SOS, MSOS, reduction using evaluation contexts, continuation-based, and Cham), as well the corresponding RLS representation of each definition. We will also characterize the relation between the RLS representations and their corresponding definitional style counterparts, pointing out some strengths and weaknesses for each style. The reader is referred to [89, 136, 123, 180, 15] for further details on the described operational semantics styles.

We assume equational definitions for basic operations on booleans and integers, and assume that any other theory defined here includes them. One of the reasons why we wrapped booleans and integers in the syntax is precisely to distinguish them from the corresponding values, and thus to prevent the "builtin" equations from reducing expressions like $3 + 5$ directly in the syntax (we wish to have full control over the computational granularity of the language), since our RLS representations aim to have the same computational granularity of each of the different styles represented.

3.2 Store

Unlike in various operational semantics, which usually abstract stores as functions, in rewriting logic we explicitly define the store as an algebraic datatype: a store is a set of bindings from variables to values, together with two operations on them, one for retrieving a value, another for setting a value. We show that well-formed stores correspond to partially defined functions. Having this abstraction in place, we can regard them as functions for all practical purposes from now on.

To define the store, we assume a pairing "binding" constructor "$\_ \mapsto \_$", associating values to variables[1], and an associative and commutative union operation "$\_\_$" with $\emptyset$ as its identity to put together such bindings. The equational definition $\mathcal{E}_{Store}$ of operations $\_[\_]$ to retrieve the value of a variable in the store and $\_[\_ \leftarrow \_]$ to update the value of a variable is given by the following equations, that operate modulo the associativity and commutativity of $\_\_$:

$$
\begin{array}{l}
(S\ X \mapsto I)[X] = I \\
(S\ X \mapsto I)[X'] = S[X'] \textbf{ if } X \neq X' \\
(S\ X \mapsto I)[X \leftarrow I'] = S\ X \mapsto I' \\
(S\ X \mapsto I)[X' \leftarrow I'] = S[X' \leftarrow I']\ X \mapsto I \textbf{ if } X \neq X' \\
\emptyset[X \leftarrow I] = X \mapsto I
\end{array}
$$

Note the $X \neq X$ appearing as a condition is not a negative condition, but rather a Boolean predicate, which can be equationally defined for any constructor-based type such as the type of variables, for example. Since these definitions are equational, from a rewriting logic semantic point of view they are invisible: transitions are performed *modulo* these equations. This way we can maintain a coarser computational granularity, while making use of auxiliary functions defined using equations. Although it might seem that, by using built-ins as integers and names, one cannot guarantee the existence of the initial model, notice that all the "built-ins" appearing in these definitions (names, booleans, integers) are definable as initial models of corresponding equational theories. And indeed, when performing formal proofs, one will make use of these equational definitions of the so-called built-ins. A store s is *well-formed* if $\mathcal{E}_{Store} \vdash s = x_1 \mapsto i_1 \ldots x_n \mapsto i_n$ for some $x_j \in \mathit{Name}$ and $i_j \in \mathit{Int}$, for all $1 \leq j \leq n$, such that $x_i \neq x_j$ for any $i \neq j$. We say that a store s is equivalent to a finite partial function $\sigma : \mathit{Name} \overset{\circ}{\to} \mathit{Int}$, written $s \simeq \sigma$, if s is well-formed and

[1] In general, one would have both an *environment*, and a *store*, with variables mapped to locations in the environment, and locations mapped to values in the store. However, for the sake of brevity, and given the simplicity of our example language, we do not use environments and map variables directly to values in the store.

behaves as σ, that is, if for any $x \in \mathit{Name}, i \in \mathit{Int}$, $\sigma(x) = i$ iff $\mathcal{E}_{\mathit{Store}} \vdash s[x] = i$. We recall that, given a store-function σ, $\sigma[i/x]$ is defined as the function mapping x to i and other variables y to $\sigma(y)$.

Proposition 2. *Let $x, x' \in \mathit{Name}$, $i, i' \in \mathit{Int}$, $s, s' \in \mathit{Store}$ and finite partial functions $\sigma, \sigma' : \mathit{Name} \overset{\circ}{\rightarrow} \mathit{Int}$.*

1. *$\emptyset \simeq \perp$ where $\perp$ is the function undefined everywhere.*
2. *$(s\ x \mapsto i) \simeq \sigma$ implies that $s \simeq \sigma[\perp /x]$ where $\sigma[\perp /x]$ is defined as σ restricted to $\mathit{Dom}(\sigma) \setminus \{x\}$.*
3. *If $s \simeq \sigma$ then also $s[x \leftarrow i] \simeq \sigma[i/x]$.*

Proof.

1. Trivial, since $\mathcal{E}_{\mathit{Store}} \not\vdash \emptyset[x] = i$ for any $x \in \mathit{Name}, i \in \mathit{Int}$.
2. Let σ' be such that $s \simeq \sigma'$. We will prove that $\mathit{Dom}(\sigma') = \mathit{Dom}(\sigma) \setminus \{x\}$ and for any $x' \in \mathit{Dom}(\sigma')$, $\sigma'(x) = \sigma(x)$. Consider an arbitrary x'. If $x' = x$, then $\mathcal{E}_{\mathit{Store}} \not\vdash s[x'] = i'$ for any i', since otherwise we would have $\mathcal{E}_{\mathit{Store}} \vdash s = s'\ x \mapsto i'$ which contradicts the well-formedness of $s\ x \mapsto i$; therefore, σ' is not defined on x'. If $x' \neq x$, then $\mathcal{E}_{\mathit{Store}} \vdash s[x'] = (s\ x \mapsto i)[x']$, therefore σ' is defined on x' iff σ is defined on x', and if so $\sigma'(x') = \sigma(x')$.
3. Suppose $s \simeq \sigma$. We distinguish two cases —if σ is defined on x or if it is not. If it is, then let us say that $\sigma(x) = i'$; in that case we must have that $\mathcal{E}_{\mathit{Store}} \vdash s[x] = i'$ which can only happen if $\mathcal{E}_{\mathit{Store}} \vdash s = s'\ x \mapsto i'$, whence $\mathcal{E}_{\mathit{Store}} \vdash s[x \leftarrow i] = s'\ x \mapsto i$. Let x' be an arbitrary variable in Name. If $x' = x$ then

$$\mathcal{E}_{\mathit{Store}} \vdash (s[x \leftarrow i])[x'] = (s'\ x \mapsto i)[x'] = i.$$

If $x' \neq x$ then

$$\mathcal{E}_{\mathit{Store}} \vdash (s[x \leftarrow i])[x'] = (s'\ x \mapsto i)[x'] = s'[x'] = (s'\ x \mapsto i')[x'] = s[x'].$$

If σ is not defined for x, it means that $\mathcal{E}_{\mathit{Store}} \not\vdash s[x] = i$ for any i, whence $\mathcal{E}_{\mathit{Store}} \not\vdash s = s'\ x \mapsto i$. If $\mathcal{E}_{\mathit{Store}} \vdash s = \emptyset$ then we are done, since $\mathcal{E}_{\mathit{Store}} \vdash (x \mapsto i)[x'] = i'$ iff $x = x'$ and $i = i'$. If $\mathcal{E}_{\mathit{Store}} \not\vdash s = \emptyset$, it must be that $\mathcal{E}_{\mathit{Store}} \vdash s = x_1 \mapsto i_1 \ldots x_n \mapsto i_n$ with $x_i \neq x$. This leads to $\mathcal{E}_{\mathit{Store}} \vdash s[x \leftarrow i] = \cdots = (x_1 \mapsto i_1 \ldots x_i \mapsto i_i)[x \leftarrow i](x_{i+1} \mapsto i_{i+1} \ldots x_n \mapsto i_n) = \cdots = \emptyset[x \leftarrow i]s = (x \mapsto i)s = s(x \mapsto i)$.

□

In the following, we will use symbols S, S', S_1, ..., to denote variables of type *Store*.

Types of configurations: $\langle \mathit{Int}, \mathit{Store}\rangle$, $\langle \mathit{Bool}, \mathit{Store}\rangle$, $\langle \mathit{AExp}, \mathit{Store}\rangle$, $\langle \mathit{BExp}, \mathit{Store}\rangle$, $\langle \mathit{Stmt}, \mathit{Store}\rangle$, $\langle \mathit{Pgm}\rangle$, $\langle \mathit{Int}\rangle$.

$$\frac{\cdot}{\langle I,\sigma\rangle \Downarrow \langle I,\sigma\rangle}$$

$$\frac{\cdot}{\langle X,\sigma\rangle \Downarrow \langle \sigma(X),\sigma\rangle}$$

$$\frac{\cdot}{\langle \texttt{++}X,\sigma\rangle \Downarrow \langle I,\sigma[I/X]\rangle},\ \texttt{if}\ \ I = \sigma(X) + 1$$

$$\frac{\langle A_1,\sigma\rangle \Downarrow \langle I_1,\sigma_1\rangle,\ \ \langle A_2,\sigma_1\rangle \Downarrow \langle I_2,\sigma_2\rangle}{\langle A_1 + A_2,\sigma\rangle \Downarrow \langle I_1 +_{\mathit{Int}} I_2,\sigma_2\rangle}$$

$$\frac{\cdot}{\langle T,\sigma\rangle \Downarrow \langle T,\sigma\rangle}$$

$$\frac{\langle A_1,\sigma\rangle \Downarrow \langle I_1,\sigma_1\rangle,\ \ \langle A_2,\sigma_1\rangle \Downarrow \langle I_2,\sigma_2\rangle}{\langle A_1 \texttt{<=} A_2,\sigma\rangle \Downarrow \langle (I_1 \leq_{\mathit{Int}} I_2),\sigma_2\rangle}$$

$$\frac{\langle B_1,\sigma\rangle \Downarrow \langle \mathit{true},\sigma_1\rangle,\ \ \langle B_2,\sigma_1\rangle \Downarrow \langle T,\sigma_2\rangle}{\langle B_1\ \texttt{and}\ B_2,\sigma\rangle \Downarrow \langle T,\sigma_2\rangle}$$

$$\frac{\langle B_1,\sigma\rangle \Downarrow \langle \mathit{false},\sigma_1\rangle}{\langle B_1\ \texttt{and}\ B_2,\sigma\rangle \Downarrow \langle \mathit{false},\sigma_1\rangle}$$

$$\frac{\langle B,\sigma\rangle \Downarrow \langle T,\sigma'\rangle}{\langle \texttt{not}\ B,\sigma\rangle \Downarrow \langle \mathit{not}(T),\sigma'\rangle}$$

$$\frac{\cdot}{\langle \mathit{skip},\sigma\rangle \Downarrow \langle \sigma\rangle}$$

$$\frac{\langle A,\sigma\rangle \Downarrow \langle I,\sigma'\rangle}{\langle X \texttt{:=} A,\sigma\rangle \Downarrow \langle \sigma'[I/X]\rangle}$$

$$\frac{\langle St_1,\sigma\rangle \Downarrow \langle \sigma''\rangle,\ \ \langle St_2,\sigma''\rangle \Downarrow \langle \sigma'\rangle}{\langle St_1; St_2,\sigma\rangle \Downarrow \langle \sigma'\rangle}$$

$$\frac{\langle St,\sigma\rangle \Downarrow \langle \sigma'\rangle}{\langle \{St\},\sigma\rangle \Downarrow \langle \sigma'\rangle}$$

$$\frac{\langle B,\sigma\rangle \Downarrow \langle \mathit{true},\sigma_1\rangle,\ \ \langle St_1,\sigma_1\rangle \Downarrow \langle \sigma_2\rangle}{\langle \texttt{if}\ B\ \texttt{then}\ St_1\ \texttt{else}\ St_2,\sigma\rangle \Downarrow \langle \sigma_2\rangle}$$

$$\frac{\langle B,\sigma\rangle \Downarrow \langle \mathit{false},\sigma_1\rangle,\ \ \langle St_2,\sigma_1\rangle \Downarrow \langle \sigma_2\rangle}{\langle \texttt{if}\ B\ \texttt{then}\ St_1\ \texttt{else}\ St_2,S\rangle \Downarrow \langle \sigma_2\rangle}$$

$$\frac{\langle B,\sigma\rangle \Downarrow \langle \mathit{false},\sigma'\rangle}{\langle \texttt{while}\ B\ St,\sigma\rangle \Downarrow \langle \sigma'\rangle}$$

$$\frac{\langle B,\sigma\rangle \Downarrow \langle \mathit{true},\sigma_1\rangle,\ \langle St,\sigma_1\rangle \Downarrow \langle \sigma_2\rangle,\ \langle \texttt{while}\ B\ St,\sigma_2\rangle \Downarrow \langle \sigma'\rangle}{\langle \texttt{while}\ B\ St,\sigma\rangle \Downarrow \langle \sigma'\rangle}$$

$$\frac{\langle St,\emptyset\rangle \Downarrow \langle \sigma\rangle,\ \ \langle A,\sigma\rangle \Downarrow \langle I,\sigma'\rangle}{\langle St.A\rangle \Downarrow \langle I\rangle}$$

Figure 3.2: The *BigStep* language definition

$$
\begin{array}{rl}
\langle X, S\rangle \rightarrow \langle S[X], S\rangle & \\
\langle \texttt{++}X, S\rangle \rightarrow \langle I, S[X \leftarrow I]\rangle & \textbf{if}\, I = S[X]+1 \\
\langle A_1 + A_2, S\rangle \rightarrow \langle I_1 +_{Int} I_2, S_2\rangle & \textbf{if}\, \langle A_1, S\rangle \rightarrow \langle I_1, S_1\rangle \wedge \langle A_2, S_1\rangle \rightarrow \langle I_2, S_2\rangle \\
\hline
\langle A_1 \texttt{<=} A_2, S\rangle \rightarrow \langle (I_1 \leq_{Int} I_2), S_2\rangle & \textbf{if}\, \langle A_1, S\rangle \rightarrow \langle I_1, S_1\rangle \wedge \langle A_2, S_1\rangle \rightarrow \langle I_2, S_2\rangle \\
\langle B_1 \;\texttt{and}\; B_2, S\rangle \rightarrow \langle T, S_2\rangle & \textbf{if}\, \langle B_1, S\rangle \rightarrow \langle \mathit{true}, S_1\rangle \wedge \langle B_2, S_1\rangle \rightarrow \langle T, S_2\rangle \\
\langle B_1 \;\texttt{and}\; B_2, S\rangle \rightarrow \langle \mathit{false}, S_1\rangle & \textbf{if}\, \langle B_1, S\rangle \rightarrow \langle \mathit{false}, S_1\rangle \\
\langle \texttt{not}\; B, S\rangle \rightarrow \langle \mathit{not}(T), S'\rangle & \textbf{if}\, \langle B, S\rangle \rightarrow \langle T, S'\rangle \\
\hline
\langle \mathit{skip}, S\rangle \rightarrow \langle S\rangle & \\
\langle X \texttt{:=} A, S\rangle \rightarrow \langle S'[X \leftarrow I]\rangle & \textbf{if}\, \langle A, S\rangle \rightarrow \langle I, S'\rangle \\
\langle \mathit{St}_1; \mathit{St}_2, S\rangle \rightarrow \langle S'\rangle & \textbf{if}\, \langle \mathit{St}_1, S\rangle \rightarrow \langle S''\rangle \wedge \langle \mathit{St}_2, S''\rangle \rightarrow \langle S'\rangle \\
\langle \{\mathit{St}\}, S\rangle \rightarrow \langle S'\rangle & \textbf{if}\, \langle \mathit{St}, S\rangle \rightarrow \langle S'\rangle \\
\langle \texttt{if}\; B \;\texttt{then}\; \mathit{St}_1 \;\texttt{else}\; \mathit{St}_2, S\rangle \rightarrow \langle S_2\rangle & \textbf{if}\, \langle B, S\rangle \rightarrow \langle \mathit{true}, S_1\rangle \wedge \langle \mathit{St}_1, S_1\rangle \rightarrow \langle S_2\rangle \\
\langle \texttt{if}\; B \;\texttt{then}\; \mathit{St}_1 \;\texttt{else}\; \mathit{St}_2, S\rangle \rightarrow \langle S_2\rangle & \textbf{if}\, \langle B, S\rangle \rightarrow \langle \mathit{false}, S_1\rangle \wedge \langle \mathit{St}_2, S_1\rangle \rightarrow \langle S_2\rangle \\
\langle \texttt{while}\; B\; \mathit{St}, S\rangle \rightarrow \langle S'\rangle & \textbf{if}\, \langle B, S\rangle \rightarrow \langle \mathit{false}, S'\rangle \\
\langle \texttt{while}\; B\; \mathit{St}, S\rangle \rightarrow \langle S'\rangle & \textbf{if}\, \langle B, S\rangle \rightarrow \langle \mathit{true}, S_1\rangle \wedge \langle \mathit{St}, S_1\rangle \rightarrow \langle S_2\rangle \\
 & \qquad \wedge \langle \texttt{while}\; B\; \mathit{St}, S_2\rangle \rightarrow \langle S'\rangle \\
\hline
\langle \mathit{St}.A\rangle \rightarrow \langle I\rangle & \textbf{if}\, \langle \mathit{St}, \emptyset\rangle \rightarrow \langle S\rangle \wedge \langle A, S\rangle \rightarrow \langle I, S'\rangle \\
\hline
\end{array}
$$

Figure 3.3: $\mathcal{R}_{\mathit{BigStep}}$ rewriting logic theory

3.3 Big-Step Operational Semantics

Introduced as natural semantics in [89], also named relational semantics in [116], or evaluation semantics, big-step semantics is “the most denotational” of the operational semantics. One can view big-step definitions as definitions of functions interpreting each language construct in an appropriate domain.

Big step semantics can be easily represented within rewriting logic. For example, consider the big-step rule defining integer division:

$$\frac{\langle A_1, \sigma\rangle \Downarrow \langle I_1, \sigma_1\rangle, \langle A_2, \sigma_1\rangle \Downarrow \langle I_2, \sigma_2\rangle}{\langle A_1/A_2, \sigma\rangle \Downarrow \langle I_1/_{Int}I_2, \sigma_2\rangle}, \text{ if } I_2 \neq 0.$$

This rule can be automatically translated into the rewrite rule:

$$\langle A_1/A_2, S\rangle \rightarrow \langle I_1/_{Int}I_2, S_2\rangle \;\textbf{if}\; \langle A_1, S\rangle \rightarrow \langle I_1, S_1\rangle \wedge \langle A_2, S_1\rangle \rightarrow \langle I_2, S_2\rangle \wedge I_2 \neq 0$$

The complete big-step operational semantics definition for our simple language, except its `halt` statement (which is discussed at the end of this section), which we call *BigStep*, is presented in Figure 3.2. We choose to exclude from the presentation the semantics for constructs entirely similar to the ones presented, such as “−”, “∗”, “/” and “`or`”. To give a rewriting logic theory for the big-step semantics, one needs to first define the various configuration constructs, which are assumed by default in *BigStep*, as corresponding operations extending the signature. Then one can define the rewrite theory $\mathcal{R}_{\mathit{BigStep}}$ corresponding to the big-step operational semantics *BigStep* entirely automatically as shown by Figure 3.3. Note that, because the rewriting relation is reflexive, we did not need to add the reflexivity rules for boolean and integer values.

Due to the one-to-one correspondence between big-step rules in *BigStep* and rewrite rules in $\mathcal{R}_{BigStep}$, it is easy to prove by induction on the length of derivations the following result:

Proposition 3. *For any $p \in Pgm$ and $i \in Int$, the following are equivalent:*

1. $BigStep \vdash \langle p \rangle \Downarrow \langle i \rangle$
2. $\mathcal{R}_{BigStep} \vdash \langle p \rangle \to^1 \langle i \rangle$

Proof. A first thing to notice is that, since all rules involve configurations, rewriting can only occur at the top, thus the general application of term rewriting under contexts is disabled by the definitional style. Another thing to notice here is that all configurations in the right hand sides are normal forms, thus the transitivity rule for rewriting logic also becomes inapplicable. Suppose $s \in Store$ and $\sigma : Name \overset{\circ}{\to} Int$ such that $s \simeq \sigma$. We prove the following statements:

1. $BigStep \vdash \langle a, \sigma \rangle \Downarrow \langle i, \sigma' \rangle$ iff $\mathcal{R}_{BigStep} \vdash \langle a, s \rangle \to^1 \langle i, s' \rangle$ and $s' \simeq \sigma'$, for any $a \in AExp, i \in Int, \sigma' : Name \overset{\circ}{\to} Int$ and $s' \in Store$.

2. $BigStep \vdash \langle b, \sigma \rangle \Downarrow \langle t, \sigma' \rangle$ iff $\mathcal{R}_{BigStep} \vdash \langle b, s \rangle \to^1 \langle t, s' \rangle$ and $s' \simeq \sigma'$, for any $b \in AExp, t \in Bool, \sigma' : Name \overset{\circ}{\to} Int$ and $s' \in Store$.

3. $BigStep \vdash \langle st, \sigma \rangle \Downarrow \langle \sigma' \rangle$ iff $\mathcal{R}_{BigStep} \vdash \langle st, s \rangle \to^1 \langle s' \rangle$ and $s' \simeq \sigma'$, for any $st \in Stmt, \sigma' : Name \overset{\circ}{\to} Int$ and $s' \in Store$.

4. $BigStep \vdash \langle p \rangle \Downarrow \langle i \rangle$ iff $\mathcal{R}_{BigStep} \vdash \langle p \rangle \to^1 \langle i \rangle$, for any $p \in Pgm$ and $i \in Int$.

Each can be proved by induction on the size of the derivation tree. To avoid lengthy and repetitive details, we discuss the corresponding proof of only one language construct in each category:

1. $BigStep \vdash \langle x\texttt{++}, \sigma \rangle \Downarrow \langle i, \sigma[i/x] \rangle$ iff
   $i = \sigma(x) + 1$ iff
   $\mathcal{E}_{Store} \subseteq \mathcal{R}_{BigStep} \vdash i = s[x] + 1$ iff
   $\mathcal{R}_{BigStep} \vdash \langle x\texttt{++}, s \rangle \to^1 \langle i, s[x \leftarrow i] \rangle$.
   This completes the proof, since $s[x \leftarrow i] \simeq \sigma[i/x]$, by 3 in Proposition 2.

2. $BigStep \vdash \langle b_1 \ \texttt{and} \ b_2, \sigma \rangle \Downarrow \langle t, \sigma' \rangle$ iff
   $(BigStep \vdash \langle b_1, \sigma \rangle \Downarrow \langle false, \sigma' \rangle$ and $t = false$
   or $BigStep \vdash \langle b_1, \sigma \rangle \Downarrow \langle true, \sigma'' \rangle$ and $BigStep \vdash \langle b_2, \sigma'' \rangle \Downarrow \langle t, \sigma' \rangle$ $)$ iff
   $(\mathcal{R}_{BigStep} \vdash \langle b_1, s \rangle \to^1 \langle false, s' \rangle$, $s' \simeq \sigma'$ and $t = false$
   or $\mathcal{R}_{BigStep} \vdash \langle b_1, s \rangle \to^1 \langle true, s'' \rangle$, $s'' \simeq \sigma''$,
   $\mathcal{R}_{BigStep} \vdash \langle b_2, s'' \rangle \to^1 \langle t, \sigma' \rangle$ and $s' \simeq \sigma'$ $)$ iff
   $\mathcal{R}_{BigStep} \vdash \langle b_1 \ \texttt{and} \ b_2, s \rangle \to^1 \langle t, s' \rangle$ and $s' \simeq \sigma'$.

3. $BigStep \vdash \langle \texttt{while} \ b \ st, \sigma \rangle \Downarrow \langle \sigma' \rangle$ iff
   $(BigStep \vdash \langle b, \sigma \rangle \Downarrow \langle false, \sigma' \rangle$

or $\mathit{BigStep} \vdash \langle b, \sigma \rangle \Downarrow \langle \mathit{true}, \sigma_1 \rangle$
and $\mathit{BigStep} \vdash \langle st, \sigma_1 \rangle \Downarrow \langle \sigma_2 \rangle$
and $\mathit{BigStep} \vdash \langle \texttt{while}\ b\ st, \sigma_2 \rangle \Downarrow \langle \sigma' \rangle$) iff
($\mathcal{R}_{\mathit{BigStep}} \vdash \langle b, s \rangle \to^1 \langle \mathit{false}, s' \rangle$ and $s' \simeq \sigma'$
or $\mathcal{R}_{\mathit{BigStep}} \vdash \langle b, s \rangle \to^1 \langle \mathit{true}, s_1 \rangle$, $s_1 \simeq \sigma_1$
and $\mathcal{R}_{\mathit{BigStep}} \vdash \langle st, s_1 \rangle \to^1 \langle s_2 \rangle$, $s_2 \simeq \sigma_2$
and $\mathcal{R}_{\mathit{BigStep}} \vdash \langle \texttt{while}\ b\ st, s_2 \rangle \to^1 \langle s' \rangle$ and $s' \simeq \sigma'$) iff
$\mathcal{R}_{\mathit{BigStep}} \vdash \langle \texttt{while}\ b\ st, s \rangle \to^1 \langle s' \rangle$ and $s' \simeq \sigma'$.

4. $\mathit{BigStep} \vdash \langle st.a \rangle \Downarrow \langle i \rangle$ iff
$\mathit{BigStep} \vdash \langle st, \bot \rangle \Downarrow \langle \sigma \rangle$ and $\mathit{BigStep} \vdash \langle a, \sigma \rangle \Downarrow \langle i, \sigma' \rangle$ iff
$\mathcal{R}_{\mathit{BigStep}} \vdash \langle st, \emptyset \rangle \to^1 \langle s \rangle$, $s \simeq \sigma$, $\mathcal{R}_{\mathit{BigStep}} \vdash \langle a, s \rangle \to^1 \langle i, s' \rangle$ and $s' \simeq \sigma'$ iff
$\mathcal{R}_{\mathit{BigStep}} \vdash \langle st.a \rangle \to^1 \langle i \rangle$

This completes the proof. □

The only apparent difference between *BigStep* and $\mathcal{R}_{\mathit{BigStep}}$ is the different notational conventions they use. However, as the above theorem shows, there is a one-to-one correspondence also between their corresponding "computations" (or executions, or derivations). Therefore, $\mathcal{R}_{\mathit{BigStep}}$ actually *is* the big-step operational semantics *BigStep*, not an "encoding" of it. Note that, in order to be faithfully equivalent to *BigStep* computationally, $\mathcal{R}_{\mathit{BigStep}}$ lacks the main strength of rewriting logic that makes it an appropriate formalism for concurrency, namely, that rewrite rules can apply under any context and in parallel (here all rules are syntactically constrained so that they can only apply at the top, sequentially).

Strengths. Big-step operational semantics allows straightforward recursive definition. It can be easily and efficiently interpreted in any recursive, functional or logical framework. It is particularly useful for defining type systems.

Weaknesses. Due to its monolithic, single-step evaluation, it is hard to debug or trace big-step semantic definitions. If the program is wrong, no information is given about where the failure occurred. Divergence is not observable in the specified evaluation relation. It may be hard or impossible to model concurrent features. It is not modular, e.g., to add side effects to expressions, one must redefine the rules to allow expressions to evaluate to pairs (value-store). It is inconvenient (and non-modular) to define complex control statements; consider, for example, adding halt to the above definition —one needs to add a special configuration *halting*(I), and the following rules:

$$
\begin{array}{rll}
\langle \texttt{halt}\ A, S \rangle \to \mathit{halting}(I) & \textbf{if} & \langle A.S \rangle \to \langle I, S' \rangle \\
\langle St_1 ; St_2, S \rangle \to \mathit{halting}(I) & \textbf{if} & \langle St_1, S \rangle \to \mathit{halting}(I) \\
\langle \texttt{while}\ B\ St, S \rangle \to \mathit{halting}(I) & \textbf{if} & \langle B, S \rangle \to \langle S' \rangle \wedge \langle St, S' \rangle \to \mathit{halting}(I) \\
\langle St.A, S \rangle \to \langle I \rangle & \textbf{if} & \langle St, \emptyset \rangle \to \mathit{halting}(I)
\end{array}
$$

3.4 Small-Step Operational Semantics

Introduced by Plotkin in [136], also called transition semantics or reduction semantics, small-step semantics captures the notion of one computational step.

One inherent technicality involved in capturing small-step operational semantics as a rewrite theory in a one-to-one notational and computational correspondence is that the rewriting relation is by definition transitive, while the small-step relation is *not* transitive (its transitive closure can be defined a posteriori). Therefore, we need to devise a mechanism to "inhibit" rewriting logic's transitive and uncontrolled application of rules. An elegant way to achieve this is to view a small step as a modifier of the current configuration. Specifically, we consider "$\cdot$" to be a modifier on the configuration which performs a "small-step" of computation; in other words, we assume an operation $\cdot\_ : \mathit{Config} \to \mathit{Config}$. Then, a small-step semantic rule, e.g.,

$$\frac{\langle A_1, S\rangle \to \langle A_1', S'\rangle}{\langle A_1 + A_2, S\rangle \to \langle A_1' + A_2, S'\rangle}$$

is translated, again automatically, into a rewriting logic rule, e.g.,

$$\cdot\langle A_1 + A_2, S\rangle \to \langle A_1' + A_2, S'\rangle \textbf{ if } \cdot\langle A_1, S\rangle \to \langle A_1', S'\rangle$$

A similar technique is proposed in [108], but there two different types of configurations are employed, one standard and the other "tagged" with the modifier. However, allowing "$\cdot$" to be a modifier rather than a part of a configuration gives more flexibility to the specification —for example, one can specify that one wants two steps simply by putting two dots in front of the configuration.

The complete [2] small-step operational semantics definition for our simple language except its `halt` statement (which is discussed at the end of this section), which we call *SmallStep*, is presented in Figure 3.4. The corresponding small-step rewriting logic theory $\mathcal{R}_{\mathit{SmallStep}}$ is given in Figure 3.5. The language described here does not involve labels on rules like in the SOS of concurrent systems. For that, one would take an approach similar to that presented in Section 3.5, that is, pushing the labels back into the configurations.

As for big-step semantics, the rewriting under context deduction rule for rewriting logic is again inapplicable, since all rules act at the top, on configurations. However, in *SmallStep* it is not the case that all right hand sides are normal forms (this actually is a key feature of small-step semantics). The "$\cdot$" operator introduced in $\mathcal{R}_{\mathit{SmallStep}}$ prevents the unrestricted application of transitivity, and can be regarded as a token given to a configuration to allow it

[2] However, for brevity's sake, we don't present the semantics of similar constructs, such as $-, *, /$, `or`.

Types of configurations: $\langle \mathit{AExp}, \mathit{Store}\rangle$, $\langle \mathit{BExp}, \mathit{Store}\rangle$, $\langle \mathit{Stmt}, \mathit{Store}\rangle$, $\langle \mathit{Pgm}, \mathit{Store}\rangle$

$$\frac{\cdot}{\langle X,\sigma\rangle \rightarrow \langle(\sigma(X)),\sigma\rangle}$$

$$\frac{\cdot}{\langle \texttt{++}X,\sigma\rangle \rightarrow \langle I,\sigma[I/X]\rangle},\ \ \texttt{if}\ \ I=\sigma(X)+1$$

$$\frac{\langle A_1,\sigma\rangle \rightarrow \langle A_1',\sigma'\rangle}{\langle A_1+A_2,\sigma\rangle \rightarrow \langle A_1'+A_2,\sigma'\rangle} \qquad \frac{\langle A_2,\sigma\rangle \rightarrow \langle A_2',\sigma'\rangle}{\langle I_1+A_2,\sigma\rangle \rightarrow \langle I_1+A_2',\sigma'\rangle}$$

$$\frac{\cdot}{\langle I_1+I_2,\sigma\rangle \rightarrow \langle I_1+_{\mathit{Int}}I_2,\sigma\rangle}$$

$$\frac{\langle A_1,\sigma\rangle \rightarrow \langle A_1',\sigma'\rangle}{\langle A_1\texttt{<=}A_2,\sigma\rangle \rightarrow \langle A_1'\texttt{<=}A_2,\sigma'\rangle} \qquad \frac{\langle A_2,\sigma\rangle \rightarrow \langle A_2',\sigma'\rangle}{\langle I_1\texttt{<=}A_2,\sigma\rangle \rightarrow \langle I_1\texttt{<=}A_2',\sigma'\rangle}$$

$$\frac{\cdot}{\langle I_1\texttt{<=}I_2,\sigma\rangle \rightarrow \langle(I_1\leq_{\mathit{Int}}I_2),\sigma\rangle}$$

$$\frac{\langle B_1,\sigma\rangle \rightarrow \langle B_1',\sigma'\rangle}{\langle B_1\ \texttt{and}\ B_2,\sigma\rangle \rightarrow \langle B_1'\ \texttt{and}\ B_2,\sigma'\rangle} \qquad \frac{\langle B,\sigma\rangle \rightarrow \langle B',\sigma'\rangle}{\langle \texttt{not}\ B,\sigma\rangle \rightarrow \langle \texttt{not}\ B',\sigma'\rangle}$$

$$\frac{\cdot}{\langle \texttt{trueand}\ B_2,\sigma\rangle \rightarrow \langle B_2,\sigma\rangle} \qquad \frac{\cdot}{\langle \texttt{nottrue},\sigma\rangle \rightarrow \langle \texttt{false},\sigma\rangle}$$

$$\frac{\cdot}{\langle \texttt{falseand}\ B_2,\sigma\rangle \rightarrow \langle \texttt{false},\sigma\rangle} \qquad \frac{\cdot}{\langle \texttt{notfalse},\sigma\rangle \rightarrow \langle \texttt{true},\sigma\rangle}$$

$$\frac{\langle A,\sigma\rangle \rightarrow \langle A',\sigma'\rangle}{\langle X\texttt{:=}A,\sigma\rangle \rightarrow \langle X\texttt{:=}A',\sigma'\rangle} \qquad \frac{\langle \mathit{St}_1,\sigma\rangle \rightarrow \langle \mathit{St}_1',\sigma'\rangle}{\langle \mathit{St}_1;\mathit{St}_2,\sigma\rangle \rightarrow \langle \mathit{St}_1';\mathit{St}_2,\sigma'\rangle}$$

$$\frac{\cdot}{\langle X\texttt{:=}I,\sigma\rangle \rightarrow \langle \texttt{skip},\sigma[I/X]\rangle} \qquad \frac{\cdot}{\langle \texttt{skip};\mathit{St}_2,\sigma\rangle \rightarrow \langle \mathit{St}_2,\sigma\rangle}$$

$$\frac{\cdot}{\langle \{\mathit{St}\},\sigma\rangle \rightarrow \langle \mathit{St},\sigma\rangle}$$

$$\frac{\langle B,\sigma\rangle \rightarrow \langle B',\sigma'\rangle}{\langle \texttt{if}\ B\ \texttt{then}\ \mathit{St}_1\ \texttt{else}\ \mathit{St}_2,\sigma\rangle \rightarrow \langle \texttt{if}\ B'\ \texttt{then}\ \mathit{St}_1\ \texttt{else}\ \mathit{St}_2,\sigma'\rangle}$$

$$\frac{\cdot}{\langle \texttt{iftruethen}\ \mathit{St}_1\ \texttt{else}\ \mathit{St}_2,\sigma\rangle \rightarrow \langle \mathit{St}_1,\sigma\rangle}$$

$$\frac{\cdot}{\langle \texttt{iffalsethen}\ \mathit{St}_1\ \texttt{else}\ \mathit{St}_2,\sigma\rangle \rightarrow \langle \mathit{St}_2,\sigma\rangle}$$

$$\frac{\cdot}{\langle \texttt{while}\ B\ \mathit{St},\sigma\rangle \rightarrow \langle \texttt{if}\ B\ \texttt{then}\ (\mathit{St};\texttt{while}\ B\ \mathit{St})\ \texttt{else}\ \texttt{skip},\sigma\rangle}$$

$$\frac{\langle \mathit{St},\sigma\rangle \rightarrow \langle \mathit{St}',\sigma'\rangle}{\langle \mathit{St}.A,\sigma\rangle \rightarrow \langle \mathit{St}'.A,\sigma'\rangle} \qquad \frac{\langle A,\sigma\rangle \rightarrow \langle A',\sigma'\rangle}{\langle \texttt{skip}.A,\sigma\rangle \rightarrow \langle \texttt{skip}.A',\sigma'\rangle}$$

$$\frac{\langle P,\emptyset\rangle \rightarrow^{*} \langle \texttt{skip}.I,\sigma\rangle}{\mathit{eval}(P) \rightarrow I}$$

Figure 3.4: The *SmallStep* language definition

$$
\begin{array}{rl}
\cdot\langle X, S\rangle \rightarrow \langle (S[X]), S\rangle & \\
\cdot\langle \texttt{++}X, S\rangle \rightarrow \langle I, S[X \leftarrow I]\rangle & \textbf{if}\; I = S[X] + 1\\
\cdot\langle A_1 + A_2, S\rangle \rightarrow \langle A_1' + A_2, S'\rangle & \textbf{if}\; \cdot\langle A_1, S\rangle \rightarrow \langle A_1', S'\rangle\\
\cdot\langle I_1 + A_2, S\rangle \rightarrow \langle I_1 + A_2', S'\rangle & \textbf{if}\; \cdot\langle A_2, S\rangle \rightarrow \langle A_2', S'\rangle\\
\cdot\langle I_1 + I_2, S\rangle \rightarrow \langle I_1 +_{Int} I_2, S\rangle & \\
\hline
\cdot\langle A_1 \texttt{<=} A_2, S\rangle \rightarrow \langle A_1' \texttt{<=} A_2, S'\rangle & \textbf{if}\; \cdot\langle A_1, S\rangle \rightarrow \langle A_1', S'\rangle\\
\cdot\langle I_1 \texttt{<=} A_2, S\rangle \rightarrow \langle I_1 \texttt{<=} A_2', S'\rangle & \textbf{if}\; \cdot\langle A_2, S\rangle \rightarrow \langle A_2', S'\rangle\\
\cdot\langle I_1 \texttt{<=} I_2, S\rangle \rightarrow \langle (I_1 \leq_{Int} I_2), S\rangle & \\
\cdot\langle B_1 \;\texttt{and}\; B_2, S\rangle \rightarrow \langle B_1' \;\texttt{and}\; B_2, S'\rangle & \textbf{if}\; \cdot\langle B_1, S\rangle \rightarrow \langle B_1', S'\rangle\\
\cdot\langle \texttt{trueand}\; B_2, S\rangle \rightarrow \langle B_2, S\rangle & \\
\cdot\langle \texttt{falseand}\; B_2, S\rangle \rightarrow \langle \texttt{false}, S\rangle & \\
\cdot\langle \texttt{not}\; B, S\rangle \rightarrow \langle \texttt{not}\; B', S'\rangle & \textbf{if}\; \cdot\langle B, S\rangle \rightarrow \langle B', S'\rangle\\
\cdot\langle \texttt{nottrue}, S\rangle \rightarrow \langle \texttt{false}, S\rangle & \\
\cdot\langle \texttt{notfalse}, S\rangle \rightarrow \langle \texttt{true}, S\rangle & \\
\hline
\cdot\langle X := A, S\rangle \rightarrow \langle X := A', S'\rangle & \textbf{if}\; \cdot\langle A, S\rangle \rightarrow \langle A', S'\rangle\\
\cdot\langle X := I, S\rangle \rightarrow \langle \texttt{skip}, S[X \leftarrow I]\rangle & \\
\cdot\langle St_1; St_2, S\rangle \rightarrow \langle St_1'; St_2, S'\rangle & \textbf{if}\; \cdot\langle St_1, S\rangle \rightarrow \langle St_1', S'\rangle\\
\cdot\langle \texttt{skip}; St_2, S\rangle \rightarrow \langle St_2, S\rangle & \\
\cdot\langle \{St\}, S\rangle \rightarrow \langle St, S\rangle & \\
\cdot\langle \texttt{if}\; B\; \texttt{then}\; St_1\; \texttt{else}\; St_2, S\rangle \qquad\qquad & \\
\rightarrow \langle \texttt{if}\; B'\; \texttt{then}\; St_1\; \texttt{else}\; St_2, S'\rangle & \textbf{if}\; \cdot\langle B, S\rangle \rightarrow \langle B', S'\rangle\\
\cdot\langle \texttt{iftruethen}\; St_1\; \texttt{else}\; St_2, S\rangle \rightarrow \langle St_1, S\rangle & \\
\cdot\langle \texttt{iffalsethen}\; St_1\; \texttt{else}\; St_2, S\rangle \rightarrow \langle St_2, S\rangle & \\
\cdot\langle \texttt{while}\; B\; St, S\rangle \qquad\qquad\qquad\qquad\qquad\qquad & \\
\rightarrow \langle \texttt{if}\; B\; \texttt{then}\; (St; \texttt{while}\; B\; St)\; \texttt{else}\; \texttt{skip}, S\rangle & \\
\hline
\cdot\langle St.A, S\rangle \rightarrow \langle St'.A, S'\rangle & \textbf{if}\; \cdot\langle St, S\rangle \rightarrow \langle St', S'\rangle\\
\cdot\langle \texttt{skip}.A, S\rangle \rightarrow \langle \texttt{skip}.A', S'\rangle & \textbf{if}\; \cdot\langle A, S\rangle \rightarrow \langle A', S'\rangle\\
\hline
\mathit{eval}(P) = \mathit{smallstep}(\langle P, \emptyset\rangle) & \\
\mathit{smallstep}(\langle P, S\rangle) = \mathit{smallstep}(\cdot\langle P, S\rangle) & \\
\mathit{smallstep}(\cdot\langle \texttt{skip}.I, S\rangle) \rightarrow I & \\
\hline
\end{array}
$$

Figure 3.5: $\mathcal{R}_{SmallStep}$ rewriting logic theory

to change to the next step. We use transitivity at the end (rules for *smallstep*) to obtain the transitive closure of the small-step relation by specifically giving tokens to the configuration until it reaches a normal form.

Again, there is a direct correspondence between SOS-style rules and rewriting rules, leading to the following result, which can also be proved by induction on the length of derivations:

Proposition 4. *For any $p \in Pgm$, $\sigma, \sigma' : Name \overset{\circ}{\to} Int$ and $s \in Store$ such that $s \simeq \sigma$, the following are equivalent:*

1. $SmallStep \vdash \langle p, \sigma\rangle \to \langle p', \sigma'\rangle$, *and*
2. $\mathcal{R}_{SmallStep} \vdash \cdot\langle p, s\rangle \to^1 \langle p', s'\rangle$ *and* $s' \simeq \sigma'$.

Moreover, the following are equivalent for any $p \in Pgm$ and $i \in Int$:

1. $SmallStep \vdash \langle p, \bot\rangle \to^* \langle \mathtt{skip}.i, \sigma\rangle$ *for some* $\sigma : Name \overset{\circ}{\to} Int$*, and*
2. $\mathcal{R}_{SmallStep} \vdash eval(p) \to i$.

Proof. As for big-step, we split the proof into four cases, by proving for each syntactical category the following facts (suppose $s \in Store, \sigma : Name \overset{\circ}{\to} Int$, $s \simeq \sigma$):

1. $SmallStep \vdash \langle a, \sigma\rangle \to \langle a', \sigma'\rangle$ iff $\mathcal{R}_{SmallStep} \vdash \cdot\langle a, s\rangle \to^1 \langle a', s'\rangle$ and $s' \simeq \sigma'$,
   for any $a, a' \in AExp$, $\sigma' : Name \overset{\circ}{\to} Int$ and $s' \in Store$.
2. $SmallStep \vdash \langle b, \sigma\rangle \to \langle b', \sigma'\rangle$ iff $\mathcal{R}_{SmallStep} \vdash \cdot\langle b, s\rangle \to^1 \langle b', s'\rangle$ and $s' \simeq \sigma'$,
   for any $b, b' \in BExp$, $\sigma' : Name \overset{\circ}{\to} Int$ and $s' \in Store$.
3. $SmallStep \vdash \langle st, \sigma\rangle \to \langle st', \sigma'\rangle$ iff $\mathcal{R}_{SmallStep} \vdash \cdot\langle st, s\rangle \to^1 \langle st', s'\rangle$ and $s' \simeq \sigma'$,
   for any $st, st' \in Stmt$, $\sigma' : Name \overset{\circ}{\to} Int$ and $s' \in Store$.
4. $SmallStep \vdash \langle p, \sigma\rangle \to \langle p', \sigma'\rangle$ iff $\mathcal{R}_{SmallStep} \vdash \cdot\langle p, s\rangle \to^1 \langle p', s'\rangle$ and $s' \simeq \sigma'$,
   for any $p, p' \in Pgm$, $\sigma' : Name \overset{\circ}{\to} Int$ and $s' \in Store$.

These equivalences can be shown by induction on the size of the derivation tree. Again, we only show one example per category:

1. $SmallStep \vdash \langle a_1 + a_2, \sigma\rangle \to \langle a_1 + a_2', \sigma'\rangle$ iff
   $a_1 = i$ and $SmallStep \vdash \langle a_2, \sigma\rangle \to \langle a_2', \sigma'\rangle$ iff
   $a_1 = i$, $R_{SmallStep} \vdash \cdot\langle a_2, s\rangle \to^1 \langle a_2', s'\rangle$ and $s' \simeq \sigma'$ iff
   $R_{SmallStep} \vdash \cdot\langle a_1 + a_2, s\rangle \to^1 \langle a_1 + a_2', s'\rangle$ and $s' \simeq \sigma'$.
2. $SmallStep \vdash \langle \mathtt{not\ true}, \sigma\rangle \to \langle \mathtt{false}, \sigma\rangle$ iff
   $R_{SmallStep} \vdash \cdot\langle \mathtt{not\ true}, s\rangle \to^1 \langle \mathtt{false}, s\rangle$.
3. $SmallStep \vdash \langle st_1; st_2, \sigma\rangle \to \langle st_1'; st_2, \sigma'\rangle$ iff
   $SmallStep \vdash \langle st_1, \sigma\rangle \to \langle st_1', \sigma'\rangle$ iff
   $R_{SmallStep} \vdash \cdot\langle st_1, s\rangle \to^1 \langle st_1', s'\rangle$ and $s' \simeq \sigma'$ iff
   $R_{SmallStep} \vdash \cdot\langle st_1; st_2, s\rangle \to^1 \langle st_1' + st_2, s'\rangle$ and $s' \simeq \sigma'$.

4. $\mathit{SmallStep} \vdash \langle st.a, \sigma\rangle \to \langle st.a', \sigma'\rangle$ iff
   $st = \texttt{skip}$ and $\mathit{SmallStep} \vdash \langle a, \sigma\rangle \to \langle a', \sigma'\rangle$ iff
   $st = \texttt{skip}$, $R_{\mathit{SmallStep}} \vdash \cdot\langle a, s\rangle \to^{1} \langle a', s'\rangle$ and $s' \simeq \sigma'$ iff
   $R_{\mathit{SmallStep}} \vdash \cdot\langle st.a, s\rangle \to \langle st.a', s'\rangle$ and $s' \simeq \sigma'$.

Let us now move to the second equivalence. For this proof let $\to^{n}$ be the restriction of $\mathcal{R}_{\mathit{SmallStep}}$ relation $\to$ to those pairs which can be provable by exactly applying $n-1$ times the **Transitivity** rule if $n > 0$, or **Reflexivity** for $n = 0$. We first prove the following more general result (suppose $p \in \mathit{Pgm}$, $\sigma : \mathit{Name} \overset{\circ}{\to} \mathit{Int}$ and $s \in \mathit{Store}$ such that $s \simeq \sigma$):

$\mathit{SmallStep} \vdash \langle p, \sigma\rangle \to^{n} \langle p', \sigma'\rangle$ iff
$\mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{smallstep}(\langle p, s\rangle) \to^{n} \mathit{smallstep}(\cdot\langle p', s'\rangle)$ and $s' \simeq \sigma'$,

by induction on n. If $n = 0$ then $\langle p, \sigma\rangle = \langle p', \sigma'\rangle$ and since $\mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{smallstep}(\langle p, s\rangle) = \mathit{smallstep}(\cdot\langle p, s\rangle)$ we are done. If $n > 0$, we have that
$\mathit{SmallStep} \vdash \langle p, \sigma\rangle \to^{n} \langle p', \sigma'\rangle$ iff
$\mathit{SmallStep} \vdash \langle p, \sigma\rangle \to \langle p_1, \sigma_1\rangle$ and $\mathit{SmallStep} \vdash \langle p_1, \sigma_1\rangle \to^{n-1} \langle p', \sigma'\rangle$ iff
$\mathcal{R}_{\mathit{SmallStep}} \vdash \cdot\langle p, s\rangle \to \langle p_1, s_1\rangle$ and $s_1 \simeq \sigma_1$ (by 1)
and $\mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{smallstep}(\langle p_1, s_1\rangle) \to^{n-1} \mathit{smallstep}(\cdot\langle p', s'\rangle)$ and $s' \simeq \sigma'$
(by the induction hypothesis) iff
$\mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{smallstep}(\cdot\langle p, s\rangle) \to^{1} \mathit{smallstep}(\langle p_1, s_1\rangle)$ and $s_1 \simeq \sigma_1$
and $\mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{smallstep}(\langle p_1, s_1\rangle) \to^{n-1} \mathit{smallstep}(\cdot\langle p', s'\rangle)$ and $s' \simeq \sigma'$ iff
$\mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{smallstep}(\cdot\langle p, s\rangle) \to^{n} \mathit{smallstep}(\cdot\langle p', s'\rangle)$ and $s' \simeq \sigma'$.
We are done, since $\mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{smallstep}(\langle p, s\rangle) = \mathit{smallstep}(\cdot\langle p, s\rangle)$.

Finally, $\mathit{SmallStep} \vdash \langle p, \bot\rangle \to^{*} \langle \texttt{skip}.i, \sigma\rangle$ iff $\mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{smallstep}(\langle p, \emptyset\rangle) \to \mathit{smallstep}(\cdot\langle \texttt{skip}.i, s\rangle)$, $s \simeq \sigma$; the rest follows from $\mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{eval}(p) = \mathit{smallstep}(\langle p, \emptyset\rangle)$ and $\mathcal{R}_{\mathit{SmallStep}} \vdash \mathit{smallstep}(\cdot\langle \texttt{skip}.i, s\rangle) = i$. □

Strengths. Small-step operational semantics precisely defines the notion of one computational step. It stops at errors, pointing them out. It is easy to trace and debug. It gives interleaving semantics for concurrency.

Weaknesses. Each small step does the same amount of computation as a big step in finding the next redex. It does not give a “true concurrency” semantics, that is, one has to choose a certain interleaving (no two rules can be applied on the same term at the same time), mainly because reduction is forced to occur only at the top. One of the reasons for introducing SOS was that abstract machines need to introduce new syntactic constructs to decompose the abstract syntax tree, while SOS would and should only work by modifying the structure of the program. We argue that this is not entirely accurate: for example, one needs to have the syntax of boolean values if one wants to have boolean expressions, and needs an `if` mechanism in the above definition to evaluate `while`. The fact that these features are common in programming languages does not mean that the languages which don’t want to allow them should be despised. It is still hard to deal with control —for example, consider adding halt to this language. One cannot simply do it as for other ordinary statements: instead, one has to add a corner case (additional rule) to each statement, as shown below:

$$\begin{aligned}
\cdot\langle \texttt{halt}\ A, S\rangle &\rightarrow \langle \texttt{halt}\ A', S'\rangle \quad \textbf{if} \quad \cdot\langle A, S\rangle \rightarrow \langle A', S'\rangle\\
\cdot\langle \texttt{halt}\ I; St, S\rangle &\rightarrow \langle \texttt{halt}\ I, S\rangle\\
\cdot\langle \texttt{halt}\ I.A, S\rangle &\rightarrow \langle \texttt{skip}.I, S\rangle
\end{aligned}$$

If expressions could also halt the program, e.g., if one adds functions, then a new rule would have to be added to specify the corner case for each halt-related arithmetic or boolean construct. Moreover, by propagating the "halt signal" through all the statements and expressions, one fails to capture the intended computational granularity of halt: it should just terminate the execution in *one step*!

3.5 MSOS Semantics

MSOS semantics was introduced by Mosses in [122, 123] to deal with the non-modularity issues of small-step and big-step semantics. The solution proposed in *MSOS* involves moving the non-syntactic state components to the labels on transitions (as provided by SOS), plus a discipline of only selecting needed attributes from the states.

A transition in *MSOS* is of the form $P \xrightarrow{\mathcal{X}} P'$, where P and P' are program expressions and $\mathcal{X}$ is a label describing the structure of the state both before and after the transition. If $\mathcal{X}$ is missing, then the state is assumed to stay unchanged. Specifically, $\mathcal{X}$ is a record containing fields denoting the semantic components; the preferred notation in *MSOS* for saying that in the label $\mathcal{X}$ the semantic component associated to the field name σ (e.g., a store name) is σ_0 (e.g., a function associating values to variables) is $\mathcal{X} = \{\sigma = \sigma_0, \ldots\}$. Modularity is achieved by the record comprehension notation "…" which indicates that more fields could follow but that they are not of interest for this transition. If record comprehension is used in both the premise and the conclusion of an MSOS rule, then all occurrences of "…" stand for the same fields with the same semantic components. Fields of a label can fall in one of the following categories: *read-only*, *read-write* and *write-only*.

Read-only fields are only inspected by the rule, but not modified. For example, when reading the location of a variable in an environment, the environment is not modified.

Read-write fields come in pairs, having the same field name, except that the "write" field name is primed. They are used for transitions modifying existing state fields. For example, a store field σ can be read and written, as illustrated by the *MSOS* rule' for assignment:

$$\frac{\mathrm{unobs}\{\sigma = \sigma_0, \sigma' = \sigma_0 \ldots\}}{X\,\texttt{:=}I \xrightarrow{\{\sigma=\sigma_0,\sigma'=\sigma_0[I/X],\ldots\}} \texttt{skip}}$$

The above rule says that, if before the transition the store was σ_0, after the transition it will become $\sigma_0[I/X]$, updating X by I. The unobs predicate is used to express that the rest of the state does not change.

Write-only fields are used to record things whose values cannot be inspected before a transition such as emission of actions to the outside world (e.g., output, recording of the trace). Their names are always primed and they have a free monoid semantics —everything written on them is actually added at the end. A good example of the usage of write-only fields would be a rule for defining a `print` language construct:

$$\frac{\text{unobs}\{\mathit{out}' = (), \ldots\}}{\texttt{print}(I) \xrightarrow{\{\mathit{out}'=I,\ldots\}} \texttt{skip}}$$

where "()" stand for monoid unit.

The state after this rule is applied will have the *out* field containing "LI", where the juxtaposition LI denotes the free monoid multiplication of L and I.

The *MSOS* description of the small-step SOS definition in Figure 3.4 is given in Figure 3.6 (we let $\mathcal{X}$ range over labels on transitions).

Because the part of the state not involved in a certain rule is hidden through the "..." notation, language extensions can be made modularly. Consider, for example, adding `halt` to the definition in Figure 3.6. One possible way to do it is to follow the technique proposed in [123] for adding non-parametric abrupt termination, with some modifications to suit our needs to abruptly terminate the program with a value. For this, we add a write-only field in the record, say *halt?* having as arrows the monoid freely generated by integer numbers, along with a language construct `stuck` to block the execution of the program. To "catch the halt signal" we extend the abstract syntax with a new construct, say **program**, applied to a top-level program. The first set of *MSOS* rules for `halt` are then:

$$\frac{A \xrightarrow{\mathcal{X}} A'}{\texttt{halt}\ \ A \xrightarrow{\mathcal{X}} \texttt{halt}\ \ A'}$$

$$\frac{\text{unobs}\{\mathit{halt?}' = (), \ldots\}}{\texttt{halt}\ \ I \xrightarrow{\{\mathit{halt?}'=I,\ldots\}} \texttt{stuck}}$$

$$\frac{P \xrightarrow{\{\mathit{halt?}'=I,\ldots\}} P'}{\texttt{program}\ \ P \xrightarrow{\{\mathit{halt?}'=I,\ldots\}} \texttt{program}\ \ \texttt{skip}.I}$$

$$\frac{P \xrightarrow{\{\mathit{halt?}'=(),\ldots\}} P'}{\texttt{program}\ \ P \xrightarrow{\{\mathit{halt?}'=(),\ldots\}} \texttt{program}\ \ P'}$$

An alternative to the above definition, which would not require the introduction of new syntax, is to make *halt?* a read-write field with possible values integers along with a default value nil and use an unobservable transition at

$$\frac{\text{unobs}\{\sigma, \ldots\},\ \sigma(X) = I}{X \xrightarrow{\{\sigma,\ldots\}} I}$$

$$\frac{\text{unobs}\{\sigma = \sigma_0, \sigma' = \sigma_0, \ldots\},\ I = \sigma_0(X) + 1}{\texttt{++}X \xrightarrow{\{\sigma=\sigma_0,\sigma'=\sigma_0[I/X],\ldots\}} I}$$

$$\frac{A_1 \xrightarrow{\mathcal{X}} A_1'}{A_1 + A_2 \xrightarrow{\mathcal{X}} A_1' + A_2} \qquad \frac{A_2 \xrightarrow{\mathcal{X}} A_2'}{I_1 + A_2 \xrightarrow{\mathcal{X}} I_1 + A_2'}$$

$$\frac{I = I_1 +_{Int} I_2}{I_1 + I_2 \rightarrow I}$$

$$\frac{A_1 \xrightarrow{\mathcal{X}} A_1'}{A_1\texttt{<=}A_2 \xrightarrow{\mathcal{X}} A_1'\texttt{<=}A_2} \qquad \frac{A_2 \xrightarrow{\mathcal{X}} A_2'}{I_1\texttt{<=}A_2 \xrightarrow{\mathcal{X}} I_1\texttt{<=}A_2'}$$

$$\frac{T = I_1 \leq_{Int} I_2}{I_1\texttt{<=}I_2 \rightarrow T}$$

$$\frac{B_1 \xrightarrow{\mathcal{X}} B_1'}{B_1 \texttt{ and } B_2 \xrightarrow{\mathcal{X}} B_1' \texttt{ and } B_2} \qquad \frac{B \xrightarrow{\mathcal{X}} B'}{\texttt{not } B \xrightarrow{\mathcal{X}}\texttt{not } B'}$$

$$\texttt{trueand } B_2 \rightarrow B_2 \qquad \texttt{nottrue}\rightarrow\texttt{false}$$

$$\texttt{falseand } B_2 \rightarrow\texttt{false} \qquad \texttt{notfalse}\rightarrow\texttt{true}$$

$$\frac{A \xrightarrow{\mathcal{X}} A'}{X\texttt{:=}A \xrightarrow{\mathcal{X}} X\texttt{:=}A'} \qquad \frac{St_1 \xrightarrow{\mathcal{X}} St_1'}{St_1; St_2 \xrightarrow{\mathcal{X}} St_1'; St_2}$$

$$\frac{\text{unobs}\{\sigma = \sigma_0, \sigma' = \sigma_0, \ldots\}}{X\texttt{:=}I \xrightarrow{\{\sigma=\sigma_0,\sigma'=\sigma_0[I/X],\ldots\}} \texttt{skip}} \qquad \texttt{skip}; St_2 \rightarrow St_2$$

$$\{St\} \rightarrow St$$

$$\frac{B \xrightarrow{\mathcal{X}} B'}{\texttt{if } B \texttt{ then } St_1 \texttt{ else } St_2 \xrightarrow{\mathcal{X}}\texttt{if } B' \texttt{ then } St_1 \texttt{ else } St_2}$$

$$\texttt{iftruethen } St_1 \texttt{ else } St_2 \rightarrow St_1$$

$$\texttt{iffalsethen } St_1 \texttt{ else } St_2 \rightarrow St_2$$

$$\texttt{while } B\ St \rightarrow\texttt{if } B \texttt{ then } (St; \texttt{while } B\ St) \texttt{ else skip}$$

$$\frac{St \xrightarrow{\mathcal{X}} St'}{St.A \xrightarrow{\mathcal{X}} St'.A} \qquad \frac{A \xrightarrow{\mathcal{X}} A'}{\texttt{skip}.A \xrightarrow{\mathcal{X}} \texttt{skip}.A'}$$

Figure 3.6: The *MSOS* language definition

top to terminate the program:

$$\frac{A \xrightarrow{\mathcal{X}} A'}{\texttt{halt}\ A \xrightarrow{\mathcal{X}} \texttt{halt}\ A'}$$

$$\frac{\text{unobs}\{halt? = nil, halt?' = nil, \ldots\}}{\texttt{halt}\ I \xrightarrow{\{halt?=nil, halt?'=I, \ldots\}} \texttt{stuck}}$$

$$\frac{\text{unobs}\{halt? = I, \ldots\}}{P \xrightarrow{\{halt?=I, \ldots\}} \texttt{skip}.I}$$

However, since the last rule is based on observation of the state, the program is not forced to terminate as soon as halt is consumed (as was the case in the first definition), since in the case of non-determinism, for example, there might be other things which are still computable.

To give a faithful representation of *MSOS* definitions in rewriting logic, we here follow the methodology in [108]. Using the fact that labels describe changes from their source state to their destination state, one can move the labels back into the configurations. That is, a transition step $P \xrightarrow{u} P'$ is modeled as a rewrite step $\cdot\langle P, u^{pre}\rangle \to \langle P', u^{post}\rangle$, where u^{pre} and u^{post} are *records* describing the state before and after the transition. Notice again the use of the "$\cdot$" operator to emulate small steps by restricting transitivity. State records can be specified equationally as wrapping (using a constructor "$\{\_\}$") a set of fields built from *fields* as constructors, using an associative and commutative concatenation operation "$\_,\_$". Fields are constructed from state attributes; for example, the store can be embedded into a field by a constructor "$\sigma : \_$".

Records u^{pre} and u^{post} are computed from u in the following way:

- For unobservable transitions, $u^{pre} = u^{post}$; same applies for unobservable attributes in premises;

- *Read-only* fields of u are added to both u^{pre} and u^{post}.

- *Read-write* fields of u are translated by putting the read part in u^{pre} and the (now unprimed) write part in u^{post}. The assignment rule, for example, becomes:

  $$\cdot\langle X\texttt{:=}I, \{\sigma : S_0, W\}\rangle \to \langle \texttt{skip}, \{\sigma : S_0[X \leftarrow I], W\}\rangle$$

  Notice that the "$\ldots$" notation gets replaced by a generic field-set variable W.

- *Write-only* fields $i' = v$ of u are translated as follows: $i : L$, with L a fresh new variable, is added to u^{pre}, and $i : Lv$ is added to u^{post}. For example, the `print` rule above becomes:

  $$\cdot\langle \texttt{print}(I), \{out : L, W\}\rangle \to \langle \texttt{skip}, \{out : LI, W\}\rangle$$

- When dealing with observable transitions, both state records meta-variables and . . . operations are represented in u^{pre} by some variables, while in u^{post} by others. For example, the first rule defining addition in Figure 3.6 is translated into:

$$\cdot\langle A_1 + A_2, R\rangle \to \langle A_1' + A_2, R'\rangle \quad \textbf{if} \quad \cdot\langle A_1, R\rangle \to \langle A_1', R'\rangle$$

The key thing to notice here is that modularity is preserved by this translation. What indeed makes *MSOS* definitions modular is the record comprehension mechanism. A similar comprehension mechanism is achieved in rewriting logic by using sets of fields and matching modulo associativity and commutativity. That is, the extensibility provided by the "..." record notation in *MSOS* is here captured by associative and commutative matching on the W variable, which allows new fields to be added.

The relation between *MSOS* and R_{MSOS} definitions assumes that *MSOS* definitions are in a certain *normal form* [108] and is made precise by the following theorem, strongly relating MSOS and modular rewriting semantics.

Theorem 3. *[108] For each normalized MSOS definition, there is a strong bisimulation between its transition system and the transition system associated to its translation in rewriting logic.*

The above presented translation is the basis for the `Maude-MSOS` tool [28], which has been used to define and analyze complex language definitions, such as Concurrent ML [27].

Figure 3.7 presents the rewrite theory corresponding to the *MSOS* definition in Figure 3.6. The only new variable symbols introduced are R,R', standing for records, and W standing for the remainder of a record.

Strengths. As it is a framework on top of any operational semantics, it inherits the strengths of the semantics for which it is used; moreover, it adds to those strengths the important new feature of *modularity*. It is well-known that SOS definitions are typically highly unmodular, so that adding a new feature to the language often requires the entire redefinition of the SOS rules.

Weaknesses. Control is still not explicit in MSOS, making combinations of control-dependent features (e.g., call/cc) impossible to specify [123, page 223]. Also, MSOS still does not allow to capture the intended computational granularity of some defined language statements. For example, the desired semantics of "halt i" is "stop the execution with the result i"; unfortunately, MSOS, like its SOS ancestors, still needs to "propagate" the halting signal along the syntax all the way to the top.

$$\begin{aligned}
\cdot\langle X, \{\sigma : S, W\}\rangle &\to \langle I, \{\sigma : S, W\}\rangle \textbf{ if } I = S[X] \\
\cdot\langle \texttt{++}X, \{\sigma : S_0, W\}\rangle &\to \langle I, \{S_0[X \leftarrow I], W\}\rangle \textbf{ if } I = S_0[X] + 1 \\
\cdot\langle A_1 + A_2, R\rangle &\to \langle A_1' + A_2, R'\rangle \textbf{ if } \cdot\langle A_1, R\rangle \to \langle A_1', R'\rangle \\
\cdot\langle I_1 + A_2, R\rangle &\to \langle I_1 + A_2', R'\rangle \textbf{ if } \cdot\langle A_2, R\rangle \to \langle A_2', R'\rangle \\
\cdot\langle I_1 + I_2, R\rangle &\to \langle I_1 +_{Int} I_2, R\rangle
\end{aligned}$$

$$\begin{aligned}
\cdot\langle A_1 \texttt{<=} A_2, R\rangle &\to \langle A_1' \texttt{<=} A_2, R'\rangle \textbf{ if } \cdot\langle A_1, R\rangle \to \langle A_1', R'\rangle \\
\cdot\langle I_1 \texttt{<=} A_2, R\rangle &\to \langle I_1 \texttt{<=} A_2', R'\rangle \textbf{ if } \cdot\langle A_2, R\rangle \to \langle A_2', R'\rangle \\
\cdot\langle I_1 \texttt{<=} I_2, R\rangle &\to \langle I_1 \leq_{Int} I_2, R\rangle \\
\cdot\langle B_1 \texttt{ and } B_2, R\rangle &\to \langle B_1' \texttt{ and } B_2, R'\rangle \textbf{ if } \cdot\langle B_1, R\rangle \to \langle B_1', R'\rangle \\
\cdot\langle \texttt{trueand } B_2, R\rangle &\to \langle B_2, R\rangle \\
\cdot\langle \texttt{falseand } B_2, R\rangle &\to \langle \texttt{false}, R\rangle \\
\cdot\langle \texttt{not } B, R\rangle &\to \langle \texttt{not } B', R'\rangle \textbf{ if } \cdot\langle B, R\rangle \to \langle B', R'\rangle \\
\cdot\langle \texttt{nottrue}, R\rangle &\to \langle \texttt{false}, R\rangle \\
\cdot\langle \texttt{notfalse}, R\rangle &\to \langle \texttt{true}, R\rangle
\end{aligned}$$

$$\begin{aligned}
\cdot\langle X \texttt{:=} A, R\rangle &\to \langle X \texttt{:=} A', R'\rangle \textbf{ if } \cdot\langle A, R\rangle \to \langle A', R'\rangle \\
\cdot\langle X \texttt{:=} I, \{\sigma : S_0, W\}\rangle &\to \langle \texttt{skip}, \{\sigma : S_0[X \leftarrow I], W\}\rangle \\
\cdot\langle St_1; St_2, R\rangle &\to \langle St_1'; St_2, R'\rangle \textbf{ if } \cdot\langle St_1, R\rangle \to \langle St_1', R'\rangle \\
\cdot\langle \texttt{skip}; St_2, R\rangle &\to \langle St_2, R\rangle \\
\cdot\langle \{St\}, R\rangle &\to \langle St, R\rangle
\end{aligned}$$

$$\begin{aligned}
&\cdot\langle \texttt{if } B \texttt{ then } St_1 \texttt{ else } St_2, R\rangle \\
&\quad \to \langle \texttt{if } B' \texttt{ then } St_1 \texttt{ else } St_2, R'\rangle \quad \textbf{if } \cdot\langle B, R\rangle \to \langle B', R'\rangle \\
&\cdot\langle \texttt{iftruethen } St_1 \texttt{ else } St_2, R\rangle \to \langle St_1, R\rangle \\
&\cdot\langle \texttt{iffalsethen } St_1 \texttt{ else } St_2, R\rangle \to \langle St_2, R\rangle \\
&\cdot\langle \texttt{while } B\ St, R\rangle \\
&\quad \to \langle \texttt{if } B \texttt{ then } (St; \texttt{while } B\ St) \texttt{ else skip}, R\rangle
\end{aligned}$$

$$\begin{aligned}
\cdot\langle St.A, R\rangle &\to \langle St'.A, R'\rangle \textbf{ if } \cdot\langle St, R\rangle \to \langle St', R'\rangle \\
\cdot\langle \texttt{skip}.A, R\rangle &\to \langle \texttt{skip}.A', R'\rangle \textbf{ if } \cdot\langle A, R\rangle \to \langle A', R'\rangle
\end{aligned}$$

Figure 3.7: R_{MSOS} rewriting logic theory

3.6 Reduction Semantics with Evaluation Contexts

Introduced in [180], also called context reduction, the evaluation contexts style improves over small-step definitional style in two ways:

1. it gives a more compact semantics to context-sensitive reduction, by using parsing to find the next redex rather than small-step rules; and

2. it provides the possibility of also modifying the context in which a reduction occurs, making it much easier to deal with control-intensive features. For example, defining halt is done now using only one rule, $C[\texttt{halt}\ I] \to I$, preserving the desired computational granularity. Additionally, one can also incorporate the configuration as part of the evaluation context, and thus have full access to semantic information on a "by need basis"; the PLT-Redex implementation of context reduction, for example, supports this approach. Notice how the assignment rule, for example, modifies both the redex, transforming it to `skip`, and the evaluation context, altering the state which can be found at its top. In this framework, constructs like `call/cc` can be defined with little effort.

$$\begin{array}{rcl}
\mathit{CConf} & ::= & \langle \mathit{CPgm}, \mathit{Store} \rangle \\
\mathit{CPgm} & ::= & [\,] \mid \mathtt{skip}.\mathit{CAExp} \mid \mathit{CStmt}.\mathit{AExp} \\
\mathit{CStmt} & ::= & [\,] \mid \mathit{CStmt}; \mathit{Stmt} \mid X\,\mathtt{:=}\mathit{CAExp} \\
 & \mid & \mathtt{if}\ \mathit{CBExp}\ \mathtt{then}\ \mathit{Stmt}\ \mathtt{else}\ \mathit{Stmt} \\
 & \mid & \mathtt{halt}\ \mathit{CAExp} \\
\mathit{CBExp} & ::= & [\,] \mid \mathit{Int}\mathtt{<=}\mathit{CAExp} \mid \mathit{CAExp}\mathtt{<=}\mathit{AExp} \\
 & \mid & \mathit{CBExp}\ \mathtt{and}\ \mathit{BExp}\ |\mathtt{not}\ \mathit{CBExp} \\
\mathit{CAExp} & ::= & [\,] \mid \mathit{Int} + \mathit{CAExp} \mid \mathit{CAExp} + \mathit{AExp}
\end{array}$$

$$\frac{E \to E'}{C[E] \to C[E']}$$

$$\begin{array}{l}
I_1 + I_2 \to (I_1 +_{\mathit{Int}} I_2) \\
\langle P, \sigma \rangle [X] \to \langle P, \sigma \rangle [(\sigma(X))] \\
\langle P, \sigma \rangle [\mathtt{++}X] \to \langle P, \sigma[I/X] \rangle [I] \text{ when } I = \sigma(X) + 1
\end{array}$$

$$\begin{array}{l}
I_1\mathtt{<=}I_2 \to (I_1 \leq_{\mathit{Int}} I_2) \\
\mathtt{true}\ \mathtt{and}\ B \to B \\
\mathtt{false}\ \mathtt{and}\ B \to \mathtt{false} \\
\mathtt{not}\ \mathtt{true} \to \mathtt{false} \\
\mathtt{not}\ \mathtt{false} \to \mathtt{true}
\end{array}$$

$$\begin{array}{l}
\mathtt{if}\ \mathtt{true}\ \mathtt{then}\ St_1\ \mathtt{else}\ St_2 \to St_1 \\
\mathtt{if}\ \mathtt{false}\ \mathtt{then}\ St_1\ \mathtt{else}\ St_2 \to St_2 \\
\mathtt{skip}; St \to St \\
\{St\} \to St \\
\langle P, \sigma \rangle [X\,\mathtt{:=}I] \to \langle P, \sigma[I/X] \rangle [\mathtt{skip}] \\
\mathtt{while}\ B\ St \to \mathtt{if}\ B\ \mathtt{then}\ (St; \mathtt{while}\ B\ St)\ \mathtt{else}\ \mathtt{skip} \\
C[\mathtt{halt}\ I] \to \langle I \rangle
\end{array}$$

$$C[\mathtt{skip}.I] \to \langle I \rangle$$

Figure 3.8: The *CxtRed* language definition

In a context reduction semantics of a language, one typically starts by defining the syntax of *evaluation contexts*. An evaluation context is a program with a "hole", the hole being a placeholder where the next computational step takes place. If C is such a context and E is some expression whose type fits into the type of the hole of C, then $C[E]$ is the program formed by replacing the hole of C by E. The characteristic reduction step underlying context reduction is:

$$\frac{E \to E'}{C[E] \to C[E']},$$

extending the usual "only-at-the-top" reduction by allowing reduction steps to take place under any desired evaluation context. Therefore, an important part of a context reduction semantics is the definition of evaluation contexts, which is typically done by means of a context-free grammar. The definition of evaluation contexts for our simple language is found in Figure 3.8 (we let [] denote the "hole").

In this BNF definition of evaluation contexts, S is a store variable. Therefore, a "top level" evaluation context will also contain a store in our simple language definition. There are also context-reduction definitions which operate only on syntax (i.e., no additional state is needed), but instead one needs to employ some substitution mechanism (particularly in definitions of λ-calculus based languages). The rules following the evaluation contexts grammar in Figure 3.8 complete the context reduction semantics of our simple language, which we call *CxtRed*.

By making the evaluation context explicit and changeable, context reduction is, in our view, a significant improvement over small-step SOS. In particular, one can now define control-intensive statements like `halt` *modularly* and at the desired level of computational granularity. Even though the definition in Figure 3.8 gives one the feeling that evaluation contexts and their instantiation come "for free", the application of the "rewrite in context" rule presented above can be expensive in practice. This is because one needs either to parse/search the entire configuration to put it in the form $C[E]$ for some appropriate C satisfying the grammar of evaluation contexts, or to maintain enough information in some special data-structures to perform the split $C[E]$ using only local information and updates. Moreover, this "matching-modulo-the-CFG-of-evaluation-contexts" step needs to be done at every computation step during the execution of a program, so it may easily become the major bottleneck of an executable engine based on context reduction. *Direct* implementations of context reduction such as `PLT-Redex` cannot avoid paying a significant performance penalty [181]. Danvy and Nielsen propose in [41] a technique for efficiently interpreting a restricted form of reduction semantics definitions by means of "refocusing" functions which yield efficient abstract machines. Although these refocusing functions are equationally definable, since we aim here to achieve minimal representational distance, we prefer to translate the definitions into rewriting logic by leaving the rules unchanged and implementing the decompose and plug functions from reduction semantics by means of equations. Next section will present an abstract-machine

$$
\begin{array}{lr}
\mathit{s2c}(\langle P, S\rangle) = \langle C, S\rangle[R] & \textbf{if } C[R] = \mathit{s2c}(P) \\
\hline
\mathit{s2c}(\texttt{skip}.I) = [\,][\texttt{skip}.I] & \\
\mathit{s2c}(\texttt{skip}.A) = (\texttt{skip}.C)[R] & \textbf{if } C[R] = \mathit{s2c}(A) \\
\mathit{s2c}(St.A) = (C.A)[R] & \textbf{if } C[R] = \mathit{s2c}(St) \\
\hline
\mathit{s2c}(\texttt{halt } I) = [\,][\texttt{halt } I] & \\
\mathit{s2c}(\texttt{halt } A) = (\texttt{halt } C)[R] & \textbf{if } C[R] = \mathit{s2c}(A) \\
\mathit{s2c}(\texttt{while } B\ St) = [\,][\texttt{while } B\ St] & \\
\mathit{s2c}(\texttt{if } T \texttt{ then } St_1 \texttt{ else } St_2) = [\,][\texttt{if } T \texttt{ then } St_1 \texttt{ else } St_2] & \\
\mathit{s2c}(\texttt{if } B \texttt{ then } St_1 \texttt{ else } St_2) = (\texttt{if } C \texttt{ then } St_1 \texttt{ else } St_2)[R] & \textbf{if } C[R] = \mathit{s2c}(B) \\
\mathit{s2c}(\{St\}) = [\,][\{St\}] & \\
\mathit{s2c}(\texttt{skip}; St_2) = [\,][\texttt{skip}; St_2] & \\
\mathit{s2c}(St_1; St_2) = (C; St_2)[R] & \textbf{if } C[R] = \mathit{s2c}(St_1) \\
\mathit{s2c}(X \texttt{:=} I) = [\,][X \texttt{:=} I] & \\
\mathit{s2c}(X \texttt{:=} A) = (X \texttt{:=} C)[R] & \textbf{if } C[R] = \mathit{s2c}(A) \\
\hline
\mathit{s2c}(I_1 \texttt{<=} I_1) = [\,][I_1 \texttt{<=} I_2] & \\
\mathit{s2c}(I \texttt{<=} A) = (I \texttt{<=} C)[R] & \textbf{if } C[R] = \mathit{s2c}(A) \\
\mathit{s2c}(A_1 \texttt{<=} A_2) = (C \texttt{<=} A_2)[R] & \textbf{if } C[R] = \mathit{s2c}(A_1) \\
\mathit{s2c}(T \texttt{ and } B_2) = [\,][T \texttt{ and } B_2] & \\
\mathit{s2c}(B_1 \texttt{ and } B_2) = (C \texttt{ and } B_2)[R] & \textbf{if } C[R] = \mathit{s2c}(B_1) \\
\mathit{s2c}(\texttt{not } T) = [\,][\texttt{not } T] & \\
\mathit{s2c}(\texttt{not } B) = (\texttt{not } C)[R] & \textbf{if } C[R] = \mathit{s2c}(B) \\
\hline
\mathit{s2c}(X) = [\,][X] & \\
\mathit{s2c}(\texttt{++}X) = [\,][\texttt{++}X] & \\
\mathit{s2c}(I_1 + I_2) = [\,][I_1 + I_2] & \\
\mathit{s2c}(I + A) = (I + C)[R] & \textbf{if } C[R] = \mathit{s2c}(A) \\
\mathit{s2c}(A_1 + A_2) = (C + A_2)[R] & \textbf{if } C[R] = \mathit{s2c}(A_1) \\
\hline
\end{array}
$$

Figure 3.9: Equational definition of *s2c*

definition of a programming language in rewriting logic, resembling Felleisen's CK machine [59], which is obtained by applying Danvy and Nielsen's technique.

Context reduction is trickier to faithfully capture as a rewrite theory, since rewriting logic, by its locality, always applies a rule *in* context, without actually having the capability of changing the given context. Also, from a rewriting point of view, context-reduction captures context-sensitive rewriting, which, although supported by rewriting logic in the form of congruence restricted to the non-frozen arguments of each operator, cannot be captured "as-is" in its full generality within rewriting logic.

To faithfully model context-reduction, we make use of two equationally-defined operations: *s2c*, which splits a piece of syntax into a context and a redex, and *c2s*, which plugs a piece of syntax into a context. In our rewriting logic definition, $C[R]$ is *not a parsing convention*, but rather a *constructor* conveniently representing the pair (context C, redex R). In order to have an algebraic representation of contexts we extend the signature by adding a constant $[\,]$, representing the hole, for each syntactic category. The operation *s2c*, presented in Figure 3.9, has an effect similar to what one achieves by parsing in context reduction, in the sense that given a piece of syntax it yields $C[R]$. It is a straight-forward, equational definition of the *decompose* function used in context-reduction implementations based on the syntax of contexts. We here

$$\mathit{c2s}([\,][H]) = H$$

$$\mathit{c2s}(\langle P, S\rangle[H]) = \langle \mathit{c2s}(P[H]), S\rangle$$
$$\mathit{c2s}(\langle I\rangle[H]) = \langle I\rangle$$

$$\mathit{c2s}(E_1.E_2[H]) = \mathit{c2s}(E_1[H]).\mathit{c2s}(E_2[H])$$

$$\mathit{c2s}(\texttt{halt}\ E[H]) = \texttt{halt}\ \mathit{c2s}(E[H])$$
$$\mathit{c2s}(\texttt{while}\ E_1\ E_2[H]) = \texttt{while}\ \mathit{c2s}(E_1[H])\ \mathit{c2s}(E_2[H])$$
$$\mathit{c2s}(\texttt{if}\ E\ \texttt{then}\ E_1\ \texttt{else}\ E_2[H]) = \texttt{if}\ \mathit{c2s}(E[H])\ \texttt{then}\ \mathit{c2s}(E_1[H])\ \texttt{else}\ \mathit{c2s}(E_2[H])$$
$$\mathit{c2s}(\{E\}[H]) = \{\mathit{c2s}(E[H])\}$$
$$\mathit{c2s}(E_1; E_2[H]) = \mathit{c2s}(E_1[H]); \mathit{c2s}(E_2[H])$$
$$\mathit{c2s}(X\texttt{:=}E[H]) = X\texttt{:=}\mathit{c2s}(E[H])$$
$$\mathit{c2s}(\texttt{skip}[H]) = \texttt{skip}$$

$$\mathit{c2s}(E_1\texttt{<=}E_2[H]) = \mathit{c2s}(E_1[H])\texttt{<=}\mathit{c2s}(E_2[H])$$
$$\mathit{c2s}(E_1\ \texttt{and}\ E_2[H]) = \mathit{c2s}(E_1[H])\ \texttt{and}\ \mathit{c2s}(E_2[H])$$
$$\mathit{c2s}(\texttt{not}\ E[H]) = \texttt{not}\ \mathit{c2s}(E[H])$$
$$\mathit{c2s}(\texttt{true}[H]) = \texttt{true}$$
$$\mathit{c2s}(\texttt{false}[H]) = \texttt{false}$$

$$\mathit{c2s}(\texttt{++}X[H]) = \texttt{++}X$$
$$\mathit{c2s}(E_1 + E_2[H]) = \mathit{c2s}(E_1[H]) + \mathit{c2s}(E_2[H])$$
$$\mathit{c2s}(I[H]) = I$$

Figure 3.10: Equational definition of *c2s*

assume the same restrictions on the context syntax as in [41], namely that the grammar defining them is context-free and that there is always a unique decomposition of an expression into a context and a redex. The operation *c2s*, presented in Figure 3.10, is the equational definition of the *plug* function used in interpreting context-reduction definitions, and it is a morphism on the syntax. Notice that (from the defining equations) we have the guarantee that it will be applied only to "well-formed" contexts (i.e., contexts containing only one hole). The rewrite theory $\mathcal{R}_{\mathit{CxtRed}}$ is obtained by adding the rules in Figure 3.11 to the equations of *s2c* and *c2s*.

The $\mathcal{R}_{\mathit{CxtRed}}$ definition is a faithful representation of context reduction semantics: indeed, it is easy to see that *s2c* recursively finds the redex taking into account the syntactic rules defining a context in the same way a parser would, and in the same way as other current implementations of this technique do it. Also, since parsing issues are abstracted away using equations, the computational granularity is the same, yielding a one-to-one correspondence between the computations performed by the context reduction semantics rules and those performed by the rewriting rules.

Theorem 4. *Suppose that $s \simeq \sigma$. Then the following hold:*

1. *$\langle p, \sigma\rangle$ parses in CxtRed as $\langle c, \sigma\rangle[r]$ iff $\mathcal{R}_{\mathit{CxtRed}} \vdash \mathit{s2c}(\langle p, s\rangle) = \langle c, s\rangle[r]$;*

2. *$\mathcal{R}_{\mathit{CxtRed}} \vdash \mathit{c2s}(c[r]) = c[r/[\,]]$ for any valid context c and appropriate redex r;*

3. *CxtRed $\vdash \langle p, \sigma\rangle \to \langle p', \sigma'\rangle$ iff $\mathcal{R}_{\mathit{CxtRed}} \vdash \cdot(\langle p, s\rangle) \to^1 \langle p', s'\rangle$ and $s' \simeq \sigma'$;*

4. *CxtRed $\vdash \langle p, \sigma\rangle \to \langle i\rangle$ iff $\mathcal{R}_{\mathit{CxtRed}} \vdash \cdot(\langle p, s\rangle) \to^1 \langle i\rangle$;*

$$\begin{array}{rl}
\cdot(I_1 + I_2) \to (I_1 +_{Int} I_2) & \\
\cdot(\langle P,S\rangle[X]) \to \langle P,S\rangle[(S[X])] & \\
\cdot(\langle P,S\rangle[\texttt{++}X]) \to \langle P,S[X \leftarrow I]\rangle[I] \ \textbf{if}\ I = s(S[X]) & \\
\hline
\cdot(I_1\texttt{<=}I_2) \to (I_1 \leq_{Int} I_2) & \\
\cdot(\texttt{trueand}\ B) \to B & \\
\cdot(\texttt{falseand}\ B) \to \texttt{false} & \\
\cdot(\texttt{nottrue}) \to \texttt{false} & \\
\cdot(\texttt{notfalse}) \to \texttt{true} & \\
\hline
\cdot(\texttt{iftruethen}\ St_1\ \texttt{else}\ St_2) \to St_1 & \\
\cdot(\texttt{iffalsethen}\ St_1\ \texttt{else}\ St_2) \to St_2 & \\
\cdot(\texttt{skip}; St) \to St & \\
\cdot(\{St\}) \to St & \\
\cdot(\langle P,S\rangle[X\texttt{:=}I]) \to \langle P,S[X \leftarrow I]\rangle[\texttt{skip}] & \\
\cdot(\texttt{while}\ B\ St) & \\
\to \texttt{if}\ B\ \texttt{then}\ (St; \texttt{while}\ B\ St)\ \texttt{else}\ \texttt{skip} & \\
\cdot(C[\texttt{halt}\ I]) \to \langle I\rangle[[]] & \\
\hline
\cdot(C[\texttt{skip}.I]) \to \langle I\rangle[[]] & \\
\hline
\cdot(C[R]) \to C[R'] \ \textbf{if}\ \cdot(R) \to R' & \\
\cdot(Cfg) \to c2s(C[R]) \ \textbf{if}\ \cdot(s2c(Cfg)) \to C[R] & \\
\hline
eval(P) = reduction(\langle P, \emptyset\rangle) & \\
reduction(Cfg) = reduction(\cdot(Cfg)) & \\
reduction(\langle I\rangle) = I & \\
\hline
\end{array}$$

Figure 3.11: $\mathcal{R}_{CxtRed}$ rewriting logic theory

5. *$CxtRed \vdash \langle p, \bot\rangle \to^* \langle i\rangle$ iff $\mathcal{R}_{CxtRed} \vdash eval(p) \to i$.*

Proof. 1. By induction on the number of context productions applied to parse the context, which is the same as the length of the derivation of $\mathcal{R}_{CxtRed} \vdash s2c(syn) = c[r]$, respectively, for each syntactical construct *syn*. We only show some of the more interesting cases.

Case ++ x: $\texttt{++}x$ parses as $[][\texttt{++}x]$. Also $\mathcal{R}_{CxtRed} \vdash s2c(\texttt{++}x) = [][\texttt{++}x]$ in one step (it is an instance of an axiom).

Case a_1 <= a_2: a_1 <= a_2 parses as a_1 <= $c[r]$ iff
$a_1 \in Int$ and a_2 parses as $c[r]$ iff
$a_1 \in Int$ and $\mathcal{R}_{CxtRed} \vdash s2c(a_2) = c[r]$ iff
$\mathcal{R}_{CxtRed} \vdash s2c(a_1\texttt{<=}a_2) = (a_1\texttt{<=}c)[r]$.

Case x := a: x := a parses as $[][x\texttt{:=}a]$ iff $a \in Int$, iff
$\mathcal{R}_{CxtRed} \vdash s2c(x\texttt{:=}i) = [][x\texttt{:=}i]$.

Case $st.a$: $st.a$ parses as $st.c[r]$ iff
$st = \texttt{skip}$ and a parses as $c[r]$, iff
$st = \texttt{skip}$ and $\mathcal{R}_{CxtRed} \vdash s2c(a) = c[r]$ iff
$\mathcal{R}_{CxtRed} \vdash s2c(at.a) = st.c[r]$.

Case $\langle p, \sigma\rangle$: $\langle p, \sigma\rangle$ parses as $c[r]$ iff
p parses as $c'[r]$ and $c = \langle c', s\rangle$ iff

$\mathcal{R}_{\mathit{CxtRed}} \vdash \mathit{s2c}(p) = c'[r]$ and $c = \langle c', s\rangle$ iff
$\mathcal{R}_{\mathit{CxtRed}} \vdash \mathit{s2c}(\langle p, s\rangle) = \langle c', s\rangle[r]$.

2. From the way it was defined, *c2s* acts as a morphism on the structure of syntactic constructs, changing $[]$ in C by R. Since *c2s* is defined for all constructors, it will work for any valid context C and pluggable expression e. Note, however, that *c2s* works as stated also on multi-contexts (i.e., on contexts with multiple holes), but this aspect does not interest us here.

3. There are several cases again to analyze, depending on the particular reduction that provoked the derivation $\mathit{CxtRed} \vdash \langle p, \sigma\rangle \to \langle p', \sigma\rangle$. We only discuss some cases; the others are treated similarly.

   $\mathit{CxtRed} \vdash \langle p, \sigma\rangle \to \langle p', \sigma'\rangle$ because of $\mathit{CxtRed} \vdash \langle c, \sigma\rangle[x] \to \langle c, \sigma\rangle[\sigma(x)]$ iff
   $\langle p, \sigma\rangle$ parses as $\langle c, \sigma\rangle[x]$ and $\langle p', \sigma'\rangle$ is $\langle c, \sigma\rangle[\sigma(x)]$ (in particular $\sigma' = \sigma$) iff
   $\mathcal{R}_{\mathit{CxtRed}} \vdash \mathit{s2c}(\langle p, s\rangle) = \langle c, s\rangle[x]$, $\mathcal{R}_{\mathit{CxtRed}} \vdash s[x] = i$ where $i = \sigma(x)$ and $\mathcal{R}_{\mathit{CxtRed}} \vdash \mathit{c2s}(\langle c, s\rangle[i]) = \langle p', s\rangle$ iff
   $\mathcal{R}_{\mathit{CxtRed}} \vdash \cdot(\langle p, s\rangle) \to^1 \langle p', s\rangle$, because $\mathcal{R}_{\mathit{CxtRed}} \vdash \cdot(\langle c, s\rangle[x]) \to^1 \langle c, s\rangle[i]$.

   $\mathit{CxtRed} \vdash \langle p, \sigma\rangle \to \langle p', \sigma\rangle$ because of $\frac{\texttt{not true}\to\texttt{false}}{c[\texttt{not true}]\to c[\texttt{false}]}$ for some evaluation context c iff
   $\langle p, \sigma\rangle$ parses as $c[\texttt{not true}]$ and $\langle p', \sigma\rangle$ is $c[\texttt{false}]$ iff
   $\mathcal{R}_{\mathit{CxtRed}} \vdash \mathit{s2c}(\langle p, s\rangle) = c[\texttt{not true}]$ and $\mathcal{R}_{\mathit{CxtRed}} \vdash \mathit{c2s}(c[\texttt{false}]) = \langle p', s\rangle$ iff
   $\mathcal{R}_{\mathit{CxtRed}} \vdash \cdot(\langle p, s\rangle) \to^1 \langle p', s\rangle$, because $\mathcal{R}_{\mathit{CxtRed}} \vdash \cdot(c[\texttt{not true}]) \to^1 c[\texttt{false}]$ (which follows since $\mathcal{R}_{\mathit{CxtRed}} \vdash \cdot(\texttt{not true}) \to^1 \texttt{false}$).

   $\mathit{CxtRed} \vdash \langle p, \sigma\rangle \to \langle p', \sigma'\rangle$ because of
   $\mathit{CxtRed} \vdash \langle c, \sigma\rangle[x\texttt{:=}i] \to \langle c, \sigma[i/x][\texttt{skip}]\rangle$ iff
   $\langle p, \sigma\rangle$ parses as $\langle c, \sigma\rangle[x\texttt{:=}i]$, $\sigma' = \sigma[i/x]$ and $\langle p', \sigma'\rangle$ is $\langle c, \sigma'\rangle[\texttt{skip}]$ iff
   $\mathcal{R}_{\mathit{CxtRed}} \vdash \mathit{s2c}(\langle p, s\rangle) = \langle c, s\rangle[x\texttt{:=}i]$, $s' = s[x \leftarrow i] \simeq \sigma'$ and $\mathcal{R}_{\mathit{CxtRed}} \vdash \mathit{c2s}(\langle c, s'\rangle[\texttt{skip}]) = \langle p', s'\rangle$ iff
   $\mathcal{R}_{\mathit{CxtRed}} \vdash \cdot(\langle p, s\rangle) \to^1 \langle p', s'\rangle$, because
   $\mathcal{R}_{\mathit{CxtRed}} \vdash \cdot(\langle c, s\rangle[x\texttt{:=}i]) \to^1 \langle c, s'\rangle[\texttt{skip}]$.

4. $\mathit{CxtRed} \vdash \langle p, \sigma\rangle \to \langle i\rangle$ because of $\mathit{CxtRed} \vdash c[\texttt{skip}.i] \to \langle i\rangle$ iff
   $\langle p, \sigma\rangle$ parses as $\langle [], \sigma\rangle[\texttt{skip}.i]$ iff
   $\mathcal{R}_{\mathit{CxtRed}} \vdash \mathit{s2c}(\langle p, s\rangle) = \langle [], s\rangle[\texttt{skip}.i]$ iff
   $\mathcal{R}_{\mathit{CxtRed}} \vdash \cdot(\langle p, s\rangle) = \langle i\rangle$, since $\mathcal{R}_{\mathit{CxtRed}} \vdash \cdot(\langle [], \sigma\rangle[\texttt{skip}.i]) \to^1 \langle i\rangle[[]]$ and since $\mathcal{R}_{\mathit{CxtRed}} \vdash \mathit{c2s}(\langle i\rangle[[]]) = \langle i\rangle$.

   Also, $\mathit{CxtRed} \vdash \langle p, \sigma\rangle \to \langle i\rangle$ because of $\mathit{CxtRed} \vdash c[\texttt{halt}\ \ i] \to \langle i\rangle$ iff
   $\langle p, \sigma\rangle$ parses as $\langle c, \sigma\rangle[\texttt{halt}\ \ i]$ iff
   $\mathcal{R}_{\mathit{CxtRed}} \vdash \mathit{s2c}(\langle p, s\rangle) = \langle c, s\rangle[\texttt{halt}\ \ i]$ iff
   $\mathcal{R}_{\mathit{CxtRed}} \vdash \cdot(\langle p, s\rangle) = \langle i\rangle$ since $\mathcal{R}_{\mathit{CxtRed}} \vdash \cdot(\langle c, \sigma\rangle[\texttt{halt}\ \ i]) \to^1 \langle i\rangle[[]]$ and since $\mathcal{R}_{\mathit{CxtRed}} \vdash \mathit{c2s}(\langle i\rangle[[]]) = \langle i\rangle$.

5. This part of the proof follows the same pattern as that for the similar property for *SmallStep* (Proposition 4), using the above properties and replacing *smallstep* by *reduction*.

□

Strengths. Context reduction semantics divides SOS rules into computational rules and rules needed to find the redex; the latter are transformed into grammar rules generating the allowable contexts. This makes definitions more compact. It improves over SOS semantics by allowing the context to be changed by execution rules. It can easily deal with control-intensive features. It is more modular than SOS.

Weaknesses. It still only allows "interleaving semantics" for concurrency. Although context-sensitive rewriting might seem to be easily implementable by rewriting, in fact all current implementations of context reduction work by transforming context grammar definitions into traversal functions, thus being as (in)efficient as the small-step implementations (one has to perform an amount of work linear in the size of the program for each computational step). However, one might obtain efficient implementations for restricted forms of context-reduction definitions by applying refocusing techniques [41].

3.7 A Continuation-Based Semantics

The idea of continuation-based interpreters for programming languages and their relation to abstract machines has been well studied (see, for example, [59]). In this section we propose a rewriting logic theory based on a structure that provides a *first-order* representation of continuations [179, 60]; this is the only reason why we call this structure a "continuation"; but notice that it can just as well be regarded as a post-order representation of the abstract syntax tree of the program, so one needs no prior knowledge of continuations [59] in order to understand this section. In particular, the definition presented here is very close to the representation of $\mathbb{K}$ computations described in the following chapter, distinguishing itself from the traditional first-order representation of continuations by the fact that the redex is here kept with the continuation, more precisely at its top. We will show the equivalence of this theory to the context reduction semantics theory.

Based on the desired order of evaluation, the program is sequentialized by transforming it into a list of tasks to be performed in order. This is done once and for all at the beginning, the benefit being that at any subsequent moment in time we know precisely where the next redex is: at the top of the list of tasks. We call this list of tasks a *continuation*, but is nothing more than a pure first-order flattening of the program and can be easily introduced without appealing to high-order constructs. For example $aexp(A_1 + A_2) = (aexp(A_1), aexp(A_2)) \curvearrowright +$

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{aexp}(I) = I} \\
\multicolumn{2}{l}{\mathit{aexp}(A_1 + A_2) = (\mathit{aexp}(A_1), \mathit{aexp}(A_2)) \curvearrowright +} \\
\multicolumn{2}{l}{k(\mathit{aexp}(X) \curvearrowright K)\ \mathit{store}(\mathit{Store}) \to k(\mathit{Store}[X] \curvearrowright K)\ \mathit{store}(\mathit{Store})} \\
\multicolumn{2}{l}{k(\mathit{aexp}(\texttt{++}X) \curvearrowright K)\ \mathit{store}((X = I)\,\mathit{Store})} \\
\multicolumn{2}{r}{\to k(s(I) \curvearrowright K)\ \mathit{store}((X = s(I))\,\mathit{Store})} \\
\multicolumn{2}{l}{k(I_1, I_2 \curvearrowright + \curvearrowright K) \to k(I_1 +_{\mathit{Int}} I_2 \curvearrowright K)} \\
\hline
\mathit{bexp}(\mathit{true}) = \mathit{true} & \mathit{bexp}(\mathit{false}) = \mathit{false} \\
\multicolumn{2}{l}{\mathit{bexp}(A_1 \texttt{<=} A_2) = (\mathit{aexp}(A_1), \mathit{aexp}(A_2)) \curvearrowright \leq} \\
\multicolumn{2}{l}{\mathit{bexp}(B_1 \ \texttt{and}\ B_2) = \mathit{bexp}(B_1) \curvearrowright \mathit{and}(\mathit{bexp}(B_2))} \\
\multicolumn{2}{l}{\mathit{bexp}(\texttt{not}\ B) = \mathit{bexp}(B) \curvearrowright \mathit{not}} \\
\multicolumn{2}{l}{k(I_1, I_2 \curvearrowright \leq \curvearrowright K) \to k(I_1 \leq_{\mathit{Int}} I_2 \curvearrowright K)} \\
\multicolumn{2}{l}{k(\mathit{true} \curvearrowright \mathit{and}(K_2) \curvearrowright K) \to k(K_2 \curvearrowright K)} \\
\multicolumn{2}{l}{k(\mathit{false} \curvearrowright \mathit{and}(K_2) \curvearrowright K) \to k(\mathit{false} \curvearrowright K)} \\
\multicolumn{2}{l}{k(T \curvearrowright \mathit{not} \curvearrowright K) \to k(\mathit{not}_{\mathit{Bool}} T \curvearrowright K)} \\
\hline
\multicolumn{2}{l}{\mathit{stmt}(\texttt{skip}) = \mathit{nothing}} \\
\multicolumn{2}{l}{\mathit{stmt}(X := A) = \mathit{aexp}(A) \curvearrowright \mathit{write}(X)} \\
\mathit{stmt}(\mathit{St}_1; \mathit{St}_2) = \mathit{stmt}(\mathit{St}_1) \curvearrowright \mathit{stmt}(\mathit{St}_2) & \mathit{stmt}(\{\mathit{St}\}) = \mathit{stmt}(\mathit{St}) \\
\multicolumn{2}{l}{\mathit{stmt}(\texttt{if}\ B\ \texttt{then}\ \mathit{St}_1\ \texttt{else}\ \mathit{St}_2) = \mathit{bexp}(B) \curvearrowright \mathit{if}(\mathit{stmt}(\mathit{St}_1), \mathit{stmt}(\mathit{St}_2))} \\
\multicolumn{2}{l}{\mathit{stmt}(\texttt{while}\ B\ \mathit{St}) = \mathit{bexp}(B) \curvearrowright \mathit{while}(\mathit{bexp}(B), \mathit{stmt}(\mathit{St}))} \\
\multicolumn{2}{l}{\mathit{stmt}(\texttt{halt}\ A) = \mathit{aexp}(A) \curvearrowright \mathit{halt}} \\
\multicolumn{2}{l}{k(I \curvearrowright \mathit{write}(X) \curvearrowright K)\ \mathit{store}(\mathit{Store}) \to k(K)\ \mathit{store}(\mathit{Store}[X \leftarrow I])} \\
\multicolumn{2}{l}{k(\mathit{true} \curvearrowright \mathit{if}(K_1, K_2) \curvearrowright K) \to k(K_1 \curvearrowright K)} \\
\multicolumn{2}{l}{k(\mathit{false} \curvearrowright \mathit{if}(K_1, K_2) \curvearrowright K) \to k(K_2 \curvearrowright K)} \\
\multicolumn{2}{l}{k(\mathit{true} \curvearrowright \mathit{while}(K_1, K_2) \curvearrowright K) \to k(K_2 \curvearrowright K_1 \curvearrowright \mathit{while}(K_1, K_2) \curvearrowright K)} \\
\multicolumn{2}{l}{k(\mathit{false} \curvearrowright \mathit{while}(K_1, K_2) \curvearrowright K) \to k(K)} \\
\multicolumn{2}{l}{k(I \curvearrowright \mathit{halt} \curvearrowright K) \to k(I)} \\
\hline
\multicolumn{2}{l}{\mathit{pgm}(\mathit{St}.A) = \mathit{stmt}(\mathit{St}) \curvearrowright \mathit{aexp}(A)} \\
\hline
\multicolumn{2}{l}{\langle P \rangle = \mathit{result}(k(\mathit{pgm}(P))\ \mathit{store}(\mathit{empty}))} \\
\multicolumn{2}{l}{\mathit{result}(k(I)\ \mathit{store}(\mathit{Store})) = I} \\
\hline
\end{array}$$

Figure 3.12: Rewriting logic theory $\mathcal{R}_K$ (continuation-based definition)

precisely encodes the order of evaluation: first A_1, then A_2, then add the values. Also, $\mathit{stmt}(\texttt{if}\ B\ \texttt{then}\ St_1\ \texttt{else}\ St_2) = B \curvearrowright \mathit{if}(\mathit{stmt}(St_1), \mathit{stmt}(St_2))$ says that St_1 and St_2 are dependent on the value of B for their evaluation. The fact that we denote the above relation by equality, although we operationally interpret it from left to right, indicates that the two terms are structurally equal (and in fact, they *are* equal in the initial model of the specification) —at any time during the evaluation one could apply the equations backwards and reconstitute the current state of the program being executed.

The top level configuration is constructed by an operator "$\_\,\_$" putting together the store (wrapped by a constructor *store*) and the continuation (wrapped by k). Also, syntax is added for the continuation items. Here the distinction between equations and rules becomes even more obvious: equations are used to prepare the context in which a computation step can be applied, while rewrite rules exactly encode the computation steps semantically, yielding the intended computational granularity. Specifically *pgm*, *stmt*, *bexp*, *aexp* are used to flatten the program to a continuation, taking into account the order of evaluation. The continuation is defined as a list of tasks, where the list constructor "$\_ \curvearrowright \_$" is associative, having as identity a constant "*nothing*". We also use lists of values and continuations, each having an associative list append constructor "$\_,\_$" with identity ".". We use variables K and V to denote continuations and values, respectively; also, we use *Kl* and *Vl* for lists of continuations and values, respectively. The rewrite theory $\mathcal{R}_K$ specifying the continuation-based definition of our example language is given in Figure 3.12. Lists of expressions are evaluated using the following (equationally defined) mechanism:

$$k((Vl, Ke, Kel) \curvearrowright K) = k(Ke \curvearrowright (Vl, \mathit{nothing}, Kel) \curvearrowright K)$$

Because in rewriting engines equations are also executed by rewriting, one would need to split the above rule into two rules:

$$\begin{aligned}
&k((Vl, Ke, Kel) \curvearrowright K = k(Ke \curvearrowright (Vl, \mathit{nothing}, Kel) \curvearrowright K)\\
&k(V \curvearrowright (Vl, \mathit{nothing}, Kel) \curvearrowright K) = k((Vl, V, Kel) \curvearrowright K)
\end{aligned}$$

The semantics we obtain here for this simple sequential language is an abstract machine, similar in spirit to the one obtainable by applying CPS transformers on an interpreter as in [150] or that obtained by applying refocusing [41] on the context-reduction definition. One slight difference is that we keep the state and the continuation as distinct entities at the top level, rather than embedding the state as part of the context/continuation structure. In a computational logic framework like rewriting logic where the gap between ”implementations” and ”specifications” is almost inexistent, this continuation-like style can be used to define languages, not only to efficiently interpret them.

An important benefit of this definitional style is that of gaining locality. Now one needs to specify from the context only what is needed to perform the computation. This indeed gives the possibility of achieving “true concurrency”, since rules which do not act on the same parts of the context can be applied in parallel. That is one of the main reasons for which the $\mathbb{K}$ technique choses to represent computations using a similar first-order representation of continuations.

Strengths. In continuation-based semantics there is no need to search for a redex anymore, because the redex is always at the top. It is much more efficient than *direct* implementations of evaluation contexts or small-step SOS. Also, this style greatly reduces the need for conditional rules/equations; conditional rules/equations might involve inherently inefficient reachability analysis to check the conditions and are harder to deal with in parallel environments. An important “strength” specific to the rewriting logic approach is that reductions can now apply wherever they match, in a *context-insensitive* way. Additionally, continuation-based definitions in the RLS style above are very modular (particularly due to the use of matching modulo associativity and commutativity).

Weaknesses. The program is now hidden in the continuation: one has to either learn to like it like this, or to write a backwards mapping to retrieve programs from continuations[3]; to flatten the program into a continuation structure, several new operations (continuation constants) need to be introduced, which “replace” the corresponding original language constructs.

Relation with Context Reduction

We next show the equivalence between the continuation-based and the context reduction rewriting logic definitions. The specification in Figure 3.13 relates the two semantics, showing that at each computational “point” it is possible to extract from our continuation structure the current expression being evaluated. For each syntactical construct $Syn \in \{AExp, BExp, Stmt, Pgm\}$, we equationally define two (partial) functions:

- *k2Syn* takes a continuation encoding of *Syn* into *Syn*; and
- *kSyn* extracts from the tail of a continuation a *Syn* and returns it together with the remainder prefix continuation.

Together, these two functions can be regarded as a parsing process, where the continuation plays the role of “unparsed” syntax, while *Syn* is the abstract syntax tree, i.e., the “parsed” syntax. The formal definitions of *k2Syn* and *kSyn* are given in Figure 3.13.

[3] However, we regard these as minor syntactic details. After all, the program needs to be transformed into an abstract syntax tree (AST) in any of the previous formalisms. Whether the AST is kept in prefix versus postfix order is somewhat irrelevant.

$$\mathit{k2Pgm}(K) = \mathit{k2Stmt}(K').A \textbf{ if } \{K', A\} = \mathit{kAExp}(K)$$

$$\mathit{k2Stmt}(\mathit{nothing}) = \texttt{skip}$$
$$\mathit{k2Stmt}(K) = \mathit{k2Stmt}(K'); \mathit{St} \textbf{ if } \{K', \mathit{St}\} = \mathit{kStmt}(K) \wedge K' \neq \mathit{nothing}$$
$$\mathit{k2Stmt}(K) = \mathit{St} \textbf{ if } \{K', \mathit{St}\} = \mathit{kStmt}(K) \wedge K' = \mathit{nothing}$$
$$\mathit{kStmt}(K \curvearrowright \mathit{write}(X)) = \{K', X \texttt{:=} A\} \textbf{ if } \{K', A\} = \mathit{kAExp}(K)$$
$$\mathit{kStmt}(K \curvearrowright \mathit{while}(K_1, K_2)) = \{K', \texttt{if } B \texttt{ then } \{\mathit{St}; \texttt{while } B_1 \mathit{St}\} \texttt{ else skip}\}$$
$$\quad \textbf{if } \{K', B\} = \mathit{kBExp}(K) \wedge B_1 = \mathit{k2BExp}(K_1) \wedge \mathit{St} = \mathit{k2Stmt}(K_2) \wedge B \neq B_1$$
$$\mathit{kStmt}(K \curvearrowright \mathit{while}(K_1, K_2)) = \{K', \texttt{while } B\ \mathit{St}\}$$
$$\quad \textbf{if } \{K', B\} = \mathit{kBExp}(K) \wedge B_1 = \mathit{k2BExp}(K_1) \wedge \mathit{St} = \mathit{k2Stmt}(K_2) \wedge B = B_1$$
$$\mathit{kStmt}(K \curvearrowright \mathit{if}(K_1, K_2)) = \{K', \texttt{if } B \texttt{ then } \mathit{k2Stmt}(K_1) \texttt{ else } \mathit{k2Stmt}(K_2)\}$$
$$\quad \textbf{if } \{K', B\} = \mathit{kBExp}(K)$$
$$\mathit{kStmt}(K \curvearrowright \mathit{halt}) = \{K', \texttt{halt } A\} \textbf{ if } \{K', A\} = \mathit{kAExp}(K)$$

$$\mathit{k2AExp}(K) = A \textbf{ if } \{\mathit{nothing}, A\} = \mathit{kAExp}(K)$$
$$\mathit{kAExp}(K \curvearrowright \mathit{kv}(\mathit{Kl}, \mathit{Vl}) \curvearrowright K') = \mathit{kAExp}(\mathit{Vl}, K, \mathit{Kl} \curvearrowright K')$$

$$\mathit{kAExp}(K \curvearrowright \mathit{aexp}(A)) = \{K, A\}$$
$$\mathit{kAExp}(K \curvearrowright I) = \{K, I\}$$
$$\mathit{kAExp}(K \curvearrowright K_1, K_2 \curvearrowright +) = \{K, \mathit{k2AExp}(K_1) + \mathit{k2AExp}(K_2)\}$$

$$\mathit{k2BExp}(K) = B \textbf{ if } \{\mathit{nothing}, B\} = \mathit{kBExp}(K)$$
$$\mathit{kBExp}(K \curvearrowright \mathit{kv}(\mathit{Kl}, \mathit{Vl}) \curvearrowright K') = \mathit{kBExp}(\mathit{Vl}, K, \mathit{Kl} \curvearrowright K')$$

$$\mathit{kBExp}(K \curvearrowright T) = \{K, T\}$$
$$\mathit{kBExp}(K \curvearrowright K_1, K_2 \curvearrowright \leq) = \{K, \mathit{k2AExp}(K_1) \texttt{<=} \mathit{k2AExp}(K_2)\}$$
$$\mathit{kBExp}(K \curvearrowright \mathit{and}(K_2)) = \{K_1, B_1 \texttt{ and } \mathit{k2BExp}(K_2)\} \textbf{ if } \{K_1, B_1\} = \mathit{kBExp}(K)$$
$$\mathit{kBExp}(K \curvearrowright \mathit{not}) = \{K', \texttt{not } B\} \textbf{ if } \{K', B\} = \mathit{kBExp}(K)$$

Figure 3.13: Recovering the abstract syntax trees from continuations

We will show below that for any step *CxtRed* takes, $\mathcal{R}_K$ performs at most one step to reach the same[4] configuration. No steps are performed for `skip`, or for dissolving a block (because these were dealt with when we transformed the syntax into continuation form), or for dissolving a statement into a skip (there is no need for that when using continuations). Also, no steps will be performed for loop unrolling, because this is *not* a computational step; it is a straightforward structural equivalence. In fact, note that, because of its incapacity to distinguish between computational steps and structural equivalences, *CxtRed* does not capture the intended granularity of `while`: it wastes a computation step for unrolling the loop and one when dissolving the while into skip; neither of these steps has any computational content.

In order to clearly explain the relation between reduction contexts and continuations, we go a step further and define a new rewrite theory $\mathcal{R}_{K'}$ which, besides identifying `while` with its unrolling, adds to $\mathcal{R}_K$ the idea of contexts, holes, and pluggable expressions. More specifically, we add a new constant "[]" and the following equation, again for each syntactical category *Syn*:

$$k(\mathit{syn}(\mathit{Syn}) \curvearrowright K') = k(\mathit{syn}(\mathit{Syn}) \curvearrowright \mathit{syn}([]) \curvearrowright K'),$$

replacing the equation for evaluating lists of expressions, namely,

$$k((\mathit{Vl}, \mathit{Ke}, \mathit{Kel}) \curvearrowright K) = k(\mathit{Ke} \curvearrowright (\mathit{Vl}, \mathit{nothing}, \mathit{Kel}) \curvearrowright K),$$

by the following equation which puts in a hole instead of nothing:

$$k((\mathit{Vl}, \mathit{Ke}, \mathit{Kel}) \curvearrowright K) = k(\mathit{Ke} \curvearrowright (\mathit{Vl}, \mathit{syn}([]), \mathit{Kel}) \curvearrowright K)$$

The intuition for the first rule is that, as we will next show, for any well-formed continuation (i.e., one obtained from a syntactic entity) having a syntactic entity as its prefix, its corresponding suffix represents a valid context where the prefix syntactic entity can be plugged in. As expected, $\mathcal{R}_{K'}$ does not bring any novelty to $\mathcal{R}_K$, that is,for any term t in $\mathcal{R}_K$, $\mathit{Tree}_{\mathcal{R}_K}(t)$ is bisimilar to $\mathit{Tree}_{\mathcal{R}_{K'}}(t)$.

Proposition 5. *For each arithmetic context c in CxtRed and $r \in AExp$, we have that $\mathcal{R}_{K'} \vdash k(\mathit{aexp}(c[r])) = k(\mathit{aexp}(r) \curvearrowright \mathit{aexp}(c)))$. Similarly for any possible combination for c and r among AExp, BExp, Stmt, Pgm, Cfg.*

(Note that r in the proposition above needs not be a redex, but can be any expression of the right syntactical category, i.e., pluggable in the hole.)

Proof. $\texttt{++}x = [][\texttt{++}x]$**:** $\mathcal{R}_{K''} \vdash k(\mathit{aexp}(\texttt{++}x)) = k(\mathit{aexp}(\texttt{++}x) \curvearrowright \mathit{aexp}([]))$

$a_1 + a_2 = [] + a_2[a_1]$**:** $\mathcal{R}_{K''} \vdash k(\mathit{aexp}(a_1 + a_2)) = k((\mathit{aexp}(a_1), \mathit{aexp}(a_2)) \curvearrowright +)$
$\quad = k(\mathit{aexp}(a_1) \curvearrowright (\mathit{aexp}([]), \mathit{aexp}(a_2)) \curvearrowright +) = k(\mathit{aexp}(a_1) \curvearrowright \mathit{aexp}([] + a_2))$

[4] "same" modulo irrelevant but equivalent syntactic notational conventions.

$i_1 + a_2 = i_1 + [][a_2]$**:** $\mathcal{R}_{K''} \vdash k(\mathit{aexp}(i_1 + a_2)) = k((\mathit{aexp}(i_1), \mathit{aexp}(a_2)) \curvearrowright +)$
$\quad = k(\mathit{aexp}(a_2) \curvearrowright (i_1, \mathit{aexp}([])) \curvearrowright +) = k(\mathit{aexp}(a_2) \curvearrowright \mathit{aexp}(i_1 + [])).$

b_1 `and` $b_2 = []$ `and` $b_2[b_1]$**:**
$\quad \mathcal{R}_{K''} \vdash k(\mathit{bexp}(b_1 \ \mathtt{and}\ b_2)) = k(\mathit{bexp}(b_1) \curvearrowright \mathit{and}(\mathit{bexp}(b_2)))$
$\quad = k(\mathit{bexp}(b_1) \curvearrowright \mathit{bexp}([]) \curvearrowright \mathit{and}(\mathit{aexp}(b_2))) = k(\mathit{bexp}(b_1) \curvearrowright \mathit{bexp}([]\mathtt{and}\ b_2)).$

t `and` $b_2 = [][t$ `and` $b_2]$**:**
$\quad \mathcal{R}_{K''} \vdash k(\mathit{bexp}(t \ \mathtt{and}\ b_2)) = k(\mathit{bexp}(t \ \mathtt{and}\ b_2) \curvearrowright \mathit{bexp}([])).$

$st.a = [].a[st]$**:** $\mathcal{R}_{K''} \vdash k(\mathit{pgm}(st.a)) = k(\mathit{stmt}(st) \curvearrowright \mathit{aexp}(a))$
$\quad = k(\mathit{stmt}(st) \curvearrowright \mathit{stmt}([]) \curvearrowright \mathit{aexp}(a)) = k(\mathit{stmt}(st) \curvearrowright \mathit{pgm}([].a)).$

$\mathtt{skip}.a = \mathtt{skip}.[][a]$**:** $\mathcal{R}_{K''} \vdash k(\mathit{pgm}(\mathtt{skip}.a)) = k(\mathit{stmt}(\mathit{skip}) \curvearrowright \mathit{aexp}(a))$
$\quad = k(\mathit{aexp}(a)) = k(\mathit{aexp}(a) \curvearrowright \mathit{aexp}([]))$
$\quad = k(\mathit{aexp}(a) \curvearrowright \mathit{stmt}(\mathit{skip}) \curvearrowright \mathit{aexp}([])) = k(\mathit{aexp}(a) \curvearrowright \mathit{pgm}(\mathtt{skip}.[])).$

All other constructs are dealt with in a similar manner. □

Lemma 1. *$\mathcal{R}_{K'} \vdash k(k_1) = k(k_2)$ implies that for any k_{rest}, $\mathcal{R}_{K'} \vdash k(k_1 \curvearrowright k_{rest}) = k(k_2 \curvearrowright k_{rest})$*

Proof. We can replay all steps in the first proof, for the second proof, since all equations only modify the head of a continuation. □

By structural induction on the equational definitions, thanks to the one-to-one correspondence of rewriting rules, we obtain the following result:

Theorem 5. *Suppose $s \simeq \sigma$.*

1. *If $\mathit{CxtRed} \vdash \langle p, \sigma \rangle \to \langle p', \sigma' \rangle$ then $\mathcal{R}_{K'} \vdash k(\mathit{pgm}(p))\, \mathit{store}(s) \to^{\leq 1} k(\mathit{pgm}(p'))\, \mathit{store}(s')$ and $s' \simeq \sigma'$, where $\to^{\leq 1} = \to^0 \cup \to^1$.*

2. *If $\mathcal{R}_{K'} \vdash k(\mathit{pgm}(p))\, \mathit{store}(s) \to k(k')\, \mathit{store}(s')$ then there exists p' and σ' such that $\mathit{CxtRed} \vdash \langle p, \sigma \rangle \to^* \langle p', \sigma' \rangle$, $\mathcal{R}_{K'} \vdash k(\mathit{pgm}(p')) = k(k')$ and $s' \simeq \sigma'$.*

3. *$\mathit{CxtRed} \vdash \langle p, \bot \rangle \to^* i$ iff $\mathcal{R}_{K'} \vdash \langle p \rangle \to i$ for any $p \in \mathit{Pgm}$ and $i \in \mathit{Int}$.*

Proof. (Sketch)

1. First, one needs to notice that rules in $\mathcal{R}_{K'}$ correspond exactly to those in *CxtRed*. For example, for $i_1 + i_2 \to i_1 +_{\mathit{Int}} i_2$, which can be read as $\langle c, \sigma \rangle[i_1 + i_2] \to \langle c, \sigma \rangle[i_1 +_{\mathit{Int}} i_2]$ we have the rule $k((i_1, i_2) \curvearrowright + \curvearrowright k_{rest}) \to k((i_1 +_{\mathit{Int}} i_2) \curvearrowright k_{rest})$ which, taking into account the above results, has, as a particular instance: $k(\mathit{pgm}(c[i_1 + i_2])) \to k(\mathit{pgm}(c[i_1 +_{\mathit{Int}} i_2])$. For $\langle c, \sigma \rangle[x\mathtt{:=}i] \to \langle c, \sigma[i/x] \rangle[\mathit{skip}]$ we have $k(i \curvearrowright \mathit{write}(x) \curvearrowright k)\ \mathit{store}(s) \to k(k)\ \mathit{store}(s[x \leftarrow i])$ which again has as an instance:
$k(\mathit{pgm}(c[x\mathtt{:=}i])\ \mathit{store}(s) \to k(c[\mathtt{skip})\ \mathit{store}(s[x \leftarrow i]).$

2. Actually σ' is uniquely determined by s' and p' is the program obtained by advancing p all non-computational steps —which were dissolved by *pgm*, or are equationally equivalent in $\mathcal{R}_{K'}$, such as unrolling the loops—, then performing the step similar to that in $\mathcal{R}_{K'}$.

3. Using the previous two statements, and the rules for halt or end of the program from both definitions. We exemplify only halt, the end of the program is similar, but simpler. For $\langle c, \sigma\rangle[\texttt{halt}\ i] \to i$ we have $k(i \curvearrowright \mathit{halt} \curvearrowright k) \to k(i)$, and combined with $\mathcal{R}_{K'} \vdash \mathit{result}(k(i)\ \mathit{store}(s)) = i$ we obtain $\mathcal{R}_{K'} \vdash \mathit{result}(k(\mathit{pgm}(c[\texttt{halt}\ i]))\ \mathit{store}(s)) \to i$.

□

3.8 The Chemical Abstract Machine

Berry and Boudol's *chemical abstract machine*, or *Cham* [15], is both a model of concurrency and a specific style of giving operational semantics definitions. Properly speaking, it is not an SOS definitional style. Berry and Boudol identify a number of limitations inherent in SOS, particularly its lack of true concurrency, and what might be called SOS's rigidity and slavery to syntax [15]. They then present the Cham as an *alternative* to SOS. In fact, as already pointed out in [103], what the Cham is, is a particular definitional style *within* RLS. That is, every Cham *is*, by definition, a specific kind of rewrite theory; and Cham computation is precisely concurrent rewriting computation; that is, proof in rewriting logic.

The basic metaphor giving its name to the Cham is inspired by Banâtre and Le Métayer's GAMMA language [10]. It views a distributed state as a "solution" in which many "molecules" float, and understands concurrent transitions as "reactions" that can occur simultaneously in many points of the solution. It is possible to define a variety of chemical abstract machines. Each of them corresponds to a rewrite theory satisfying certain common conditions.

There is a common syntax shared by all chemical abstract machines, with each machine possibly extending the basic syntax by additional function symbols. The common syntax is typed, and can be expressed as the following order-sorted signature Ω:

sorts Molecule, Molecules, Solution .
subsorts Solution $<$ *Molecule* $<$ *Molecules* .
op $\lambda :\longrightarrow$ *Molecules* .
op $\_,\_$: *Molecules Molecules* $\longrightarrow$ *Molecules* .
op $\{|\_|\}$: *Molecules* $\longrightarrow$ *Solution* . `*** membrane operator`
op $\_ \triangleleft \_$: *Molecule Solution* $\longrightarrow$ *Molecule* . `*** airlock operator`

A *Cham* is then a rewrite theory $\mathcal{C} = (\Sigma, AC, R)$, with $\Sigma \supseteq \Omega$, together with a partition $R = \mathit{Reaction} \uplus \mathit{Heating} \uplus \mathit{Cooling} \uplus \mathit{AirlockAx}$. The associativity and commutativity (AC) axioms are asserted of the operator $\_,\_$, which has identity

λ. The rules in R may involve variables, but are subject to certain syntactic restrictions that guarantee an efficient form of AC matching [15]. *AirlockAx* is the bidirectional rule[5] $\{\!|m, M|\!\} \rightleftharpoons \{\!|m \triangleright \{\!|M|\!\}|\!\}$, where m is a variable of sort *Molecule* and M a variable of sort *Molecules*. The purpose of this axiom is to choose one of the molecules m in a solution as a candidate for reaction with other molecules outside its membrane. The *Heating* and *Cooling* rules can typically be paired, with each rule $t \longrightarrow t' \in \mathit{Heating}$ having a symmetric rule $t' \longrightarrow t \in \mathit{Cooling}$, and vice-versa, so that we can view them as a single set of bidirectional rules $t' \rightleftharpoons t$ in *Heating-Cooling*.

Berry and Boudol [15] make a distinction between *rules*, which are rewrite rules specific to each Cham —and consist of the *Reaction*, *Heating*, and *Cooling* rules— and *laws* which are general properties applying to all Chams for governing the admissible transitions. The first three laws, the *Reaction*, *Chemical* and *Membrane* laws, just say that the Cham evolves by AC-rewriting. The fourth law states the axiom *AirlockAx*. The *Reaction* rules are the heart of the Cham and properly correspond to state transitions. The rules in *Heating-Cooling* express *structural equivalence*, so that the *Reaction* rules may apply after the appropriate structurally equivalent syntactic form is found. A certain strategy is typically given to address the problem of finding the right structural form, for example to perform "heating" as much as possible. In rewriting logic terms, a more abstract alternative view is to regard each Cham as a rewrite theory $\mathcal{C} = (\Sigma, \mathit{ACI} \cup \mathit{Heating\text{-}Cooling} \cup \mathit{AirlockAx}, \mathit{Reaction})$, in which the *Heating-Cooling* rules and the *AirlockAx* axiom have been made part of the theory's equational axioms. That is, we can more abstractly view the *Reaction* rules as applied *modulo* $\mathit{ACI} \cup \mathit{Heating\text{-}Cooling} \cup \mathit{AirlockAx}$.

As Berry and Boudol demonstrate in [15], the Cham is particularly well-suited to give semantics to concurrent calculi, yielding considerably simpler definitions than those afforded by SOS. In particular, [15] presents semantic definitions for the TCCS variant of CCS, a concurrent λ-calculus, and Milner's π-calculus. Milner himself also used Cham ideas to provide a compact formulation of his π-calculus [114]. Since our example language is sequential, it cannot take full advantage of the Cham's true concurrent capabilities. Nevertheless, there are interesting Cham features that, as we explain below, turn out to be useful even in this sequential language application. A Cham semantics for our language is given in Figure 3.14. Note that, since the Cham is itself a rewrite theory, in this case there is no need for a representation in RLS, nor for a proof of correctness of such a representation; that is, the "representational distance" in this case is equal to 0. Again, RLS does not advocate any particular definitional style: the Cham style is just one possibility among many, having its own advantages and limitations.

The *Cham* definition for our simple programming language takes the *CxtRed* definition in Figure 3.8 as a starting point. More precisely, we follow the "refocusing" technique [41]. We distinguish two kinds of molecules: syntactic

[5] Which is of course understood as a pair of rules, one in each direction.

$$
\begin{array}{l}
St.A \rightleftharpoons [St \mid [[].A]] \\
\texttt{skip}.A \rightleftharpoons [A \mid [\texttt{skip}.[]]] \\
[X\,\texttt{:=}A \mid C] \rightleftharpoons [A \mid [X\,\texttt{:=}[] \mid C]] \\
[St_1; St_2 \mid C] \rightleftharpoons [St_1 \mid [[]; St_2 \mid C]] \\
[\texttt{if}\ B\ \texttt{then}\ St_1\ \texttt{else}\ St_2 \mid C] \rightleftharpoons [B \mid [\texttt{if}\ []\ \texttt{then}\ St_1\ \texttt{else}\ St_2 \mid C]] \\
[\texttt{halt}\ A \mid C] \rightleftharpoons [A \mid [\texttt{halt}\ [] \mid C]] \\
[A_1\texttt{<=}A_2 \mid C] \rightleftharpoons [A_1 \mid [[]\texttt{<=}A_2 \mid C]] \\
[I\texttt{<=}A \mid C] \rightleftharpoons [A \mid [I\texttt{<=}[] \mid C]] \\
[B_1\ \texttt{and}\ B_2 \mid C] \rightleftharpoons [B_1 \mid [[]\ \texttt{and}\ B_2 \mid C]] \\
[\texttt{not}\ B \mid C] \rightleftharpoons [B \mid [\texttt{not}\ [] \mid C]] \\
[A_1 + A_2 \mid C] \rightleftharpoons [A_1 \mid [[] + A_2 \mid C]] \\
[I + A \mid C] \rightleftharpoons [A \mid [I + [] \mid C]] \\
\hline
I_1 + I_2 \rightarrow (I_1 +_{Int} I_2) \\
[X \mid C], \{\!|(X, I) \rhd \sigma|\!\} \rightarrow [I \mid C], \{\!|(X, I) \rhd \sigma|\!\} \\
[\texttt{++}X \mid C], \{\!|(X, I) \rhd \sigma|\!\} \rightarrow [I +_{Int} 1 \mid C], \{\!|(X, I +_{Int} 1) \rhd \sigma|\!\} \\
\hline
I_1\texttt{<=}I_2 \rightarrow (I_1 \leq_{Int} I_2) \\
\texttt{true and}\ B \rightarrow B \\
\texttt{falseand}\ B \rightarrow \texttt{false} \\
\texttt{not true} \rightarrow \texttt{false} \\
\texttt{notfalse} \rightarrow \texttt{true} \\
\hline
\texttt{iftruethen}\ St_1\ \texttt{else}\ St_2 \rightarrow St_1 \\
\texttt{iffalsethen}\ St_1\ \texttt{else}\ St_2 \rightarrow St_2 \\
\texttt{skip}; St \rightarrow St \\
\{St\} \rightarrow St \\
[X\,\texttt{:=}I \mid C], \{\!|(X, I') \rhd \sigma|\!\} \rightarrow [\texttt{skip} \mid C], \{\!|(X, I) \rhd \sigma|\!\} \\
[\texttt{while}\ B\ St \mid C] \rightarrow [\texttt{if}\ B\ \texttt{then}\ (St; \texttt{while}\ B\ St)\ \texttt{else}\ \texttt{skip} \mid C] \\
[\texttt{halt}\ I \mid C], \sigma \rightarrow I \\
\hline
\texttt{skip}.I, \sigma \rightarrow I \\
\hline
\end{array}
$$

Figure 3.14: The *Cham* language definition

molecules and store molecules. Syntactic molecules are either language constructs or evaluation contexts and we will use "[_ | _]" as a molecule constructor for stacking molecules. We let C range over syntactic molecules representing stacked contexts. Store molecules are pairs (x, i), where x is a variable and i is an integer. The store is a solution containing store molecules. Then the definition of "refocusing" functions is translated into heating/cooling rules, bringing the redex to the top of the syntactic molecule. This allows for the reduction rules to only operate at the top, in a conceptually identical way as for continuation based definitions in Figure 3.12, both of them basically following the $\mathbb{K}$ technique.

One can notice a strong relation between our *Cham* and *CxtRed* definitions, in the sense that a step performed using reduction under evaluation contexts is equivalent to a suite of heating steps followed by one transition step and then by as many cooling steps as possible. That is, given programs P, P' and states σ, σ':

$$
\mathit{CxtRed} \vdash \langle P, \sigma \rangle \rightarrow \langle P', \sigma' \rangle \iff \mathit{Cham} \vdash P, \{\!|\sigma|\!\} \rightharpoonup^*; \rightarrow^1; \leftharpoondown^* P', \{\!|\sigma'|\!\}
$$

Note that we could not use the existing airlock mechanism to stack evaluation contexts since that could lead to unsound computations. Indeed, say one would use constructs $\{\!|\_ \rhd \_|\!\}$ to stack contexts, replacing the $[\_ \mid \_]$ construct. Then by applying heating on $\texttt{skip}; 3/4/5$, one can obtain the following sequence (of structurally equivalent molecules):

$$\begin{aligned}&\texttt{skip}; 5/(2/x) \rightharpoonup \{\!|5/(2/x) \rhd \{\!|\texttt{skip}; []|\!\}|\!\} \rightharpoonup \{\!|2/x \rhd \{\!|5/[] \rhd \{\!|\texttt{skip}; []|\!\}|\!\}|\!\}\\ &\rightharpoonup \{\!|x \rhd \{\!|2/[] \rhd \{\!|5/[] \rhd \{\!|\texttt{skip}; []|\!\}|\!\}|\!\}|\!\}\end{aligned}$$

Now, by applying the cooling, then heating rules for airlock, one obtains the following sequence (of, again, equivalent molecules):

$$\begin{aligned}&\{\!|x \rhd \{\!|2/[] \rhd \{\!|5/[] \rhd \{\!|\texttt{skip}; []|\!\}|\!\}|\!\}|\!\} \rightharpoondown \{\!|x \rhd \{\!|2/[] \rhd \{\!|5/[], \texttt{skip}; []|\!\}|\!\}|\!\}\\ &\rightharpoondown \{\!|x \rhd \{\!|2/[], 5/[], \texttt{skip}; []|\!\}|\!\} \rightharpoonup \{\!|x \rhd \{\!|5/[] \rhd \{\!|2/[], \texttt{skip}; []|\!\}|\!\}|\!\}\\ &\rightharpoonup \{\!|x \rhd \{\!|5/[] \rhd \{\!|2/[] \rhd \{\!|\texttt{skip}; []|\!\}|\!\}|\!\}|\!\}\end{aligned}$$

Finally, by applying cooling rules for contexts, we obtain the sequence:

$$\begin{aligned}&\{\!|x \rhd \{\!|5/[] \rhd \{\!|2/[] \rhd \{\!|\texttt{skip}; []|\!\}|\!\}|\!\}|\!\} \rightharpoondown \{\!|5/x \rhd \{\!|2/[] \rhd \{\!|\texttt{skip}; []|\!\}|\!\}|\!\}\\ &\rightharpoondown \{\!|2/(5/x) \rhd \{\!|\texttt{skip}; []|\!\}|\!\} \rightharpoondown \texttt{skip}; 2/(5/x)\end{aligned}$$

However, $\texttt{skip}; 5/(2/x)$ and $\texttt{skip}; 2/(5/x)$ are obviously *not* structurally equivalent.

The above language definition does not exhibit the strengths of the Cham, since Cham was designed to handle easily concurrent constructs, which are missing from our language. However, making the above language concurrent in Cham comes at no additional effort. One can execute multiple programs at the same time, sharing the store, simply by putting them together, and together with the store at the top-level solution and replacing the rule for the end of the program by $\texttt{skip}.I \to I$, to allow all programs to finish their evaluation and keep the results.

When Cham definitions follow the style in Figure 3.14, i.e., taking a context-reduction-like approach, one could use as evaluation strategies *heating only on redexes* and *cooling only on values*, which would lead to a deterministic abstract-machine. Moreover, one can notice that airlock rules were introduced to select elements from a set without specifying the rest of the set, abstracted by a molecule. Efficient implementations should probably do exactly the opposite, that is, matching in the sets. To do that in our rewrite framework, one would orient the airlock rules in the sense of inserting back the "airlocked" molecules into their original solution and to apply them on the terms of the existing rules, to make the definition executable. The only rules changing in the definition above are those involving the store; for example, the assignment rule is transformed into:

$$[X \texttt{:=} I \mid C], \{\!|(X, I'), \sigma|\!\} \to [\texttt{skip} \mid C], \{\!|(X, I), \sigma|\!\}$$

One should notice that the specification obtained by these transformations is equivalent to the initial one, since it does not change the equivalence classes and the transitions. The main advantage of the newly obtained specification is that it is also executable in a deterministic fashion, that is, there is no need to search for a final state anymore.

Strengths. Being a special case of rewriting logic, it inherits many of the benefits of rewriting logic, being specially well-suited for describing truly concurrent computations and concurrent calculi.

Weaknesses. Heating/cooling rules are hard to implement efficiently in general —an implementation allowing them to be bidirectional in an uncontrolled manner would have to *search* for final states, possibly leading to a combinatorial explosion. Rewriting strategies such as those in [17, 177, 49] can be of help for solving particular instances of this problem. Although this solution-molecule paradigm seems to work pretty well for languages in which the structure of the state is simple enough, it is not clear how one could represent the state for complex languages, with threads, locks, environments, an so on. Finally, Chams provide no mechanism to freeze the current molecular structure as a ”value”, and then to store or retrieve it, as we would need in order to define language features like call/cc. Even though it was easy to define halt because we simply discarded the entire solution, it would seem hard or impossible to define more complex control-intensive language features in Cham.

3.9 Experiments

RLS definitions, being executable, actually *are* also interpreters for the programming languages they define. One can take an RLS executable definition *as is* and execute it on top of a rewrite engine.

However, one should not wrongly conclude from this that in order to make any use of RLS definitions of programming languages, in particular of those following the various definitional styles proposed in this chapter, one must have an advanced rewrite engine. In fact, one can implement interpreters for languages given an RLS definition using one's programming language of choice. Although the proposed RLS definitions follow the same style and intuitions, and have the same strengths and limitations as their original formulation in their corresponding definitional styles, we believe that automating the process of generating interpreters from the rewriting logic language definitions following a specific operational semantics style should be easier than doing it directly from the original definition, since the rewriting logic definition *is already executable*. Furthermore, since most of the definitional styles presented in this chapter use a restricted from of rewriting, one can hope for automatic translations of those definitions into interpreters in programming languages offering a limited support for matching and rewriting. To test this claim, we have manually but mechanically translated the RLS definitions for all styles (except for MSOS and the Cham) in Haskell, Ocaml and Prolog. Section 3.10 discusses our translation procedures into these programming languages.

We compare the running times and memory requirements of the interpreters derived mechanically using the above-mentioned procedures, with those of the "free" interpreters given by executing the definition "as-is" on two rewrite engines (marked with $\star$ in the tables), namely ASF+SDF `1.5` (a compiler) and `Maude 2.2` (a fast interpreter with good tool support), as well as with those obtained executing off-the-shelf interpreter implementations in Scheme, used in teaching programming languages (marked with $\sharp$ in the tables). For Scheme we have used PLT-Scheme as an interpreter and language interpreter implementations from [60], chapters 3.9 (evaluation semantics) and 7.3 (continuation based semantics), and a PLT-Redex definition given as example in the installation package (for context reduction). Big-step interpreters are also compared against bc, a C-written interpreter for a subset of C working only with integers (bc comes as part of UNIX; type "man bc" for documentation), and two interpreters implemented using monads in Haskell and Ocaml (we mark these interpreters with $\flat$ in Figure 3.16). Since RLS representations of MSOS and Cham definitions rely intensively on matching modulo associativity and commutativity, which is only supported by Maude, we have only performed some experiments on their RLS definitions in Maude. For Cham we preferred to give the times obtained by using the novel transformations and strategies presented in Section 3.8 for making the specification "more executable". Using the specification *as is*, Cham is extremely ineffective when executed: it takes about 1205MB of memory and 188 seconds to search for the solution of running the Collatz program (explained below) up to 3.

One may naturally ask: "What is the point of all these experiments? They show little or nothing to support the RLS resulting definitions compared to their original definitions, and only show what programs (interpreters) in what programming languages are more efficient than others." Our goal here is to convey the reader our strong belief, supported by empirical evaluation, that the working language designer may be better off in practice *formally defining* a desired language, using some preferred definitional style, than *implementing an interpreter in an ad-hoc way* for that language, even in a preferred programming language. Unfortunately, the latter approach is also how programming language concepts are being taught in many places. Formal definitions tend to be significantly more compact, easier to read and more modular than ad-hoc language implementations, so they are easier to change and experiment with. Additionally, they can serve as a mathematical object capturing the essence of the desired language. One can then use this mathematical object for many other purposes in addition to executing programs, including formal analyses such as theorem proving and model-checking, static analysis, partial evaluation, compiler generation, and so on. Of course, this belief transcends the boundaries

```
x0 := 0;
while (++x0<=2){
  x1:=0;
  while (++x1<=2){
      ...
          x18:=0;
          while (++x18<=2){
            skip;
          }
      ...
  }
}.0
```

(a)

```
nr:=300;
while (not (nr<=2)){
  n:=nr;
  nr:=nr − 1;
  while (not (n==1)){
    steps:=steps + 1;
    r:=n;
    q:=0;
    while (not (r<=1)){
      r:=r − 2;
      q:=q + 1
    };
    if (r==0)
      then n:=q
      else n:=3 ∗ n + 1
    }
  }
}.steps
```

(b)

Figure 3.15: Programs used in evaluation: (a) A tower of loops, each performing two iterations; (b) Program testing Collatz's conjecture up to 300.

of rewriting logic; what RLS gives us here is a unified framework, with a uniform notation supported by a rigorous computational logic, in which one can formally define programming languages using any of the desired styles. None of the translations from RLS definitions into programming languages has been implemented, because that is not the focus of this dissertation. Nevertheless, we strongly believe that they can be implemented with relatively little effort.

One of the programs chosen to test various implementations consists of n nested loops, each of 2 iterations, parameterized by n. The other program tests the Collatz's conjecture up to 300. Collatz's conjecture states that starting from any positive number n and performing the following operations:

- if n is even then divide it by 2;
- if n is odd then multiply it by 3 and add 1;

after a finite number of steps, the value of n will become 1. To make the program more computation-intensive (and also to maximize the number of language constructs used), we here use repeated subtraction to compute division. We also count in `steps` the cumulative number of operations performed until 1 is reached for all numbers tested and return it as the result of the program. The source code for the programs used is presented in Figure 3.15.

Figures 3.16, 3.17, 3.18, and 3.19, give for each definitional style the running time of the various interpreters. For the largest number n (18) of nested loops, peak memory usage was also recorded. Times are expressed in seconds. A limit

	N nested loops(1..2)				Collatz' conjecture
N	15	16	18	Memory for 18	up to 300
⋆Asf+Sdf	1.7	2.9	11.6	13mb	265.1
♭BC	0.3	0.6	2.3	<1mb	13.8
Haskell	0.3	0.7	2.8	4mb	32.1
♭Haskell (monads)	0.6	1.4	4.4	3mb	58.7
⋆Maude	3.8	7.7	31.5	6mb	184.5
Ocaml	0.5	1.1	5.0	1mb	10.2
♭Ocaml (monads)	0.5	0.9	3.8	2mb	21.5
Prolog	1.6	1.9	7.6	316mb	-
♯Scheme [60]	3.8	7.4	30.2	13mb	122.3

Figure 3.16: Execution times for Big Step definitions

	N nested loops(1..2)				Collatz' conjecture
N	15	16	18	Memory for 18	up to 300
⋆Asf+Sdf	11.9	25.7	115.0	9mb	769.6
Haskell	3.2	7.0	31.64	3mb	167.4
⋆Maude	63.4	131.2	597.4	6mb	>1000
Ocaml	1.0	2.2	9.9	1mb	21.0
Prolog	7.0	14.5	-	>700mb	-

Figure 3.17: Execution times for Small Step definitions

of 700mb was set on memory usage, to avoid swapping; the symbol "-" found in a table cell signifies that the memory limit was reached. For Haskell we have used the `ghc 6.4.2` compiler. For Ocaml we have used the `ocamlcopt 3.09.3` compiler. For Prolog we have compiled the programs using the `gprolog 1.3.0` compiler. For Scheme we have used the PLT-Scheme (`mzscheme 3.7.1`) interpreter. Tests were performed on an Intel Pentium 4@2GHz with 1GB RAM, running Linux.

To have an overview of execution times obtained by using the RLS definition *as is* for all the styles presented, Figure 3.20 shows, side by side, their execution times in Maude.

	N nested loops(1..2)					Collatz' conjecture
N	9	15	16	18	Memory for 18	up to 300
⋆Asf+Sdf	0.6	88.7	214.4	1008.6	10mb	891.3
Haskell	0.1	5.8	12.0	53.9	3mb	157.2
⋆Maude	0.8	76.2	162.8	713.2	6mb	1931.6
Ocaml	0.0	1.8	3.8	16.7	1mb	11.0
Prolog	0.1	9.4	-	-	>700mb	-
♯PLT-Redex	198.2	-	-	-	>700mb	-

Figure 3.18: Execution times for Context Reduction definitions

	N nested loops(1..2)				Collatz' conjecture
N	15	16	18	Memory for 18	up to 300
⋆Asf+Sdf	2.5	4.7	18.3	13mb	344.7
Haskell	0.6	1.1	4.4	4mb	41.1
⋆Maude	8.4	15.6	63.2	7mb	483.9
Ocaml	0.5	1.1	5.0	1mb	10.9
Prolog	3.0	6.2	24.0	≈500mb	-
♯Scheme [60]	5.9	11.3	45.2	10mb	323.6

Figure 3.19: Execution times for Continuation based definitions

	N nested loops(1..2)				Collatz' conjecture
N	15	16	18	Memory for 18	up to 300
Big-Step	3.8	7.7	31.5	6mb	184.5
Small-Step	63.4	131.2	597.4	6mb	1249.1
Context-Reduction	76.2	162.8	713.2	6mb	1931.6
Continuation-Based	8.4	15.6	63.2	7mb	483.9
MSOS	61.9	127.4	566.3	6mb	1421.5
Cham	15.7	31.5	129.2	6mb	618.0

Figure 3.20: Execution times for RLS definitions interpreted in Maude

Prolog yields pretty fast interpreters. However, for backtracking reasons, it needs to maintain the stack of all predicates tried on the current path, thus the amount of memory grows with the number of computational steps. The style promoted in [60] seems to also take into account efficiency. Its only drawback is the fact that it looks more like an implementation, the representational distance to the big-step definition being much bigger than in interpreters based on RLS. The PLT-Redex implementation of context reduction seems to serve more a didactic purpose. It compensates for lack of speed by providing a nice interface and the possibility to visually trace a run. The rewriting logic implementations seem to be quite efficient in terms of speed and memory usage, while keeping a minimal representational distance to the operational semantics definitions. In particular, RLS definitions interpreted in Maude are comparable in terms of efficiency with the interpreters in Scheme, while having the advantage of being formal definitions. The main reason for Maude and Scheme being slower than the others, is because they are both interpreters while the others are compilers. It is well known that compilers usually generate executables one order of magnitude faster than their interpreted versions. Also, it is good to notice that the interpreter obtained by mechanically compiling the RLS definition in Ocaml can reach the speed of the hand-optimized, C-written `bc` interpreter.

3.10 Obtaining Interpreters from RLS Definitions

Since the definitions presented above are deterministic and use a restricted form of rewriting (with the exception of *MSOS* and *Cham*), we believe it is straightforward to generate interpreters from them in languages having built-in support for pattern matching and abstract data types. The main principle we use is to translate rewriting rules into evaluation functions. Since the store was defined separately and relies on matching modulo associativity and commutativity, we abstract it away, assuming each such language comes with a pre-defined store.

In the following we will show, with the definitions of assignment from big-step and continuation semantics how their translation appears as part of the chosen implementation languages. Since functional languages have a particular way of declaring abstract data types, you will notice that the syntax of the program looks different in different languages. However, assuming the existence of an external parser, we could ask from that parser to give as output terms of the abstract data type in the corresponding language.

3.10.1 Big-Step Based Definitions

The rewriting rule for assignment in big-step is:

$$\langle X\texttt{:=}A, S\rangle \to \langle S'[X \leftarrow I]\rangle \quad \textbf{if} \quad \langle A, S\rangle \to \langle I, S'\rangle$$

Asf+Sdf Since Asf+Sdf is a rewriting engine, translating RLS specifications to Asf+Sdf interpreters is mostly a matter of using a different notation. In fact Asf+Sdf adopts a notation with setting the premises above the line, close to the original semantics.

```
[] <I,S1> := <A,S>
   =========================
   <X := A,S> = bind(S1,X,I)
```

Haskell We use `Scgf(st,s)` and `Acfg(a,s)`, etc., to encode configurations $\langle st, s\rangle$ and $\langle a, s\rangle$, respectively. We define an evaluation function for each type of configuration, for example `eStmt` is the function evaluating `Scfg` configurations and `eAExp` is evaluating `Acfg` configurations. The matching of the evaluation of premises is performed by using the let construct.

```
eStmt (Scfg (Assign x a) s) =
       let (Acfg (Int i) s1) = eAExp (Acfg a s)
           in (bind s1 x i)
```

Maude Since Maude is the standard execution engine for rewriting logic specifications, the rules here *are* the ones in the specification.

```
rl < X := A,S > => S1[X <- I] if < A,S > => {I,S1} .
```

Ocaml Since Ocaml supports polymorphic functions, we only need to define one evaluation function for all constructs. Then matching is used to obtain the starting term and `match ... with ...` is used for evaluating the premises.

```
let rec eval = function
 ...
  | Scfg(Assign(x,a),s) ->
      (match eval (Acfg(a,s)) with Acfg(Int(i),s1) ->
                                           (bind s1 x i))
```

Prolog In Prolog we define a relation for each type of configuration and use unification for matching only purposes. Note that while in Ocaml, constructors of abstract data types start with capital letter, in Prolog this would correspond to variables, so we need to use `scfg`, `acfg`, etc., to encode configurations.

```
eStmt(scfg(X = A,S),S2) :- eAExp(acfg(AE,S),acfg(I,S1)),
                           bind(S1,X,I,S2).
```

3.10.2 Continuation Based Definitions

Recall that the RLS semantics for assignment consists of an equation and a rule:

$$stmt(X := A) = aexp(A) \curvearrowright write(X)$$
$$k(I \curvearrowright write(X) \curvearrowright K)\ \ store(Store) \to k(K)\ \ store(Store[X \leftarrow I])$$

Asf+Sdf Again, the translation to Asf+Sdf implies minimal or no modifications. Note that Asf+Sdf makes no distinction between equations and rules, all of them being written as equations.

```
[] stmt(X := A) = aexp(A) -> write(X)
[] k(int(I) -> write(X) -> K) store(Store)
 = k(K) store(bind(Store,X,I))
```

Haskell The continuation concatenation is replaced by list concatenation. The evaluation rules are transformed into a recursive evaluation function acting at the top of the state.

```
stmt (Assign x a)) = (aexp a) ++ [Kwrite x]
result (Kval (Vint i):Kwrite x:k) s = result k (bind s x i)
```

Maude Representation in Maude is the exact rewriting logic definition.

```
eq stmt(X := A) = aexp(A) -> write(X) .
rl k(int(I) -> write(X) -> K) store(Store)
=> k(K) store(Store[X <- I]) .
```

Ocaml A similar approach as that for Haskell.

```
let rec stmt = function
    ...
    | Assign(x, a) -> (aexp a) @ [Kwrite x]
let rec result s = function
    ...
    | (Kval (Vint i)::Kwrite x::k) -> result (bind s x i) k
```

Prolog Same approach as for the functional languages above, but we now define (functional) evaluation relations for functions decomposing the program and a one-step rewrite relation for the top-level evaluation process.

```
stmt(X = A,K) :- aexp(A,KA), append(KA,[write(X)],K) .
step(conf(store(S),v([I]),k([write(X)|K])),
     conf(store(S1),v(Vl),k(K)))
     :- bind(S,X,I,S1).
```

3.11 Discussion

In this chapter we have tried to show how RLS can be used as a logical framework for operational semantics definitions of programming languages. In particular, by showing in detail how it can faithfully capture big-step and small-step SOS, MSOS, context reduction, continuation-based semantics, and the Cham, we hope to have illustrated what might be called its *ecumenical* character; that is, its flexible support for a wide range of definitional styles, without forcing or pre-imposing any given style. In fact, we think that this flexibility makes RLS useful as a way of exploring *new* definitional styles.

However, existing language definitional styles bring their own limitation inside rewriting logic, and thus fail to fully make use of the flexibility of the logic itself. Moreover, rewriting logic is a meta-logic, and thus does not give any recipe for defining programming languages. Therefore, a natural question arises:

> *What would be the ideal language definitional framework which can be based on rewriting logic?*

This remainder of this dissertation advances the 𝕂 framework—a rewriting-based language definitional framework, which attempts to combine the strengths of all the definitional styles presented in this chapter with that of unrestricted use of rewriting—as a serious candidate for answering the question above.

Acknowledgments. We thank our fellow researchers in the RLS project, including Wolfgang Ahrendt, Musab Al-Turki, Marcelo d'Amorim, Eyvind W. Axelsen, Christiano Braga, Illiano Cervesato, Fabricio Chalub, Feng Chen, Manuel Clavel, Azadeh Farzan, Alejandra Garrido, Mark Hills, Einar Broch Johnsen, Ralph Johnson, Michael Katelman, Laurentiu Leustean, Narciso Martí-Oliet, Olaf Owe, Stefan Reich, Andreas Roth, Juan Santa-Cruz, Ralf Sasse, Koushik Sen, Mark-Oliver Stehr, Carolyn Talcott, Prasanna Thati, Ram Prasad Venkatesan, and Alberto Verdejo, for their many contributions, which have both advanced the project and stimulated our ideas. We also thank the students at UIUC who attended courses on programming language design, semantics and formal methods, who provided important feedback and suggestions. Last but not least, we thank Mark Hills for fruitful discussions and his help in adjusting the PLT-Redex implementation to suit our needs.

Chapter 4

An Overview of the 𝕂 Semantic Framework

The 𝕂 framework, started in 2003 by Roșu [142] for teaching a programming languages class, and continuously developed since then [80, 146, 144], is a specialized framework for defining and analyzing programming languages based on rewriting. 𝕂 applies the lessons learned from representing and evaluating existing language definitional frameworks in rewriting logic (Chapter 3) with the full power of rewriting, to allow developing powerful, modular, versatile, and clear definitions of programming languages. Moreover, through its representation in rewriting logic, it gains executability and access to the generic, but powerful, execution and analysis tools available for rewrite theories (debugging and tracing, BFS exploration, LTL model checker, inductive theorem prover).

The introduction and development of the 𝕂 framework was largely motivated by the observation that after more than 40 years of systematic research in programming language semantics, the quest for an ideal language definitional framework remains largely open to the working programming language designer, but also to the entire research community. In our view, which synthesizes views taken by existing approaches, such an ideal framework should satisfy at least the following requirements.

Versatility. It should give a unified approach to define not only languages but also language-related abstractions, such as type checkers, type inferencers, abstract interpreters, safety policy or domain-specific checkers. The current state-of-the art is that language designers use different approaches or styles to define different aspects of a language, sometimes even to define different components of the same aspect.

Expression power. It should be able to define arbitrarily complex language features, including, obviously, all those found in existing languages, while being able to also capture their intended computational granularity. For example, features like call-with-current-continuation and true concurrency are hard or impossible to define in many existing frameworks.

Modularity. It should be modular, i.e., adding new language features does not require modifying existing definitions of unrelated features. Modularity is crucial for scalability and reuse.

Non-determinism&concurrency. It should provide good support for non-determinism and concurrency, at any desired granularity.

Generality. Is should be generic, that is, not tied to any particular programming language or paradigm.

Executability. Is should be executable, so one can "test" language or formal analyzer definitions, as if one already had an interpreter or a compiler for one's language. Efficient executability of language definitions may even eliminate the need for additional interpreters.

Analyzability. It should provide state-exploration capabilities, including exhaustive behavior analysis (e.g., model-checking), when one's language is non-deterministic or/and concurrent.

Provability. It should provide corresponding initial-model or axiomatic semantics (to allow inductive or Hoare-style proofs), so that one can formally reason about programs.

The list above contains a *minimal* set of desirable features that an ideal language definitional framework should have. There are additional desirable requirements of an ideal language definitional framework that are more subjective and thus more difficult to quantify. For example, it should be simple and easy to understand, teach and use by mainstream enthusiastic language designers, not only by language experts—in particular, an ideal framework should not require its users to have advanced understanding of category theory, logics, or type theory in order to use it. Also, it should have good data representation capabilities and should allow proofs of theorems about programming languages that are easy to comprehend. Additionally, a framework providing support for parsing programs directly in the desired language syntax may be desirable, so that an external parser is not needed.

The requirements above are nevertheless ambitious. Proponents of existing language definitional frameworks may argue that their favorite framework has these properties; however, a careful analysis of existing language definitional frameworks reveals that they actually fail to satisfy some of these ideal features (see Chapter 8 for more details on that). Others may argue that their favorite framework has some of the properties above, the "important ones", declaring the other properties either "not interesting" or "something else". For example, one may say that what is important in one's framework is to achieve a dynamic semantics of a language, but that defining type systems, proving properties about programs, model checking, etc., are "something else" and therefore are allowed to require a different "encoding" of the language. Our position is that an ideal language definitional framework should not compromise any of the requirements above.

Whether $\mathbb{K}$ satisfies all the requirements above or not is, and probably will always be, open. What we can mention with regards to this aspect, though, is that

$\mathbb{K}$ was motivated and stimulated by the observation that the existing language definitional frameworks fail to fully satisfy these minimal requirements; consequently, $\mathbb{K}$'s design and development were conducted aiming *explicitly* to fulfill all requirements discussed above, promoting none of them at the expense of others.

The $\mathbb{K}$ framework consists of two components: the $\mathbb{K}$ *technique*, discussed in this chapter, and $\mathbb{K}$ *rewriting*, a concurrent rewriting semantics for $\mathbb{K}$ discussed in Chapter 6. Like a term rewrite system, a $\mathbb{K}$-system consists of a signature for building terms and of a set of rules for iteratively rewriting terms. Emerging from rewriting logic [103], $\mathbb{K}$ rules can be applied concurrently and unrestricted by context. Moreover, $\mathbb{K}$ rules inherit features from graph rewrite rules, containing information about what part of the matched term is left unchanged by the rule (called the *read-only* part), similar to interfaces in graph rewriting [44]. Besides offering a more compact notation for the rewrite rules, identifying the read-only part can also potentially enhance the concurrency: two overlapping rules can be applied in parallel if they only overlap on their read-only part, similar to the concept of parallel independence in graph rewriting [44]. However, if one is not interested in the degree of concurrency allowed by a $\mathbb{K}$ definition, one can safely ignore the information regarding the read-only part, and view $\mathbb{K}$ rules as an alternative notation for rewrite rules. Our prototype implementation, described in Chapter 7, takes this approach, which allows $\mathbb{K}$ definitions to benefit from the existing infrastructure and analysis tools provided for rewriting logic theories by the Maude [34] rewrite engine.

The remainder of this chapter is structured as follows. Section 4.1 discusses the $\mathbb{K}$ framework intuitively, by means of defining a simple imperative language and an extension of it; this section should give the reader quite a clear feel for what $\mathbb{K}$ is about and how it operates. Section 4.2 presents the $\mathbb{K}$ technique, explaining essentially how rewriting can be used to define programming language semantics by means of nested-cell configurations, computations and rewrite rules. To illustrate the power of the $\mathbb{K}$ framework, Section 4.3 presents an increasingly complex, modular, and concurrent definition of AGENT, a pedagogical multi-agent and multi-threaded functional programming language with imperative and control features. Finally, Section 4.4 provides connections with the remainder of the dissertation.

4.1 $\mathbb{K}$ Overview by Example

The role of this section is threefold: (1) it gives the reader a better understanding of the $\mathbb{K}$ framework before we proceed to define it rigorously in the remainder sections; (2) it shows how $\mathbb{K}$ avoids some of the limitations of other semantic approaches; and (3) it shows that $\mathbb{K}$ is actually easy to use. We use as concrete examples the IMP language, a very simple imperative language, and IMP++, an extension of IMP with: increment to exhibit side-effects for expressions; input and output; halt, to show how $\mathbb{K}$ deals with abrupt termination; and spawning of threads, to show how concurrency is handled. We define both an executable semantics and a type system for these languages. The type system is included mainly for demonstration purposes, to show that one can use the same framework, $\mathbb{K}$, to define both dynamic and static semantics of languages.

Programming languages, calculi, as well as type systems or formal analyzers can be defined in $\mathbb{K}$ by making use of special, potentially nested *($\mathbb{K}$) cell* structures, and *($\mathbb{K}$) (rewrite) rules*. There are two types of $\mathbb{K}$ rules: *computational rules*, which count as computational steps, and *structural rules*, which do not count as computational steps. The role of the structural rules is to rearrange the term so that the computational rules can apply. $\mathbb{K}$ rules are *unconditional* (they may have ordinary side conditions, though, as rule schemata), and they are *context-insensitive*, so $\mathbb{K}$ rules apply concurrently as soon as they match, without any contextual delay or restrictions.

Computations One sort has a special meaning in $\mathbb{K}$, namely the sort K of *computations*. The intuition for terms of sort K is that they have computational contents, such as programs or fragments of programs have; indeed, computations extend the syntax of the original language. Computations have a list structure, capturing the intuition of computation sequentialization, with list constructor $\_ \curvearrowright \_$ (read “followed by”) and unit “$\cdot$” (the empty computation). Computations give an elegant and uniform means to define and handle evaluation contexts [180] and/or continuations [59]. Indeed, a computation “$v \curvearrowright C$” can be thought of as “$C[v]$, that is, evaluation context C applied to v” or as “passing v to continuation C”. Computations can be handled like any other term in a rewriting environment, that is, they can be matched, moved from one place to another, modified, or even deleted. A term may contain an arbitrary number of computations, which can evolve concurrently; they can be thought of as execution threads. Rules corresponding to inherently sequential operations (such as lookup/assignment of variables in the same thread) must be designed with care, to ensure that they are applied only at the top of computations.

The distinctive feature of $\mathbb{K}$, compared to other term rewriting approaches to defining programming languages, is that $\mathbb{K}$ allows rewrite rules to apply *concurrently* even in cases when they overlap, provided that they do not change the overlapped portion of the term. This allows for truly concurrent semantics. For example, two threads can read the same location of memory concurrently, even though the corresponding rules overlap on the store location being read.

$\mathbb{K}$ achieves, in one uniform framework, the benefits of both the chemical abstract machines (or Chams) and reduction semantics with evaluation contexts, at the same time avoiding what might be called the “rigidity to chemistry” of the former and the “rigidity to syntax” of the latter. Like the other semantic approaches that can be represented in rewriting logic presented in Chapter 3, $\mathbb{K}$ can also be represented in rewriting logic. This dissertation describes two such representations: an executable one, which gives an interleaving semantics for concurrency with sharing of resources, and a completely faithful one, obtained through an intermediate embedding into graph rewriting.

The concurrent semantics for $\mathbb{K}$ rewriting and its faithful representation in rewriting logic are presented in detail in Chapter 6. The executable representation of $\mathbb{K}$ in rewriting logic, whose implementation in the Maude [34] rewrite engine is presented in Chapter 7, can be used to execute $\mathbb{K}$ definitions, thus providing "interpreters for free" directly from formal language definitions; additionally, general-purpose formal analysis techniques and tools developed for rewriting logic, such as state space exploration for safety violations or model-checking, give us corresponding techniques and tools for the defined languages, at no additional development cost. The fact that this executable representation looses part of the true concurrency of $\mathbb{K}$ does not limit its access to the execution and analysis tools for rewrite theories, as the implementation of rewriting logic in the Maude rewrite engine also considers a fully interleaved semantics when executing or exploring transitions systems for rewrite theories.

4.1.1 $\mathbb{K}$ Semantics of Imp

Figure 4.1 shows the complete $\mathbb{K}$ definition of Imp, except for the configuration; the Imp configuration is explained separately below. The left column gives the Imp syntax. The middle column contains special syntax $\mathbb{K}$ annotations, called strictness attributes, stating the evaluation strategy of some language constructs. Finally, the right column gives the semantic rules.

$\mathbb{K}$ makes intensive use of the context-free grammar (CFG) notation for syntax and for configurations, extended with specialized "algebraic" notation for lists, sets, multisets (bag) and maps. For any sort S, the sort $\mathsf{List}^{\dagger}_{\star}[S]$ (or $\mathsf{Bag}^{\dagger}_{\star}[S]$, or $\mathsf{Set}^{\dagger}_{\star}[S]$) defines the $\star$-separated lists (or bags, or sets) of elements of sort S, with identity $\dagger$. If unspecified, by default $\star$ is $\_,\_$ for lists and $\_\_$ for bags and sets, and $\dagger$ is "$\cdot$". For example, $\mathsf{List}[S]$ defines comma-separated lists of elements of type S, and could be expressed with the lower-level CFG productions $\mathsf{List}[S] ::= \cdot \mid S(,S)^{*}$. Similarly, sort $\mathsf{Map}^{\dagger}_{\star}[S_1 \mapsto S_2]$ contains a set of mappings *source* $\mapsto$ *target*, with *source* of sort S_1 and *target* of sort S_2, separated by $\star$ and with identity $\dagger$; be default, $\star$ is $\_\_$ and $\dagger$ is "$\cdot$".

Like in the Cham, program or system configurations in $\mathbb{K}$ are organized as potentially nested structures of *cells* (we call them cells instead of molecules to avoid confusion with terminology in Cham and chemistry). However, unlike the Cham which only provides multisets (or bags), $\mathbb{K}$ also provides list, set and map cells in addition to multiset cells; $\mathbb{K}$'s cells may be labeled to distinguish them from each other. We use angle brackets as cell wrappers.

The $\mathbb{K}$ configuration of IMP can be defined as:

$$\mathit{Configuration}_{\mathrm{IMP}} \quad \equiv \quad \langle \langle K\rangle_{\mathsf{k}}\ \langle \mathsf{Map}[\mathit{Id} \mapsto \mathit{Int}]\rangle_{\mathsf{state}}\rangle_{\top}$$

In words, Imp configurations consist of a top cell $\langle\rangle_\top$ containing two other cells inside: a cell $\langle\rangle_{\mathsf{k}}$ which holds a term of sort K (terms of sort K are called computations and extend the original language syntax as explained in the next paragraph) and a cell $\langle\rangle_{\mathsf{state}}$ which holds a map from variables to integers. For example, "$\langle\langle x = 1\ ;\ y = x + 1\ ;\rangle_{\mathsf{k}}\ \langle\cdot\rangle_{\mathsf{state}}\rangle_\top$" is a configuration holding program "$x = 1\ ;\ y = x + 1\ ;$" and empty state, while the configuration "$\langle\langle x = 1\ ;\ y = x + 1\ ;\rangle_{\mathsf{k}}\ \langle x \mapsto 0, y \mapsto 1\rangle_{\mathsf{state}}\rangle_\top$" holds the same program and a state mapping x to 0 and y to 1.

$\mathbb{K}$ provides special notational support for *computational structures*, or simply *computations*. Computations have the sort K, which is therefore builtin in the $\mathbb{K}$ framework; the intuition for terms of sort K is that they have computational contents, such as, for example, a program or a fragment of program has. Computations extend the original language/calculus/system syntax with special "$\curvearrowright$"-separated lists "$T_1 \curvearrowright T_2 \curvearrowright \cdots \curvearrowright T_n$" comprising *(computational) tasks*, thought of as having to be "processed" sequentially ("$\curvearrowright$" reads "followed by"). The identity of the "$\curvearrowright$" associative operator is "$\cdot$". Like in reduction semantics with evaluation contexts, $\mathbb{K}$ allows one to define evaluation contexts over the language syntax. However, unlike in reduction semantics, parsing does not play any crucial role in $\mathbb{K}$, because $\mathbb{K}$ replaces the hard-to-implement split/plug operations of evaluation contexts by plain, context-insensitive rewriting. Therefore, instead of defining evaluation contexts using context-free grammars and relying on splitting syntactic terms (via parsing) into evaluation contexts and redexes, in $\mathbb{K}$ we define evaluation contexts using special rewrite rules. For example, the evaluation contexts of sum, comparison and conditional in Imp can be defined as follows, by means of *structural rules* (the sum "+" is non-deterministic, i.e., the evaluation procedure for its arguments is not fixed and the comparison "<=" is sequential):

$$\begin{array}{rcl}
a_1 + a_2 & \rightleftharpoons & a_1 \curvearrowright \square + a_2 \\
a_1 + a_2 & \rightleftharpoons & a_2 \curvearrowright a_1 + \square \\
a_1 \texttt{ <= } a_2 & \rightleftharpoons & a_1 \curvearrowright \square \texttt{ <= } a_2 \\
i_1 \texttt{ <= } a_2 & \rightleftharpoons & a_2 \curvearrowright i_1 \texttt{ <= } \square \\
\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2 & \rightleftharpoons & b \curvearrowright \texttt{if}\ \square\ \texttt{then}\ s_1\ \texttt{else}\ s_2
\end{array}$$

The symbol $\rightleftharpoons$ stands for two structural rules, one left-to-right and another right-to-left.

The right-hand sides of the structural rules above contain, besides the task sequentialization operator $\curvearrowright$, *freezer* operators containing $\square$ in their names, such as "$\square$ + _", "_ + $\square$", etc. The first rule above says that in any expression of the form "$a_1 + a_2$", a_1 can be scheduled for processing while a_2 is being held for future processing. Since the rules above are bi-directional, they can be used at will to structurally re-arrange the computations for processing. Thus, when iteratively applied left-to-right they fulfill the role of *splitting* syntax into an evaluation context (the tail of the resulting sequence of computational tasks) and a redex (the head of the resulting sequence), and when applied right-to-

Original language syntax	Strictness	Semantics
$AExp ::= Int \mid Id$		$\langle \underset{i}{\underline{x}}\ \cdots\rangle_{\mathsf{k}}\ \langle\cdots\ x \mapsto i\ \cdots\rangle_{\mathsf{state}}$
$\mid\ AExp\ \texttt{+}\ AExp$	$[strict]$	$i_1\ \texttt{+}\ i_2 \to i_1 +_{Int} i_2$
$\mid\ AExp\ \texttt{/}\ AExp$	$[strict]$	$i_1\ \texttt{/}\ i_2 \to i_1 /_{Int} i_2$ when $i_1 \neq 0$
$BExp ::= AExp\ \texttt{<=}\ AExp$	$[seqstrict]$	$i_1\ \texttt{<=}\ i_2 \to i_1 \leq_{Int} i_2$
$\mid\ \texttt{not}\ BExp$	$[strict]$	$\texttt{not}\ t \to \neg_{Bool} t$
$\mid\ BExp\ \texttt{and}\ BExp$	$[strict(1)]$	$true\ \texttt{and}\ b \to b$
		$false\ \texttt{and}\ b \to \texttt{false}$
$Stmt ::= \texttt{skip;}$		$\texttt{skip;} \to \cdot$
$\mid\ Id\ \texttt{=}\ AExp\ \texttt{;}$	$[strict(2)]$	$\langle \underset{\cdot}{\underline{x\ \texttt{=}\ i\ \texttt{;}}}\ \cdots\rangle_{\mathsf{k}}\ \langle\cdots\ x \mapsto \underset{i}{\underline{\_}}\ \cdots\rangle_{\mathsf{state}}$
$\mid\ Stmt\ \ Stmt$		$s_1\ s_2 \rightharpoonup s_1 \curvearrowright s_2$
$\mid\ \texttt{if}\ BExp\ \texttt{then}\ Stmt$	$[strict(1)]$	$\texttt{if}\ true\ \texttt{then}\ s\ \texttt{else}\ \_ \to s$
$\texttt{else}\ Stmt$		$\texttt{if}\ false\ \texttt{then}\ \_\ \texttt{else}\ s \to s$
$\mid\ \texttt{while}\ BExp\ \texttt{do}\ Stmt$		$\langle \underset{\texttt{if}\ b\ \texttt{then}(s\ \texttt{while}\ b\ \texttt{do}\ s)\ \texttt{else skip;}}{\underline{\texttt{while}\ b\ \texttt{do}\ s}}\ \cdots\rangle_{\mathsf{k}}$
$Pgm ::= \texttt{var}\ \mathsf{List}[Id]\ \texttt{;}\ Stmt$		$\langle \underset{s}{\underline{\texttt{var}\ xl\ \texttt{;}\ s}}\rangle_{\mathsf{k}}\ \langle \underset{xl \mapsto 0}{\underline{\cdot}}\rangle_{\mathsf{state}}$

Figure 4.1: $\mathbb{K}$ definition of Imp: syntax (left), annotations (middle) and semantics (right); $x \in Id$, $xs \in \mathsf{List}[Id]$, $i, i_1, i_2 \in Int$, $t \in Bool$, $b \in BExp$, $s, s_1, s_2 \in Stmt$ (b, s, s_1, s_2 can also be in K)

left they fulfill the role of *plugging* syntax into context. Such structural rules are called *heating/cooling rules* in $\mathbb{K}$, since they are reminiscent of the Cham heating/cooling rules; for example, $a_1\ \texttt{+}\ a_2$ is "heated" into $a_1 \curvearrowright \square\ \texttt{+}\ a_2$, while $a_1 \curvearrowright \square\ \texttt{+}\ a_2$ is "cooled" into $a_1\ \texttt{+}\ a_2$. A language definition can use structural rules not only for heating/cooling but also to give the semantics of some language constructs; this will be discussed later in this section.

To avoid writing obvious heating/cooling structural rules like the above, we prefer to use the *strictness attribute* syntax annotations in $\mathbb{K}$, as shown in the middle column in Figures 4.1 and 4.2: "*strict*" means non-deterministically strict in all enlisted arguments (given by their positions) or by default in all arguments if none enlisted, meaning that all specified arguments must be evaluated before evaluating the construct itself, and "*seqstrict*" is like *strict* but each argument is fully processed before moving to the next one (see the second structural rule of "<=" above).

The structural rules corresponding to strictness attributes (or the heating/cooling rules) decompose and eventually push the tasks that are ready for processing to the top (or the left) of the computation. Semantic rules then tell how to process the atomic tasks. The right column in Figure 4.1 shows the semantic $\mathbb{K}$ rules of Imp. To understand them, let us first discuss the important notion of a $\mathbb{K}$ *rule*, which is a strict generalization of the usual notion of a rewrite rule. To take full advantage of $\mathbb{K}$'s support for concurrency, $\mathbb{K}$ rules explicitly mention the parts of the term that they read, write, or do not care about. The underlined parts are those which are written by the rule; the term underneath the line is the new subterm replacing the one above the line.

All writes in a $\mathbb{K}$ rule are applied in *one parallel step*, and, with some reasonable restrictions discussed in Chapter 6 (that avoid read/write and write/write conflicts), writes in multiple $\mathbb{K}$ rule instances can also apply in parallel. The operations which are not underlined represent the read-only part of the term: they need to stay unchanged during the application of the rule. Ellipses "$\cdots$" at the beginning or end of cells are used to locate in the cell the specified content, by indicating that there could be more items in the cell before or after the specified content, respectively. While cells holding lists (including the computation cell) can have ellipses on any (or both) ends of the cell, we convene that cells holding sets, bags, or maps should always use ellipses on both ends, as the contents are to be located "in the middle" of the cell. The anonymous variables "$\_$" are used to represent parts of the term that the current rule does not need to refer to.

For example, the lookup rule (the first rule in Figure 4.1) says that once program variable x reaches the top (beginning) of the computation, it is replaced by the value to which it is mapped in the state, regardless of the remaining computation (which is indicated by the ellipses on the right side of the $\langle\rangle_{\mathsf{k}}$ cell) or the other mappings in the state (which is indicated by ellipses in both sides of the $\langle\rangle_{\mathsf{state}}$ cell). Similarly, the assignment rule says that once the assignment statement "$x = i\ ;$" reaches the top of the computation, the value of x in the store is replaced by i and the statement dissolves; "$\cdot$" is the unit (or empty) computation ("$\cdot$" tends to be used in $\mathbb{K}$ as a polymorphic unit of most if not all list, set and multiset structures). The rule for variable declarations in Figure 4.1 (last one) expects an empty state and allocates and initializes with 0 all the declared variables; the dotted or dashed lines signify that the rule is structural, which is discussed next.

$\mathbb{K}$ rules are split in two categories: *computational rules* and *structural rules*. Computational rules capture the intuition of computational steps in the execution of the defined system or language, while structural rules capture the intuition of structural rearrangement, rather than computational evolution, of the system. We use dashed or dotted lines in the structural rules to convey the idea that they are lighter-weight than the computational rules. Ordinary rewrite rules are a special case of $\mathbb{K}$ rules, when the entire term is replaced; in this case, we prefer to use the standard notation $l \rightarrow r$ as syntactic sugar for computational rules and the notation $l \rightharpoondown r$ or $l \rightharpoonup r$ as syntactic sugar for structural rules. We have seen several structural rules at the beginning of this section, namely the heating/cooling rules corresponding to the strictness attributes. Figure 4.1 shows three more: $s_1\, s_2$ is rearranged as $s_1 \curvearrowright s_2$, loops are unrolled when they reach the top of the computation (unconstrained unrolling would lead to undesirable non-termination), and declared variables are allocated in the state. There are no rigid requirements on when rules should be computational versus structural

Original language syntax		Semantics
$AExp$ ::=		
$\mid$	$\texttt{++}\,Id$	$\langle \frac{\texttt{++}\,x}{i +_{Int} 1} \cdots\rangle_{\mathsf{k}}\ \langle \cdots\ x \mapsto \frac{i}{i +_{Int} 1} \cdots\rangle_{\mathsf{state}}$
$\mid$	$\texttt{read}$	$\langle \frac{\texttt{read}}{i} \cdots\rangle_{\mathsf{k}}\ \langle \frac{i}{\cdot} \cdots\rangle_{\mathsf{in}}$
$Stmt$ ::=		
$\mid$	$\texttt{print}\,AExp\ \texttt{;}$ $[strict]$	$\langle \frac{\texttt{print}\,i\ \texttt{;}}{\cdot} \cdots\rangle_{\mathsf{k}}\ \langle \cdots\ \frac{\cdot}{i}\rangle_{\mathsf{out}}$
$\mid$	$\texttt{spawn}\,Stmt$	$\langle \frac{\texttt{spawn}\,s}{\cdot} \cdots\rangle_{\mathsf{k}}\ \frac{\cdot}{\langle s\rangle_{\mathsf{k}}}$
$\mid$	$\texttt{haltThread;}$	$\langle \frac{\texttt{haltThread;} \curvearrowright \_}{\cdot}\rangle_{\mathsf{k}}$
		$\langle\cdot\rangle_{\mathsf{k}} \rightharpoondown \cdot$

Figure 4.2: $\mathbb{K}$ definition of IMP++ (extends that of IMP in Figure 4.1, *without changing anything*)

and, in the latter case, on when one should use $l \rightharpoondown r$ or $l \rightharpoonup r$ as syntactic sugar. We (subjectively) prefer to use structural rules for desugaring (like for sequential composition), loop unrolling and declarations, and we prefer to use "$\rightharpoonup$" when syntax is split into computational tasks and "$\rightharpoondown$" when computational tasks are put back into the original syntax.

Each $\mathbb{K}$ rule can be easily "desugared" into a standard term rewrite rule by combining all its changes into one top-level change. The relationship between $\mathbb{K}$ rewriting and conventional term rewriting and rewriting logic is discussed in Chapter 6. The main point is that through this desugaring, the resulting conventional rewrite system associated to a $\mathbb{K}$-system lacks the full potential for concurrency of the original $\mathbb{K}$-system.

4.1.2 $\mathbb{K}$ Semantics of IMP++

Figure 4.2 shows how the $\mathbb{K}$ semantics of IMP can be seamlessly extended into a semantics for IMP++. To accommodate input and output, two new cells need to be added to the configuration:

$$\mathit{Configuration}_{\text{IMP++}} \quad\equiv\quad \langle\langle K\rangle_{\mathsf{k}}\ \langle \mathsf{Map}[Id \mapsto Int]\rangle_{\mathsf{state}}\ \langle \mathsf{List}[Int]\rangle_{\mathsf{out}}\ \langle \mathsf{List}[Int]\rangle_{\mathsf{in}}\rangle_{\top}$$

However, note that none of the existing IMP rules needs to change, because each of them only matches what it needs from the configuration. The increment construct "`++ _`" introduces side effects for expressions: it increments the value of the identifier and evaluates to that value. The rule for the `read` construct uses the first element from the $\langle\rangle_{\mathsf{input}}$ cell (and replaces it by the unit "$\cdot$") to replace the `read` expression. The `print` construct is strict and its rule adds the value of its argument to the end of the $\langle\rangle_{\mathsf{out}}$ buffer (matches and replaces the unit "·" at the end of the buffer). The rule for `haltThread` dissolves the current computation, and the rule for `spawn` creates a new $\langle\rangle_{\mathsf{k}}$ cell initialized with the spawned statement. The code in this new cell will be processed concurrently with the other threads. The last rule "cools" down a terminated thread by simply dissolving it; it is a structural rule because, again, we do not want it to count as a computation.

We conclude this section with a discussion on the concurrency of the $\mathbb{K}$ definition of IMP++. Since in $\mathbb{K}$ rule instances can share read-only data, various instances of the look up rule can apply concurrently, in spite of the fact that they overlap on the state. Similarly, since the rules for variable assignment and increment do not update anything else in the $\langle\rangle_{\mathsf{state}}$ cell except the mapping corresponding to the variable, multiple assignments, increments and reads of distinct variables can happen concurrently. However, if two threads want to write the same variable, or if one wants to write it while another wants to read it, then the two corresponding rules need to interleave, because the two rule instances are in a concurrency conflict. Note also that the rules for `read`/`print` match and change the beginning/end of the $\langle\rangle_{\mathsf{in}}/\langle\rangle_{\mathsf{out}}$ cell. That means, in particular, that multiple `read`/`print` statements by various threads need to be interleaved for the same reason as above; however, one `read` could be executed in parallel with one `print` command. On the other hand, the rule for `spawn` matches any empty top-level position and replaces it by the new thread, so threads can spawn threads concurrently. Similarly, multiple threads can be dissolved concurrently when they are done (last "cooling" structural rule). These desirable concurrency aspects of IMP++ are possible to define formally thanks to the specific nature of the $\mathbb{K}$ rules. If we used standard rewrite rules instead of $\mathbb{K}$ rules, then many of the concurrent steps above would need to be interleaved because rewrite rule instances which overlap cannot be applied concurrently.

4.1.3 $\mathbb{K}$ Type System for IMP/IMP++

The $\mathbb{K}$ semantics of IMP/IMP++ discussed above can be used to execute even ill-typed IMP/IMP++ programs, which may be considered undesirable by some language designers. Indeed, one may want to define a type checker for a desired typing policy, and then use it to discard as inappropriate programs that do not obey the desired typing policy. We next show how to define a type system for IMP/IMP++ using the very same $\mathbb{K}$ framework. The type system is defined like an (executable) semantics of the language, but one in the more abstract domain of types rather than in the concrete domain of values. The technique is general and has been used to define more complex type systems, such as higher-order polymorphic ones [52].

The typing policy that we want to enforce on IMP/IMP++ programs is easy: all variables in a program have by default integer type and must be declared, arithmetic/Boolean operations are applied only on expressions of corresponding

Original language syntax	Strictness	Semantics
$AExp ::= Int$		$i \rightarrow int$
$\mid\ Id$		$\langle \underset{int}{\underline{x}}\ \cdots\rangle_{\mathsf{k}}\ \langle\cdots\ x\ \cdots\rangle_{\mathsf{vars}}$
$\mid\ AExp\ \texttt{+}\ AExp$	$[strict]$	$int\ \texttt{+}\ int \rightarrow int$
$\mid\ AExp\ \texttt{/}\ AExp$	$[strict]$	$int\ \texttt{/}\ int \rightarrow int$
$\mid\ \texttt{++}\,Id$		$\langle \underset{int}{\underline{\texttt{++}\,x}}\ \cdots\rangle_{\mathsf{k}}\ \langle\cdots\ x\ \cdots\rangle_{\mathsf{vars}}$
$\mid\ \texttt{read}$		$\texttt{read} \rightarrow int$
$BExp ::= AExp\ \texttt{<=}\ AExp$	$[strict]$	$int\ \texttt{<=}\ int \rightarrow bool$
$\mid\ \texttt{not}\ BExp$	$[strict]$	$\texttt{not}\ bool \rightarrow bool$
$\mid\ BExp\ \texttt{and}\ BExp$	$[strict]$	$bool\ \texttt{and}\ bool \rightarrow bool$
$Stmt ::= \texttt{skip;}$		$\texttt{skip;} \rightarrow stmt$
$\mid\ Id\ \texttt{=}\ AExp\ \texttt{;}$	$[strict(2)]$	$\langle \underset{stmt}{\underline{x\ \texttt{=}\ int\ \texttt{;}}}\ \cdots\rangle_{\mathsf{k}}\ \langle\cdots\ x\ \cdots\rangle_{\mathsf{vars}}$
$\mid\ Stmt\ \ Stmt$	$[strict]$	$stmt\ \ stmt \rightarrow stmt$
$\mid\ \texttt{if}\ BExp\ \texttt{then}\ Stmt$ $\texttt{else}\ Stmt$	$[strict]$	$\texttt{if}\ bool\ \texttt{then}\ stmt \rightarrow stmt$ $\texttt{else}\ stmt$
$\mid\ \texttt{while}\ BExp\ \texttt{do}\ Stmt$	$[strict]$	$\texttt{while}\ bool\ \texttt{do}\ stmt \rightarrow stmt$
$\mid\ \texttt{print}\ AExp\ \texttt{;}$	$[strict]$	$\texttt{print}\ int\ \texttt{;} \rightarrow stmt$
$\mid\ \texttt{haltThread;}$		$\texttt{haltThread;} \rightarrow stmt$
$\mid\ \texttt{spawn}\ Stmt$	$[strict]$	$\texttt{spawn}\ stmt \rightarrow stmt$
$Pgm ::= \texttt{var}\ \mathsf{List}[Id]\ \texttt{;}\ Stmt$		$\langle \underset{s\ \curvearrowright\ pgm}{\underline{\texttt{var}\ xl\ \texttt{;}\ s}}\rangle_{\mathsf{k}}\ \langle \underset{xl}{\underline{\cdot}}\rangle_{\mathsf{vars}}$ $stmt \curvearrowright pgm \rightarrow pgm$

Figure 4.3: $\mathbb{K}$ type system for IMP++ (and IMP)

types, etc. Since programs and fragments of programs are now going to be rewritten into their types, we need to add to computations some basic types. Also, in addition to the computation to be typed, configurations must also hold the declared variables. Thus, we define the following (the "..." in the definition of K includes all the default syntax of computations, such as the original language syntax, "$\curvearrowright$", freezers, etc.):

$$\begin{array}{rcl} K & ::= & \ldots \mid int \mid bool \mid stmt \mid pgm \\ Configuration^{Type}_{\textsc{Imp++}} & \equiv & \langle\langle K\rangle_{\mathsf{k}}\ \langle \mathsf{Set}[Id]\rangle_{\mathsf{vars}}\rangle_{\top} \end{array}$$

Figure 4.3 shows the IMP/IMP++ type system as a $\mathbb{K}$ system over such configurations. Constants reduce to their types, and types are propagated through each language construct in a straightforward manner. Note that almost each language construct is strict now, because we want to type all its arguments in almost all cases in order to apply the typing policy of the construct. Two constructs are exceptions, namely the increment and the assignment. The typing policy of these constructs is that they take precisely a variable and not something that types to int. If we defined, e.g., the assignment strict and with rule $int = int$, then our type system would allow ill-formed programs like "$x + y = 0\ ;$". Note how we defined the typing policy of programs "$\mathtt{var}\,xl$; s": the declared variables xl are stored into the $\langle\rangle_{\mathsf{vars}}$ cell (expected to be initially empty) and the statement is scheduled for typing (using a structural rule), placing a "reminder" in the computation that the *pgm* type is eventually expected; once/if the statement is correctly typed, the type *pgm* is generated.

4.2 The $\mathbb{K}$ Technique

Q/A

Q: *What are these Question/Answer boxes in this section?*
A: Each subsection in this section introduces an important component of the $\mathbb{K}$ technique, such as configurations, computations, or semantic rules. Each Q/A box captures the essence of the corresponding subsection *from a user perspective*. They will ease the understanding of how the various components fit together.

In this section we present the $\mathbb{K}$ *technique*, which consists of a series of guidelines and notations that turn $\mathbb{K}$ into an effective framework for defining programming languages or calculi. The development of the $\mathbb{K}$ technique has been driven by practical needs, and it is the result of our efforts to define various programming languages, paradigms, and calculi as rewrite or $\mathbb{K}$-systems. We would like to make two important observations before we proceed:

1. The $\mathbb{K}$ technique is *flexible* and *open-ended*. Our current guidelines and notations are convenient enough to define the range of languages, features and calculi that we considered so far. Some readers may, however, prefer different or new notations. As an analogy, there are no rigid rules for how to write an SOS configuration [136]: one may use the angle-bracket notation $\langle code, state, \ldots\rangle$, the square bracket notation $[code, state, \ldots]$, or even the simple tuple notation $(code, state, \ldots)$; also, one may use a different (from comma) symbol to separate the various configuration ingredients and, even further, one could use writing conventions (such as the "state" or "exception" conventions in [116]) to simplify the writing of SOS definitions. Even though we believe that our notational conventions discussed in this section should be sufficient for any definitional task, we still encourage our reader to feel free to change our notations or propose new ones if needed to better fit one's needs or style. Nevertheless, our current prototype implementation of $\mathbb{K}$ described in Chapter 7 relies on our current notation as described in this section; therefore, to use our tool one needs to obey our notation.

2. The $\mathbb{K}$ technique yields a *semantic definitional style*. As an analogy, no matter what notations one uses for configurations and other ingredients

in SOS definitions (see item above), or even whether one uses rewriting logic or any other computational framework to represent and execute SOS definitions or not, SOS still remains SOS, with all its advantages and limitations; the same holds true for any other definitional style. Similarly, we expect that the 𝕂 technique can be represented or implemented in various back-end computational frameworks. Indeed, the same way the various conventional language definitional styles become *definitional methodologies or styles* within rewriting logic as shown in Chapter 3, the 𝕂 technique can also be cast as a definitional methodology or style within other computational frameworks; in Chapter 7 we show how this can be done for rewriting logic and Maude, for example. We prefer 𝕂 rewriting as the intended semantics for the 𝕂 technique because we believe that it ensures the maximum of concurrency one can hope for in 𝕂 definitions. However, if one is not sensitive to this true concurrency aspect or if one prefers a certain computational framework over anything else, then one can very well use the 𝕂 technique in that framework.

4.2.1 𝕂 Configurations: Nested Cell Structures

Q/A

Q: *Do I need to define a configuration for my language?*
A: No, but it is strongly recommended to define one whenever your language is non-trivial. Even if you define no configuration, you still need to define the cells used later on in the semantic rules; otherwise the rules will not parse.
Q: *How can I define a configuration?*
A: All you need is to define a potentially *nested-cell* structure like in Figure 4.4, which is a cell term over the simple cell grammar described below. By defining the configuration you have three benefits:

- You implicitly define all the needed cells, which is required anyway;
- You can reuse existing semantic rules that were conceived for more abstract configurations, via a process named *context transformation*; and
- You have a better understanding of all the semantic ingredients that you need for your subsequent semantics as well as their role.

In 𝕂 definitions, the programming language, calculus or system configuration is represented as a potentially *nested cell* structure. This is similar in spirit to how configurations are represented in chemical abstract machines (Chams, see [15]) or in membrane systems (P-systems, see [131]), except that $\mathbb{K}$'s cells can hold more varied data and are not restricted to certain means to communicate with their environment. The various cells in a $\mathbb{K}$ configuration hold the infrastructure needed to process the remaining computation, including the computation itself; cells can hold, for example, computations (these are discussed in depth in Section 4.2.2), environments, heaps or stores, remaining input, output, analysis results, resources held, bookkeeping information, and so on. The number and type of cells that appear in a configuration is not fixed and is typically different from definition to definition. $\mathbb{K}$ assumes and makes intensive use of the entire range of structures allowed by algebraic CFGs, such as lists, sets, multisets, and maps.

Formally, $\mathbb{K}$ configurations have the following simple, nested-cell structure:

$$\begin{array}{rcl} \mathit{Cell} & ::= & \langle \mathit{CellContents} \rangle_{\mathit{CellLabel}} \\ \mathit{CellContents} & ::= & \mathit{Sort} \mid \mathsf{Bag}_{\textvisiblespace}[\mathit{Cell}\,] \\ \mathit{CellLabel} & ::= & \mathit{CellName} \mid \mathit{CellName}{*} \\ \mathit{CellName} & ::= & \top \mid \mathsf{k} \mid \textvisiblespace \mid \mathsf{env} \mid \mathsf{store} \mid \ldots \\ & & \text{(language-specific cell names; } \top,\ \mathsf{k} \text{ are common)} \end{array}$$

where *Sort* can be *any sort name*, including arbitrary list ($\mathsf{List}[\mathit{Sort}]$), set ($\mathsf{Set}[\mathit{Sort}]$), bag ($\mathsf{Bag}[\mathit{Sort}]$) or map ($\mathsf{Map}[\mathit{Sort}_1 \mapsto \mathit{Sort}_2]$) sorts. Many $\mathbb{K}$ definitions share the cell labels $\top$ (which stands for "*top*") and k (which stays for "*computation*"). They are built-in in our implementation of $\mathbb{K}$ in Maude described in Chapter 7, so one needs not declare them in each language definition. The white-space or "invisible" label "␣" may be preferred as an alternative to $\top$ and/or k, particularly when there is a need for only one cell type, like in the definitions of CCS and Pi calculi. The cells with starred labels say that there could be multiple instances, or clones, of that cell. This multiplicity information is optional[1], but can be useful for context transformation (Section 4.2.5).

We have seen so far three $\mathbb{K}$ configurations, for IMP, for IMP++, and for their type system:

$$\begin{array}{rcl} \mathit{Configuration}_{\text{IMP}} & \equiv & \langle \langle K \rangle_{\mathsf{k}}\ \langle \mathsf{Map}[\mathit{Id} \mapsto \mathit{Int}] \rangle_{\mathsf{state}} \rangle_{\top} \\ \mathit{Configuration}_{\text{IMP++}} & \equiv & \langle \langle K \rangle_{\mathsf{k}}\ \langle \mathsf{Map}[\mathit{Id} \mapsto \mathit{Int}] \rangle_{\mathsf{state}}\ \langle \mathsf{List}[\mathit{Int}] \rangle_{\mathsf{in}}\ \langle \mathsf{List}[\mathit{Int}] \rangle_{\mathsf{out}} \rangle_{\top} \\ \mathit{Configuration}^{\mathit{Type}}_{\text{IMP++}} & \equiv & \langle \langle K \rangle_{\mathsf{k}}\ \langle \mathsf{Set}[\mathit{Id}\,] \rangle_{\mathsf{vars}} \rangle_{\top} \end{array}$$

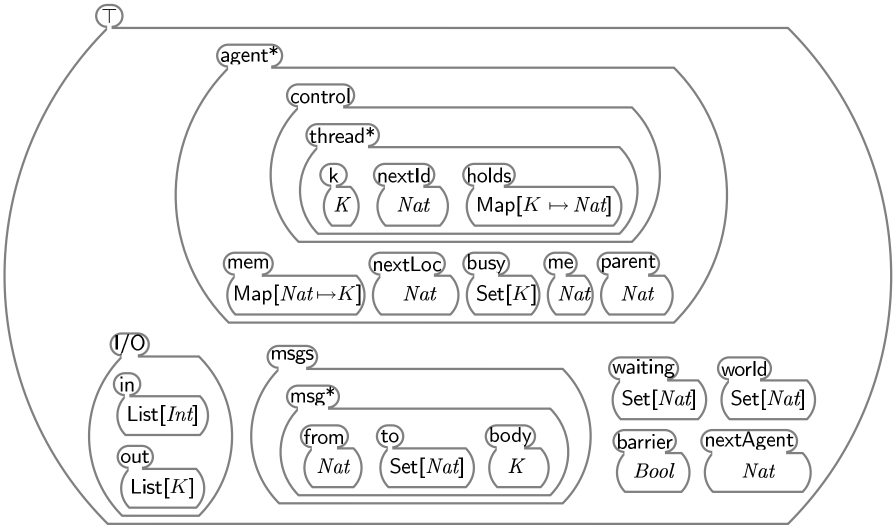
Notice that they all obey the general cell grammar above, that is, they are nested cell structures; the bottom cells only contain a sort and no other cells.

As a more complex example, Figure 4.4 presents the $\mathbb{K}$ configuration of AGENT (see Section 4.3), a pedagogical language conceived to challenge and expose the limitations of the various language definitional frameworks.

Figure 4.4 presents both a textual representation of AGENT configurations (as the ones for IMP and IMP++ described above), as well as a graphical one,

[1] Note, in particular, that we omitted it for the k label in the IMP++ configuration (IMP++ is multithreaded).

$$\begin{aligned}
\mathit{Configuration}_{\textsc{Agent}} &\equiv \langle \mathit{Agents}_{\textsc{Agent}}\ \mathit{I/O}_{\textsc{Agent}}\ \mathit{Messages}_{\textsc{Agent}}\ \mathit{Rest}_{\textsc{Agent}} \rangle_{\top}\\
\mathit{Agents}_{\textsc{Agent}} &\equiv \left\langle \begin{array}{c} \mathit{Threads}_{\textsc{Agent}} \langle \mathsf{Map}[\mathit{Nat} \mapsto K] \rangle_{\mathsf{mem}} \langle \mathit{Nat} \rangle_{\mathsf{nextLoc}} \\ \langle \mathsf{Set}[K] \rangle_{\mathsf{busy}}\ \langle \mathit{Nat} \rangle_{\mathsf{me}}\ \langle \mathit{Nat} \rangle_{\mathsf{parent}} \end{array} \right\rangle_{\mathsf{agents}*}\\
\mathit{Threads}_{\textsc{Agent}} &\equiv \langle\ \langle\ \langle \mathit{Nat} \rangle_{\mathsf{k}}\ \langle \mathsf{Map}[K \mapsto \mathit{Nat}] \rangle_{\mathsf{holds}}\ \rangle_{\mathsf{thread}*}\ \rangle_{\mathsf{control}}\\
\mathit{I/O}_{\textsc{Agent}} &\equiv \langle \langle \mathsf{List}[\mathit{Int}] \rangle_{\mathsf{in}}\ \langle \mathsf{List}[K] \rangle_{\mathsf{out}} \rangle_{\mathsf{I/O}}\\
\mathit{Messages}_{\textsc{Agent}} &\equiv \langle \langle \langle \mathit{Nat} \rangle_{\mathsf{from}}\ \langle \mathit{Nat} \rangle_{\mathsf{to}}\ \langle K \rangle_{\mathsf{body}} \rangle_{\mathsf{msg}*} \rangle_{\mathsf{msgs}}\\
\mathit{Rest}_{\textsc{Agent}} &\equiv \langle \mathsf{Set}[\mathit{Nat}] \rangle_{\mathsf{waiting}} \langle \mathsf{Set}[\mathit{Nat}] \rangle_{\mathsf{world}} \langle \mathit{Bool} \rangle_{\mathsf{barrier}} \langle \mathit{Nat} \rangle_{\mathsf{nextAgent}}
\end{aligned}$$



$\langle\rangle_{\top}$: top cell, holding everything
$\langle\rangle_{\mathsf{agent}*}$: one agent, can multiply
$\langle\rangle_{\mathsf{control}}$: the control unit of the agent
$\langle\rangle_{\mathsf{thread}*}$: one thread, can multiply
$\langle\rangle_{\mathsf{k}}$: thread's computation
$\langle\rangle_{\mathsf{nextId}}$: next substitution id
$\langle\rangle_{\mathsf{holds}}$: thread's locks
$\langle\rangle_{\mathsf{mem}}$: agent's store
$\langle\rangle_{\mathsf{nextLoc}}$: the next available location
$\langle\rangle_{\mathsf{busy}}$: agent's busy locks
$\langle\rangle_{\mathsf{me}}$: agent's id
$\langle\rangle_{\mathsf{parent}}$: agent's parent id

$\langle\rangle_{\mathsf{I/O}}$: the system's input/output
$\langle\rangle_{\mathsf{in}}$: the system input
$\langle\rangle_{\mathsf{out}}$: the system output
$\langle\rangle_{\mathsf{msgs}}$: all the pending messages
$\langle\rangle_{\mathsf{msg}*}$: one message, can multiply
$\langle\rangle_{\mathsf{from}}$: the message sender's id
$\langle\rangle_{\mathsf{to}}$: the message receivers' ids
$\langle\rangle_{\mathsf{body}}$: the message's value/body
$\langle\rangle_{\mathsf{waiting}}$: the ids of agents waiting at barrier
$\langle\rangle_{\mathsf{world}}$: the ids of all the agents
$\langle\rangle_{\mathsf{barrier}}$: a flag saying if barrier is active
$\langle\rangle_{\mathsf{nextAgent}}$: the next available agent id

Figure 4.4: The configuration of the AGENT language in both term (top) and graphical (middle) representation, with short explanation of cell contents (bottom)

which can be automatically generated by the K2Latex component of our current implementation of $\mathbb{K}$ in Maude described in Chapter 7. The AGENT configurations have five levels of cell-nesting and several cells labels are starred, meaning that there can be multiple instances of those cells. For example, the top cell may contain multiple $\langle\rangle_{\mathsf{agent}}$ cells; each agent may contain, besides information like a local store, busy resources (used as locks for thread synchronization), etc., an arbitrary number of $\langle\rangle_{\mathsf{thread}}$ cells grouped in a $\langle\rangle_{\mathsf{control}}$ cell; each thread contains a local computation and a number of resources (locks) held by the thread. As one may expect, real life language definitions tend to employ rather complex configurations. For example the $\mathbb{K}$ definition of C [51] uses more than fifty cells with five levels of nesting.

The advantage of representing configurations as nested cell-structures is that, like in MSOS [123], subsequent rules only need to mention those configuration items that are needed for those particular rules, as opposed to having to mention the entire configuration, whether needed or not, like in conventional SOS. We can add or remove items from a configuration as we like, only impacting the rules that use those particular configuration items. Rules that do not need the changed configuration items do not need to be touched. This is an important aspect of $\mathbb{K}$, which significantly contributes to its modularity.

Defining a configuration for a $\mathbb{K}$ semantics of a language, calculus or system is an optional step, in that it suffices to only define the desirable cell syntax so that configurations like the desired one parse as ordinary cell terms. That indeed provides all the necessary infrastructure to give the semantic $\mathbb{K}$ rules. However, providing a specific configuration term is useful in practice for at least two reasons. First, the configuration can serve as an intuitive skeleton for writing the subsequent semantic rules, one which can be consulted to quickly find out, for example, what kind of cells are available and where they can be found. Second, the configuration structure is the basis for *context transformation* (see Section 4.2.5), which gives more modularity to $\mathbb{K}$ rules by allowing them to be reusable in language extensions that require changes in the structure of the configuration.

4.2.2 $\mathbb{K}$ Computations: $\curvearrowright$-Separated Nested Lists of Tasks

Q/A

Q: *What are $\mathbb{K}$ computations?*

A: Computations are an intrinsic part of the $\mathbb{K}$ framework. They extend abstract syntax with a special nested-list structure and can be thought of as sequences of fragments of program that need to be processed sequentially.

Q: *Do I need to define computations myself?*

A: What is required is to define an abstract syntax of your language (discussed below) and desired evaluation strategies for the language constructs (discussed in Section 4.2.3), which need to be defined no matter what framework you prefer. By doing so, you implicitly define the basic $\mathbb{K}$ computational infrastructure. In many cases you do not need to define any other computation constructs.

Q: *Do I need to understand in depth what computations are in order to use $\mathbb{K}$?*

A: Not really. If you follow a purely syntactic definitional style mimicking reduction semantics with evaluation contexts [180] in $\mathbb{K}$, then the only computations that you will ever see in your rules are abstract syntax terms.

Q: *What is the benefit of using more complex (than abstract syntax) computations?*

A: Using $\mathbb{K}$ at its full strength. Many complex languages are very hard or impossible to define purely syntactically, while they admit elegant and natural definitions using proper $\mathbb{K}$ computations. For example, the Agent language in Section 4.3.

$\mathbb{K}$ takes a very abstract view of language syntax and, in theory, it is not concerned *at all* with parsing aspects[2]. More precisely, in $\mathbb{K}$ there is only one top-level sort[3] associated to all the language syntax, called K and standing for *computational structures* or *computations*, and terms t of sort K have the abstract syntax tree (AST) representation $l(t_1, \ldots, t_n)$, where l is some $\mathbb{K}$ *label* and t_1,...,t_n are terms of sort K, extended with the list (infix) construct "$\curvearrowright$", read "followed by" or "and then"; for example, if t_1, t_2, ..., t_n are computations then $t_1 \curvearrowright t_2 \curvearrowright \cdots \curvearrowright t_n$ is also a computation, namely the one *sequentializing* t_1, t_2, ..., t_n. All the original language constructs, including constants and program

[2] In practice, like in all other language semantics frameworks, some parser is always assumed or effectively used as a front-end to $\mathbb{K}$ to parse and transform the language syntax into its abstract $\mathbb{K}$ syntax.

[3] Technically, one can define more than one top-level computation sort; however, for simplicity we prefer to keep only one computation sort for now.

variables, as well as all the freezers (discussed below), are regarded as labels. For notational convenience, we continue to write K-terms using the original syntax instead of the harder to read AST notation. Formally, computations are defined as follows:

$$\begin{array}{rcl} K & ::= & \mathit{KLabel}(\mathsf{List}^{\sqcup}[K]) \mid \mathsf{List}_{\curvearrowright}[K] \\ \mathit{KLabel} & ::= & \text{(one per language construct, plus auxiliary ones as needed)} \end{array}$$

The first construct scheme for K abstractly captures any programming language syntax as an AST, provided that one adds one *KLabel* for each language construct. For example, in the case of the IMP language, we add to *KLabel* all the following labels corresponding to the Imp syntax:

$$\begin{array}{rcl} \mathit{KLabel}_{\mathrm{IMP}} & ::= & \mathit{Int} \mid \mathit{Id} \mid \_\,\texttt{+}\,\_ \mid \_\,\texttt{/}\,\_ \mid \_\,\texttt{<=}\,\_ \mid \texttt{not}\,\_ \mid \_\,\texttt{and}\,\_ \mid \texttt{skip;} \mid \_\,\texttt{=}\,\_\,\texttt{;} \mid \_\;\_ \\ & \mid & \texttt{if}\,\_\,\texttt{then}\,\_\,\texttt{else}\,\_ \mid \texttt{while}\,\_\,\texttt{do}\,\_ \mid \texttt{var}\,\_\,\texttt{;}\,\_ \end{array}$$

We recommend the use of the *mix-fix notation* for labels, like in the above labels corresponding to the Imp language; the mix-fix notation was introduced by the OBJ language [65] and followed by many other similar languages, where underscores in the name of an operation mark the places of its arguments. In addition to the language syntax, *KLabel* may include additional labels for semantic reasons; e.g, labels corresponding to semantic domain values which may have not been automatically included in the syntax of the language, such as the *Bool* domain in the case of Imp. We take the liberty to call $\mathbb{K}$ *constants* those computations which are labels applied to, and always intended to be applied to, an empty list of arguments (e.g., `skip;` (), `true` (), 1(), 2(), etc.)

Most K labels always take a fixed number of arguments; e.g., the label $\texttt{if}\,\_\,\texttt{then}\,\_\,\texttt{else}\,\_$ above takes 3 arguments. Even though the simplistic syntax of $\mathbb{K}$ cannot enforce the fixed number of arguments in the semantics, one can show that the semantic rules never change the number of arguments of such labels, so they will always have the original number of arguments as given by the original parsing of the program. There are syntactic language constructs, however, which are allowed to take an arbitrary number of arguments. A typical example is, for example, lists of expressions or lists of variables in variable declarations. A list of expressions "$e_1 \texttt{ , } e_2 \texttt{ , } \dots \texttt{ , } e_n$" is captured as a $\mathbb{K}$ computation of the form "$\_\texttt{ , }\_(t_1, t_2, \dots, t_n)$" (with $\_\texttt{ , }\_()$ for the empty list), where t_i is the $\mathbb{K}$ representation of e_i. Therefore, lists of expressions are regarded as labels applied to an arbitrary number of arguments; the name of the label is inspired from list constructs being thought of as binary associative operations.

It is convenient in many $\mathbb{K}$ definitions to distinguish syntactically between proper computations and computations which are finished. A similar phenomenon is common and also accepted in other definitional styles, which distinguish between proper expressions and values, for example. To make this distinction smooth, we add the *KResult* syntactic subcategory of K which is constructed using corresponding labels (all labels in *KResultLabel* are also in *KLabel*):

$$\begin{array}{rcl} \mathit{KResult} & ::= & \mathit{KResultLabel}(\mathsf{List}_{\sqcup}[K]) \\ \mathit{KResultLabel} & ::= & \text{(one per construct of terminated computations,} \\ & & \text{e.g., values, results, etc.)} \end{array}$$

Among the labels in *KResultLabel* one may have certain language constants, such as `true`, 0, 1, etc., but also labels that correspond to non-constant terms, for example $\lambda\_.\_$; indeed, in some λ-calculi, λ-abstractions $\lambda x.e$ (or $\lambda\_.\_(x, e)$ in AST form), are values (or finished computations).

There is yet another category of labels that turns out to be useful in semantic definitions, namely *hybrid* labels, which are intended to "hold data", i.e., take lists of K results into a K result:

$$\begin{array}{rcl} \mathit{KResult} & ::= & \mathit{KHybridLabel}(\mathsf{List}_{\sqcup}[K]) \\ \mathit{KHybridLabel} & ::= & \text{(one per construct that does not reduce} \\ & & \text{ once its arguments are reduced)} \end{array}$$

For example, the "list" label $\_\,\texttt{,}\,\_$ above should be declared hybrid, since we want $\_\,\texttt{,}\,\_(t_1, t_2, ..., t_n)$ to be considered evaluated in the semantics whenever each t_i is evaluated. On the other hand, labels like $\_\,\texttt{+}\,\_$ are obviously not hybrid. In fact, hybrid labels are rather rare. It may also be worth noting that, unlike the result labels, the hybrid labels are more of a convenience than a necessity. Indeed, one can always introduce a new result label for any label intended to be hybrid, e.g. $\_\,\texttt{,}_{\mathit{result}}\,\_$, together with a rule replacing the label with its result counterpart whenever its arguments become results, e.g., "$\_\,\texttt{,}\,\_(t_1, t_2, ..., t_n) \rightharpoondown \_\,\texttt{,}_{\mathit{result}}\,\_(t_1, t_2, ..., t_n)$ when $t_1, t_2, ..., t_n \in \mathit{KResult}$" (a structural rule, see Section 4.2.3). However, this would be inconvenient in many cases.

We take the liberty to write language or calculus syntax either in AST form, like in $\lambda\_.\_(x, e)$, $\texttt{if}\ \_\ \texttt{then}\ \_\ \texttt{else}\ \_(b, s_1, s_2)$, and $\texttt{skip}()$, or in mixfix form, like in $\lambda x.e$, $\texttt{if}\, b\, \texttt{then}\, s_1\, \texttt{else}\, s_2$, and `skip`. For example, in Figure 4.1 we preferred to write the rule for addition as $i_1 \texttt{+} i_2 \rightarrow i_1 +_{\mathit{Int}} i_2$ instead of $\_\,\texttt{+}\,\_(i_1(), i_2()) \rightarrow (i_1\ +_{\mathit{Int}} i_2)()$. In our Maude implementation of $\mathbb{K}$ described in Chapter 7, thanks to Maude's builtin support for mixfix notation and parsing capabilities, we actually write programs using the mixfix notation. Even though theoretically unnecessary, this is actually very convenient in practice, because it makes language definitions more readable and, consequently, less error-prone. Additionally, programs in the defined languages can be regarded as terms the way they are, without any intermediate AST representation. In other implementations of $\mathbb{K}$, one may need to use an explicit parser or to get used to reading syntax in AST representation. Either way, from here on we assume that programs, or fragments of programs, parse as computations in $\mathbb{K}$.

The second construct scheme for $\mathbb{K}$ allows one to sequentialize computational tasks. Intuitively, $k_1 \curvearrowright k_2$ says "process k_1 then k_2". How this is used and what

$$\begin{array}{l}
(\texttt{if true then} \cdot \texttt{else} \cdot) \curvearrowright \texttt{while false do} \cdot \\
a_1 \curvearrowright \square + a_2 \\
a_2 \curvearrowright a_1 + \square \\
a_3 \curvearrowright (a_1 + a_2) + \square \\
a_3 \curvearrowright (a_1 \curvearrowright \square + a_2) + \square \\
b \curvearrowright \texttt{if}\, \square \,\texttt{then}\, s_1 \,\texttt{else}\, s_2 \\
b \curvearrowright \texttt{if}\, \square \,\texttt{then}\, (s \curvearrowright \texttt{while}\, b \,\texttt{do}\, s) \,\texttt{else}\, \cdot
\end{array}$$

Figure 4.5: Examples of $\mathbb{K}$ computations

is the exact meaning of "process" is left open and depends upon the particular definition. For example, in a concrete semantic language definition it can mean "evaluate k_1 then k_2", while in a type inferencer definition it can mean "type and accumulate type constraints in k_1 then do the same for k_2", etc. Figure 4.5 shows examples of computations making use of the $\mathsf{List}_{\curvearrowright}[K]$ structure of K (we use parentheses for disambiguation). The "$\cdot$" in the first and last computations in Figure 4.5 is the unit of K (given by $\mathsf{List}_{\curvearrowright}[K]$). Note that $\curvearrowright$-separated lists of computations can be nested. Most importantly note that, unlike in evaluation contexts, $\square$ is not a "hole" in K, but rather part of a *KLabel*; the *KLabel*s involving $\square$ in Figure 4.5 are "$\_\ \texttt{+}\ \square$", "$\square\ \texttt{+}\ \_$", and "$\texttt{if}\ \square\ \texttt{then}\ \_\ \texttt{else}\ \_$". The $\square$ carries the "plug here" intuition; e.g., one may think of "$a_1 \curvearrowright \square + a_2$" as "process a_1, then plug its result in the hole in $\square + a_2$". The user of $\mathbb{K}$ is not expected to declare these special labels. We assume them whenever needed. In our implementation of $\mathbb{K}$ in Maude [159], all these are generated automatically as constants of sort *KLabel* after a simple analysis of the language syntax.

Freezers To distinguish the labels containing $\square$ in their name from the labels that encode the syntax of the language under consideration, we call the former *freezers*. The role of the freezers is therefore to store the enclosing computations for future processing. One can freeze computations at will in $\mathbb{K}$, using freezers like the ones above, or even by defining new freezers. In complex $\mathbb{K}$ definitions, one may need many computation freezers, making definitions look heavy and hard to read if one makes poor choices for freezer names. Therefore, we adopt the following *freezer naming convention*, respected by all the freezers above:

> If a computation can be seen as $c[k, x_1, \ldots, x_n]$ for a multicontext c and a freezer is needed to freeze everything except k, then its name is "$c[\square, \_, \ldots, \_]$".

Additionally, to increase readability, we take the freedom to generalize the adopted mixfix notation in $\mathbb{K}$ and "plug" the remaining computations in the freezer, that is, we write $c[\square, k_1, \ldots, k_n]$ instead of $c[\square, \_, \ldots, \_](k_1, \ldots, k_n)$. For instance, if $\_@\_$ is some binary operation and if, for some reason, in contexts of the form $(e_1 @ e_2) @ (e_3 @ e_4)$ one wishes to freeze e_1, e_3 and e_4 (in order to, e.g., process e_2), then, when there is no confusion, one may write $(e_1 @ \square) @ (e_3 @ e_4)$

instead of $((\_@\square)@(\_@\_))(e_1, e_3, e_4)$. This convention is particularly useful when one wants to follow a reduction semantics with evaluation contexts style in $\mathbb{K}$, because one can mechanically associate such a freezer to each context-defining production. For example, the freezer $(\_@\square)@(\_@\_)$ above would be associated to a production of the form "*Cxt* ::= (*Exp*@*Cxt*)@(*Exp*@*Exp*)".

4.2.3 $\mathbb{K}$ Rules: Computational and Structural

Q/A

Q: *How are the* $\mathbb{K}$ *rules different from conventional rewrite rules?*
A: The $\mathbb{K}$ framework builds upon $\mathbb{K}$ rewriting; how the $\mathbb{K}$ rewriting rules differ from standard rules is detailed in Chapter 6.
Q: *What do I lose if I think of* $\mathbb{K}$ *rules as sugared variants of standard rules?*
A: Not much if you are *not* interested in *true concurrency*.
Q: *Does that mean that I can execute* $\mathbb{K}$ *definitions on any rewrite engine?*
A: Yes. However, it is desirable to use a rewrite engine with support at least for associative matching. In fact, our current implementation of $\mathbb{K}$ described in Chapter 7 does so.

$$\begin{array}{cc}
\textit{Computational rules} & \textit{Structural rules} \\
p[\underset{r_1}{\underline{l_1}}, \underset{r_2}{\underline{l_2}}, \ldots, \underset{r_n}{\underline{l_n}}] & p[\underset{r_1}{\underset{\cdots}{l_1}}, \underset{r_2}{\underset{\cdots}{l_2}}, \ldots, \underset{r_n}{\underset{\cdots}{l_n}}]
\end{array}$$

Figure 4.6: $\mathbb{K}$ rules

The $\mathbb{K}$ framework builds upon $\mathbb{K}$ rewriting. $\mathbb{K}$ rules can be split into computational and structural. From here on, we distinguish them as shown in Figure 4.6. They both consist of a local context, or pattern, p, with some of its subterms underlined and rewritten to corresponding subterms underneath the line. The idea is that the underlined subterms represent the "read-write" part of the rule, while the operations in p which are not underlined represent the "read-only" part of the rule and can be shared by concurrent rule instances. The difference between computational and structural rules is that rewrite steps using the latter do not count as computational steps, their role being to rearrange the structure of the term so that computational rules can apply. There are no rigid requirements on when a $\mathbb{K}$ semantic rule should be computational versus structural. While in most cases the distinction between the two is natural, there are situations where one needs to subjectively choose one or the other; for example, we chose the rule for variable declarations in the IMP semantics in Figure 4.1 to be structural, but some language designers may prefer it to be computational.

$$a_1 + a_2 \rightleftharpoons a_1 \curvearrowright \Box + a_2$$
$$a_1 + a_2 \rightleftharpoons a_2 \curvearrowright a_1 + \Box$$

Figure 4.7: Rules for _ + _ *strict*

Recall from Section 4.1.1 that we prefer to use the conventional rewrite rule notations "$l \rightarrow r$" and "$l \rightharpoonup r$" for computational and structural $\mathbb{K}$ rules, respectively, when $p = \Box$ (that is, when there is only one read-write part, the entire pattern, and no read-only part). There is not much to say here about $\mathbb{K}$ rules besides what has already been said in Section 4.1. The will be formally defined and discussed at length in Chapter 6. We would only like to elaborate a bit further on the heating/cooling rules and their corresponding strictness attributes.

Heating/Cooling Structural Rules

Q/A

Q: *What is the role of the heating/cooling rules?*
A: These are $\mathbb{K}$'s mechanism to define evaluation strategies of language constructs. They allow you to decompose fragments of programs into sequences of smaller computations, and to compose smaller computations back into fragments of programs.
Q: *Do I need to define such heating/cooling rules myself?*
A: Most likely no. It usually suffices to define *strictness attributes*, as discussed below; these are equivalent to defining evaluation contexts. Strictness attributes serve as a notational convenience for defining obvious heating/cooling structural rules.

After defining the desired language syntax so that programs or fragments of programs become terms of sort K, called computations, the very first step towards giving a $\mathbb{K}$ semantics is to define the evaluation strategies or strictness of the various language constructs by means of heating/cooling rules, or more conveniently, by means of the special attributes described shortly. The heating/cooling rules allow us to regard computations many different, but completely equivalent ways. For example, "$a_1 + a_2$" in IMP may be regarded also as "$a_1 \curvearrowright \Box + a_2$", with the intuition "schedule a_1 for processing and *freeze* a_2 in freezer $\Box + \_$", but also as "$a_2 \curvearrowright a_1 + \Box$" (recall from Section 4.1.1 that, in IMP, addition is intended to be non-deterministic). As discussed in Section 4.2.2, freezers are nothing but special labels whose role is to store computations for future processing.

Heating/cooling structural rules tell how to "pass in front" of the computation fragments of the program that need to be processed, and also how to "plug their results back" once processed. In most language definitions, *all* such rules can be extracted automatically from $\mathbb{K}$ strictness operator attributes as explained below; Figure 4.1 shows several examples of strictness attributes. For example, the *strict* attribute of _ + _ is equivalent to the two heating/cooling pairs of $\mathbb{K}$ rules in Figure 4.7 (a_1 and a_2 range over computations in K). The symbol "$\rightleftharpoons$" is borrowed from the chemical abstract machine (Cham) [15], as a shorthand for combinations of a heating rule ("$\rightharpoonup$") and a cooling rule ("$\leftharpoondown$"). Indeed, one can think of the first rule as follows: to process $a_1 + a_2$, let us first "heat" a_1, applying the rule from left to right; once a_1 is processed (using other rules in the semantics) producing some result, place that result back into context via a "cooling" step, applying the rule from right to left. These heating/cooling rules can be applied at any moment and in any direction, since they are regarded not as computational steps but as structural rearrangements. For example, one can use the heating/cooling rules for "_ + _" to pick and pass in front either a_1 or a_2, then rewrite it one step only using semantic rules (defined shortly), then plug it back into the sum, then pick and pass in front either a_1 or a_2 again and rewrite it one step only, and so on, thus obtaining the desired non-deterministic operational semantics of _ + _.

The general idea to define a certain evaluation context, say $c[\square, N_1, \ldots, N_n]$, where $N_1, \ldots, N_n$ are the various syntactic categories involved (or non-terminals in the CFG of the language), is to define a *KLabel* freezer $c[\square, \_, \ldots, \_]$ like discussed in Section 4.2.2, together with a heating/cooling rule pair

$$c[k, k_1, \ldots, k_n] \rightleftharpoons k \curvearrowright c[\square, k_1, \ldots, k_n].$$

One should be aware that in $\mathbb{K}$ "$\square$" is nothing but a symbol that we prefer to use as part of label names. In particular, "$\square$" is *not* a computation (recall that in reduction semantics with evaluation contexts "$\square$" is a special context, called a "hole"). For example, a hasty reader may think that $\mathbb{K}$'s approach to strictness is unsound, because one can "prove" wrong correspondences as follows:

$$\begin{array}{lll} a_1 + a_2 & \rightharpoonup a_1 \curvearrowright \square + a_2 & \text{(by the first rule above applied left-to-right)} \\ & \rightharpoonup a_1 \curvearrowright a_2 \curvearrowright \square + \square & \text{(by the second rule above applied left-to-right)} \\ & \rightharpoondown a_1 \curvearrowright a_2 + \square & \text{(by the first rule above applied right-to-left)} \\ & \rightharpoondown a_2 + a_1 & \text{(by the second rule above applied right-to-left)} \end{array}$$

What is wrong in the above "proof" is that one cannot apply the second rule in the second step above, because $\square + a_2$ is nothing but a convenient way to write the frozen computation $\square +_{\_}(a_2)$. One may say that there is no problem with the above, because _ + _ is intended to be commutative anyway; unfortunately, the same could be proved for any non-deterministic construct, for example for a division operation, "/", if that was to be included in our language. Since the heating/cooling rules are thought of as structural rearrangements, so that computational steps take place *modulo* them, then it would certainly be wrong to have both "a_1/a_2" and "a_2/a_1" in the same computational class. One of $\mathbb{K}$'s most subtle technical aspects, which fortunately is transparent to users, is to find the right (i.e., as weak as possible) restrictions on the applications of heating/cooling

$$\begin{array}{l}
x * (y + 2) \\
x \curvearrowright (\Box * (y + 2)) \\
x \curvearrowright (\Box * (y \curvearrowright (\Box + 2))) \\
x \curvearrowright (\Box * (2 \curvearrowright (y + \Box))) \\
(y + 2) \curvearrowright (x * \Box) \\
y \curvearrowright (\Box + 2) \curvearrowright (x * \Box) \\
2 \curvearrowright (y + \Box) \curvearrowright (x * \Box) \\
x * (y \curvearrowright (\Box + 2)) \\
x * (2 \curvearrowright (y + \Box))
\end{array}$$

Figure 4.8: Computational class

$$\begin{array}{l}
a_1 \;\texttt{<=}\; a_2 \rightleftharpoons a_1 \curvearrowright \Box \;\texttt{<=}\; a_2 \\
r_1 \;\texttt{<=}\; a_2 \rightleftharpoons a_2 \curvearrowright r_1 \;\texttt{<=}\; \Box
\end{array}$$

Figure 4.9: Rules for _ <= _ *seqstrict*

equations, so that each computational class contains no more than one fragment of program. The idea is to only allow heating and/or cooling of operator arguments that are proper syntactic computations (i.e., terms over the original syntax, i.e., different from “$\cdot$” and containing no “$\curvearrowright$”). With that, for example, Figure 4.8 shows the computational class of the expression $x * (y + 2)$ in the context of a language definition with non-deterministically strict binary $+$ and $*$. Note that there is only one syntactic computation in the computation class above, namely the original expression itself. This is a crucial desired property of $\mathbb{K}$.

4.2.4 Strict and Hybrid Attributes

In $\mathbb{K}$ definitions, one typically defines zero, one, or more heating/cooling rules per language construct, depending on its intended evaluation/processing strategy. These rules tend to be straightforward and boring to write, so in $\mathbb{K}$ we prefer a higher-level and more compact and intuitive approach: we annotate the language syntax with *strictness attributes*. A language construct annotated as *strict*, such as for example the “_ + _” in Figure 4.1, is automatically associated a heating/-cooling pair of rules as above for each of its subexpressions. If an operator is intended to be strict in only some of its arguments, then the positions of the strict arguments are listed as arguments of the *strict* attribute. For example, note that the strictness attribute of if _ then _ else _ in Figure 4.1 is *strict*(1); that means that a heating/cooling equation is added only for the first subexpression of the conditional, namely the rule pair “$\texttt{if}\;b\;\texttt{then}\;s_1\;\texttt{else}\;s_2 \rightleftharpoons b \curvearrowright \texttt{if}\;\Box\;\texttt{then}\;s_1\;\texttt{else}\;s_2$”.

The two pairs of heating/cooling rules corresponding to the strictness attribute *strict* of _ + _ above did not enforce any particular order in which the two subexpressions were processed. It is often the case that one wants a deterministic order in which the strict arguments of a language construct are processed, typically from left to right. Such an example is the relational operator _<=_ in Figure 4.1, which was declared with the strictness attribute *seqstrict*, saying that its subexpressions are processed deterministically, from left to right. The attribute *seqstrict* requires the definition of the syntactic category of result computations *KResult*, as discussed in Section 4.2.2, and it can be desugared automatically as follows: generate a heating/cooling pair of rules for each argument like in the case of *strict*, but requiring that all its previous arguments are in *KResult*. For example, the *seqstrict* attribute of _ <= _ desugars into the rules in Figure 4.9 ($a_1, a_2 \in K$ and $r_1 \in KResult$). Like the *strict* attribute, *seqstrict* can also take a list of numbers as argument and then the heating/cooling rules are generated so that the corresponding arguments are processed in that order.

The strictness attributes also apply to labels taking an arbitrary number of arguments. For example, an expression-list construct _ , _ can be declared *strict*, meaning that all the expressions appearing in the list need to evaluate whenever the expression-list is asked to evaluate. The general desugaring rules of strictness can be best given in terms of $\mathbb{K}$ label representations. For example, a *strict* attribute associated to $label \in KLabel$ is syntactic sugar for rules like the ones below ($kl_1, kl_2 \in \mathsf{List}[K]$, $k \in K$, $r \in KResult$):

$$\begin{array}{l} label(kl_1, k, kl_2) \rightleftharpoons k \curvearrowright label(\_, \square, \_)(klist(kl_1), klist(kl_2)) \\ r \curvearrowright label(\_, \square, \_)(klist(kl_1), klist(kl_2)) \rightleftharpoons label(kl_1, r, kl_2) \end{array}$$

where $label(\_, \square, \_)$ and $klist$ are labels. This way, the lists kl_1 and kl_2 are unambiguously frozen until k is reduced to a result. For *seqstrict*, replace $kl_1 \in \mathsf{List}[K]$ by $rl_1 \in \mathsf{List}[KResult]$ above.

Our most general strictness declaration in $\mathbb{K}$, also supported by our current implementation described in Chapter 7, is to declare a certain syntactic context (a derived term) strict in a certain position designated by a special marker □. For example, in the REF module of the $\mathbb{K}$ definition of AGENT in Section 4.3, we declare the context "* □ = _", with the meaning that the assignment statement applied to a pointer needs to first evaluate the pointer expression before other semantic rules can apply.

Language constructs can also be annotated with the *hybrid* attribute, to indicate that their corresponding label is hybrid (see Section 4.2.2).

4.2.5 Context Transformation

We next introduce one of the most advanced feature of $\mathbb{K}$, the *context transformation*, which gives $\mathbb{K}$ an additional degree of modularity. The process of context transformation is concerned with automatically modifying existing $\mathbb{K}$ rules according to the cell structure defined by the desired configuration of a target language. The benefit of context transformation is that it allows us to define semantic rules more abstractly, without worrying about the particular details of the concrete final language configuration. This way, it implicitly enhances the modularity and reuse of language definitions: existing rules do not need to change as the configuration of the languages changes to accommodate additional language features, and language features defined generically once and for all can be reused across different languages with different configuration structures.

Defining a configuration (see Section 4.2.1) is a necessary step in order to make use of $\mathbb{K}$'s context transformation. Assuming that the various cell-labels forming the configuration are distinct, then one can use the structure of the configuration to *automatically* transform abstract rules, i.e., ones that do not obey the intended cell-structure of the configuration, into concrete ones that are well-formed within the current configuration structure. The context transformation process can be thought of as being applied statically on all rules, before the $\mathbb{K}$-system is executed.

Consider, for example, the $\mathbb{K}$ semantic rule for `print` in Imp++ (see Figure 4.2):

$$\langle \underset{\texttt{skip}}{\underline{\texttt{print}\, v}} \ \cdots\rangle_{\mathsf{k}} \ \langle \cdots \ \underset{v}{\underline{\cdot}}\rangle_{\mathsf{out}}$$

This rule captures the semantics of the output statement in the most abstract and compact possible way: if "`print` V" is the next computational task (where V is a value), then append V to the end of the $\langle\rangle_{\mathsf{out}}$ buffer and evaluate the `print` statement to the unit for statements. This rule matched the configuration structure of Imp++, because the Imp++ configuration structure was very simple: a top level cell containing all the other cells inside as simple, non-nested cells. Consider now defining a more complex language, like the Agent language in Section 4.3 whose configuration is shown in Figure 4.4. The particular cell arrangement in the Agent configuration makes the rule above directly inapplicable; to be precise, even though the rule context still parses as a *CellContents*-term, it will never match/apply when used in the context of the Agent configuration.

Context transformation is about automatic adaptation of $\mathbb{K}$ rules like above to new configurations. Indeed, note that there is only one way to bring the cells $\langle\rangle_{\mathsf{k}}$ and $\langle\rangle_{\mathsf{out}}$ mentioned in the rule above together: to wrap them with the cells above them declared in the Agent configuration until a common cell contents is reached, namely to transform the rule into the following one:

$$\langle \cdots \ \langle \cdots \ \langle \cdots \ \langle \underset{\texttt{skip}}{\underline{\texttt{print}\, V}} \ \cdots\rangle_{\mathsf{k}} \ \cdots\rangle_{\mathsf{thread}} \ \cdots\rangle_{\mathsf{control}} \ \cdots\rangle_{\mathsf{agent}} \ \langle \cdots \ \langle \cdots \ \underset{V}{\underline{\cdot}}\rangle_{\mathsf{out}} \ \cdots\rangle_{\mathsf{I/O}}$$

Context transformation can be defined as the process of customizing the paths to the various cells used in a rule according to the configuration of the target language. As part of this customization process, variables are used for the remaining parts of the introduced cells, so that other rule instances concerned with those parts of the cells can apply concurrently with the transformed rule.

The Locality Principle

The rule above was rather simple, in that there was no confusion on how to complete the paths to the referred cells. Consider instead a general-purpose $\mathbb{K}$ rule for pointer dereferencing:

$$\langle \underset{V}{\underline{*\,N}}\ \cdots\rangle_{\mathsf{k}}\ \langle\cdots\ N\mapsto V\ \cdots\rangle_{\mathsf{mem}}$$

This says that if dereferencing of location N is the next computational task and if value V is stored at location N, then $*\,N$ rewrites to V. The configuration of Agent considers the cells $\langle\rangle_{\mathsf{k}}$ and $\langle\rangle_{\mathsf{mem}}$ at different levels in the structure, so a context transformation operation is necessary to adapt this rule to Agent's concrete configuration. However, without care, there are two ways to do it:

$$\langle\cdots\ \langle\cdots\ \langle \underset{V}{\underline{*\,N}}\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{thread}}\ \cdots\rangle_{\mathsf{control}}\ \langle\cdots\ N\mapsto V\ \cdots\rangle_{\mathsf{mem}},\text{ or}$$

$$\langle\cdots\ \langle\cdots\ \langle\cdots\ \langle \underset{V}{\underline{*\,N}}\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{thread}}\ \cdots\rangle_{\mathsf{control}}\ \cdots\rangle_{\mathsf{agent}}\ \langle\cdots\ \langle\cdots\ N\mapsto V\ \cdots\rangle_{\mathsf{mem}}\ \cdots\rangle_{\mathsf{agent}}.$$

The first rule above says that the thread containing the dereferencing and the store are part of the same agent, while the second rule says that they are in different agents (why we are allowed to multiply the agent cells is explained shortly). Even though we obviously meant the first one, both these rules make sense according to the configuration of Agent.

To avoid such conflicts, context transformation relies on the *locality principle*: rules are transformed in a way that makes them as local as possible, or, in other words, in a way that the resulting rule matches as deeply as possible in the concrete configuration. Thus, the locality principle rules out the second rule transformation above, since it is less local than the former.

If, for some reason (which makes no sense for Agent) one means a non-local transformation of a rule context, then one should add more cell-structure to the abstract rule for disambiguation. For example, if one really meant the second context transformation of the dereferencing rule above, then one should have written the abstract rule, for example, as follows:

$$\langle\cdots\ \langle \underset{V}{\underline{*\,N}}\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{agent}}\ \langle\cdots\ N\mapsto V\ \cdots\rangle_{\mathsf{mem}}$$

Now there is only one way to transform it to fit the configuration of Agent, namely like in the second Agent-concrete rule above. Indeed, the $\langle\rangle_{\mathsf{mem}}$ cell can only be within an $\langle\rangle_{\mathsf{agent}}$ cell and the $\langle\rangle_{\mathsf{k}}$ cell inside the declared $\langle\rangle_{\mathsf{agent}}$ cell can only be inside an intermediate $\langle\rangle_{\mathsf{thread}}$ cell, which must reside in a $\langle\rangle_{\mathsf{control}}$ cell. Context transformation applies at all levels in the rule context.

The Cell-Cloning Principle

There are $\mathbb{K}$ rules in which one wants to refer to two or more cells having the *same label*. An artificial example was shown above, where more than one $\langle\rangle_{\mathsf{agent}}$ cell was needed. A more natural rule involving two cells with the same label would be one for thread communication or synchronization, in which the two threads are directly involved in the said action. For example, consider adding a rendezvous synchronization mechanism to Imp++ whose intended semantics is the following: a thread whose next computational task is a rendezvous barrier statement "$\texttt{rendezvous}\,v$" blocks until another thread also reaches an identical "$\texttt{rendezvous}\,v$" statement, and, in that case, both threads unblock and continue their execution. The following $\mathbb{K}$ rule captures this desired behavior of rendezvous synchronization:

$$\langle \underset{\texttt{skip}}{\underline{\texttt{rendezvous}\,v}}\ \cdots\rangle_{\mathsf{k}}\ \langle \underset{\texttt{skip}}{\underline{\texttt{rendezvous}\,v}}\ \cdots\rangle_{\mathsf{k}}$$

Since this $\mathbb{K}$ rule captures the essence of the intended rendezvous synchronization, we would like to reuse it unchanged in language definitions which are more complex than Imp++, such as the Agent language in Section 4.3. Unfortunately, this rule will never match/apply as is on Agent configurations, because two $\langle\rangle_{\mathsf{k}}$ cells can never appear next to each other. A context transformation operation is therefore necessary, but it is not immediately clear how the rule context should be changed. The *cell-cloning principle* applies when abstract rules refer to two or more cells with the same name, and it states that context transformation should be consistent with the cell cloning, or multiplicity, information provided as part of the configuration definition; this can be done using starred labels, as explained in Section 4.2.1. Note that, for example, the Agent configuration in Figure 4.4 declares both the agent and the thread cells clonable. Thus, using the cell-cloning principle in combination with the locality principle, the abstract rule above can be transformed into the following Agent-concrete rule:

$$\langle\cdots\ \langle \underset{\texttt{skip}}{\underline{\texttt{rendezvous}\,v}}\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{thread}}\ \langle\cdots\ \langle \underset{\texttt{skip}}{\underline{\texttt{rendezvous}\,v}}\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{thread}}$$

The cell-cloning principle can therefore only be applied when one defines a configuration for one's language and, moreover, when one also provides the desired cell-cloning information (by means of starred labels). However, in our experience with defining languages in $\mathbb{K}$, it is actually quite useful to spend the time and add the cell-cloning information to one's configuration; one not only gets the convenience and modularity that comes with context transformation for free, but also a better insight on how one's language configurations look when programs are executed and thus, implicitly, a better understanding of one's language semantics.

Context Transformation with Default Values.

When creating a new agent, or spawning a new thread, a new cell for that agent/thread must be created, together with all the structure of nested cells below it. Moreover, all leaf cells need to be initialized with appropriate values. However, while the structure of the newly created cell can be quite complex, only a few cells are initialized with non-default values. Also, having to specify the entire cell makes the rule depend on a particular configuration, and thus non-modular: if the designer decides to add a new cell or to re-organize existing cells, the rule would have to be updated. To address all these issues, we allow partially specified configurations to appear in the replacement part of a $\mathbb{K}$ rule, and context-transform them by adding all missing cells and initializing the leaf ones with their default values. The default values can either be given for each sort or for each cell once and for all; in our K-Maude tool presented in Chapter 7, for example, we place them inside each cell when we define the configuration. The rule of agent creation can then be written as ("_" variables are used in the replacement part to indicate that it must be context-transformed):

$$\left(\begin{array}{l} \langle\cdots\ \langle \frac{\texttt{newAgent}\ S}{N_2}\ \cdots\rangle_{\mathsf{k}}\ \langle N_1\rangle_{\mathsf{me}}\ \cdots\rangle_{\mathsf{agent}}\ \langle \frac{N_2}{N_2 +_{Int} 1}\rangle_{\mathsf{nextAgent}} \\ \qquad \frac{\cdot}{\langle\cdots\ \langle S\rangle_{\mathsf{k}} \langle N_2\rangle_{\mathsf{me}}\ \langle N_1\rangle_{\mathsf{parent}}\ \cdots\rangle_{\mathsf{agent}}}\ \langle\cdots\ \frac{\cdot}{N_2}\ \cdots\rangle_{\mathsf{world}} \end{array}\right)$$

Upon applying the context transformation, the $\langle\rangle_{\mathsf{agent}}$ cell is completed to:

$$\left\langle \begin{array}{c} \langle\langle\langle S\rangle_{\mathsf{k}}\ \langle\cdot\rangle_{\mathsf{holds}}\ \rangle_{\mathsf{thread}}\rangle_{\mathsf{control}}\ \langle\cdot\rangle_{\mathsf{mem}}\ \langle 0\rangle_{\mathsf{nextLoc}} \\ \langle\cdot\rangle_{\mathsf{busy}}\ \langle N_2\rangle_{\mathsf{me}}\ \langle N_1\rangle_{\mathsf{parent}} \end{array} \right\rangle_{\mathsf{agent}}$$

Next section shows the $\mathbb{K}$ technique at work by defining Agent, a highly non-trivial concurrent programming language.

4.3 The Agent Language

This section gives the $\mathbb{K}$ semantic definition of Agent, a non-trivial experimental programming language aimed at challenging existing or future language definitional frameworks. All language constructs of Agent can be found in existing languages, possibly in a more general form and possibly using a different syntax. None of Agent's features are artificially crafted and the language, as a whole, is plausible and powerful.

The rationale for choosing features that already exist in popular languages is to avoid criticism, from proponents of frameworks that cannot handle the proposed features, that those features are useless, and thus do not deserve attention. Our standpoint is that the very fact that a feature exists in a mainstream language is sufficient evidence that the feature is useful, so an ideal semantic framework should unarguably support it. While language designers may chose not to include such a feature in their languages, a language-design framework designer aiming at an ideal framework has no choice and should support it. Agent's features, and particularly their combination, have been chosen to capture *the essence* of a certain category of conventional language concepts; if a semantic framework cannot handle a certain Agent feature then that framework cannot handle a potentially wide range of desirable features.

The language starts with arithmetic expressions, then gradually grows in complexity by adding: functions; recursion; call with current continuation and abrupt termination; statements; input/output; multithreading with shared memory and synchronization; and agents with synchronous and asynchronous communication, and with broadcasting and barriers. To stress the independence among language features, as well as the modularity induced by the $\mathbb{K}$ framework, we introduce each feature in its separate module, as they were developed in the K-Maude tool. The modules are displayed using the LaTeX code produced by the tool, but massaging pagination to avoid wasting space. Being solely interested in defining the semantics here, and not in parsing, we use a single syntactic category, K to stand for the AST.

4.3.1 Generic Expression Features

Expressions. The arithmetic and boolean expressions used in Agent are defined in a module EXP similarly to the expressions used in IMP, except that they all share the same syntactic category K. EXP includes integers and booleans as primary values, and allows arithmetic expressions built from them using operators like addition, multiplication and division, as well as comparison operators and logical connectives. All operators are strict and act on values as their corresponding builtin operators.

Module EXP

imports K+PL-INT

$\mathit{KResult} ::= \mathit{Bool} \mid \mathit{Int}$

$K ::= K \texttt{ + } K$ [strict]

$\mid K \texttt{ * } K$ [strict]

$\mid K \texttt{ / } K$ [strict]

$\mid K \leq K$ [seqstrict]

$\mid K \texttt{ == } K$ [strict]

$\mid \texttt{not } K$ [strict]

$\mid K \texttt{ and } K$ [strict(1)]

K rules:

$I_1 \texttt{ + } I_2 \Rightarrow I_1 \texttt{ +}_{Int} I_2$

$I_1 \texttt{ * } I_2 \Rightarrow I_1 \texttt{ *}_{Int} I_2$

$I_1 \texttt{ / } I_2 \Rightarrow I_1 \texttt{ /}_{Int} I_2$

when $I_2 \texttt{ =/=}_{Bool} \mathsf{0}$

$I_1 \leq I_2 \Rightarrow I_1 \leq_{Int} I_2$

$V_1 \texttt{ == } V_2 \Rightarrow V_1 \texttt{ ==}_{Bool} V_2$

$\texttt{not } T \Rightarrow \texttt{not}_{Bool} T$

$\mathsf{true\ and}\ E \Rightarrow E$

$\mathsf{false\ and}\ E \Rightarrow \mathsf{false}$

end module

Conditional. The conditional construct is strict in its first argument, expecting it to be evaluated to a boolean value, and then chooses one of the branches based on that value.

MODULE IF
  IMPORTS K+PL-BOOL
  $KResult$ ::= $Bool$
  K ::= `if` K `then` K `else` K [strict(1)]

  K RULES:
    `if` true `then` E `else` _ $\Rightarrow$ E
    `if` false `then` _ `else` E $\Rightarrow$ E
END MODULE

4.3.2 Functional Features

Call-by-value λ-calculus. λ-abstraction and the (free) variables are results, and the application construct is strict in both arguments (call-by-value). β-reduction is only applied at the top of the computation, to inhibit reductions inside λ-abstractions:

MODULE LAMBDA
  IMPORTS SUBSTITUTION
  K ::= $\lambda Id.K$
      | K K [strict]
  $KResult$ ::= Id | $\lambda X.E$

K RULES:

$$\langle \underset{E\ [\ E'\ /\ X\ ]}{\underline{\lambda X.E\ \ E'}} \cdots\rangle_{\mathsf{k}}$$

END MODULE

No initial configuration is necessary in the above because the cell $\langle\rangle_{\mathsf{k}}$ is already defined in the module K, which is imported by SUBSTITUTION.

Fixpoint recursion. AGENT allows recursion through the standard fix-point constructor '$\mu\_.\_$', whose semantics is given as a by-need unrolling when it reaches the top of computation.

MODULE MU
  IMPORTS SUBSTITUTION
  K ::= $\mu Id.K$

K RULES:

$$\langle \underset{E\ [\ \mu X.E\ /\ X\ ]}{\underline{\mu X.E}} \cdots\rangle_{\mathsf{k}}$$

END MODULE

4.3.3 Control Features

Call with current continuation. The CALLCC module extends module LAMBDA (as it relies on the application construct) with a strict '`callcc_`' construct. The argument of callcc is expected to evaluate to a function to which the current continuation is passed as a value wrapped by '$cc(\_)$'. If that value becomes the first argument of an application construct, then second argument of the application is passed to the original continuation.

Module CALLCC

imports LAMBDA

$K ::= \texttt{callcc}\ K$ [strict]

$KResult ::= cc(K)$

K rules:

$$\langle \underset{V\ \ cc(K)}{\underline{\texttt{callcc}\ V}} \curvearrowright K \rangle_{\mathsf{k}}$$

$$\langle \underset{V \curvearrowright K}{\underline{cc(K)\ \ V \curvearrowright \_}} \rangle_{\mathsf{k}}$$

end module

Abrupt termination. Although callcc allows the encoding of most control-intensive constructs, including halt, we include a definition of halt not depending on any of the existing features.

Module HALT

imports K

$K ::= \texttt{halt}\ K$ [strict]

K rules:

$$\langle \underset{V}{\underline{\texttt{halt}\ V \curvearrowright \_}} \rangle_{\mathsf{k}}$$

end module

4.3.4 Imperative Features

Sequential composition. Sequential composition is achieved through the construct '_;_' and its unit value '`skip`'. The semantics of '_;_' is that it evaluates the first argument, and then it discards its value, leaving the second argument to be evaluated. Note that we allow the first argument to evaluate to any value, and thus allow any expression to be used as a statement.

Module SEQ

imports K

$K ::= K\ ;\ K$ [strict(1)]

$KResult ::= \texttt{skip}$

K rules:

$V\ ;\ S \rightharpoonup S$

end module

Input/Output. As input and output transcend the computation structure, we need to specify the structure of the configuration required to give semantics to these constructs. The '`read`' expression evaluates to the first integer extracted from the list in the $\langle\rangle_{\mathsf{input}}$ cell, while the '`print_`' statement evaluates its argument, and then it appends it to the list in the $\langle\rangle_{\mathsf{output}}$ cell.

Module IO

imports PL-INT+SEQ

$K ::= \mathit{Int}$

$\quad \mid \texttt{read}$

$\quad \mid \texttt{print}\ K$ [strict]

initial configuration:

$$\left\langle \cdot K \right\rangle_{\mathsf{k}} \quad \left\langle \cdot \mathit{List} \right\rangle_{\mathsf{in}} \quad \left\langle \cdot \mathit{List} \right\rangle_{\mathsf{out}}$$

K rules:

$$\langle \underset{I}{\underline{\texttt{read}}}\ \cdots\rangle_{\mathsf{k}}\ \langle \underset{\cdot}{\underline{I}}\ \cdots\rangle_{\mathsf{in}}$$

$$\langle \underset{\texttt{skip}}{\underline{\texttt{print}\ V}}\ \cdots\rangle_{\mathsf{k}}\ \langle \cdots\ \underset{V}{\underline{\cdot}}\ \rangle_{\mathsf{out}}$$

end module

Loop statement. Although loops could be easily simulated using recursion, we here give it a direct (imperative) semantics, by unrolling it when on-top of the computation.

Module WHILE

imports IF+SEQ

$K ::= \texttt{while}\ K\ \texttt{do}\ K$

K rules:

$$\langle \underset{\texttt{if}\ E\ \texttt{then}\ S\ \texttt{;}\ \texttt{while}\ E\ \texttt{do}\ S\ \texttt{else}\ \texttt{skip}}{\underline{\texttt{while}\ E\ \texttt{do}\ S}}\ \cdots\rangle_{\mathsf{k}}$$

end module

Memory. To be able to write meaningful sequential programs we introduce memory. However, to keep in tone with our functional flavor, we prefer references instead of variable declarations. For that, we require that the configuration contains a $\langle\rangle_{\mathsf{mem}}$ cell containing a map (initially empty) of locations (naturals) to values, and a $\langle\rangle_{\mathsf{nextLoc}}$ cell containing the next available location (initially 0). The ‘ref_’ expression evaluates its argument, and then it allocates it in the memory at the next available location, evaluating itself to that location. ‘*_’ evaluates its argument and then dereferences it to the value it maps to in the memory. ‘_:=_’ expects a dereferencing expression as its first argument, and updates the value in the memory mapped to by it with the value of the second argument. The evaluation of the location in the first argument is ensured through a context declaration.

Module REF

imports SEQ

$K ::= \mathit{Nat}$

$\quad \mid \texttt{ref}\ K$ [strict]

$\quad \mid \texttt{*}\ K$ [strict]

$\quad \mid K\ \texttt{:=}\ K$ [strict(2)]

initial configuration:

$\left\langle \cdot K \right\rangle_{\mathsf{k}} \; \left\langle \cdot \mathit{Map} \right\rangle_{\mathsf{mem}} \; \left\langle 0 \right\rangle_{\mathsf{nextLoc}}$

K RULES:

CONTEXT: $\texttt{*}\,\square\ \texttt{:=}\ \_$

$\langle \underset{N}{\underline{\texttt{ref}\ V}}\ \cdots\rangle_{\mathsf{k}}\ \langle\cdots\ \underset{N \mapsto V}{\underline{\quad\cdot\quad}}\ \cdots\rangle_{\mathsf{mem}}\ \langle \underset{\mathsf{s}_{\mathit{Nat}}\ N}{\underline{\quad N\quad}} \rangle_{\mathsf{nextLoc}}$

$\langle \underset{V}{\underline{\texttt{*}\ N}}\ \cdots\rangle_{\mathsf{k}}\ \langle\cdots\ N \mapsto V\ \cdots\rangle_{\mathsf{mem}}$

$\langle \underset{\texttt{skip}}{\underline{\texttt{*}\ N\ \texttt{:=}\ V}}\ \cdots\rangle_{\mathsf{k}}\ \langle\cdots\ N \mapsto \underset{V}{\underline{\ \_\ }}\ \cdots\rangle_{\mathsf{mem}}$

END MODULE

4.3.5 Multithreading Features

Let us now define a generic multi-threading module. Note that this module is independent of the previous ones; the only module required is SEQ, as we want our multi-threading statements to evaluate to '`skip`'. As we need to allow multiple execution threads synchronized using locks, our multi-threading minimal configuration is written to reflect that: the $\langle\rangle_{\mathsf{thread}}$ cell has multiplicity "zero-or-more" (indicated by the $*$ suffix) and must contain a computation $\langle\rangle_{\mathsf{k}}$ cell and a $\langle\rangle_{\mathsf{holds}}$ cell to account for the locks held; besides the multiple threads, the system must also contain a $\langle\rangle_{\mathsf{busy}}$ cell containing the acquired but not yet released locks. The semantics of locks is that any value can act as a lock, and that locks are re-entrant (this is why the $\langle\rangle_{\mathsf{holds}}$ cell contains a map from values to naturals). The '`spawn_`' statement creates a new thread containing the given argument as its initial computation and having default initial values for other potential cells within thread (e.g., the $\langle\rangle_{\mathsf{holds}}$ cell is initialized with the empty map, as specified by the configuration). When the computation of a thread is reduced to a value, the thread is dissolved and its resources are freed. Lock acquire is defined through two rules, depending whether the lock is already held by the thread (in which case its counter is increased), or it is available (in which case it is added to the $\langle\rangle_{\mathsf{busy}}$ cell and mapped to 0 in the $\langle\rangle_{\mathsf{holds}}$ cell). The rules for lock release mirror those for lock acquiring. Finally, '`rendezvous_`' evaluates its argument and then blocks until another thread requires a rendez-vous on the same value; then, the two threads advance together.

MODULE THREADS

IMPORTS PL-NAT+SEQ

K ::= Nat

| `spawn` K

| `acquire` K [strict]

| `release` K [strict]

| `rendezvous` K [strict]

INITIAL CONFIGURATION:

$$\left\langle \left\langle \cdot K \right\rangle_{\mathsf{k}} \; \left\langle \cdot Map \right\rangle_{\mathsf{holds}} \right\rangle_{\mathsf{thread*}} \; \left\langle \cdot Set \right\rangle_{\mathsf{busy}}$$

K RULES:

$$\langle \cdots \; \langle \frac{\texttt{spawn}\ S}{\texttt{skip}} \; \cdots \rangle_{\mathsf{k}} \; \cdots \rangle_{\mathsf{thread}} \quad \frac{\cdot}{\langle \cdots \; \langle S \rangle_{\mathsf{k}} \; \cdots \rangle_{\mathsf{thread}}}$$

$$\frac{\langle \cdots \; \langle V \rangle_{\mathsf{k}} \; \langle Holds \rangle_{\mathsf{holds}} \; \cdots \rangle_{\mathsf{thread}}}{\cdot} \quad \langle \frac{Busy}{Busy \; \texttt{-}_{Set} \; \texttt{keys}\ Holds} \rangle_{\mathsf{busy}}$$

$$\langle \frac{\texttt{acquire}\ V}{\texttt{skip}} \; \cdots \rangle_{\mathsf{k}} \; \langle \cdots \; \frac{\cdot}{V \mapsto 0} \; \cdots \rangle_{\mathsf{holds}} \; \langle Busy \; \frac{\cdot}{V} \rangle_{\mathsf{busy}}$$

$$\text{when } \texttt{not}_{Bool}\ V \ \texttt{in}\ Busy$$

$$\langle \frac{\texttt{acquire}\ V}{\texttt{skip}} \; \cdots \rangle_{\mathsf{k}} \; \langle \cdots \; V \mapsto \frac{N}{\texttt{s}_{Nat}\ N} \; \cdots \rangle_{\mathsf{holds}}$$

$$\langle \frac{\texttt{release}\ V}{\texttt{skip}} \; \cdots \rangle_{\mathsf{k}} \; \langle \cdots \; V \mapsto \frac{\texttt{s}_{Nat}\ N}{N} \; \cdots \rangle_{\mathsf{holds}}$$

$$\langle \frac{\texttt{release}\ V}{\texttt{skip}} \; \cdots \rangle_{\mathsf{k}} \; \langle \cdots \; \frac{V \mapsto 0}{\cdot} \; \cdots \rangle_{\mathsf{holds}} \; \langle \cdots \; \frac{V}{\cdot} \; \cdots \rangle_{\mathsf{busy}}$$

$$\langle \frac{\texttt{rendezvous}\ V}{\texttt{skip}} \; \cdots \rangle_{\mathsf{k}} \; \langle \frac{\texttt{rendezvous}\ V}{\texttt{skip}} \; \cdots \rangle_{\mathsf{k}}$$

END MODULE

4.3.6 Communicating Agents

The AGENTS module allows dynamic creation and termination of agents. Agents are self-aware and creator-aware, and communicate by means of asynchronous and synchronous message-send commands, by targeted and non-targeted receive expressions, and by broadcasting and global barriers. Given the complexity of interaction we want to achieve, the configuration required to give their semantics must contain several new cells. First, each agent is contained in an $\langle\rangle_{\mathsf{agent}}$ cell, and contains at least a $\langle\rangle_{\mathsf{control}}$ cell with a $\langle\rangle_{\mathsf{k}}$ cell within it, and $\langle\rangle_{\mathsf{me}}$ and $\langle\rangle_{\mathsf{parent}}$ cells providing identification information. The reason for the $\langle\rangle_{\mathsf{control}}$ cell is that we want to allow the control mechanism of a thread to be more complex, while still being able to completely stop an agent if needed. Additional cells are: $\langle\rangle_{\mathsf{nextAgent}}$—a counter for the next available agent, $\langle\rangle_{\mathsf{world}}$—a set containing the agents currently in the system, $\langle\rangle_{\mathsf{barrier}}$ and $\langle\rangle_{\mathsf{waiting}}$—containing the current status of the global barrier, and the agents waiting at it, respectively, and $\langle\rangle_{\mathsf{msgs}}$ which contains a bag of messages, each wrapped into a $\langle\rangle_{\mathsf{msg}}$ cell and containing cells describing the sender ($\langle\rangle_{\mathsf{from}}$), the set of recipients ($\langle\rangle_{\mathsf{to}}$), and the message itself ($\langle\rangle_{\mathsf{body}}$). Since this module introduces substantially more language constructs than the preceding ones, we next explain the rules inline within the definition.

MODULE AGENTS

IMPORTS PL-INT+SEQ

$$\begin{array}{rl} K ::= & \mathit{Int} \mid \mathit{Bool} \\ & \mid \texttt{newAgent}\ K \quad \mid \texttt{haltAgent} \\ & \mid \texttt{me} \quad \mid \texttt{parent} \\ & \mid \texttt{receive} \quad \mid \texttt{receiveFrom}\ K\ \text{[strict]} \\ & \mid \texttt{send}\ K\ \texttt{to}\ K\ \text{[strict]} \quad \mid \texttt{sendSynch}\ K\ \texttt{to}\ K\ \text{[strict]} \\ & \mid \texttt{barrier} \quad \mid \texttt{broadcast}\ K\ \text{[strict]} \end{array}$$

INITIAL CONFIGURATION:

$$\left\langle \begin{array}{c} \left\langle \langle 0\rangle_{\mathsf{me}}\ \langle \texttt{-Int 1}\rangle_{\mathsf{parent}}\ \langle\langle \cdot K\rangle_{\mathsf{k}}\rangle_{\mathsf{control}} \right\rangle_{\mathsf{agent}*} \\ \langle 1\rangle_{\mathsf{nextAgent}}\ \langle 0\rangle_{\mathsf{world}}\ \langle \mathsf{true}\rangle_{\mathsf{barrier}}\ \langle \cdot \mathit{Set}\rangle_{\mathsf{waiting}} \\ \left\langle \left\langle \langle \cdot K\rangle_{\mathsf{from}}\ \langle \cdot \mathit{Set}\rangle_{\mathsf{to}}\ \langle \cdot K\rangle_{\mathsf{body}} \right\rangle_{\mathsf{message}*} \right\rangle_{\mathsf{messages}} \end{array} \right\rangle$$

The '`newAgent_`' expression creates a new agent, setting as its non-default values the computation cell (initialized with the given argument), the $\langle\rangle_{\mathsf{me}}$ cell (initialized as the next available agent id, which is incremented), and the $\langle\rangle_{\mathsf{parent}}$ cell (initialized as the id of the creating agent); additionally, the new agent id is registered in $\langle\rangle_{\mathsf{world}}$ and is returned as the value of '`newAgent_`'.

$$\left(\begin{array}{ll} \langle \cdots\ \langle \underset{N_2}{\underline{\texttt{newAgent}\ S}}\ \cdots\rangle_{\mathsf{k}}\ \langle N_1\rangle_{\mathsf{me}}\ \cdots\rangle_{\mathsf{agent}} & \langle\cdots\ \underset{N_2}{\underline{\ \cdot\ }}\ \cdots\rangle_{\mathsf{world}} \\ \dfrac{\cdot}{\langle\cdots\ \langle N_2\rangle_{\mathsf{me}}\ \langle N_1\rangle_{\mathsf{parent}}\ \langle S\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{agent}}} & \langle \underset{\mathsf{s}_{\mathit{Nat}}\ N_2}{\underline{\ N_2\ }} \rangle_{\mathsf{nextAgent}} \end{array}\right)$$

When the control of an agent becomes empty, the agent is dissolved and unregistered from the $\langle\rangle_{\mathsf{world}}$ cell; hence '`haltAgent`' only needs to empty the contents of the $\langle\rangle_{\mathsf{control}}$ cell.

$$\underset{\cdot}{\underline{\langle\cdots\ \langle\cdot\rangle_{\mathsf{control}}\ \langle N\rangle_{\mathsf{me}}\ \cdots\rangle_{\mathsf{agent}}}}\quad \langle\cdots\ \underset{\cdot}{\underline{N}}\ \cdots\rangle_{\mathsf{world}}$$

$$\langle\cdots\ \langle \texttt{haltAgent}\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{control}} \Rightarrow \langle\cdot\rangle_{\mathsf{control}}$$

'`me`' and '`parent`' have the straightforward semantics, yielding the contents of the $\langle\rangle_{\mathsf{me}}$ and $\langle\rangle_{\mathsf{parent}}$ cells of the enclosing agent:

$$\langle \underset{N}{\underline{\texttt{me}}}\ \cdots\rangle_{\mathsf{k}}\ \langle N\rangle_{\mathsf{me}} \qquad\qquad \langle \underset{N}{\underline{\texttt{parent}}}\ \cdots\rangle_{\mathsf{k}}\ \langle N\rangle_{\mathsf{parent}}$$

An agent can send any results (including agent ids) to other agents, provided it knows their identity. To model asynchronous communication, $\langle\rangle_{\mathsf{msg}}$ cells hold the sender of the message, the intended set of receivers, and a message body:

$$\langle N_1\rangle_{\mathsf{me}}\ \langle \underset{\texttt{skip}}{\underline{\texttt{send}\ V\ \texttt{to}\ N_2}}\ \cdots\rangle_{\mathsf{k}}\quad \dfrac{\cdot}{\langle\langle N_1\rangle_{\mathsf{from}}\ \langle N_2\rangle_{\mathsf{to}}\ \langle V\rangle_{\mathsf{body}}\rangle_{\mathsf{message}}}$$

An agent can request to receive a message from a certain agent, or from any agent, waiting until that happens. Upon receiving, the agent's id is removed from the $\langle\rangle_{\mathsf{to}}$ cell of the message:

$$\langle N\rangle_{\mathsf{me}}\ \langle \underset{V}{\underline{\texttt{receive}}}\ \cdots\rangle_{\mathsf{k}}\ \langle\cdots\ \langle\cdots\ \underset{\cdot}{\underline{N}}\ \cdots\rangle_{\mathsf{to}}\ \langle V\rangle_{\mathsf{body}}\ \cdots\rangle_{\mathsf{message}}$$

$$\langle\langle N_2\rangle_{\mathsf{from}}\ \langle\cdots\ \underset{\cdot}{\underline{N_1}}\ \cdots\rangle_{\mathsf{to}}\ \langle V\rangle_{\mathsf{body}}\rangle_{\mathsf{message}}\ \langle N_1\rangle_{\mathsf{me}}\ \langle\underset{V}{\underline{\mathtt{receiveFrom}\ N_2}}\ \cdots\rangle_{\mathsf{k}}$$

A message can also be broadcast to all agents in the $\langle\rangle_{\mathsf{world}}$ cell:

$$\langle N\rangle_{\mathsf{me}}\ \langle\underset{\mathtt{skip}}{\underline{\mathtt{broadcast}\ V}}\ \cdots\rangle_{\mathsf{k}}\ \langle W\rangle_{\mathsf{world}}\ \ \underset{\langle\langle N\rangle_{\mathsf{from}}\ \langle W\rangle_{\mathsf{to}}\ \langle V\rangle_{\mathsf{body}}\rangle_{\mathsf{message}}}{\underline{\qquad\qquad\qquad\cdot\qquad\qquad\qquad}}$$

Once a message has no receivers, it can be removed:

$$\langle\cdots\ \langle\cdot\rangle_{\mathsf{to}}\ \cdots\rangle_{\mathsf{message}}\ \rightharpoonup\ \cdot$$

Agents can also communicate synchronously if the sender chooses so, in which case the sender and the receiver need to be matched together for the exchange to occur:

$$\left(\begin{array}{c}\langle\cdots\ \langle N_1\rangle_{\mathsf{me}}\ \langle\underset{\mathtt{skip}}{\underline{\mathtt{sendSynch}\ V\ \mathtt{to}\ N_2}}\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{agent}}\\ \langle\cdots\ \langle N_2\rangle_{\mathsf{me}}\ \langle\underset{V}{\underline{\mathtt{receiveFrom}\ N_1}}\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{agent}}\end{array}\right)$$

$$\langle\underset{\mathtt{skip}}{\underline{\mathtt{sendSynch}\ V\ \mathtt{to}\ N_2}}\ \cdots\rangle_{\mathsf{k}}\ \langle\cdots\ \langle N_2\rangle_{\mathsf{me}}\ \langle\underset{V}{\underline{\mathtt{receive}}}\ \cdots\rangle_{\mathsf{k}}\ \cdots\rangle_{\mathsf{agent}}$$

AGENT supports global synchronization by means of barriers. When an agent reaches a barrier and the barrier is on, the agent adds itself to the $\langle\rangle_{\mathsf{waiting}}$ cell:

$$\langle N\rangle_{\mathsf{me}}\ \langle\mathtt{barrier}\ \cdots\rangle_{\mathsf{k}}\ \langle\mathsf{true}\rangle_{\mathsf{barrier}}\ \langle W\ \ \underset{N}{\underline{\cdot}}\rangle_{\mathsf{waiting}}$$

When all agents in $\langle\rangle_{\mathsf{world}}$ are waiting, the barrier is lifted:

$$\langle\underset{\mathsf{false}}{\underline{\mathsf{true}}}\rangle_{\mathsf{barrier}}\ \langle W\rangle_{\mathsf{waiting}}\ \ \langle W\rangle_{\mathsf{world}}$$

$$\text{when } W\ \mathtt{=/=}_{Bool}\ \cdot$$

Then, agents unregister from the $\langle\rangle_{\mathsf{waiting}}$ cell and proceed:

$$\langle N\rangle_{\mathsf{me}}\ \langle\underset{\mathtt{skip}}{\underline{\mathtt{barrier}}}\ \cdots\rangle_{\mathsf{k}}\ \langle\mathsf{false}\rangle_{\mathsf{barrier}}\ \langle\cdots\ \underset{\cdot}{\underline{N}}\ \cdots\rangle_{\mathsf{waiting}}$$

Once all the agents proceeded, the barrier is reseted to true:

$$\langle\underset{\mathsf{true}}{\underline{\mathsf{false}}}\rangle_{\mathsf{barrier}}\ \ \langle\cdot\rangle_{\mathsf{waiting}}$$

END MODULE

The existence of a read-only pattern on which rule instances can overlap, allows for example, that two agents send or receive messages simultaneously, while accessing the same $\langle\rangle_{\mathsf{msgs}}$ cell, including the case of multiple agents simultaneously reading the same broadcast message and removing themselves from the receivers set.

4.3.7 Reflective Features

In this section we identify a common pattern in defining reflective features, that of an AST visitor, and we show how it can be instantiated for both code generation and for the generic substitution module used in the AGENT definition.

Generic AST Visitor. The AST visitor pattern defined below provides a mechanism which allows the visiting of a computation structure while looking for nodes containing specified labels and applying other labels to the subtrees rooted in them, which potentially implies (or not) additional processing done to those subtrees. To do that, "boxed constructors" are created for the constructors of the computation (double comma, all labels, curved arrow), to stand for visited parts of the computation. In addition to maintaining the visited status, these constructors are strict, with the purpose of bringing their subcomputations out for visiting.

MODULE K-VISITOR

IMPORTS K-WRAPPERS+K

$K ::= \text{apply } \mathit{KLabel} \text{ to } \mathit{Set} \text{ in } \mathit{List}\{K\}$

$\quad \mid K \boxed{\curvearrowright} K \;\; [\text{strict}]$

$\quad \mid K \boxed{,,} K \;\; [\text{strict}]$

$\quad \mid \mathit{unbox}(K) \;\; [\text{strict}]$

$\mathit{KResult} ::= \boxed{\mathit{List}\{K\}}$

$\mathit{KLabel} ::= \boxed{\mathit{KLabel}}$

K RULES:

$\text{apply } A \text{ to } \mathit{Labels}\;\; \mathit{Label} \text{ in } \mathit{Label}\,(\,\mathit{Kl}\,) \rightharpoonup A\,(\,\mathit{Label}\,(\,\mathit{Kl}\,)\,)$

$\text{apply } A \text{ to } \mathit{Labels} \text{ in } \mathit{Label}\,(\,\mathit{Kl}\,) \rightharpoonup \boxed{\mathit{Label}}\,(\,\text{apply } A \text{ to } \mathit{Labels} \text{ in } \mathit{Kl}\,)$
$\qquad \text{when } \mathtt{not}_{\mathit{Bool}}\; \mathit{Label}\; \mathtt{in}\; \mathit{Labels}$

$\text{apply } A \text{ to } \mathit{Labels} \text{ in } \cdot \rightharpoonup \boxed{\cdot}$

$\text{apply } A \text{ to } \mathit{Labels} \text{ in } K_1 \curvearrowright K_2 \rightharpoonup \text{apply } A \text{ to } \mathit{Labels} \text{ in } K_1 \boxed{\curvearrowright}$
$\qquad \text{apply } A \text{ to } \mathit{Labels} \text{ in } K_2$
$\qquad \text{when } K_1 \;\mathtt{=/=}_{\mathit{Bool}}\; \cdot \;\mathtt{and}_{\mathit{Bool}}\; K_2 \;\mathtt{=/=}_{\mathit{Bool}}\; \cdot$

$\text{apply } A \text{ to } \mathit{Labels} \text{ in } \cdot\mathit{List}\{K\} \rightharpoonup \boxed{\cdot\mathit{List}\{K\}}$

$\text{apply } A \text{ to } \mathit{Labels} \text{ in } K_1 \,,,\, \mathit{NeKl} \rightharpoonup \text{apply } A \text{ to } \mathit{Labels} \text{ in } K_1 \boxed{,,}$
$\qquad \text{apply } A \text{ to } \mathit{Labels} \text{ in } \mathit{NeKl}$

CONTEXT: $\boxed{\mathit{Label}}\,(\,\_ \,,,\, \square \,,,\, \_\,)$

$\mathit{unbox}(\boxed{K}) \rightharpoonup K$

$\boxed{\mathit{Label}}\,(\,\boxed{\mathit{Kl}}\,) \rightharpoonup \boxed{\mathit{Label}\,(\,\mathit{Kl}\,)}$

$\boxed{\mathit{Kl}} \boxed{,,} \boxed{\mathit{Kl}'} \rightharpoonup \boxed{\mathit{Kl} \,,,\, \mathit{Kl}'}$

$\boxed{K_1} \boxed{\curvearrowright} \boxed{K_2} \rightharpoonup \boxed{K_1 \curvearrowright K_2}$

END MODULE

Generic substitution. The module below defines a generic binding construct "$x.E$" which specifies that variable x is bound in expression E, and thus does not commit to a single binding operations, allowing it to be used in combination with any binding operation such as λ and μ. This binding construct is accompanied by a substitution construct which is aware of it, in the sense of avoiding variable capturing during substitution.

To define the substitution, we use the AST visitor pattern defined above to visit the term in which the substitution needs to be made, looking for either the variable to be replaced, or for the binding construct, and applying to them a label which encodes the substitution. If the variable to be replaced is reached, then it is substituted by the "boxed" version of the term to replace it. If the binding construct is reached, then first the bound variable is substituted by a fresh variable, to avoid variable capturing, then the original substitution is applied recursively on the substituted term.

Module SUBSTITUTION

imports K-VISITOR+PL-INT+PL-ID

$K \ ::= \ Id \mid Nat$

$\mid \ Id.K$

$\mid \ K \ \texttt{[} \ K \ \texttt{/} \ Id \ \texttt{]}$

$\mid \ K[\boxed{K}/Id]$

$\mid \ eval(K)[\boxed{K}/Id]$ [strict(1)]

$KLabel \ ::= \ [\boxed{K}/Id]$

$Id \ ::= \ id_{Nat}$

initial configuration:

$\langle \cdot K \rangle_{\mathsf{k}} \quad \langle 0 \rangle_{\mathsf{nextId}}$

K rules:

$$\langle \underset{unbox(K'[\boxed{K}/Y])}{\underline{K' \ \texttt{[} \ K \ \texttt{/} \ Y \ \texttt{]}}} \ \cdots \rangle_{\mathsf{k}}$$

macro: $K'[\boxed{K}/Y] = \text{apply } [\boxed{K}/Y] \text{ to } \_.\_ \ getKLabel(Y) \text{ in } K'$

$$[\boxed{K}/Y] \ (\ Y\ ) \ \rightharpoonup \ \boxed{K}$$

$$\langle \underset{\boxed{\_.\_} \ (\ \boxed{id_N}\boxed{,,}eval(K'[\boxed{id_N}/X])[\boxed{K}/Y]\ )}{\underline{[\boxed{K}/Y] \ (\ X.K'\ )}} \ \cdots \rangle_{\mathsf{k}} \quad \langle \underset{\mathsf{s}_{Nat} \ N}{\underline{\ N\ }} \rangle_{\mathsf{nextId}}$$

$$eval(\boxed{K'})[\boxed{K}/Y] \ \rightharpoonup \ K'[\boxed{K}/Y]$$

end module

Code generation. Once the visitor pattern has been extracted, the code generation definition becomes relatively simple, focusing only on the relevant parts of the quote/unquote mechanism.

Module QUOTE-UNQUOTE

imports PL-NAT+K-VISITOR

$$
\begin{aligned}
K \ &::= \ \texttt{quote}\ K \\
&\quad \mid \ \texttt{unquote}\ K \\
&\quad \mid \ \texttt{lift}\ K \ \ [\text{strict}] \\
&\quad \mid \ \texttt{eval}\ K \ \ [\text{strict}] \\
&\quad \mid \ \mathit{quote}(K, \mathit{Nat}) \\
\mathit{KLabel} \ &::= \ \mathit{quote}_{\mathit{Nat}}
\end{aligned}
$$

K RULES:

$$\langle \underset{\mathit{quote}(K,\mathsf{0})}{\underline{\ \texttt{quote}\ K\ }} \ \cdots\rangle_{\mathsf{k}}$$

MACRO: $\mathit{quote}(K, N) = $ apply quote_N to $\texttt{quote\_}$ $\texttt{unquote\_}$ in K

$$\mathit{quote}_N\ (\ \texttt{quote}\ K\ ) \Rightarrow \boxed{\texttt{quote\_}}\ (\ \mathit{quote}(K, \mathsf{s}_{\mathit{Nat}}\ N)\ )$$

$$\mathit{quote}_{\mathsf{0}}\ (\ \texttt{unquote}\ K\ ) \ \Rightarrow \ K$$

$$\mathit{quote}_{\mathsf{s}_{\mathit{Nat}}\ N}\ (\ \texttt{unquote}\ K\ ) \Rightarrow \boxed{\texttt{unquote\_}}\ (\ \mathit{quote}(K, N)\ )$$

$$\texttt{lift}\ V \ \Rightarrow \ \boxed{V}$$

$$\texttt{eval}\,\boxed{K} \ \Rightarrow \ K$$

END MODULE

There are two important aspects of code generation. One aspect is that code quoting and unquoting can be nested, and an unquoted code can be evaluated only if it is guarded by as many `unquote` constructs as `quote` ones; to achieve this we need to maintain a counter for the depth of quotation –we prefer to do it as part of a special K label. The other aspect is that the generated code may need to be eventually evaluated; however, this is very simple using our AST visitor pattern, as we can simply view code as being wrapped by the visited box.

Initially, a quoted expression starts at the quoting level 0, and uses the visitor pattern to visit all $\mathbb{K}$ constructors, looking for the `quote`/`unquote` labels, which need to increment/decrement the counter. If the quoting level is 0 and an `unquote` (K) expression is encountered, then K is released for evaluation; its evaluation will indeed take place, because the visited labels were all defined strict. Each computation rooted in a visited label eventually evaluates to a visited fragment; finally the resulting visited fragment is “unboxed” once every node has been visited.

4.3.8 Putting Them All Together

The main module of the language, named AGENT loads all modules defined so far, and defines a running configuration consistent with previous configurations (i.e., extending the transitive closure of the subcell relation).

MODULE AGENT
IMPORTS EXP+IF+LAMBDA+MU+CALLCC
IMPORTS HALT+SEQ+IO+REF+WHILE+THREADS+AGENTS
$\mathit{Bag} \ ::= \ \texttt{run(}\ K\ \texttt{,}\ \mathit{List}\ \texttt{)}$

MACRO: $\mathtt{run(}\ K\ \mathtt{,}\ L\ \mathtt{)} = \langle \cdots\ \langle K \rangle_{\mathsf{k}}\ \langle L \rangle_{\mathsf{in}}\ \cdots \rangle_{\mathsf{T}}$

INITIAL CONFIGURATION:

$$\left\langle \begin{array}{c} \left\langle \begin{array}{c} \left\langle \left\langle \left\langle \cdot K \right\rangle_{\mathsf{k}} \left\langle \mathsf{0} \right\rangle_{\mathsf{nextId}} \left\langle \cdot Map \right\rangle_{\mathsf{holds}} \right\rangle_{\mathsf{thread*}} \right\rangle_{\mathsf{control}} \left\langle \cdot Set \right\rangle_{\mathsf{busy}} \\ \left\langle \cdot Map \right\rangle_{\mathsf{mem}} \left\langle \mathsf{0} \right\rangle_{\mathsf{nextLoc}} \left\langle \mathsf{0} \right\rangle_{\mathsf{me}} \left\langle -_{Int}\ \mathsf{1} \right\rangle_{\mathsf{parent}} \end{array} \right\rangle_{\mathsf{agent*}} \\ \left\langle \mathsf{1} \right\rangle_{\mathsf{nextAgent}} \left\langle \mathsf{0} \right\rangle_{\mathsf{world}} \left\langle \mathsf{true} \right\rangle_{\mathsf{barrier}} \left\langle \cdot Set \right\rangle_{\mathsf{waiting}} \\ \left\langle \left\langle \left\langle \cdot K \right\rangle_{\mathsf{from}} \left\langle \cdot Set \right\rangle_{\mathsf{to}} \left\langle \cdot K \right\rangle_{\mathsf{body}} \right\rangle_{\mathsf{message*}} \right\rangle_{\mathsf{messages}} \\ \left\langle \left\langle \cdot List \right\rangle_{\mathsf{in}} \left\langle \cdot List \right\rangle_{\mathsf{out}} \right\rangle_{\mathsf{I/O}} \end{array} \right\rangle_{\mathsf{T}}$$

END MODULE

For example, the $\langle\rangle_{\mathsf{control}}$ cell of an agent contains here a pool of threads, each with its own $\langle\rangle_{\mathsf{k}}$ cell, instead of just one $\langle\rangle_{\mathsf{k}}$ cell. However, thanks to context abstraction provided by $\mathbb{K}$ (Section 4.2.5), this does not affect the applicability of the already defined rules.

Since this is the final module of the AGENT definition, we also define a '`run( _ , _ )`' macro, which given an initial computation and input list sets the contents of the $\langle\rangle_{\mathsf{k}}$ cell and the $\langle\rangle_{\mathsf{in}}$ cell with the provided values in the initial configuration of the system.

4.4 Discussion

We believe that the overview of the $\mathbb{K}$ framework presented in this chapter as well as the examples accompanying it bring enough evidence that the $\mathbb{K}$ is a by now a mature, versatile, and powerful framework for modularly defining (concurrent) programming languages. Nevertheless, there are some questions which are not fully addressed in this chapter. Among them, there are the following:

- What can we do with a language definition once we have managed to define it in $\mathbb{K}$?
- We have claimed that $\mathbb{K}$ definitions offer a high degree of concurrency; but how much concurrency is there really available for $\mathbb{K}$ definitions?
- What does it take to obtain an implementation of the $\mathbb{K}$ framework?

In the remainder of the dissertation we will try to give additional evidence on the usefulness of $\mathbb{K}$ definitions beyond their ability to express program languages semantics in a formal setting, we will describe the tool support available for writing and using $\mathbb{K}$ definitions, and present further development regarding the amount of concurrency which can be captured through the $\mathbb{K}$ framework.

Chapter 5

From Language Definitions to (Runtime) Analysis Tools

The rewriting logic representation of $\mathbb{K}$ definitions gives them access to the arsenal of generic tools for rewriting logic available through the Maude rewrite engine [34]: state space exploration, LTL model checking, inductive theorem proving, and so on. This collection of analysis tools is by itself enough to provide more information about the behaviors of a program than one would get by simply testing the program using an interpreter or a compiler for that language. Nevertheless, the effort of defining the semantics pays back in more than just one way: by relatively few alterations to the definition, one can use the same generic tools to obtain type checkers and type inferencers [52], static policy checking tools [81, 78], runtime verification tools [149], and even Hoare-like program verification tools [141].

The purpose of this chapter is to offer a glimpse on the process of turning definitions into testing and analysis tools. One instance of this process has already been presented in Section 4.1.3 where, by simply interpreting the dynamic semantics for IMP++ into an abstract domain, we obtained (relatively mechanically) a type checker for the language. In this chapter we argue that $\mathbb{K}$ definitions can be used to test and analyze the executions of programs written in real-life languages either directly or by transforming the definitions into runtime analysis tools.

To stress the "real-life" aspect, we choose as our running example a subset of the C programming language, named KERNELC, which is completely defined in Section 5.1. In Section 5.2 we show how this definition can be easily turned into a runtime verification tool for *strong* memory safety [149]. Section 5.3 presents a simple, sequentially consistent, definition for KERNELCC, an extension of KERNELC with thread creation and synchronization constructs, and shows how this definition can be adjusted for (1) checking whether the executions of a program are datarace free; or (2) instrumenting the execution to obtain traces for applying predictive runtime analysis techniques. Section 5.4 defines an alternative semantics for KERNELCC based on a relaxed memory model inspired from the x86-TSO memory model [128] and shows that the differences between this model and the sequential consistent one can be tested using the available tools.

5.1 KernelC: A C-like Language

The running example for this chapter is KernelC, a non-trivial subset of the C language (including memory allocation and pointer arithmetic), which is defined in this section, and then adapted in the following sections (with relatively minimal effort) to obtain testing and runtime analysis tools for programs written in this language, as well as in KernelCC, an extension of KernelC with concurrency constructs.

5.1.1 Syntax

KernelC allows writing C programs with addition and subtraction, increment, assignment, basic comparison operators and logical connectives, ternary if, basic input/output library functions, expression statements, statement composition and blocks, conditional, while loop, function declaration and invocations, variable declarations, memory allocation, freeing, and dereferencing (including array dereferencing). We additionally allow `#include` directives, which make our programs fully specified by including the standard library headers, and thus compilable and executable with a C compiler.The module below presents the syntax of KernelC:

```
Module KERNELC-SYNTAX
  imports PL-ID+PL-INT
  Exp ::= Int | PointerId | DeclId
        | Exp + Exp [strict]  | Exp - Exp [strict]  | Exp ++
        | Exp = Exp [strict(2)]
        | Exp == Exp [strict]  | Exp != Exp [strict]  | Exp <= Exp [strict]
        | ! Exp   | Exp && Exp   | Exp || Exp   | Exp ? Exp : Exp
        | printf("%d;", Exp ) [strict]  | scanf("%d", & Exp )
        | Id ( List{Exp} ) [strict(2)]  | Id ()
        | (int*)malloc( Exp *sizeof(int)) [strict]  | free( Exp ) [strict]
        | NULL  | * Exp [strict]  | Exp [ Exp ]  | scanf("%d", Exp ) [strict]
  PointerId ::= Id   | * PointerId [strict]
  DeclId ::= int Exp   | void PointerId
  Stmt ::= Exp ; [strict]  | {}   | { StmtList }
         | if( Exp ) Stmt   | if( Exp ) Stmt else Stmt [strict(1)]
         | while( Exp ) Stmt
         | DeclId  List{DeclId} { StmtList }
         | DeclId  List{DeclId} { StmtList return Exp ;}
         | #include< StmtList >
  Id ::= main
  StmtList ::= Stmt   | StmtList  StmtList
  Pgm ::= StmtList
END MODULE
```

```
int * arrRead() {
  int n;
  scanf(”%d”,&n);
  int * a = (int *)malloc((n+1)*sizeof(int));
  int i = 0;
  while (i != n) {
    scanf(”%d”,a+i);
    i++;
  }
  a[n]=0;
  return a;
}
```

```
void arrPrint(int *a) {
   while (* a) {
     printf (”%d;”,* a++);
   }
}

void arrCpy(int * a, int * b) {
   while (*a++=*b++) {}
}

int arrLen(int * a) {
   int l = 0;
   while(* a++) l++;
   return l;
}
```

Figure 5.1: `array.h`: a library for manipulating null-terminated arrays

Restrictions. To keep things simple, we only allow int and void as a basic types. Moreover, we only allow the reference operator for the `scanf` function, which basically means that one cannot get the address for local variables (for heap allocated variables the address is returned automatically by the allocation function). Also, to avoid dealing with control, as it is not particularly relevant in the context of runtime verification, we only allow return statements at the end of a function body.

While in this chapter we use KernelC, the results that we present, including the extensions (such as the memory safety, or the extension with threads), can be applied to the full C language. Ellison and Roşu [51] present a comprehensive $\mathbb{K}$ definition of the C language following the C99 standard [85] and are already applying our extensions.

How expressive is KernelC? Although comprising only a limited subset of C, KernelC is nevertheless quite expressive. The above defined syntax and the preprocessing lexer for K-Maude allows us to write library functions like those in Figure 5.1 and implementations of algorithms like that in Figure 5.2, which make extensive use of allocated memory and pointer arithmetic. Moreover, these programs abide to the C syntax, and can be successfully compiled using a standard C compiler (we have used gcc).

Figure 5.1 presents a collections of functions for creating, displaying, copying, and getting the length of null-terminated arrays of integers, that is, arrays meant to hold non-zero integers, whose end is signaled by a zero element. Functions `arrCpy` and `arrLen` are similar to the C library functions `strcpy` and `strlen`.

The function `quickSort` in Figure 5.2 is an implementation of the Quicksort algorithm [82], which, in addition to sorting in place, makes direct use of pointers instead of indexes.

```
void quickSort(int *b, int *e) {
  int t;
  if (! (e <= b + 1)) {
    int p = *b; int *l = b+1; int *r = e;
    while (l+1<= r) {
      if (*l <= p) {
        l = l + 1;
      } else {
        r = r - 1;
        t=*l;*l=*r;*r=t;
      }
    }
    l = l - 1;
    t=*l;*l=*b;*b=t;
    quickSort(b,l); quickSort(r,e);
  }
}
```

Figure 5.2: `qsort.h`: A C implementation of the Quicksort algorithm

The semantics of KernelC presented in next section allows programs using the functions above to be executed, traced, and debugged.

5.1.2 Semantics of KernelC

Before we define the execution rules for KernelC constructs, let us perform some simplifications. In essence, in the module presented below we express some syntactic constructs in terms of others and we remove information about types, identifying all types with int.

Module KERNELC-DESUGARED-SYNTAX
  imports KERNELC-SYNTAX
    macro: ! *E* = *E* ? 0 : 1
    macro: E_1 && E_2 = E_1 ? E_2 : 0
    macro: E_1 || E_2 = E_1 ? 1 : E_2
    macro: if(*E*) *St* = if(*E*) *St* else {}
    macro: NULL = 0
    macro: *I* () = *I* (())
    macro: *DeclId*(*DeclIds*) { *Sts* } = *DeclId*(*DeclIds*) { *Sts* return 0 ;}
    macro: void *PointerId* = int *PointerId*
    macro: int * *PointerId* = int *PointerId*
    macro: int * *PointerId* = *E* = int *PointerId* = *E*
    macro: #include< *Stmts* > = *Stmts*
    macro: E_1 [E_2] = * (E_1 + E_2)
    macro: scanf("%d", & * *E*)=scanf("%d", *E*)
    macro: int *X* = *E* ;= int *X* ; *X* = *E* ;
end module

The above module simplified the syntax of our language, reducing the number of constructs which need to be semantically defined. The module below presents a complete semantics for "desugared" KernelC. Assuming the reader is already familiar with the language definitions presented in Chapter 4, this definition should be pretty easy to read. Therefore, let us only describe the structure of the configuration and some design choices.

The top level of the configuration can hold the $\langle\rangle_{\mathsf{T}}$ cell, containing the state of the execution, or the $\langle\rangle_{\mathsf{result}}$ cell, which should be set to the output of the program once its execution completes successfully. Inside the $\langle\rangle_{\mathsf{T}}$ cell, one can find the computation cell $\langle\rangle_{\mathsf{k}}$, the local environment $\langle\rangle_{\mathsf{env}}$ (mapping local variables to their values) the set of functions $\langle\rangle_{\mathsf{funs}}$ structured as a map indexed by function name, the input ($\langle\rangle_{\mathsf{in}}$) and output ($\langle\rangle_{\mathsf{out}}$) cells, heap allocated memory $\langle\rangle_{\mathsf{mem}}$ as a mapping from locations to values, and a counter cell $\langle\rangle_{\mathsf{next}}$ used to generate fresh locations.

One important design choice is that we have decided to clearly distinguish between the heap allocated memory which is kept in the $\langle\rangle_{\mathsf{mem}}$ cell and the local variables memory which is kept in the $\langle\rangle_{\mathsf{env}}$ cell as a direct map from variables to values. Due to this choice, it is impossible to obtain the address of a variable, and this is enforced by the non-existence of the reference operator in KernelC; in fact, the C reference operator "`&`" only appears syntactically as a variant of the `scanf` function, to allow reading local variables from the input list. Another simplifying design decision was to not deal with scoping. The semantic rules presented below assume that once a variable is declared, it is visible for the remainder of the enclosing function execution, and therefore there should not be duplicated declarations of the same variable during the execution of the function.

Another important design decision we took was about the granularity of computation steps. We deliberately made all rules structural except for the rules which read/write dynamically allocated memory. This basically means that in the associated transition system we will only have a transition when an access to the $\langle\rangle_{\mathsf{mem}}$ cell is performed; one could think about the other transitions as being "silent".

Module KERNELC-SEMANTICS

imports PL-CONVERSION+K+KERNELC-DESUGARED-SYNTAX

$KResult ::= List\{Val\}$

$K ::= List\{Exp\} \mid List\{PointerId\} \mid List\{DeclId\} \mid StmtList \mid Pgm \mid String$
$\quad \mid \texttt{restore(}\ Map\ \texttt{)}$

$Exp ::= Val$

$Val ::= Int$

$List\{K\} ::= Nat\ \texttt{..}\ Nat$

initial configuration:

$$\left\langle \begin{array}{c} \langle \cdot K\rangle_{\mathsf{k}}\ \ \langle \cdot Map\rangle_{\mathsf{env}}\ \ \langle \cdot Map\rangle_{\mathsf{funs}} \\ \langle \cdot List\rangle_{\mathsf{in}}\ \ \langle \texttt{""}\rangle_{\mathsf{out}}\ \ \langle \cdot Map\rangle_{\mathsf{mem}}\ \ \langle \cdot Map\rangle_{\mathsf{ptr}}\ \ \langle 1\rangle_{\mathsf{next}} \end{array} \right\rangle_{\mathsf{T}} \langle \texttt{""}\rangle_{\mathsf{result}}$$

K RULES:

$I_1 \texttt{ == } I_2 \rightharpoonup \texttt{Bool2Int(}\ I_1 \texttt{ ==}_{\mathit{Int}}\ I_2 \texttt{)}$

$I_1 \texttt{ != } I_2 \rightharpoonup \texttt{Bool2Int(}\ I_1 \texttt{ !=}_{\mathit{Int}}\ I_2 \texttt{)}$

$I_1 \texttt{ + } I_2 \rightharpoonup I_1 \texttt{ +}_{\mathit{Int}}\ I_2$

$I_1 \texttt{ - } I_2 \rightharpoonup I_1 \texttt{ -}_{\mathit{Int}}\ I_2$

$I_1 \texttt{ <= } I_2 \rightharpoonup \texttt{Bool2Int(}\ I_1 \leq_{\mathit{Int}} I_2 \texttt{)}$

$\texttt{\_?\_:\_} \rightharpoonup \texttt{if(\_)\_else\_}$

$\texttt{if(}\ I\ \texttt{)}\ \_\ \texttt{else}\ St \rightharpoonup St \qquad \text{when } I \texttt{ ==}_{\mathit{Int}}\ 0$

$\texttt{if(}\ I\ \texttt{)}\ St\ \texttt{else}\ \_ \rightharpoonup St \qquad \text{when } \texttt{not}_{\mathit{Bool}}\ I \texttt{ ==}_{\mathit{Int}}\ 0$

$V \texttt{ ;} \rightharpoonup \cdot$

$\langle \underset{V}{\underline{X}} \ \cdots\rangle_{\mathsf{k}} \quad \langle\cdots\ X \mapsto V\ \cdots\rangle_{\mathsf{env}}$

$\langle \underset{I}{\underline{X \texttt{ ++}}} \ \cdots\rangle_{\mathsf{k}} \quad \langle\cdots\ X \mapsto \underset{I \texttt{ +}_{\mathit{Int}} 1}{\underline{I}}\ \cdots\rangle_{\mathsf{env}}$

$\langle \underset{V}{\underline{X \texttt{ = } V}} \ \cdots\rangle_{\mathsf{k}} \quad \langle\cdots\ X \mapsto \underset{V}{\underline{\_}}\ \cdots\rangle_{\mathsf{env}}$

$\langle \underset{\texttt{if(}\ E\ \texttt{) \{}\ St\ \ \texttt{while(}\ E\ \texttt{)}\ St\ \texttt{\} else \{\}}}{\underline{\texttt{while(}\ E\ \texttt{)}\ St}} \ \cdots\rangle_{\mathsf{k}}$

$\langle \underset{\texttt{void}}{\underline{\texttt{printf("\%d;",}\ I\ \texttt{)}}} \ \cdots\rangle_{\mathsf{k}} \quad \langle \underset{S \texttt{ +}_{\mathit{String}}\ \texttt{Int2String(}\ I\ \texttt{)} \texttt{ +}_{\mathit{String}}\ \texttt{";"}}{\underline{S}} \rangle_{\mathsf{out}}$

$\langle \underset{\texttt{void}}{\underline{\texttt{scanf("\%d",}\ N\ \texttt{)}}} \ \cdots\rangle_{\mathsf{k}} \quad \langle\cdots\ N \mapsto \underset{I}{\underline{\_}}\ \cdots\rangle_{\mathsf{mem}} \quad \langle \underset{\cdot}{\underline{I}}\ \cdots\rangle_{\mathsf{in}}$

$\langle \underset{\texttt{void}}{\underline{\texttt{scanf("\%d", \&}\ X\ \texttt{)}}} \ \cdots\rangle_{\mathsf{k}} \quad \langle\cdots\ X \mapsto \underset{I}{\underline{\_}}\ \cdots\rangle_{\mathsf{env}} \quad \langle \underset{\cdot}{\underline{I}}\ \cdots\rangle_{\mathsf{in}}$

$\texttt{\{}\ Sts\ \texttt{\}} \rightharpoonup Sts$

$\texttt{\{\}} \rightharpoonup \cdot$

$St\ \ Sts \rightharpoonup St \curvearrowright Sts$

$\langle \underset{V}{\underline{\texttt{*}\ N}} \ \cdots\rangle_{\mathsf{k}} \quad \langle\cdots\ N \mapsto V\ \cdots\rangle_{\mathsf{mem}}$

CONTEXT: $(\texttt{*}\ \square)\ \texttt{++}$

$\langle \underset{I}{\underline{(\texttt{*}\ N)\ \texttt{++}}} \ \cdots\rangle_{\mathsf{k}} \quad \langle\cdots\ N \mapsto \underset{I \texttt{ +}_{\mathit{Int}} 1}{\underline{I}}\ \cdots\rangle_{\mathsf{mem}}$

CONTEXT: $\texttt{*}\ \square\ \texttt{=}\ \_$

$\langle \underset{V}{\underline{\texttt{*}\ N \texttt{ = } V}} \ \cdots\rangle_{\mathsf{k}} \quad \langle\cdots\ N \mapsto \underset{V}{\underline{\_}}\ \cdots\rangle_{\mathsf{mem}}$

$\langle \underset{\cdot}{\underline{\texttt{int}\ X(Xl)\texttt{\{}Sts\ \texttt{return}\ E\texttt{;\}}}} \ \cdots\rangle_{\mathsf{k}} \quad \langle\cdots\ \underset{X \mapsto \texttt{int}\ X(Xl)\texttt{\{}Sts\ \texttt{return}\ E\texttt{;\}}}{\underline{\cdot}}\ \cdots\rangle_{\mathsf{funs}}$

$$\left(\begin{array}{c} \langle \underset{Sts \curvearrowright E \curvearrowright \texttt{restore}\ (Env)}{\underline{X\ \texttt{(}\ Vl\ \texttt{)}}} \ \cdots\rangle_{\mathsf{k}} \quad \langle \underset{\texttt{eraseKLabel}\ (\texttt{int\_}\ \texttt{,}\ Xl) \mapsto Vl}{\underline{Env}} \rangle_{\mathsf{env}} \\ \langle\cdots\ X \mapsto \texttt{int}\ X\ \ Xl\ \texttt{\{}\ Sts\ \texttt{return}\ E\ \texttt{;\}}\ \cdots\rangle_{\mathsf{funs}} \end{array}\right)$$

$\langle \underset{0}{\underline{\texttt{int}\ X}} \ \cdots\rangle_{\mathsf{k}} \quad \langle\cdots\ \underset{X \mapsto 0}{\underline{\cdot}}\ \cdots\rangle_{\mathsf{env}}$

```
#include<stdio.h>
#include<stdlib.h>
#include<array.h>

int main() {
  int * x = arrRead(); arrPrint(x);
  int * y = arrRead(); arrPrint(y);
  arrCpy(x,y);
  arrPrint(x); arrPrint(y);
  free(x); free(y);
  return 0;
}
```

(a)

```
#include<stdio.h>
#include<stdlib.h>
#include<qsort.h>
#include<array.h>

int main() {
  int * a = arrRead();
  quickSort(a,a+arrLen(a));
  arrPrint(a);
  free(a);
  return 0;
}
```

(b)

Figure 5.3: Programs testing the (a) `arrCpy` and the (b) `quickSort` functions

$$\langle V \curvearrowright \frac{\texttt{restore(}\ Env\ \texttt{)}}{\cdot} \cdots\rangle_{\mathsf{k}}\quad \langle \frac{\_}{Env} \rangle_{\mathsf{env}}$$

$$\left(\begin{array}{c} \langle \frac{\texttt{(int*)malloc(}\ N\ \texttt{*sizeof(int))}}{N'} \cdots\rangle_{\mathsf{k}}\quad \langle\cdots \frac{\cdot}{N' \mapsto N} \cdots\rangle_{\mathsf{ptr}} \\ \langle\cdots \frac{\cdot}{N'\ ..\ N +_{Nat} N' \mapsto 0} \cdots\rangle_{\mathsf{mem}}\quad \langle \frac{N'}{N +_{Nat} N'} \rangle_{\mathsf{next}} \end{array}\right)$$

$$\langle \frac{\texttt{free(}\ N\ \texttt{)}}{\texttt{void}} \cdots\rangle_{\mathsf{k}}\quad \langle\cdots \frac{N \mapsto N'}{\cdot} \cdots\rangle_{\mathsf{ptr}}\quad \langle \frac{Mem}{Mem\ [\bot\ /\ N\ ..\ N +_{Nat} N'\ ]} \rangle_{\mathsf{mem}}$$

$$N_1\ ..\ N_1 \rightharpoonup \cdot List\{K\}$$

$$N_1\ ..\ N \rightharpoonup N\ ,\ N_1\ ..\ N -_{Int} 1 \quad \text{when } N >_{Int} N_1$$

END MODULE

Test Setup. The semantics presented above allows us to execute programs written in KernelC. To test our functions for zero-terminated arrays and the Quicksort implementation, we can write programs like those in Figure 5.3 and test cases for them, and run them either using gcc, or our own definition. For the `arrCpy` test, we have selected three test cases: when the string to be copied is empty, when it is shorter than the first string, and when it is longer than the first string. Between these three, only the last one is memory unsafe. However, the program compiled through `gcc` shows no observable error for any of them. The reason for that is that `gcc` pads allocated blocks with non-allocated space, and since the buffer overflow is small enough, it does not override other allocated memory.

We can here observe the first benefit of our definition, as our definition actually detects a problem with the third test case. This happens because our naïve allocation semantics allocates blocks in order and without padding, and thus the overflow of the copy command actually overrides the beginning of the second array. However, this kind of error would have been detected by any interpreter of the language taking a similar approach to memory allocation.

Moreover, this is not really helpful; the buffer overflow was allowed to occur, and, although producing observable effects (by compromising existing data), this could be hard to trace for more complex programs. Next section shows how one can easily modify the semantics presented above to obtain a runtime verification tool for memory safety, which will precisely identify the source of our problem.

5.2 Runtime Verification of Memory Safety

Informally, memory safety means that the program cannot access a memory location which it shouldn't (e.g., exceeding arrays boundaries, addressing unallocated or freed memory, and so on).

One of the main sources of C's non-determinism comes from the underspecification of *C's memory allocator*, which implements the `malloc` and `free` functions. The C language specification guarantees that a call to `malloc(` n `)` will, if it succeeds, return a pointer to a region of n contiguous and previously unallocated locations (if it does not succeed, it will return `NULL`). These locations now become allocated. When these locations are no longer needed, the pointer, which was returned from a `malloc` call, is passed to `free` which deallocates the memory. The C language specification provides no guarantees except for that `malloc` returns unallocated locations; `free` might deallocate the memory or not.

Therefore, either our choice of allocating new locations in order in the definition above, or `gcc`'s choice of padding the allocated buffers is correct and in fact any implementation which ensures non-overlapping of allocated and not-yet-released memory blocks is acceptable. This under-specification could be captured in our $\mathbb{K}$ definition by replacing the rule for `malloc` with the rule:

$$\left(\begin{array}{c} \langle \underset{N'}{\underline{\texttt{(int*)malloc(}\ N\ \texttt{*sizeof(int))}}}\ \cdots\rangle_{\mathsf{k}}\ \ \langle\cdots\ \underset{N'\mapsto N}{\underline{\quad\cdot\quad}}\ \cdots\rangle_{\mathsf{ptr}} \\ \langle\sigma\ \underset{N'\ ..\ N\ +_{Nat}\ N'\ \mapsto\ 0}{\underline{\qquad\qquad\cdot\qquad\qquad}}\rangle_{\mathsf{mem}} \end{array}\right)$$

$$\text{when } \operatorname{Dom}\sigma \cap \{N', \ldots, N'+N\} = \emptyset$$

With this small modification, a KernelC program is *memory-safe* iff it cannot get stuck when its initial state is rewritten using this modified definition, i.e., it cannot be rewritten to a normal form whose computation cell is not well-terminated. However, the underspecification of this definition makes it non-analyzable, as infinitely many choices are now available each time one tries to apply the memory allocation rule.

A simple fix to this problem is to handle pointer allocation symbolically. That is, whenever memory is allocated, the base pointer will be a fresh symbolic positive number. This approach brings at least three benefits: (1) the allocated blocks are always guaranteed to be distinct; (2) they are also completely isolated, and thus overflow is impossible; and (3) except testing for equality or disequality, pointer comparison only makes sense between addresses belonging to the same block (as for them we can simplify the symbolic base pointer). To ensure that our executions are even safer than that, we additionally use a special computation constant `undef` (which is not a value) to be mapped to the uninitialized variables and memory locations. Not being a value, the `undef` constant has the property that it cannot be read, as our read rules only read values from the store, but can be overwritten, as our assign rules use an anonymous variable for the preexistent value.

To achieve all these benefits, the following alterations are sufficient for our symbolic definition:

- Extend the specification of natural numbers to include symbolic non-zero natural numbers, e.g., by a construct

  $\mathit{NzNat} ::= \texttt{symNzNat}\ \mathit{Nat}$

  and define reduction rules to handle basic arithmetic and comparison operations on terms involving symbolic naturals, by basically simplifying the symbolic variables whenever possible.

- Add to the computation sort a constant `undef`, which is a proper (i.e., not a result) computation, to stand for uninitialized variables/memory locations:

  $K ::= \texttt{undef}$

- Update the rule for declaring a variable to set its initial value to `undef`:

  $\langle \underset{\texttt{0}}{\underline{\texttt{int}\ X}}\ \cdots\rangle_{\mathsf{k}}\ \langle\cdots\ \underset{X \mapsto \texttt{undef}}{\underline{\cdot}}\ \cdots\rangle_{\mathsf{env}}$

- Finally, update the malloc rule to use the counter from the $\langle\rangle_{\mathsf{next}}$ cell to generate a symbolic pointer and to initialize the allocated memory to `undef`.

$$\left(\begin{array}{l} \langle \underset{\mathit{SymN}}{\underline{\texttt{(int*)malloc(}\ N\ \texttt{*sizeof(int))}}}\ \cdots\rangle_{\mathsf{k}}\ \langle\cdots\ \underset{\mathit{SymN} \mapsto N}{\underline{\cdot}}\ \cdots\rangle_{\mathsf{ptr}} \\ \langle\cdots\ \underset{\mathit{SymN}\ \texttt{..}\ N +_{\mathit{Nat}} \mathit{SymN} \mapsto \texttt{undef}}{\underline{\cdot}}\ \cdots\rangle_{\mathsf{mem}}\ \langle \underset{N' +_{\mathit{Nat}} 1}{\underline{N'}}\rangle_{\mathsf{next}} \end{array}\right)$$

when $\mathit{SymN} = \texttt{symNzNat}\ N'$

Let us call the definition obtained this way SafeKernelC. Unlike the underspecified version of the definition above, this definition is precisely specified and deterministic, and thus can be used to execute programs (although pointers are handled symbolically). We define *strong memory safety* for KernelC programs as the property of executing to completion under the SafeKernelC definition. Following a similar argument to the one presented in the paper introducing strong memory safety [149], one can prove that: (1) strong memory safety guarantees memory safety; and (2) monitoring SafeKernelC executions yields a sound procedure for runtime verification of KernelC memory safety.

The above result would be vacuous if most executions would get stuck because of not knowing how to manipulate symbolic pointers. However, this is not the case: SafeKernelC can successfully execute and analyze programs like those presented in the sections above. In particular it was able to completely execute the `Quicksort` test for any attempted input, and thus show that its execution is memory safe (for those inputs). Moreover, the `arrCpy` testing program only gets stuck for the test case which overflows, and it does so in the attempt of writing a location past the allocated memory, clearly identifying the problem. As these programs are making intensive use of pointer arithmetic and comparison, it is fair to say that this approach should work for a large class of C programs. We actually conjecture that this approach can be successfully applied to any C programs writable using the KernelC syntax which are abiding to the specifications of the C standard [85, 70], i.e., not reading uninitialized data, using only basic (addition/subtraction) pointer arithmetic, and comparing only pointers having the same base.

Note that our semantics also handles “free” and detects the common C bugs such as “double free” or accessing deallocated memory. Our rules effectively remove the memory mappings of freed memory locations, and thus further attempts to free those locations again or to access the freed memory lead to stuck configurations.

5.3 Runtime Verification of Concurrent Programs

In this section we propose KernelCC, a simple extension of KernelC with threads and synchronization constructs, together with a sequentially consistent semantics for it. We then show how this definition can be easily transformed into tools for analyzing and observing program executions. First such transformation allows checking program executions for datarace freeness. The second can be used to obtain traces which can be later analyzed by offline analysis tools, such as ones based on the happens-before relation or generic sequential consistency assumptions [95].

5.3.1 Extending KernelC with Concurrency Constructs

The proposed extension of KernelC with concurrency is very simple: we add a new construct for thread creation, `spawn`, which is supposed to call a function in a new thread, and which returns the id of the thread created. This id can be used by the `join` construct to wait for the corresponding thread to terminate. Lock `acquire` and `release` can be performed on any value (to keep things simple and avoid introducing an additional lock declaration construct). All constructs except for `spawn` take arbitrary expressions as input and are declared strict.

Exp ::= `spawn` *Exp* | `join(` *Exp* `)` [strict]
| `acquire(` *Exp* `)` [strict] | `release(` *Exp* `)` [strict]

We can now make the Quicksort implementation concurrent in KernelCC, by changing the line containing the two recursive invocations of the sorting function into two spawning constructs for the same invocations:

spawn(quickSort(b,l)); **spawn**(quickSort(r,e));

The semantics for KernelCC presented in this section does not need to modify any rules. We simply extend the syntax of the language with the introduced constructs, and we reorganize the configuration by adding some additional cells to manage multiple threads and their synchronization.

The new configuration is the following:

INITIAL CONFIGURATION:

$$\left\langle \begin{array}{l} \left\langle \left\langle \left\langle \cdot K \right\rangle_{\mathsf{k}} \left\langle \cdot Map \right\rangle_{\mathsf{env}} \left\langle \mathsf{0} \right\rangle_{\mathsf{id}} \right\rangle_{\mathsf{thread*}} \right\rangle_{\mathsf{threads}} \left\langle \cdot Map \right\rangle_{\mathsf{locks}} \left\langle \cdot Set \right\rangle_{\mathsf{cthreads}} \\ \quad \left\langle \cdot Map \right\rangle_{\mathsf{funs}} \left\langle \cdot List \right\rangle_{\mathsf{in}} \left\langle \texttt{""} \right\rangle_{\mathsf{out}} \left\langle \cdot Map \right\rangle_{\mathsf{mem}} \left\langle \cdot Map \right\rangle_{\mathsf{ptr}} \left\langle \mathsf{0} \right\rangle_{\mathsf{next}} \end{array} \right\rangle_{\mathsf{T}}$$

$$\left\langle \texttt{""} \right\rangle_{\mathsf{result}}$$

The configuration of KernelCC adds to that of KernelC a $\langle\rangle_{\mathsf{threads}}$ cell inside the top cell $\langle\rangle_{\mathsf{T}}$ containing potentially zero or more $\langle\rangle_{\mathsf{thread}}$ cells, which group the existing computation cell $\langle\rangle_{\mathsf{k}}$ and the environment of local variable declarations $\langle\rangle_{\mathsf{env}}$ with a new cell $\langle\rangle_{\mathsf{id}}$ to hold the identifier for the thread. Also inside the top cell $\langle\rangle_{\mathsf{T}}$ we add a $\langle\rangle_{\mathsf{locks}}$ cell to map each busy lock to the thread holding it, and a $\langle\rangle_{\mathsf{cthreads}}$ cell to keep the set of identifiers of the completed threads (used in giving semantics to the `join` construct). Because declared variables are separated from the allocated memory, and since one cannot take the reference of a declared variable, the only shared memory in KernelC is that which is dynamically allocated. This is a simplifying assumption which also avoids the need for synchronization for local variables, as one is always certain that the local memory is private to the thread.

To keep things looking more natural, and since we can afford design decisions, we allow the `spawn` construct to take just one argument, which is expected to be a function invocation; however, we do not declare `spawn` strict, and use a context to specify that the arguments need to be evaluated before the thread is spawned:

CONTEXT: $\texttt{spawn}\ \_\ (\ \square\ )$

Once the arguments have been evaluated, a new thread is generated to contain the function invocation as its computation. To be able to identify the threads (mainly for the `join` construct) the value of the $\langle\rangle_{\mathsf{next}}$ counter is used as an identifier for the newly spawned thread and is also returned to the parent thread as the result of the spawning construct.

$$\left\langle \frac{\texttt{spawn}\ X\ (\ Vl\ )}{N} \cdots \right\rangle_{\mathsf{k}} \left\langle \frac{N}{N +_{Nat} 1} \right\rangle_{\mathsf{next}} \frac{\cdot}{\left\langle \cdots \left\langle X\ (\ Vl\ ) \right\rangle_{\mathsf{k}} \left\langle N \right\rangle_{\mathsf{id}} \cdots \right\rangle_{\mathsf{thread}}}$$

Once a thread execution is completed, it is dissolved, and its id is added to the set of completed threads, to be used in giving semantics to the `join` construct.

The semantics of the `join` construct is that it is dissolved only when the value of its argument is among the completed thread ids.

$$\underset{\cdot}{\underline{\langle \cdots\ \langle V\rangle_{\mathsf{k}}\ \langle N\rangle_{\mathsf{id}}\ \cdots\rangle_{\mathsf{thread}}}}\ \ \langle \cdots\ \underset{N}{\underline{\ \cdot\ }}\ \cdots\rangle_{\mathsf{cthreads}}$$

$$\langle \underset{\texttt{void}}{\underline{\texttt{join(}\ N\ \texttt{)}}}\ \cdots\rangle_{\mathsf{k}}\ \ \langle \cdots\ N\ \cdots\rangle_{\mathsf{cthreads}}$$

The semantics of lock acquire and release is different than that from the previously defined languages. The reason for this difference is that we choose not to allow locks to be reentrant, and thus can afford a simpler semantics for them. The acquire/release semantics is that we allow the program to lock on any natural number (although in programs we only lock on existing memory locations), and we keep a map $\langle\rangle_{\mathsf{locks}}$ mapping each lock to the thread holding it.

$$\langle \underset{\texttt{void}}{\underline{\texttt{acquire(}\ N\ \texttt{)}}}\ \cdots\rangle_{\mathsf{k}}\ \langle N'\rangle_{\mathsf{id}}\ \ \langle \mathit{Locks}\ \ \underset{N \mapsto N'}{\underline{\qquad\cdot\qquad}}\rangle_{\mathsf{locks}}$$
$$\text{when}\ \ \texttt{not}_{\mathit{Bool}}\ N\ \texttt{in}\ \texttt{domain}\ (\mathit{Locks})$$

$$\langle \underset{\texttt{void}}{\underline{\texttt{release(}\ N\ \texttt{)}}}\ \cdots\rangle_{\mathsf{k}}\ \langle N'\rangle_{\mathsf{id}}\ \ \langle \cdots\ \underset{\cdot}{\underline{N \mapsto N'}}\ \cdots\rangle_{\mathsf{locks}}$$

We will next talk about analyzing executions obtained using this concurrent semantics to check program executions for concurrency problems, namely dataraces and deadlocks.

5.3.2 Searching for Dataraces

We here follow a rather obvious definition of dataraces, namely a datarace can be observed iff during the execution of the program there is a moment in which two threads can take as the next execution step transitions which accesses the same memory location, and at least one of the two accesses is attempting to update the location.

Although we could express this property as an assertion on states and then use Maude's model checker to check whether every possible execution of the program (for a given input) is datarace free, we here take a simpler approach by defining the race condition within $\mathbb{K}$ and then directly using the exploration capabilities of Maude to search for a datarace. If one is found, a configuration exhibiting the race is produced; if not, then the program is proven datarace free (for the given input). To do that, we add two new cells as alternatives to existing cells, $\langle\rangle_{\mathsf{race}}$ as an alternative to the $\langle\rangle_{\mathsf{k}}$ cell and $\langle\rangle_{\mathsf{raceDetected}}$ as an alternative to the top cell $\langle\rangle_{\top}$, together with two rules capturing the write-write, and write-read dataraces, respectively:

$$\langle *\, N = E\ \ \cdots\rangle_{\underset{\mathsf{race}}{\underline{\mathsf{k}}}}\qquad \langle *\, N = E'\ \ \cdots\rangle_{\underset{\mathsf{race}}{\underline{\mathsf{k}}}}$$

$$\langle *\, N = E\ \ \cdots\rangle_{\underset{\mathsf{race}}{\underline{\mathsf{k}}}}\qquad \langle *\, N\ \ \cdots\rangle_{\underset{\mathsf{race}}{\underline{\mathsf{k}}}}$$

These two structural rules ensure that any further computation is stopped for the two threads identified to be in a race, and eases their recognition in the configuration exhibiting the race. In addition to that, we add another structural rule which changes the top computation itself once a race is detected, so we can easily identify an entire configuration exhibiting a race.

$$\langle \cdots\ \langle K \rangle_{\mathsf{race}}\ \cdots \rangle_{\frac{\mathsf{T}}{\mathsf{raceDetected}}}$$

Dataraces in Quicksort. With nothing more than the addition of these rules we can now test whether a direct execution of our concurrent version of Quicksort, call it `pConcQuickSort` yields any observable dataraces for a given input, here abstracted by `pQuickSort.in`. To do that, we can use the following Maude command upon loading the compiled version of our K-Maude definition of KernelCC:

```
rewrite run('pConcQuickSort, pQuickSort.in) .
```

It turns out that by simply executing the program we can observe a datarace. However, when analyzing it, we observe that the datarace is not caused by the concurrent sorting procedure itself; instead, it occurs between the sorting procedure and the main thread, which is not prevented from outputting the array before all sorting threads have completed. To address this issue, we can attempt to join the threads before outputting the array by replacing the line for spawning the two recursive calls to `quickSort` by the following lines:

```
int t1 = spawn(quickSort(b,l));
int t2 = spawn(quickSort(r,e));
join(t1); join(t2);
```

When re-executing the thus modified program, no race is observed. We can now check for datarace freeness, by using the Maude command:

```
search[1] run('pConcQuickSort, pQuickSort.in)
        ⇒∗ ⟨ raceDetected ⟩ B:Bag ⟨/ raceDetected ⟩ .
```

which asks Maude to check for a configuration observed during the execution in which the top cell is raceDetected. As desired, this search command finds no solution, which guarantees that the program is datarace free for the given input. While this is not a proof of the general datarace freeness of the program, one can use a technique like this in conjunction with test generation tools to increase confidence in the datarace freeness, or even to prove datarace freeness given a sufficient coverage criteria.

Dataraces and Deadlocks. Let us here present an additional example, illustrating how one can detect dataraces, attempt to fix them, and then re-check the program for dataraces, but also for deadlocks which could have been introduced by the attempted fix. The following *Banking* example is a C implementation of a

```
int *newAccount(int m) {
  int *a=(int *)malloc(1*sizeof(int));
  *a=m;
  return a;
}

void deposit(int *a, int m) {
  acquire(a);
  *a=*a+m;
  release(a);
}

int balance(int *a) {
  acquire(a);
  int b=*a;
  release(a);
  return b;
}
```

```
void withdraw(int *a, int m) {
  acquire(a);
  if (m <= *a) {
    *a=*a−m;
  }
  release(a);
}

void transfer(int *a, int *b, int m) {
  acquire(a);
  if (m <= *a) {
    *a=*a−m;
    *b=*b+m;
  }
  release(a);
}
```

`account.h`

```
#include<stdio.h>
#include<stdlib.h>
#include<account.h>

void run(int *a, int *b) {
  deposit(a,300);
  withdraw(a,100);
  transfer(a,b,100);
}
```

```
int main() {
  int *one = newAccount(100);
  int *two = newAccount(20);
  printf(”%d;”, balance(one));
  printf(”%d;”, balance(two));
  int t1 = spawn(run(one, two));
  int t2 = spawn(run(two, one));
  join(t1); join(t2);
  printf(”%d;”, balance(one));
  printf(”%d;”, balance(two));
  return 0;
}
```

`pAccountDriver.c`

Figure 5.4: The account “class” and a concurrent test driver for it.

Java class exhibiting a concurrent bug pattern [54]. The class attempts to define an account and some basic operations on it: creation, deposit, balance, withdraw, and transfer to another account. Figure 5.4 presents our C implementation of it, which encodes the objects as locations holding the amount of money available and the methods of the class as functions taking the receiver object's location as their first argument. Additionally, similarly to Java, we model the synchronized attribute of the methods by locking on the location of the receiver object at the beginning of the function and unlocking it before the return.

Similarly to the concurrent Quicksort implementation, we can check the execution of the test driver for dataraces. Asking the tool to simply execute the test driver directly (i.e., to rewrite the initial configuration of program until it reaches a final state, without exploring the non-determinism) reveals no datarace. However, the search reveals a datarace instantaneously on account “`two`” between its read in the `deposit` function and its write in `transfer`.

Upon analyzing the counter-example configuration, one can extract as a reason for it the fact that the access to account `b` in the `transfer` function is not synchronized. A simple-minded fix for this problem is to additionally lock on the `b` account in the `transfer` function. Upon applying this fix we can verify that the test driver became indeed datarace free. However, this came at the expense of introducing a deadlock, which we can actually detect by either formalizing deadlock in the model checker, or by directly searching for final configurations which are not topped in the result cell, using the Maude command:

search[1] run('pAccountDriver.c) ⇒! ⟨ T ⟩ B:*Bag* ⟨/ T ⟩ .

which detects a deadlock between the two calls to `transfer`.

By following Dijkstra's [43] solution to deadlock avoidance, we can ensure datarace freeness while avoiding deadlocks. The way to achieve that is by always acquiring resources in the same order in any thread. In our concrete example, this can be done by replacing the `transfer` function with the following one:

```
void transfer(int *a, int *b, int m) {
  if (!(a <= b)) {
    acquire(a); acquire(b);
  } else {
    acquire(b); acquire(a);
  }
  if (m <= *a) {
    *a = *a-m;
    *b = *b+m;
  }
  release(b); release(a);
}
```

Using this new implementation of the `transfer` function we can now effectively check that our test driver is both datarace and deadlock free. However, although this might work in most real implementations of C (extended with a similar kind of concurrency), this approach is not completely "conformant" according to the C standard, as the program attempts to compare two unrelated pointers; this can also be captured by our strong memory safety approach: if replacing the base KernelC definition with SafeKernelC, the execution would get stuck when attempting to compare the two pointers.

5.3.3 Monitoring Executions

Although model checking (or completely exploring) all possible executions of a program for a given input is indeed desirable when one wants to guarantee datarace freeness, this can become prohibitively expensive. Runtime analysis techniques can again help in alleviating this problem by instrumenting the program to produce traces of its execution, and then, by using that execution to infer other executions, including ones that might exhibit a bug (although the original execution did not). We follow here the approach of predictive runtime analysis [157, 158, 30, 32, 162], and show how one can use monitoring to collect a trace containing all events required to replay the execution or to obtain a causal model for it.

To enable monitoring of executions, no existing rule needs to be changed. All we need to do is to extend the configuration with some additional cells representing the monitoring infrastructure and then to provide additional structural rules for updating the monitor.

We exemplify this approach below by adding monitoring infrastructure for observing store accesses and synchronizing events in KernelC executions. For the monitoring infrastructure, we only need to add two cells to the configuration: a cell $\langle\rangle_{\mathsf{event}}$ representing a flag defaulting to false, signaling whether an event-generating construct is about to be executed, which must be added to the $\langle\rangle_{\mathsf{thread}}$ cell, and and a cell $\langle\rangle_{\mathsf{events}}$, initially empty, holding the list of generated events, which must be added to the top cell $\langle\rangle_{\mathsf{T}}$. Moreover, we need additional constructors for each type of event (read/write/spawn/acquire/release), or pre-event (read/spawn) we want to monitor:

$K \ ::= \ \mathit{PreEvent} \mid \mathit{Event}$

$\mathit{Event} ::= \texttt{write(}\ \mathit{Nat} : \mathit{Nat} \leftarrow \mathit{Int}\ \texttt{)} \quad \mid \texttt{read(}\ \mathit{Nat} : \mathit{Nat} \mapsto \mathit{Int}\ \texttt{)}$

$\qquad\qquad \mid \texttt{spwn(}\ \mathit{Nat} : \mathit{Nat}\ \texttt{)} \quad \mid \texttt{acq(}\ \mathit{Nat} : \mathit{Nat}\ \texttt{)} \quad \mid \texttt{rel(}\ \mathit{Nat} : \mathit{Nat}\ \texttt{)}$

$\mathit{PreEvent} ::= \texttt{rea(}\ \mathit{Nat} : \mathit{Nat}\ \texttt{)} \quad \mid \texttt{spw(}\ \mathit{Nat}\ \texttt{)}$

Given this setup, the rules for generating the events are generated after the following pattern. When an recordable action is about to be executed, a structural rule inserts the (pre-)event in the computation after the construct which would generate it, and it sets the event flag to true to mark that the event generation was enabled and thus avoid multiple events being generated for the same action. Then, after the action takes place, another structural rule adds the event to the list of events, while resetting the flag. Since for `read` and `spawn` we have only limited information before the action takes place, a pre-event is scheduled, and then transformed into an event with another structural rule once the rule completes.

$$\langle \texttt{*}\ N \texttt{ = } V \curvearrowright \frac{\cdot}{\texttt{write(}\ T : N \leftarrow V\ \texttt{)}} \ \cdots\rangle_{\mathsf{k}} \ \langle T\rangle_{\mathsf{id}} \ \langle \frac{\mathsf{false}}{\mathsf{true}} \rangle_{\mathsf{event}}$$

$$\langle \texttt{*}\ N \curvearrowright \frac{\cdot}{\texttt{rea(}\ T : N\ \texttt{)}} \ \cdots\rangle_{\mathsf{k}} \ \langle T\rangle_{\mathsf{id}} \ \langle \frac{\mathsf{false}}{\mathsf{true}} \rangle_{\mathsf{event}}$$

$$\langle V \curvearrowright \frac{\texttt{rea(}\ T : N\ \texttt{)}}{\texttt{read(}\ T : N \mapsto V\ \texttt{)}} \ \cdots\rangle_{\mathsf{k}}$$

$$\langle \texttt{spawn}\ E \curvearrowright \frac{\cdot}{\texttt{spw(}\ T\ \texttt{)}} \ \cdots\rangle_{\mathsf{k}} \ \langle T\rangle_{\mathsf{id}} \ \langle \frac{\mathsf{false}}{\mathsf{true}} \rangle_{\mathsf{event}}$$

$$\langle N \curvearrowright \frac{\texttt{spw(}\ T\ \texttt{)}}{\texttt{spwn(}\ T : N\ \texttt{)}} \ \cdots\rangle_{\mathsf{k}}$$

$$\langle \texttt{acquire(}\ N\ \texttt{)} \curvearrowright \underset{\texttt{acq(}\ T\ \texttt{:}\ N\ \texttt{)}}{\underline{\cdot}} \cdots\rangle_{\mathsf{k}}\ \langle T\rangle_{\mathsf{id}}\ \langle\underset{\mathsf{true}}{\underline{\mathsf{false}}}\rangle_{\mathsf{event}}$$

$$\langle \texttt{release(}\ N\ \texttt{)} \curvearrowright \underset{\texttt{rel(}\ T\ \texttt{:}\ N\ \texttt{)}}{\underline{\cdot}} \cdots\rangle_{\mathsf{k}}\ \langle T\rangle_{\mathsf{id}}\ \langle\underset{\mathsf{true}}{\underline{\mathsf{false}}}\rangle_{\mathsf{event}}$$

Once the action was consumed and the pre-event was completed to an event, the generated event is recorded into the trace:

$$\langle \_ : \mathit{Val} \curvearrowright \underset{\cdot}{\underline{\mathit{Event}}} \cdots\rangle_{\mathsf{k}}\ \langle\underset{\mathsf{false}}{\underline{\mathsf{true}}}\rangle_{\mathsf{event}}\ \langle\cdots \underset{\mathit{Event}}{\underline{\cdot}}\rangle_{\mathsf{trace}}$$

An important prerequisite of this monitoring approach is that all monitorable actions must be computational rules, to ensure that the structural rules for scheduling the events apply before the actual events are generated.

5.4 A Relaxed Memory Model for KernelCC

Before we conclude, let us show how one can give another memory model semantics for the concurrent version of KernelC, and to use the available analysis tools to compare it against the sequentially consistent version of KernelCC defined in Section 5.3. We base this semantics on the x86-TSO memory model [128], regarding threads as processors, and local variables as registers.

The x86-TSO memory model specification associates to each process (in our case thread) a write buffer, which collects the local updates of memory variables, and defines the semantics of memory access and synchronization by taking into account these buffers. Therefore, the rules for all involved language constructs need to be changed in our $\mathbb{K}$ definition; nevertheless, nothing else except them and the configuration needs to be altered.

Two more cells need to be added to the $\langle\rangle_{\mathsf{thread}}$ cell: a $\langle\rangle_{\mathsf{buffer}}$ cell holding the queue of buffered writes, and a $\langle\rangle_{\mathsf{blocked}}$ cell containing a flag signaling whether the thread is blocked in waiting for a lock. Moreover, we add a list item constructor `bwrite` to represent a buffered write, that takes as parameters a location and a value; and we define a `locations` function which retrieves the set of locations from a list of buffered writes:

$$\mathit{ListItem} ::= \texttt{bwrite(}\ \mathit{Nat}\ \texttt{,}\ \mathit{Val}\ \texttt{)}$$

$$\mathit{Set} ::= \texttt{locations}\ (\mathit{List})$$

$$\texttt{locations}\ (\cdot) \rightharpoonup \cdot$$

$$\texttt{locations}\ (\texttt{bwrite(}\ N\ \texttt{,}\ V\ \texttt{)}\ \ \mathit{Mem}) \rightharpoonup N\ \ \texttt{locations}\ (\mathit{Mem})$$

In what follows we present the $\mathbb{K}$ rules specifying the new relaxed memory model semantics for concurrent KernelC preceded by their natural language description taken verbatim from Owens et al. [128]:

1. p can read v from memory at address a if p is not blocked, has no buffered writes to a, and the memory does contain v at a;

$$\langle \underset{V}{\underline{\texttt{*}\ N}} \cdots\rangle_{\mathsf{k}}\ \langle \mathit{Mem}\rangle_{\mathsf{buffer}}\ \langle\cdots\ N \mapsto V\ \cdots\rangle_{\mathsf{mem}}$$

$$\text{when}\ \ \texttt{not}_{\mathit{Bool}}\ N\ \texttt{in}\ \texttt{locations}\ (\mathit{Mem})$$

2. p can read v from its write buffer for address a if p is not blocked and has v as the newest write to a in its buffer;

$$\langle \underset{V}{\underline{\texttt{*}\ N}}\ \ \cdots\rangle_{\mathsf{k}}\ \ \langle\cdots\ \ \texttt{bwrite(}\ N\ \texttt{,}\ V\ \texttt{)}\ \ \mathit{Mem}\rangle_{\mathsf{buffer}}$$

$$\text{when}\ \texttt{not}_{\mathit{Bool}}\ N\ \texttt{in}\ \texttt{locations}\ (\mathit{Mem})$$

3. p can read the stored value v from its register r at any time;

   Since we view local variables as our registers, and since the rule is unconstrained, the existing rule for reading / writing local variables stays unchanged.

4. p can write v to its write buffer for address a at any time;

$$\langle \underset{V}{\underline{\texttt{*}\ N\ \texttt{=}\ V}}\ \ \cdots\rangle_{\mathsf{k}}\ \ \langle\cdots\ \ \underset{\texttt{bwrite(}\ N\ \texttt{,}\ V\ \texttt{)}}{\underline{\qquad\cdot\qquad}}\rangle_{\mathsf{buffer}}$$

   Additionally, in KernelCC we need to define the rules for incrementing values at memory locations, which, similarly to regular reads, have two flavors: depending on whether the location is or is not in the appropriate write buffer:

$$\langle \underset{I}{\underline{\texttt{*}\ N\ \texttt{++}}}\ \ \cdots\rangle_{\mathsf{k}}\ \ \langle\cdots\ \ \texttt{bwrite(}\ N\ \texttt{,}\ I\ \texttt{)}\ \ \mathit{Mem}\ \ \underset{\texttt{bwrite(}\ N\ \texttt{,}\ I\ +_{\mathit{Int}}\ 1\ \texttt{)}}{\underline{\qquad\qquad\cdot\qquad\qquad}}\rangle_{\mathsf{buffer}}$$

$$\text{when}\ \texttt{not}_{\mathit{Bool}}\ N\ \texttt{in}\ \texttt{domain}\ (\mathit{Mem})$$

$$\langle \underset{I}{\underline{\texttt{*}\ N\ \texttt{++}}}\ \ \cdots\rangle_{\mathsf{k}}\ \ \langle\mathit{Mem}\ \ \underset{\texttt{bwrite(}\ N\ \texttt{,}\ I\ +_{\mathit{Int}}\ 1\ \texttt{)}}{\underline{\qquad\qquad\cdot\qquad\qquad}}\rangle_{\mathsf{buffer}}\ \ \langle\cdots\ \ N\mapsto I\ \ \cdots\rangle_{\mathsf{mem}}$$

$$\text{when}\ \ \texttt{not}_{\mathit{Bool}}\ N\ \texttt{in}\ \texttt{domain}\ (\mathit{Mem})$$

5. if p is not blocked, it can silently dequeue the oldest write from its write buffer to memory;

$$\langle\mathsf{false}\rangle_{\mathsf{blocked}}\ \ \langle\underset{\cdot}{\underline{\texttt{bwrite(}\ N\ \texttt{,}\ V\ \texttt{)}}}\ \ \cdots\rangle_{\mathsf{buffer}}\ \ \langle\cdots\ \ N\mapsto\ \underset{V}{\underline{\_}}\ \ \cdots\rangle_{\mathsf{mem}}$$

6. p can write value v to one of its registers r at any time;

   Same as for item 3, the existing rule needs not be changed.

7. if p's write buffer is empty, it can execute an MFENCE (so an MFENCE cannot proceed until all writes have been dequeued, modelling buffer flushing); LFENCE and SFENCE can occur at any time, making them no-ops;

   We here assume that thread synchronization constructs, such as creation, termination, and join are all generating MFENCE operations:

$$\langle\underset{N}{\underline{\texttt{spawn}\ X\ \texttt{(}\ \mathit{Vl}\ \texttt{)}}}\ \ \cdots\rangle_{\mathsf{k}}\ \langle\underset{N\ +_{\mathit{Int}}\ 1}{\underline{\quad N\quad}}\rangle_{\mathsf{next}}\ \ \langle\cdot\rangle_{\mathsf{buffer}}\ \ \underset{\langle\cdots\ \langle X\ \texttt{(}\ \mathit{Vl}\ \texttt{)}\rangle_{\mathsf{k}}\ \ \langle N\rangle_{\mathsf{id}}\ \cdots\rangle_{\mathsf{thread}}}{\underline{\qquad\qquad\qquad\cdot\qquad\qquad\qquad}}$$

$$\underset{\cdot}{\underline{\langle\cdots\ \langle V\rangle_{\mathsf{k}}\ \langle N\rangle_{\mathsf{id}}\ \langle\cdot\rangle_{\mathsf{buffer}}\ \cdots\rangle_{\mathsf{thread}}}}\ \langle\cdots\ \underset{N}{\underline{\cdot}}\ \cdots\rangle_{\mathsf{cthreads}}$$

$$\langle\underset{0}{\underline{\texttt{join(}\ N\ \texttt{)}}}\ \cdots\rangle_{\mathsf{k}}\ \langle\cdot\rangle_{\mathsf{buffer}}\ \langle\cdots\ N\ \cdots\rangle_{\mathsf{cthreads}}$$

8. if the lock is not held, and p's write buffer is empty, it can begin a LOCK'd instruction;

$$\langle\underset{\texttt{void}}{\underline{\texttt{acquire(}\ N\ \texttt{)}}}\ \cdots\rangle_{\mathsf{k}}\ \langle N'\rangle_{\mathsf{id}}\ \langle\cdot\rangle_{\mathsf{buffer}}\ \langle \mathit{Locks}\quad \underset{N\mapsto N'}{\underline{\qquad\cdot\qquad}}\rangle_{\mathsf{locks}}$$

$$\text{when }\ \texttt{not}_{\mathit{Bool}}\ N\ \texttt{in}\ \texttt{keys}\ \mathit{Locks}$$

9. if p holds the lock, and its write buffer is empty, it can end a LOCK'd instruction.

$$\langle\underset{\texttt{void}}{\underline{\texttt{release(}\ N\ \texttt{)}}}\ \cdots\rangle_{\mathsf{k}}\ \langle N'\rangle_{\mathsf{id}}\ \langle\cdot\rangle_{\mathsf{buffer}}\ \langle\cdots\ \underset{\cdot}{\underline{N\mapsto N'}}\ \cdots\rangle_{\mathsf{locks}}$$

Two additional structural rules are used to update the flag of the $\langle\rangle_{\mathsf{blocked}}$ cell:

$$\langle\texttt{acquire(}\ N\ \texttt{)}\ \cdots\rangle_{\mathsf{k}}\ \langle\underset{\mathsf{true}}{\underline{\mathsf{false}}}\rangle_{\mathsf{blocked}}\ \langle\cdots\ N\mapsto\ \_\ \cdots\rangle_{\mathsf{locks}}$$

$$\langle\texttt{acquire(}\ N\ \texttt{)}\ \cdots\rangle_{\mathsf{k}}\ \langle\underset{\mathsf{false}}{\underline{\mathsf{true}}}\rangle_{\mathsf{blocked}}\ \langle \mathit{Locks}\rangle_{\mathsf{locks}}$$

$$\text{when }\texttt{not}_{\mathit{Bool}}\ N\ \texttt{in}\ \texttt{keys}\ \mathit{Locks}$$

The first rule says that the thread becomes blocked if it tries to acquire a lock which is already held, while the second rule unblocks the thread once the lock is released.

Thus, with precisely one rule for concurrency construct and without altering unrelated language constructs, we have defined a concurrent semantics for KernelC with a relaxed memory model.

Using this semantics, we can test, for example, that programs relying on busy-waiting synchronization are not portable from sequentially consistent memory models to relaxed memory models. Consider the KernelC specification of Peterson's software solution for mutual exclusion [132] presented in Figure 5.5. The presented implementation uses a function with three parameters, `flag`, `turn`, and `t`. `flag` is a (dynamically allocated) array, `turn` points to an integer in memory, and `t` is used as a thread identifier. To mark the critical sections, we are printing -1 and -2 for the beginning of critical section and 1 and 2 for the end of critical section for the threads identified by 0 and 1, respectively.

Using the previous (sequentially consistent) definition of concurrency for KernelC, one can verify that mutual exclusion is ensured by loading the K-Maude compiled definition and asking Maude to search for all final states obtainable upon running the program. Only two such states are obtained, with a total search space of 68 states, containing `"-1;1;-2;2;"` and `"-2;2;-1;1;"` in the $\langle\rangle_{\mathsf{result}}$ cell, respectively, effectively showing that the statements in the two critical sections cannot be interleaved.

```
#include <stdio.h>
#include <stdlib.h>

void peterson(int *flag, int *turn, int t) {
  flag [t] = 1;
  *turn = 1−t;
  while (flag[1−t] && *turn == 1−t) {}
  printf(”%d;”,−1 − t);
  printf(”%d;”, 1 + t);
  flag [t] = 0;
}

int main() {
  int* flag= (int *)malloc(2*sizeof(int));
  flag[0]= 0; flag[1]= 0 ;
  int *turn= (int *)malloc(1*sizeof(int));
  int t1= spawn(peterson(flag, turn, 0));
  int t2= spawn(peterson(flag, turn, 1));
  join(t1); join(t2);
  return 0;
}
```

Figure 5.5: An implementation of Peterson's algorithm in KernelC.

However, when exploring the executions of the same program in the relaxed memory model definition of concurrent KernelC, mutual exclusion is not ensured: indeed Maude finds 6 solutions to the same task, with a total search space of 439 states, showing that the sequences -1,1 and -2,2 can be interleaved in every possible way. We can now ask Maude to show us the sequence of rewrites required to reach the state where the result cell holds `"-1;-2;2;1;"`, for example. When analyzing this paths we can observe the following potential order of rule applications:

1. Schedule and then commit the initialization of the `flag` array (spawn needs an empty buffer to proceed);

2. Spawn both threads, in order, which also leads to silently scheduling the first two writes of each thread (in order), without committing them, as scheduling can happen at any time when it is enabled, being modeled as a structural rule;

3. Since it has no pending updates on the flag corresponding to the other thread, the first thread can read it from the memory and, being 0, can exit the busy wait loop;

4. Similarly, the second thread can read the value of first thread's flag from the memory, which is still 0, as the buffers are not required to be emptied, and then can exit the busy wait loop;

5. Both threads are now in the mutual exclusion zone which should not have been possible.

One can try to fix this problem by changing the order between the tests in the function's conditional statement. However, this "fix" does not help either (although it increases the search space to 471 states), as both of the threads can still avoid committing the buffered writes by using the value of `turn` from the local buffer and then reading the value of the flag from the shared memory. A real solution would require some kind of synchronization mechanism such as the use of `MFENCE` instructions to "flush" the write buffers and thus obtain an execution similar to the sequentially consistent ones.

Thus, by applying a simple, generic, and already available rewriting logic tool on our $\mathbb{K}$ definitions we have shown that the relaxed memory model for KernelCC defined in this section cannot be relied on for achieving mutual exclusion for programs which achieve that under the sequential consistency assumptions of the definition in Section 5.3.

5.5 Discussion

We have shown how $\mathbb{K}$ definitions of programming languages can be turned (with negligible effort) into runtime analysis tools for testing and analyzing executions of both sequential and concurrent programs, under different memory models.

Moreover, having different semantics (e.g., different memory models) of the same language formalized in the same framework opens the door for analyzing the relationship between definitions of languages and for proving meta-theorems about them. One such example for $\mathbb{K}$ definitions was achieved by Ellison [50], who proves type soundness as a relation between the dynamic semantics and the typing semantics of a language, exploiting the fact that the rules of the two definitions are essentially the same, although their semantic domain changes.

In the same vein, one could take advantage of the direct correspondence between the rules of the sequentially consistent model defined in Section 5.3 and those of the relaxed model defined in Section 5.4 to prove that, under datarace freeness assumptions, the relaxed memory model cannot produce more behaviors than the sequentially consistent one. Similar to the on-paper proof, the idea would be to show that in any possible execution under the relaxed model, accesses to the same variable must satisfy the sequential consistency requirement, because conflicting accesses to the same location need to be separated by synchronization constructs, which require the write buffers to be emptied. Achieving similar results and studying how their proof can be mechanized seems like an interesting line of research.

We do not claim here that the tools one obtains almost for free within the $\mathbb{K}$ framework completely eliminate the need of writing dedicated analysis tools in "real" programming languages. At least, not yet. Nevertheless, we strongly believe that the $\mathbb{K}$ framework can be viewed like a workbench for rapidly prototyping and experimenting with such analysis tools. Moreover, we believe that compilation techniques could be used to generate (more) competitive analysis tools directly from $\mathbb{K}$ definitions.

One could indeed argue that the techniques we have applied here to derive analysis tools can be virtually applied to other programming languages definitional frameworks, as well. While not attempting to completely refute this claim, we rather claim that, even if it is possible, it would be rather difficult and it would require additional infrastructure to be built, which might obscure the interesting parts. Indeed, we believe $\mathbb{K}$ to be better fit for this approach for at least three reasons: (1) the representation of computations in continuation-like structures, which allows having the redex always at the top of the computation cell; (2) the easiness of $\mathbb{K}$ in dealing with concurrency, as, for example, the ability to match two redexes (from two different threads) simultaneously, which is needed, for example, to allow an elegant definition for detecting dataraces; and (3) the context transformers, which allow one to focus only on the relevant parts of the configuration.

In this chapter we have only focused on runtime analysis of programs using $\mathbb{K}$ definitions. Nevertheless, definitions of languages in the $\mathbb{K}$ framework can also be used to derive other types of verification and analysis tools: type checkers and inferencers, policy checkers, and even verifiers based on an axiomatics semantics-like approach. These are related research topics within the larger $\mathbb{K}$ and RLS context, and we will not detail them in this dissertation; the relevant related work is presented in Section 8.1.

Chapter 6

A Concurrent Semantics for $\mathbb{K}$ Rewriting

Given the intrinsic potential for concurrency of rewriting logic, and of rewriting in general, it is natural to attempt (and succeed in) defining concurrent programming languages using rewriting. However, a question arising in this context is whether the framework is generous enough to be able to offer the amount of concurrency desired by the language designer. Noticing an apparent lack of concurrency related to the concurrent access to resources for the direct representation of $\mathbb{K}$ in rewriting logic, the research presented in this chapter defines a faithful concurrent semantics for $\mathbb{K}$ which captures the intended concurrency specified by the $\mathbb{K}$ rules. This semantics is obtained by speculating the resemblance between $\mathbb{K}$ rules and a graph rewriting rules, and by adapting classical concurrency results from the algebraic theory of graph rewriting [45, 39, 47]. As graph rewriting itself can be captured within rewriting logic, this new $\mathbb{K}$ semantics induces another representation of $\mathbb{K}$ into rewriting logic, which, although not directly executable, faithfully captures the intended concurrency of $\mathbb{K}$ rules.

The Quest for More Concurrency

The easiest way to represent $\mathbb{K}$ within rewriting logic is to transform $\mathbb{K}$ rules into rewrite rules by simply ignoring their precise identification of what is changed by the rule and what not. This representation has the great advantage that nothing else (terms, configurations) needs to change. Additionally, if taking an interleaving concurrency point of view, as the implementation of rewriting logic into the Maude rewrite engine [34] (and, consequently, our K-Maude implementation of $\mathbb{K}$) does, then there is no observable difference w.r.t. the obtained transition system. What is lost in this translation though is the amount of concurrency available in one execution step.

Consider rewriting the term $h(f(a), 0, 1)$ using the following canonical term rewrite system, where h is a ternary operation, g is binary, f is unary, 0, 1, a, b are constants, and x, y are variables:

$$\begin{array}{l}
(1)\ h(x, y, 1) \Rightarrow h(g(x, x), y, 0) \\
(2)\ h(x, 0, y) \Rightarrow h(x, 1, y) \\
(3)\ a \Rightarrow b \\
(4)\ f(x) \Rightarrow x
\end{array}$$

The term $h(f(a), 0, 1)$ has a unique normal form, $h(g(b, b), 1, 0)$, which can be reached in a minimum of 4 rewrite steps, e.g., $h(f(a), 0, 1) \Rightarrow h(a, 0, 1) \Rightarrow h(b, 0, 1) \Rightarrow h(b, 1, 1) \Rightarrow h(g(b, b), 1, 0)$. In spite of the fact that all four rule instances above overlap on the term $h(f(a), 0, 1)$, the concurrent semantics for $\mathbb{K}$ rewriting can achieve the same result in *one concurrent rewrite step* for an appropriate set of $\mathbb{K}$ rules corresponding to the rewrite rules above. We are not aware of any other existing term rewriting approach which can be used to rewrite $h(f(a), 0, 1)$ to $h(g(b, b), 1, 0)$ in one concurrent step.

Let us first discuss intuitively how and why the four rules above can apply concurrently on $h(f(a), 0, 1)$. First, note that rule (1) modifies the first and the third arguments of h regardless of the second argument, while rule (2) modifies the second argument of h regardless of its first and third arguments. Therefore, rules (1) and (2) can share (without changing it) the top operator h and yield complementary changes on the original term, so they can safely apply their changes in parallel on term $h(f(a), 0, 1)$. Moreover, note that none of these rules needs to know that x is specifically bound or points to $f(a)$, or what happens with $f(a)$ during their application. Therefore, we can rewrite the $f(a)$ that x points to in parallel with the application of rules (1) and (2). Using a similar argument, rules (3) and (4) can apply in parallel on $f(a)$ to rewrite it to b. Thus, rules (1), (2), (3) and (4) can in principle apply in one parallel rewrite step on $h(f(a), 0, 1)$ and produce $h(g(b, b), 1, 0)$.

One can formalize the above intuition by using the particular form of the $\mathbb{K}$ rules. $\mathbb{K}$ rules add an additional layer to the standard rewrite rules, by explicitly mentioning what part of the matched term is rewritten and what part is left unchanged. This is achieved by underlining the parts to be rewritten (read-write), and writing the changes underneath the line. For example, the rewrite rules (1) and (2) above become the following $\mathbb{K}$ rules:

$$(1)\ h(\underset{g(x, x)}{\underline{\quad x \quad}}, \_, \underset{0}{\underline{1}}) \qquad\qquad (2)\ h(\_, \underset{1}{\underline{0}}, \_)$$

The parts of the term which are not underlined are shared (read-only). Variables which are not reused in a rule (i.e., occur only once) play a purely structural role; they are called “anonymous variables” and are often replaced by a generic “_” variable (each occurrence of “_” stands for a distinct variable, like in Prolog). Conventional rewrite rules are special $\mathbb{K}$ rules, where the entire term gets rewritten; the standard notation $l \Rightarrow r$ is then allowed as syntactic sugar. In fact, the ASCII notation for $\mathbb{K}$ rules in K-Maude [159], our implementation of $\mathbb{K}$ onto Maude [34], conservatively extends the the representation of standard rewrite rules in Maude; for example, the $\mathbb{K}$-rule (1) above is written as `h(x=>g(x,x),_,1=>0)` in K-Maude.

Two or more $\mathbb{K}$ rules can apply concurrently if and only if the instances of their read/write parts do not directly overlap and a special acyclicity condition (explained below) holds. By direct overlapping we mean overlapping of the actual operation symbols specified by the rules, not including the variables; overlapping below the matching instances of rule variables is proved safe and thus allowed. For example, rules (1) and (2) above can apply concurrently, because the read/write parts of each are matched "below" the match of the variables of the other.

The example above was deliberately artificial, to explain the problem that we are attempting to solve and its subtleties using a minimal setting. Given its simplicity, it will be used as a running example in the remainder of this chapter. But, to emphasize that the kind of concurrency allowed by $\mathbb{K}$ rules is indeed desirable when defining programming languages, let us now recast the problem above in the world of programming language definitions.

Multi-threaded concurrency. Consider the following $\mathbb{K}$ rules for accessing the state: $\langle \underset{i}{\underline{*l}} \ \cdots\rangle_{\mathsf{k}} \ \langle\cdots \ l \mapsto i \ \cdots\rangle_{\mathsf{state}}$ and $\langle \underset{i}{\underline{*l = i}} \ \cdots\rangle_{\mathsf{k}} \ \langle\cdots \ l \mapsto \underset{i}{\underline{\_}} \ \cdots\rangle_{\mathsf{state}}$. Assuming an initial configuration (used to give context to the $\mathbb{K}$ rules) of the form

$$\langle\langle\langle\langle \cdot K\rangle_{\mathsf{k}} \ \langle \cdot \mathit{Map}\rangle_{\mathsf{env}}\rangle_{\mathsf{thread}*}\rangle_{\mathsf{threads}} \ \langle \cdot \mathit{Map}\rangle_{\mathsf{state}}\langle 0\rangle_{\mathsf{next}}\rangle_{\mathsf{T}},$$

and directly transforming the rules above into rewrite rules, we obtain the following two rules:

$$\langle ts\langle t\langle *l \curvearrowright k\rangle_{\mathsf{k}}\rangle_{\mathsf{thread}}\rangle_{\mathsf{threads}} \ \langle \sigma \ \ l \mapsto i\rangle_{\mathsf{state}}$$
$$\rightarrow \langle ts\langle t\langle i \curvearrowright k\rangle_{\mathsf{k}}\rangle_{\mathsf{thread}}\rangle_{\mathsf{threads}} \ \langle \sigma \ \ l \mapsto i\rangle_{\mathsf{state}},$$

and $\langle ts\langle t\langle *l = i' \curvearrowright k\rangle_{\mathsf{k}}\rangle_{\mathsf{thread}}\rangle_{\mathsf{threads}} \ \langle \sigma \ \ l \mapsto i\rangle_{\mathsf{state}}$

$$\rightarrow \langle ts\langle t\langle i' \curvearrowright k\rangle_{\mathsf{k}}\rangle_{\mathsf{thread}}\rangle_{\mathsf{threads}} \ \langle \sigma \ \ l \mapsto i'\rangle_{\mathsf{state}}.$$

The $\langle\rangle_{\mathsf{k}}$ cell is a unary operation $\langle \_ \rangle_{\mathsf{k}}$ that holds a $\curvearrowright$-separated list of tasks (i.e., $\curvearrowright$ is associative and has identity "$\cdot$"). The $\langle\rangle_{\mathsf{env}}$ and $\langle\rangle_{\mathsf{state}}$ cells hold maps, i.e. a sets of bindings of names to locations, and of locations to values, respectively, constructed with the associative and commutative (AC) concatenation operation "$\_ \ \_$", having identity "$\cdot$". The concatenation operation "$\_ \ \_$" used to put cells together is also an AC operator with identity "$\cdot$". x, k, σ, t, ts, i, and i' are variables, l standing for a location in the state, k for the rest of the computation, σ for the remainder of the state, t for the remainder of the thread, ts for the other threads, and i, i' for integers.

Consider a system containing only these rules, and choose the (ground) term to be rewritten to be

$$\left\langle \begin{array}{c} \left\langle \begin{array}{c} \langle\langle *1\rangle_{\mathsf{k}} \ \ \langle\cdot\rangle_{\mathsf{env}}\rangle_{\mathsf{thread}} \ \ \langle\langle *1\rangle_{\mathsf{k}} \ \ \langle\cdot\rangle_{\mathsf{env}}\rangle_{\mathsf{thread}} \\ \langle\langle *2 = 3\rangle_{\mathsf{k}} \ \ \langle\cdot\rangle_{\mathsf{env}}\rangle_{\mathsf{thread}} \end{array} \right\rangle_{\mathsf{threads}} \\ \langle 1 \mapsto 1 \ \ 2 \mapsto 2\rangle_{\mathsf{state}} \end{array} \right\rangle_{\mathsf{T}},$$

This configuration specifies two threads whose tasks are to read the value at location 1 in the state, and a thread updating the value at location 2. Intuitively, all threads could advance simultaneously: the first two by reading the value of at location 1 (since it is shared), and the third by updating the value at location 2 (since location 2 is independent of location 1). However, this is impossible to achieve directly in standard rewriting because "the same object cannot be shared by two simultaneous rewrites" [104].

One reason is that traditional matching modulo axioms requires that the term be rearranged to fit the pattern, and thus it limits concurrency where sharing is allowed: there is no way to re-arrange the term so that any two of the rule instances match simultaneously. To address that, we propose a special treatment for matching operators governed by axioms (Section 6.2.3). Another, more fundamental reason, is that the top set constructor operation and the state itself need to be shared for the rules to apply. The $\mathbb{K}$ rules address this issue by distinguishing the read-only part of a rule pattern, which can be shared, and by making the change local to the exact position in which the update must be applied. Thus, the read-only part of a $\mathbb{K}$ rule can be regarded as an interface connecting the parts to be rewritten with each other and within the term which is being rewritten. This resembles a similar concept of interfaces in the algebraic approach to graph rewriting [47]. Sections 6.3 and 6.4 use this resemblance in giving semantics to $\mathbb{K}$ rewriting by representing $\mathbb{K}$ rules as graph rewrite rules and terms as (term-)graphs to take advantage of the concurrency results with interface sharing available for graph rewriting.

Meseguer [104] proposes a different solution to this problem in the context of a rewriting logic specification of concurrent objects, with the help of "emulsifying" equations which "magically" create multiple copies and arrange the configuration such that concurrent access is possible. This specification takes advantage of the special structure of concurrent objects, namely that all of them live in the same multi-set configuration at the top of the term to be rewritten, and thus creating multiple copies of an objects is easy achievable by generic rules. Although this approach might not be easily extended for general terms, which would allow a direct capturing of the semantics of $\mathbb{K}$ concurrent rewriting, it is generic enough to capture the concurrency of formalisms such as graph rewriting given an appropriate representation of graphs as multisets of concurrent objects [104]. In Section 6.5 we sketch how this solution could be used to capture the concurrency of $\mathbb{K}$ rules into rewriting logic in two steps: first, represent $\mathbb{K}$ rules as graph rewriting rules, then represent graph rewriting rules as rewrite rules (with sharing) on concurrent objects.

On seriallizability. While allowing more concurrency in one step for our $\mathbb{K}$ definitions, it is highly desirable that this concurrency does not introduce additional behaviors, unintended by the specification. This property allows, for example, to rely on the direct representation of $\mathbb{K}$ in rewriting logic for proving properties about program executions—if the concurrency in one step is serializable, then any transition of the concurrent semantics can be simulated by one or several sequential steps.

However, unrestricted concurrency migh lead to unserializability. Consider, for example, the term $f(g(a), h(b))$ to be rewritten using the following two $\mathbb{K}$ rules (f, g, h, a, and b, are operation symbols, while x and y are variables):

$$f(g(\underset{x}{\underline{a}}), x) \qquad\qquad f(y, h(\underset{y}{\underline{b}}))$$

A blind concurrent application of these two rules on $f(g(a), h(b))$ would yield the term $f(g(h(b)), h(g(a)))$. However, this concurrent rewrite step is *non-serializable*, since there is no way to order the application of the two rules on $f(g(a), h(b))$ to obtain $f(g(h(b)), h(g(a)))$.

For the reasons specified above, we choose not to allow unserializable rewriting for $\mathbb{K}$ and give an acyclicity criterion that ensures it. Informally, we have a cyclic relationship which prevents the two rules from being applied concurrently on $f(g(a), h(b))$: a gets rewritten to $h(b)$, then b gets rewritten to $g(a)$, and so on. The acyclicity condition was initially unexpected; its necessity appeared while proving the serializability of $\mathbb{K}$ concurrent rewriting (Theorem 6).

The main idea in formalizing the $\mathbb{K}$ concurrent rewriting is to lift the problem to a problem of graph rewriting, then use graph rewriting to perform the concurrent step, and then recover a term, i.e. the result of the concurrent rewrite step, from the resulting graph. While lifting term rewriting to graph rewriting is not a new idea (several existing works, labeled as term-graph-rewriting, are discussed in Section 2.3), previous efforts focused on term-graph rewriting for efficiency reasons, to avoid repeating rewrites on identical subterms of the term to rewrite. Our main purpose for reducing the problem to graph rewriting is to "borrow concurrency" from a domain where concurrency with sharing of resources has been extensively researched. Unfortunately, due to the desired capability of $\mathbb{K}$ rules to explicitly state what is shared and to allow concurrent rewrites under variables, conventional notions of term graph representations could not be used unchanged in the lifting process. Also, as already mentioned, a novel and unexpected acyclicity condition was necessary in order to show that the resulting graph can be reinterpreted as a term and the obtained parallel graph rewriting, when reinterpreted as $\mathbb{K}$ rewriting, is sound, complete and serializable for conventional term rewriting (Theorems 6 and 7).

The remainder of this chapter is structured as follows. In Section 6.1 we discuss several additional examples, such as concurrent sorting, concurrent Dijkstra's all shortest paths, and an executable variant of π-calculus, whose potential for concurrency are largely increased by the $\mathbb{K}$ concurrent rewriting. Section 6.2 formalizes $\mathbb{K}$ rules and intuitively describes the desirable semantics for $\mathbb{K}$ concurrent rewriting. Section 6.3 lifts the problem to the world of graph rewriting, extending an existing term-graph rewriting approach to faithfully capture $\mathbb{K}$ rewriting. Section 6.4 defines the $\mathbb{K}$ concurrent rewriting relation on terms and shows that is is serializable and correct w.r.t. standard term rewriting. Section 6.5 sketches a novel representation of $\mathbb{K}$ into rewriting logic which, although non-executable, faithfully captures the amount of concurrency obtainable in one step by $\mathbb{K}$ definitions. Finally, Section 6.6 discusses some current limitations and ways in which those could be addressed.

6.1 Motivating Examples

All examples in this chapter are produced using the LaTeX generator of the K-Maude tool (see Chapter 7) which converts the unidimensional ASCII notation into an easier to visualize bidimensional mathematical notation. We did, though, manually adjust the generated LaTeX a little for pagination purposes. Also, all the examples available on the K-Maude tool website [88], including those discussed here, use the substitution defined using the reflective capabilities of $\mathbb{K}$ (see Section 4.3.7); the examples in this chapter importing a module SUBSTITUTION have been adjusted to use a custom substitution instead of the generic one.

Concurrent sorting. The following K-Maude module sorts a list of integers. The imported builtin PL-INT module defines the syntactic category (or sort) *Int* as well as operations on integers, such as $>_{Int}$. The module K is imported by almost all definitions; it defines the syntactic category K, which should include all syntax (note the simple production "K ::= *Int*"), and provides common semantic infrastructure such as lists, sets, maps, cells, etc.:

MODULE SORT IMPORTS PL-INT+K

$K ::= \mathit{Int}$

INITIAL CONFIGURATION:

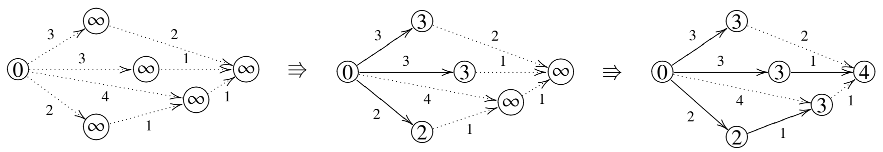
$\langle \cdot \mathit{List} \rangle_{\mathsf{sortme}}$

K RULES:

$\langle \cdots \underset{y}{\underline{x}} - \underset{x}{\underline{y}} \cdots \rangle_{\mathsf{sortme}}$ when $x >_{\mathit{Int}} y$

END MODULE

The module SORT above contains only one $\mathbb{K}$-rule, which states that any two unordered elements in the list cell can be swapped. The advantage of using $\mathbb{K}$ rewriting here, as opposed to conventional term rewriting modulo associativity, is that multiple instances of this rule can apply concurrently, even ones whose two elements are interleaved. Let us show how one can use this rule to sort the list $3, 8, 5, 7, 4, 1, 2, 6$ in three concurrent steps. We will mark how the numbers pair in the matching process by annotating the underline with indexed variables corresponding to each match. In the first concurrent step, the matching phase could mark for rewriting all positions, obtaining e.g.



(0) (1) (2)

Figure 6.1: Dijkstra's all shortest paths derivation in two concurrent steps

$\underline{3}_{x_1}, \underline{8}_{x_2}, \underline{5}_{x_3}, \underline{7}_{x_4}, \underline{4}_{y_3}, \underline{1}_{y_1}, \underline{2}_{y_2}, \underline{6}_{y_4}$. Upon applying the concurrent step, the list becomes $1, 2, 4, 6, 5, 3, 8, 7$. In the second step, the matching phase can yield $1, 2, \underline{4}_{x_1}, 6, 5, \underline{3}_{y_1}, \underline{8}_{x_2}, \underline{7}_{y_2}$, inducing a second concurrent rewrite step, to $1, 2, 3, 6, 5, 4, 7, 8$. Finally, there is only one possible rule instance left for matching, $1, 2, 3, \underline{6}_{x}, 5, \underline{4}_{y}, 7, 8$, producing the sorted list.

Concurrent Dijkstra. The following module gives a one-rule $\mathbb{K}$ specification for solving Dijkstra's all-shortest path problem:

MODULE DIJKSTRA

IMPORTS PL-ID+NAT-INF

$K ::= Id$

$SetItem ::= Edge$

$Edge ::= Id \xrightarrow{Nat} Id$

INITIAL CONFIGURATION:

$\langle graph\rangle_{\mathsf{white}} \cdot Set \quad \langle shortest\rangle_{\mathsf{white}} \cdot Map$

K RULES:

$$\langle \cdots\ x_1 \xrightarrow{w} x_2\ \cdots\rangle_{\mathsf{graph}}\ \langle \cdots\ x_1 \mapsto c_1\ \text{–}\ x_2 \mapsto \underset{c_1 +_{Nat} w}{\underline{\quad c_2 \quad}}\ \cdots\rangle_{\mathsf{shortest}}$$

when $c_1 +_{Nat} w <_{Nat} c_2$

END MODULE

The module PL-ID introduces identifiers as constants of sort *Id*, and NAT-INF introduces natural numbers with infinity. A graph is represented as a set of weighted edges $x_1 \xrightarrow{w} x_2$, saying that there is an edge from x_1 to x_2 of cost w. All shortest paths are represented as a mapping, which is a set of bindings $x_i \mapsto c_i$; each such binding states that the shortest path from the root to node x_i is c_i.

The initial term to rewrite should contain the graph in the $\langle\rangle_{\mathsf{graph}}$ cell, and a map mapping each node to ∞ in the $\langle\rangle_{\mathsf{shortest}}$ cell, except for the root node node, say a, which is mapped to 0. The rule above matches an edge $x_1 \xrightarrow{w} x_2$ in the graph, so that the current shortest path to x_2 is larger than the shortest path to x_1 plus w; if that is the case, the cost of the shortest path to x_2 is updated. We are only interested in the costs of the shortest paths here, not the shortest paths themselves. Those are easy to compute as well (e.g., storing x_1 next to the new cost of x_2), but we do not do it here. Note that everything is shared by the rule, except for the part it changes, the cost of x_2. Thus, many rules instances can apply in parallel, as far as they do not write the same shortest path costs. The graphical representation of a two-concurrent-step run of this system is presented in Figure 6.1. Initially all graph edges are dotted while the nodes contain the initial minimal costs. As the rewriting proceeds, costs in the nodes are updated and the edges considered are depicted with full lines.

By Theorem 7, the concurrent rewrite steps produced by the $\mathbb{K}$ system above are serializable, so standard term rewriting analysis techniques can apply. For example, the corresponding term rewrite system terminates (the sum of the non-infinity shortest path costs decreases with the application of each rewrite rule) and is confluent (its critical pairs are joinable), so it admits unique normal forms. The normal forms give the shortest path costs, because any path computation can be mimicked with applications of the rule above. This may be one of the simplest implementations and proofs of correctness for Dijkstra's algorithm.

Executable π-calculus. The K-Maude module below contains a $\mathbb{K}$ definition for a simple executable variant of the π-calculus.

MODULE EXECUTABLE-PI IMPORTS SUBSTITUTION

$$\begin{aligned} \mathit{Proc} ::=\ & !\mathit{Proc} \\ \mid\ & \mathit{Action}.\mathit{Proc} \\ \mid\ & \langle \mathsf{Bag}[\langle \mathsf{Bag}[\mathit{Proc}]\rangle_{\mathsf{sum}}]\rangle_{\mathsf{par}} \end{aligned} \qquad \begin{aligned} \mathit{Action} ::=\ & \overline{\mathit{Id}}\langle \mathit{Id}\rangle \\ \mid\ & \mathit{Id}(\mathit{Id}) \end{aligned}$$

K RULES:

$$\langle \underset{P}{\underline{\_\ \ \overline{C}\langle X\rangle.P}}\rangle_{\mathsf{sum}} \quad \langle \underset{Q\ [\ X\ /\ Y\ ]}{\underline{\ \_\ \ C(\,Y).Q\ }}\rangle_{\mathsf{sum}}$$

$$\langle \underset{P}{\underline{\_\ \ \overline{C}\langle X\rangle.P}}\rangle_{\mathsf{sum}} \quad \langle \underset{\cdot}{\underline{\_}}\ \ !C(\,Y).Q\rangle_{\mathsf{sum}} \quad \underset{\langle Q\ [\ X\ /\ Y\ ]\rangle_{\mathsf{sum}}}{\underline{\qquad\quad\cdot\qquad\quad}}$$

$$\langle R\ \ \langle Q\ \ (\nu X)P\rangle_{\mathsf{sum}}\rangle_{\mathsf{par}} \rightharpoonup (\nu\,Y)\langle R\ \ \langle Q\ \ P\ [\ Y\ /\ X\ ]\rangle_{\mathsf{sum}}\rangle_{\mathsf{par}}$$

when Y is fresh

$$\langle\langle P\rangle_{\mathsf{par}}\rangle_{\mathsf{sum}} \rightharpoonup P$$

END MODULE

The syntax defined above is quite similar to the original syntax of the π-calculus [115]. However, similar to the approach using Cham [20], to enhance the parallel communication inside processes, we use bags to represent the choice operator (the $\langle\rangle_{\mathsf{sum}}$ cell) and the parallel composition (the $\langle\rangle_{\mathsf{par}}$ cell). Thus, processes are represented as controlled nested cells: a $\langle\rangle_{\mathsf{par}}$ cell holds a bag of $\langle\rangle_{\mathsf{sum}}$ cells, each containing a bag of processes to be chosen amongst. This convention does not alter the expressivity, since any of the cells could contain only one element. It is standard to assume guarded choice; we do it, too: each process in a $\langle\rangle_{\mathsf{sum}}$ cell must start with an action. The 0 process is represented by $\langle\langle\cdot\rangle_{\mathsf{sum}}\rangle_{\mathsf{par}}$. For executability, we follow Pict [135] and only allow replication for input expressions. According to [135], this does not limit the formal power of the calculus.

The $\mathbb{K}$ system above only contains four $\mathbb{K}$ rules. The first two are for communication: the first is standard (note that the non-communicating processes are discarded from the two $\langle\rangle_{\mathsf{sum}}$ cells), while the second defines replication triggered by input. The third rule defines scope extrusion by pushing the ν binder up. The fourth and final rule "releases" a bag of parallel-composed processes once they have reached the top of a sum cell.

6.2 $\mathbb{K}$ Rewriting: Intuitive Semantics

In this section we formally define $\mathbb{K}$ rules and $\mathbb{K}$ systems and intuitively describe how applications of $\mathbb{K}$ rules can be combined concurrently.

6.2.1 $\mathbb{K}$ Rules and $\mathbb{K}$ Systems

$\mathbb{K}$ rules describe how a term can be transformed into another term by altering some of its parts. They share the idea of match-and-replace of standard term rewriting; however, each $\mathbb{K}$-rule identifies a read-only pattern, the local context of the rule. This pattern is used to glue together read-write patterns, that is, subparts to be rewritten. Moreover, through its variables, it also provides information which can be used and shared by the read-write patterns. To some extent, the read-only pattern plays here the same role played by interfaces in graph rewriting [44].

To focus on the core of concurrent rewriting, in this section only we make the following three simplifying assumptions: (1) all $\mathbb{K}$ rules are unconditional; (2) all $\mathbb{K}$ rules are left linear; and (3) there are no lists, sets, bags, or maps involved. $\mathbb{K}$ rules are typically unconditional and, when conditional, they have only very simple conditions anyway, which can be regarded as side conditions (as opposed to premises) that can be checked within their mathematical domain, without recursively invoking the $\mathbb{K}$ rewriting. Also, (2) can be reduced to (1) by checking that two terms are equal; only syntactic equality is considered in $\mathbb{K}$ rules (as opposed to provability), which again can be checked easily without recursively invoking $\mathbb{K}$ rewriting. (3) is the most subtle, because it may seem that one needs to extend $\mathbb{K}$ to work "modulo" list or multiset axioms. However, that is not the case, because working modulo such axioms actually inhibits concurrency: indeed, having to restructure the term to rewrite in order for the rule to match is not only expensive, but may also be in conflict with other rules attempting to concurrently apply. In Section 6.2.3 we show how, by dynamically changing the rules during the matching process instead of the term to be rewritten, one can achieve the same effect of matching modulo axioms without altering the term to be rewritten. Thus, we believe that our simplifying assumptions are acceptable.

A signature Σ is a pair (S, F) where S is a set of *sorts* and F is a set of operations $f : w \to s$, where f is an operation symbol, $w \in S^*$ is its arity, and $s \in S$ is its result sort. If w is the empty word ϵ then f is a constant. T_Σ

is the universe of (ground) terms over Σ and $T_\Sigma(\mathcal{X})$ is that of Σ-terms with variables from the S-sorted set X.

Given term $t \in T_\Sigma(\mathcal{X})$, let *vars*$(t)$ be the variables from $\mathcal{X}$ appearing in t. Given an ordered set of variables, $\mathcal{W} = \{\Box_1, \dots, \Box_n\}$, named *context variables*, or *holes*, a $\mathcal{W}$*-context over* $\Sigma(\mathcal{X})$ (assume that $\mathcal{X} \cap \mathcal{W} = \emptyset$) is a term $C \in T_\Sigma(\mathcal{X} \cup \mathcal{W})$ in which each variable in $\mathcal{W}$ occurs once.

The instantiation of a $\mathcal{W}$-context C with an n-tuple $\bar{t} = (t_1, \dots, t_n)$, written $C[\bar{t}]$ or $C[t_1, \dots, t_n]$, is the term $C[t_1/\Box_1, \dots, t_n/\Box_n]$. One can regard $\bar{t}$ as a substitution $\bar{t} : \mathcal{W} \to T_\Sigma(X)$, defined by $\bar{t}(\Box_i) = t_i$, in which case $C[\bar{t}] = \bar{t}(C)$.

Definition 1. *A* $\mathbb{K}$***-rule*** $\rho : (\forall \mathcal{X})\ k[\ L \Rightarrow R\ ]$ *over a signature* $\Sigma = (S, F)$ *is a tuple* $(\mathcal{X}, k, L, R)$*, where:*

- $\mathcal{X}$ *is an* S*-sorted set, called the* ***variables*** *of the rule* ρ*;*

- k *is a* $\mathcal{W}$*-context over* $\Sigma(\mathcal{X})$*, called the* ***rule pattern****, where* $\mathcal{W}$ *are the* ***holes*** *of* k*;* k *can be thought of as the "read-only" part or the "local" context of* ρ*;*

- $L, R : \mathcal{W} \to T_\Sigma(\mathcal{X})$ *associate to each hole in* $\mathcal{W}$ *the* ***original term*** *and its* ***replacement term****, respectively;* L*,* R *can be thought of as the "read/write" part of* ρ*.*

We may write $(\forall \mathcal{X})\ k[\ \underset{r_1}{\underline{l_1}}, \dots, \underset{r_n}{\underline{l_n}}\ ]$ *instead of* $(\forall \mathcal{X})\ k[\ L \Rightarrow R\ ]$ *whenever* $\mathcal{W} = \{\Box_1, \cdots, \Box_n\}$ *and* $L(\Box_i) = l_i$ *and* $R(\Box_i) = r_i$*; this way, the holes are implicit and need not be mentioned.*

A set of $\mathbb{K}$ *rules is called a* $\mathbb{K}$***-system****.*

The variables in $\mathcal{W}$ are only used to formally identify the positions in k where rewriting takes place; in practice we typically use the compact notation above, that is, underline the to-be-rewritten subterms in place and write their replacement underneath. When the set of variables $\mathcal{X}$ is clear, it can be omitted.

Let us discuss how this definition captures the visual intuition by formally describing the rules from our running example form the beginning of this chapter. For each of the four rules, the corresponding elements of a $\mathbb{K}$-rule are described in below:

	1	2	3	4
X	$\{x, y\}$	$\{x\}$	$\emptyset$	$\{x\}$
$\mathcal{W}$	$\{\Box_1, \Box_2\}$	$\{\Box\}$	$\{\Box\}$	$\{\Box\}$
p	$h(\Box_1, y, \Box_2)$	$h(x, \Box, y)$	$\Box$	$\Box$
L	$\Box_1 \mapsto x\ ;\ \Box_2 \mapsto 1$	$\Box \mapsto 0$	$\Box \mapsto a$	$\Box \mapsto f(x)$
R	$\Box_1 \mapsto g(x, x)\ ;\ \Box_2 \mapsto 0$	$\Box \mapsto 1$	$\Box \mapsto b$	$\Box \mapsto x$

6.2.2 Relation to Rewrite Rules

Here we analyze the relationship between rewrite rules and $\mathbb{K}$ rules, showing that the latter are a conservative extension of the former. Consider the two mappings defined below:

$$\mathit{R2K}((\forall\mathcal{X})\ l \to r) = (\forall\mathcal{X})\ \underset{r}{\underline{l}}$$

$$\mathit{K2R}((\forall\mathcal{X})\ k[\ \underset{R}{\underline{L}}\ ]) = (\forall\mathcal{X})\ L(k) \to R(k)$$

R2K associates to each rewrite rule a $\mathbb{K}$ rule quantified by the same variables, having as the read-only pattern just a hole $\square$ which is mapped by L and R to the left-hand-side and right-hand-side of the rewrite rule, respectively. Conversely, *K2R* associates to each $\mathbb{K}$-rule a rewrite rule by forgetting the additional information contained by the $\mathbb{K}$ rule and flattening it by applying L and R to the read-only pattern to obtain the lhs and the rhs of the rewrite rule, respectively.

$\mathbb{K}$ rules faithfully capture conventional rewrite rules, since $\mathit{K2R}(\mathit{R2K}(\varrho)) = \varrho$ for any rewrite rule ϱ. Note, however, that the opposite does not hold, that is, $\mathbb{K}$ rules are strictly more informative than their corresponding rewrite rules, because the latter lose their “locality” of the changes; in particular, it is impossible to recover the $\mathbb{K}$ rule from the obtained rewrite rule, because it is not always straightforward to determine what should be shared and what not.

Given a $\mathbb{K}$-rule $\rho : (\forall\mathcal{X})\ k[\ L \Rightarrow R\ ]$, its associated 0-*sharing* $\mathbb{K}$-*rule* is $\rho_0 = \mathit{R2K}(\mathit{K2R}(\rho)) : (\forall\mathcal{X})\ \square[\ \underset{R(k)}{\underline{L(k)}}\ ]$, that is a rule specifying the same transformation but without sharing anything. A $\mathbb{K}$ rule is *proper* if its read-only pattern k is a proper term.

How much to share Another possible transformation from rewrite rules to $\mathbb{K}$ rules is one that would enforce maximal data-sharing. Let us formally define such a transformation, say $\mathit{R2K}_{max}$. Given a rewrite rule ϱ: $(\forall\mathcal{X})\ l \to r$, the *maximally sharing* $\mathbb{K}$ *rule* associated to ϱ is $\mathit{R2K}_{max}(\varrho) = (\forall\mathcal{X})\ k[\ \underset{R}{\underline{L}}\ ]$ satisfying that $\mathit{K2R}(\mathit{R2K}_{max}(\varrho)) = \varrho$, and that, for any other $\mathbb{K}$ rule $\rho' : (\forall\mathcal{X})\ k'[\ \underset{R'}{\underline{L'}}\ ]$ such that $\mathit{K2R}(\rho') = \varrho$, k' is a specialization of k, that is, there exists a substitution $\theta : \mathit{vars}(k) \to T_\Sigma(\mathcal{X} \cup \mathit{vars}(k'))$ such that $k' = \theta(k)$.

It might seem that maximal sharing is always desirable; however maximal sharing does not always model the intended behavior. Consider the rewrite rule $a\ * \to b\ *$, where a, b, and $*$ are constants of sort s and “$\_\ \_ : ss \to s$” is an associative and commutative (AC) operation composing elements of sort s. If the desired behavior is that $*$ is a catalyst allowing the transformation from a to b to happen, and therefore we want it to be potentially read by other rules, then the corresponding $\mathbb{K}$ rule is $\underset{b}{\underline{a}}\ *$; as seen shortly, such a rule allows a term $a\ a\ *$ to rewrite in one concurrent step to $b\ b\ *$. However, if $*$ is to be regarded as a token ensuring mutual exclusion, then the corresponding $\mathbb{K}$ rule is $\underset{b\ *}{\underline{a\ *}}$; as seen shortly, such rules cannot be applied concurrently on $a\ a\ *$, requiring, due to the overlapping of the token "$*$", two interleaved steps.

The remainder of this section gives a theoretical account of the $\mathbb{K}$ rewriting method, arguing that the read-only information contained in $\mathbb{K}$ rules can be used to enhance the potential of parallelism of rewriting.

6.2.3 Matching $\mathbb{K}$ Rules

One could give a straightforward definition for what it means for a $\mathbb{K}$ rule to match a term: *one* $\mathbb{K}$ rule matches a term if and only if its corresponding rewrite rule matches it:

Definition 2. *A $\mathbb{K}$ rule $\rho : (\forall \mathcal{X})\ k[\ \underset{R}{\underline{L}}\ ]$* ***matches*** *a Σ-term t using context C and substitution θ if and only if its corresponding rewrite rule $K2R(\rho)$ matches term t using the same context C and substitution θ, that is, if $t = C[\theta(L(k))]$.*

Hence, when analyzed in isolation, a $\mathbb{K}$ rule is no more special than its corresponding rewrite rule. Only when trying to apply rules in parallel we can observe the benefits of the $\mathbb{K}$ rules.

As seen in Chapter 4, the $\mathbb{K}$ framework employs lists, sets, bags, and maps to give semantics. Moreover, it is precisely the bag structure of the configuration that allows $\mathbb{K}$ to give truly concurrent semantics to languages as seen in most of the examples presented in this dissertation. However, concurrent matching modulo (ACU) axioms in the presence of sharing appears to be a rather difficult problem. Recall the motivating example from the beginning of this chapter, in which we attempted to concurrently read and write in the state. The structure of $\mathbb{K}$ rules introduced above, explicitly representing read-only parts of the rules, allows multiple rules to overlap on the $\langle\rangle_{\mathsf{store}}$ cell at the same time. However, the problem of matching multiple rules simultaneously still remains, because the traditional treatment of matching modulo axioms requires the term to be rearranged (using the axioms) to conform to the rule, and thus it makes unclear what does it mean for multiple rules to actually match at the same time. The remainder part of this subsection shows how concurrency can be enhanced by handling matching modulo (ACU) axioms differently.

The main idea is to think of the matching process as *changing the rule to fit the term rather than changing the term to fit the rule*; in that sense, rewrite rules become rule schemata. Our goal is to modify the rule to match a concrete representation of the term. The reason for requiring adjustments to the rule is that while the term is concrete, and, for example, each list constructor has only two arguments, the rule is specified modulo axioms.

Assume a generic $\mathbb{K}$ rule $\rho :\ (\forall\mathcal{X})\ k[\underset{R}{\underline{L}}]$. In what follows we describe a sequence of steps, which, if applied in order, generate all concrete instances of ρ needed to replace matching modulo (ACU) axioms by plain term matching. Before going into more technical details, let us briefly describe each step. First step deals with the unit axiom, generating additional rules to account for variables being matched to the unit of an operation. Second step prepares the terrain for dealing with associativity; each variable which could stand for a term topped in an associative operation is replaced by an arbitrary number of variables separated by that operation. In step 3 we deal with commutativity, by generating rule instances for all permutations of arguments of commutative operations. Finally, in step 4, we deal with the associativity axiom, properly parenthesizing all parts of the rule containing associative operations. Note that, although both steps 2 and 4 deal with associativity, steps 3 needs to be inserted between them to generate all permutations needed for the AC operations.

This generative process of generating all matching instances for rules serves only for a theoretical purpose, as it actually generates an infinite number of concrete rule instances. In a practical implementation of these ideas, we expect that the rule schema would dynamically be adjusted in the process of matching, creating concrete rule instances by-need; additionally, an implementation using graph rewriting might choose appropriate representations for lists and sets which would allow matching inside without having to specify the list/set context.

1. Resolving Unit We assume that the concrete terms to be matched are always kept in normal form w.r.t. the unit axioms, that is, the unit $\dagger$ of an operation $\star$ cannot appear as an argument of that operation. This can be obtained by either reducing the term after each rewriting step, or by reducing it only once at the beginning, and ensuring that the rewrite steps preserve this property. Assuming this, to address matching modulo unit it is sufficient then that for each variable x of the same sort as $\dagger$ appearing as an argument of operation $\star$, say in a subterm $\star(x, p)$, we generate an additional rule in which $\star(x, p)$ is replaced by p (and similarly if x is the second argument of $\star$). Moreover, if x appears at the top of a replacement term $R(\square)$, then $R(\square)$ must be $\dagger$ in the additional generated rule. As a matching example, for the pattern $\langle \sigma\ \ l \mapsto i\rangle_{\mathsf{state}}$ we need to generate an additional pattern $\langle l \mapsto i\rangle_{\mathsf{state}}$ since σ could match $\cdot$, the unit of $\_\ \_$.

2. Multiplying associative variables For simplicity, we assume that each sort has at most one associative operator defined on it; our definitions satisfy that—in fact, besides the K sort, which itself is a list, all other sorts with associative operators allowed by $\mathbb{K}$ are lists, bags, sets, and maps. Moreover, we will assume that all rules topped in an associative operator $\star$ of sort S have by default two (or only one if $\star$ is also commutative) anonymous variables of sort S at the top of the rule, one on each side of the read-only pattern, to account for the fact that the rule may match in a context. The associativity will be resolved in two steps. The first step, described here, is that for each rule containing a variable xss of a sort S constructed with an associative operator $\_\star\_$ and for each natural number $n \geq 2$, a rule in which xs is replaced by $xs_1 \star xs_2 \star \cdots \star xs_n$ (where xs_i is a fresh variable for each i) must be added to the existing rules. Continuing our example above, the matching pattern instances associated to $\langle \sigma \ \ l \mapsto i \rangle_{\mathsf{state}}$ would now be (including the one from desugaring the unit axiom): $\langle l \mapsto i \rangle_{\mathsf{state}}$, $\langle \sigma \ \ l \mapsto i \rangle_{\mathsf{state}}$, $\langle \sigma_1 \ \ \sigma_2 \ \ l \mapsto i \rangle_{\mathsf{state}}$, and so on.

3. Resolving Commutativity For each occurrence of a subterm $\star(t_1, t_2)$ in a rule, with $\star$ being commutative, add (if it doesn't already exist) a rule in which $\star(t_1, t_2)$ is replaced by $\star(t_2, t_1)$, effectively generating all permutations for terms built with AC operators. The patterns above are enriched to the following patterns: $\langle l \mapsto i \rangle_{\mathsf{state}}$, $\langle \sigma \ \ l \mapsto i \rangle_{\mathsf{state}}$, $\langle l \mapsto i \ \ \sigma \rangle_{\mathsf{state}}$, $\langle \sigma_1 \ \ \sigma_2 \ \ l \mapsto i \rangle_{\mathsf{state}}$, $\langle \sigma_1 \ \ l \mapsto i \ \ \sigma_2 \rangle_{\mathsf{state}}$, $\langle l \mapsto i \ \ \sigma_1 \ \ \sigma_2 \rangle_{\mathsf{state}}$ (and the ones equivalent to them modulo renamings of the fresh variables), and so on.

Next step is only needed if associative operators are handled as ordinary binary operations when representing the term. If, for example, associative operations are represented as operations with a variable number of arguments this step may be skipped. However, we here prefer to keep this step in order to preserve the algebraic structure of the terms.

4. Resolving Associativity For each rule containing subterms of the form $t_1 \star t_2 \star \cdots \star t_n$ where $\star$ is an associative operator, generate rules containing all possible ways to put parentheses so that each occurrence of $\_\star\_$ has only two arguments. Note that matching terms containing subterms built from new variables using only $\star$ these rules need not be considered, as they will be equivalent with rules containing just one new variable instead of that subterm. Keeping this in mind, the following patterns are the final concrete patterns associated to the ones presented above: $\langle l \mapsto i \rangle_{\mathsf{state}}$, $\langle \sigma \ \ l \mapsto i \rangle_{\mathsf{state}}$, $\langle l \mapsto i \ \ \sigma \rangle_{\mathsf{state}}$, $\langle \sigma_1 \ \ (\sigma_2 \ \ l \mapsto i) \rangle_{\mathsf{state}}$, $\langle (\sigma_1 \ \ l \mapsto i) \ \ \sigma_2 \rangle_{\mathsf{state}}$, $\langle \sigma_1 \ \ (l \mapsto i \ \ \sigma_2) \rangle_{\mathsf{state}}$, and $\langle (l \mapsto i \ \ \sigma_1) \ \ \sigma_2 \rangle_{\mathsf{state}}$, and so on.

Note that, since now parts of the original variables might be grouped together with other parts of the matching pattern, parenthesizing makes virtually impossible to rewrite those variables, or even associative operators, unless the entire list is being rewritten. Therefore, we require that for each sort S containing an associative operation $\star$, and any variable xs of sort S, whenever $\star$ or xs appears at top in a term to be replaced, i.e., $t = L[\square]$, $\square$ must not be an argument of $\star$ in the read-only pattern p. The restriction concerning xs may indeed inhibit parallelism when rewriting list variables. However, this situation does not seem to be very common in practice; in particular, it does not appear in any of our current definitions using $\mathbb{K}$. This restriction concerning $\star$ can be satisfied in two ways. The first one is to push the hole $\square$ up in the term as long as $\star$ operations are on top of it, and thus inhibit the parallelism. The second, is to push the hole down, by moving $\star$ into the pattern, and splitting $\square$ into two new holes, requiring L to map them to the two arguments of $\star$, and updating R accordingly (including the possibility that R maps one of the holes to $\cdot$, while the other to $R(\square)$).

Example: Matching the state read/write rules Let us now show that interpreting $\mathbb{K}$ rules as rule schemata as described above allows multiple concurrent matching instances "modulo" associativity and commutativity. Recall the two rules defining store access for the Imp++ language, now with context transformations being applied on them, and naming the variables:

$$\rho_r : \langle ts \langle t \langle \underset{i}{\underline{*l}} \curvearrowright k \rangle_{\mathsf{k}} \rangle_{\mathsf{thread}} \rangle_{\mathsf{threads}} \ \langle \sigma \ \ l \mapsto i \rangle_{\mathsf{state}}$$

$$\text{and } \rho_w : \langle ts \langle t \langle \underset{i'}{\underline{*l = i'}} \curvearrowright k \rangle_{\mathsf{k}} \rangle_{\mathsf{thread}} \rangle_{\mathsf{threads}} \ \langle \sigma \ \ l \mapsto \underset{i'}{\underline{i}} \rangle_{\mathsf{state}}$$

Recall also the ground termed to be matched, this time parenthesized:

$$\left\langle \left\langle \begin{array}{c} (\langle\langle *1 \rangle_{\mathsf{k}} \ \ \langle \cdot \rangle_{\mathsf{env}} \rangle_{\mathsf{thread}} \ \ \langle\langle *1 \rangle_{\mathsf{k}} \ \ \langle \cdot \rangle_{\mathsf{env}} \rangle_{\mathsf{thread}}) \\ \langle\langle *2 = 3 \rangle_{\mathsf{k}} \ \ \langle \cdot \rangle_{\mathsf{env}} \rangle_{\mathsf{thread}} \\ \langle 1 \mapsto 1 \ \ 2 \mapsto 2 \rangle_{\mathsf{state}} \end{array} \right\rangle_{\mathsf{threads}} \right\rangle_{\mathsf{T}},$$

Then, the three concretizations of the 2 schema rules above which can match this term are:

$$\rho_{r,1} : \langle ((\langle\langle \underset{i}{\underline{*l}} \rangle_{\mathsf{k}} \ t_1 \rangle_{\mathsf{thread}} \ ts_1) ts_2 \rangle_{\mathsf{threads}} \ \langle l \mapsto i \ \ \sigma \rangle_{\mathsf{state}}$$

$$\rho_{r,2} : \langle (ts_1 \ \ \langle\langle \underset{i}{\underline{*l}} \rangle_{\mathsf{k}} \ t_1 \rangle_{\mathsf{thread}}) ts_2 \rangle_{\mathsf{threads}} \ \langle l \mapsto i \ \ \sigma \rangle_{\mathsf{state}}$$

$$\rho_{w,1} : \langle ts_1 \ \ \langle\langle \underset{i}{\underline{*l = i'}} \rangle_{\mathsf{k}} \ t_1 \rangle_{\mathsf{thread}} \rangle_{\mathsf{threads}} \ \langle \sigma \ \ l \mapsto \underset{i'}{\underline{i}} \rangle_{\mathsf{state}}$$

6.2.4 $\mathbb{K}$ Concurrent Rewriting—Intuition

In Section 6.2.1 we claimed that $\mathbb{K}$ rules can achieve more locality than usual rewrite-rules. Indeed, by allowing rules to share their read-only pattern, parallel rewriting using $\mathbb{K}$ rules can capture more concurrent computations than directly achievable by applying rewriting logic deduction using the rewrite rules obtained by ignoring the sharing of information. In this section we intuitively describe how a term is matched by multiple rules, and indicate how all of the matched rules can be applied concurrently. We will formalize this intuitive description through an embedding in graph rewriting in Section 6.3.

As described above, the intuition in a $\mathbb{K}$ rule $(\forall \mathcal{X})\ k[\ \underset{R}{\underline{L}}\ ]$ is that k represents a read-only pattern, which can be shared with other rules, while the terms given by

L should be regarded as read-write, because no two rules should simultaneously modify the same part, and no rule should read it while another rule writes it.

We next give a visual intuition for combining multiple instances of $\mathbb{K}$ rules. Assume that the term to be rewritten is initially uncolored (or black), that is, all its positions are available to all rules. Whenever a $\mathbb{K}$ rule matches, it colors the matched part of the term using two colors, green for the read-only pattern, which can be shared, and red for the read-write parts, so that they would not be touched by any other rule. When combining multiple matches concurrently, the following natural coloring policies apply:

1. Uncolored, or black, can be colored in any color by any $\mathbb{K}$ rule;

2. No rule is allowed to color (green or red) a position which is already red;

3. Once a position is green, it cannot be colored in red by any $\mathbb{K}$ rule.

In short, "red cannot be repainted and green can only be repainted green."

The first policy says that unconstrained parts of the term can be safely matched, in order to be rewritten, by any $\mathbb{K}$ rule. The second policy says that a part of a term which is being written by some rule cannot be read or written by any other rule. Finally, the third policy says that a part of a term that is being read by some rule can be read but not written by other rules.

Analyzing this coloring through the resemblance between $\mathbb{K}$ rules and graph-rewriting rules, one can notice that the policy imposed by the above coloring rules is in direct correspondence with the notion of parallel independence [44] of rules applications in graph rewriting, in the sense that the rule instances are allowed to overlap on their patterns, but not on anything else.

Let us conclude this section by showing how we can finally achieve the goal of concurrently advancing all threads from the motivating example at the beginning of this section. To do that, we use the coloring policies described above to combine the concrete matching rules associated to rule schemata ρ_r and ρ_w from the end of previous subsection. Let us start with the original term

$$\left\langle \left\langle \begin{array}{c} (\langle\langle *1\rangle_{\mathsf{k}}\ \ \langle\cdot\rangle_{\mathsf{env}}\rangle_{\mathsf{thread}}\ \ \langle\langle *1\rangle_{\mathsf{k}}\ \ \langle\cdot\rangle_{\mathsf{env}}\rangle_{\mathsf{thread}}) \\ \langle\langle *2 = 3\rangle_{\mathsf{k}}\ \ \langle\cdot\rangle_{\mathsf{env}}\rangle_{\mathsf{thread}} \\ \langle 1 \mapsto 1\ \ 2 \mapsto 2\rangle_{\mathsf{state}} \end{array} \right\rangle_{\mathsf{threads}} \right\rangle_{\top},$$

and let us match rule $\rho_{r,2}$ first, yielding the coloring:

$$\left\langle \left\langle \begin{array}{c} (\langle\langle *1\rangle_{\mathsf{k}}\ \ \langle\cdot\rangle_{\mathsf{env}}\rangle_{\mathsf{thread}}\ \ \langle\langle *1\rangle_{\mathsf{k}}\ \ \langle\cdot\rangle_{\mathsf{env}}\rangle_{\mathsf{thread}}) \\ \langle\langle *2 = 3\rangle_{\mathsf{k}}\ \ \langle\cdot\rangle_{\mathsf{env}}\rangle_{\mathsf{thread}} \\ \langle 1 \mapsto 1\ \ 2 \mapsto 2\rangle_{\mathsf{state}} \end{array} \right\rangle_{\mathsf{threads}} \right\rangle_{\top},$$

Rule $\rho_{r,1}$ also matches since the colors are consistent, and colors the first continuation cell in green and the a inside it in red:

$$\left\langle \begin{array}{c} \left\langle \begin{array}{c} (\langle\langle *1\rangle_{\mathsf{k}}\ \ \langle\cdot\rangle_{\mathsf{env}}\rangle_{\mathsf{thread}}\ \ \langle\langle *1\rangle_{\mathsf{k}}\ \ \langle\cdot\rangle_{\mathsf{env}}\rangle_{\mathsf{thread}}) \\ \langle\langle *2 = 3\rangle_{\mathsf{k}}\ \ \langle\cdot\rangle_{\mathsf{env}}\rangle_{\mathsf{thread}} \end{array} \right\rangle_{\mathsf{threads}} \\ \langle 1 \mapsto 1\ \ 2 \mapsto 2\rangle_{\mathsf{state}} \end{array} \right\rangle_{\top},$$

Finally, rule $\rho_{w,1}$ can also match without conflicts:

$$\left\langle \begin{array}{c} \left\langle \begin{array}{c} (\langle\langle *1\rangle_{\mathsf{k}}\ \ \langle\cdot\rangle_{\mathsf{env}}\rangle_{\mathsf{thread}}\ \ \langle\langle *1\rangle_{\mathsf{k}}\ \ \langle\cdot\rangle_{\mathsf{env}}\rangle_{\mathsf{thread}}) \\ \langle\langle *2 = 3\rangle_{\mathsf{k}}\ \ \langle\cdot\rangle_{\mathsf{env}}\rangle_{\mathsf{thread}} \end{array} \right\rangle_{\mathsf{threads}} \\ \langle 1 \mapsto 1\ \ 2 \mapsto 2\rangle_{\mathsf{state}} \end{array} \right\rangle_{\top},$$

Since all matches succeeded, we can apply all three rules simultaneously, obtaining the term

$$\left\langle \begin{array}{c} \left\langle \begin{array}{c} (\langle\langle 1\rangle_{\mathsf{k}}\ \ \langle\cdot\rangle_{\mathsf{env}}\rangle_{\mathsf{thread}}\ \ \langle\langle 1\rangle_{\mathsf{k}}\ \ \langle\cdot\rangle_{\mathsf{env}}\rangle_{\mathsf{thread}}) \\ \langle\langle 3\rangle_{\mathsf{k}}\ \ \langle\cdot\rangle_{\mathsf{env}}\rangle_{\mathsf{thread}} \end{array} \right\rangle_{\mathsf{threads}} \\ \langle 1 \mapsto 1\ \ 2 \mapsto 3\rangle_{\mathsf{state}} \end{array} \right\rangle_{\top},$$

In the subsequent sections we formalize this intuition of $\mathbb{K}$ term rewriting through an embedding into graph rewriting theory. The reasons for our choice are: (1) (term) graph rewriting [12, 68, 138] was shown to be sound and complete for term rewriting, which we want to preserve for $\mathbb{K}$ rewriting; (2) the intuition that the pattern k of a $\mathbb{K}$-rule is meant to be "shared" with competing concurrent rule instances is conceptually captured by the notion of interface graphs of graph rewrite rules in the DPO (double-pushout) algebraic approach to graph rewriting [44, 39]; and (3) the results in the DPO theory of graph rewriting showing that if graph rule instances only overlap on the interface graphs, then they can be concurrently applied and the obtained rewrite step is serializable [45, 93, 69], which precisely matches the intuition for $\mathbb{K}$ rewriting described above.

6.3 $\mathbb{K}$ Graph Rewriting

Although the theory of graph rewriting has early on shown the potential for parallelism with sharing of context, the existing term-graph rewriting approaches aim at efficiency: rewrite common subterms only once. More specifically, they do not attempt to use the context-sharing information for enhancing the potential for concurrency, as we want to do in $\mathbb{K}$ rewriting. Consequently, the one-step concurrency achieved by current term-graph rewriting approaches is no better than that of standard rewriting, or even worse if we consider that subterm sharing can inhibit behaviors.

As our interests fall at the convergence of term-graph rewriting (for being sound and complete w.r.t. term rewriting) and the DPO approach to graph rewriting

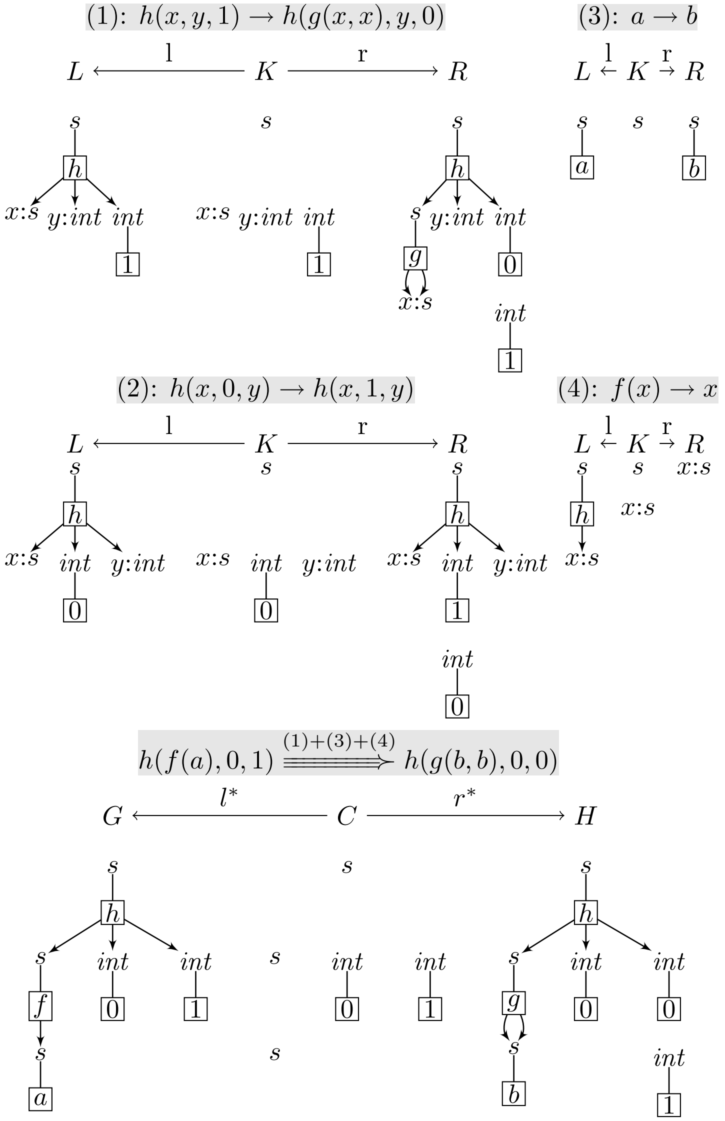

(1): $h(x, y, 1) \to h(g(x, x), y, 0)$
$L \xleftarrow{l} K \xrightarrow{r} R$
s
h
x:s y:int int
1
g
0
int
(3): $a \to b$
$L \xleftarrow{l} K \xrightarrow{r} R$
a
b
(2): $h(x, 0, y) \to h(x, 1, y)$
x:s int y:int
(4): $f(x) \to x$
$h(f(a), 0, 1) \xRightarrow{(1)+(3)+(4)} h(g(b, b), 0, 0)$
$G \xleftarrow{l^*} C \xrightarrow{r^*} H$
f

Figure 6.2: Jungle representations for the rewrite rules corresponding to $\mathbb{K}$ rules (1)–(4) from the motivating example and a possible concurrent step.

(for concurrency with sharing of context), the subsequent graph embedding of $\mathbb{K}$ rewriting can be seen as an extension (enhancing the concurrency, while conserving soundness and completeness) of the jungle hypergraph rewriting [68, 38] incarnation of term-graph rewriting.

In what follows, we assume that the notions regarding graph rewriting and jungle evaluation, as presented in Sections 2.2 and 2.3 are known. For example, Figure 6.2 presents the jungle rules representing the rewrite rules (1)-(4) from our running example.

$\mathbb{K}$ *graph rewriting* uses the same mechanisms and intuitions of jungle rewriting, but relaxes the definitions of both graph jungles and graph evaluation rules to increase the potential for concurrency in the case of context sharing.

The relaxation at the level of rules is that, similarly to the original definition of jungle rules [68], instead of practically removing the entire left-hand-side of an evaluation rule during the evaluation step (by sectioning the root of L from the rest in K), $\mathbb{K}$ *graph rewrite rules* allow more of the local context (precisely, the k part of a $\mathbb{K}$ rule) to be preserved by a rule, and thus potentially allow other rules to share it for parallel rewriting. However, departing from the definition of jungle rules, we relax the requirement that the order between the nodes of K and variables of R should be the same as in L, to allow rules such as reading or writing the value of a variable from a store.

$\mathbb{K}$ *term-graphs* are closely related to the graph jungles—they actually coincide for ground terms. The difference is that the $\mathbb{K}$ term-graph representation allows certain variables (the anonymous and the pattern-hole variables) to be omitted from the graph. By reducing the number of nodes that need to be shared (i.e., by not forcing these variable nodes to be shared in the interface graph), this "partiality" allows terms at those positions to be concurrently rewritten by other rules.

6.3.1 $\mathbb{K}$ Term-Graphs

The top-half of Figure 6.3 shows the $\mathbb{K}$ term-graphs involved in the graph representations of the $\mathbb{K}$ rules (1)–(4) of our running example. For example the representation of variable x can be observed as the (singleton) graph R for rule (4), the constants a and b as graphs L and R from rule (3), and the term $f(x)$ as graph L in rule (4); all these $\mathbb{K}$ term-graphs are also graph jungles. The bottom-half of Figure 6.3 shows the $\mathbb{K}$ term-graphs involved in the graph transformation which uses all four rules combined to rewrite the graph representation of $h(f(a), 0, 1)$ (graph G) to one that can be used to retrieve $h(g(b, b), 1, 0)$ (graph H).

The novel aspect of our representation is that, unlike the graph jungles, the $\mathbb{K}$ term-graphs are *partial*: they do not require each operation node to have outward edges for all sorts in its arity. This partiality plays a key role in "abstracting away" the anonymous variables and the holes of the pattern. For example, the number of outward edges specified for the nodes labeled with h have all possible values between 3 (its normal arity) in graphs G and H, to 0, e.g., in graph K for rule (1). This flexibility is crucial for enhancing concurrency; only through it rules (1) and (2) can apply in parallel, as it allows the outward edge of h labeled with 1 to be rewritten by rule (1), while h is still shared with rule (2). This is achieved by relaxing the 1.(ii) property of Proposition 1 to allow partially specified operations. For self-containedness reasons, we write the entire definition, but follow the same structure as in Proposition 1.

Definition 3. *Given a signature $\Sigma = (S, F)$, a $\mathbb{K}$ Σ-**term-graph** is a graph G over labels $(S \cup F, \{\epsilon\} \cup Nat)$ satisfying the following:*

0. *G is bipartite, partitions given by nodes with labels in S—**sort nodes**—, and F—**operation nodes**—;*

1. *every operation node labeled by $f : s_1 \cdots s_n \to s$ is*

   (i) the target of exactly one edge, labeled with 0 and having its source labeled with s, and

   *(ii) the source of **at most** n edges having distinct labels in $\{1, \cdots, n\}$, such that $\mathrm{lv}(\mathrm{target}(e)) = s_{\mathrm{le}(e)}$ for each such edge e;*

2. *every sort node has at most one outward edge; and*

3. *G is acyclic.*

Let $\mathbf{KGraph}_{\boldsymbol{\Sigma}}$ denote the full subcategory of $\mathbf{Graph}(\mathbf{S} \cup \mathbf{F}, \{\epsilon\} \cup Nat)$ having $\mathbb{K}$ Σ-term-graphs as objects.

Note that any graph jungle is a $\mathbb{K}$ term-graph. In the sequel, for notational simplicity $\mathbb{K}$ term-graphs will be referred to as just term-graphs. Therefore, most of the definitions from graph jungles can be easily extended for term-graphs.

Given a set of anonymous variables $A \subseteq X$, an A-anonymizing variable-collapsed tree representing of a term $t \notin A$ with variables from X is obtained from a variable-collapsed tree representing t by removing the variable nodes corresponding to variables in A and their adjacent edges.

The root nodes of a term-graph G, $ROOT_G$ are no different than for graph jungles; however, VAR_G now only captures the non-anonymous variables. To capture all variables, we need to additionally identify partially specified operation nodes.

Open, and variable nodes. Let G be a term-graph over $\Sigma = (S, F)$. The set $OPEN_G$ of *open (or incomplete) operation nodes* of G, consists of the operation nodes whose outward edges are incompletely specified. Formally, $OPEN_G = \{v \in \mathrm{lv}^{-1}(S) \mid |s^{-1}(v)| < \mathit{arity}(\mathrm{lv}(v))\}$. The set of *term variables* of G, $TVARS_G$ consists from the variables of G and the positions of the unspecified outward edges for open operation nodes (which stand for anonymous variables). Formally, $TVARS_G = VAR_G \cup \{x_{v,i} \mid v \in OPEN_G, 1 \leq i \leq \mathit{arity}(\mathrm{lv}(v)) \wedge i \notin \mathrm{le}(\mathrm{source}^{-1}(v))\}$.

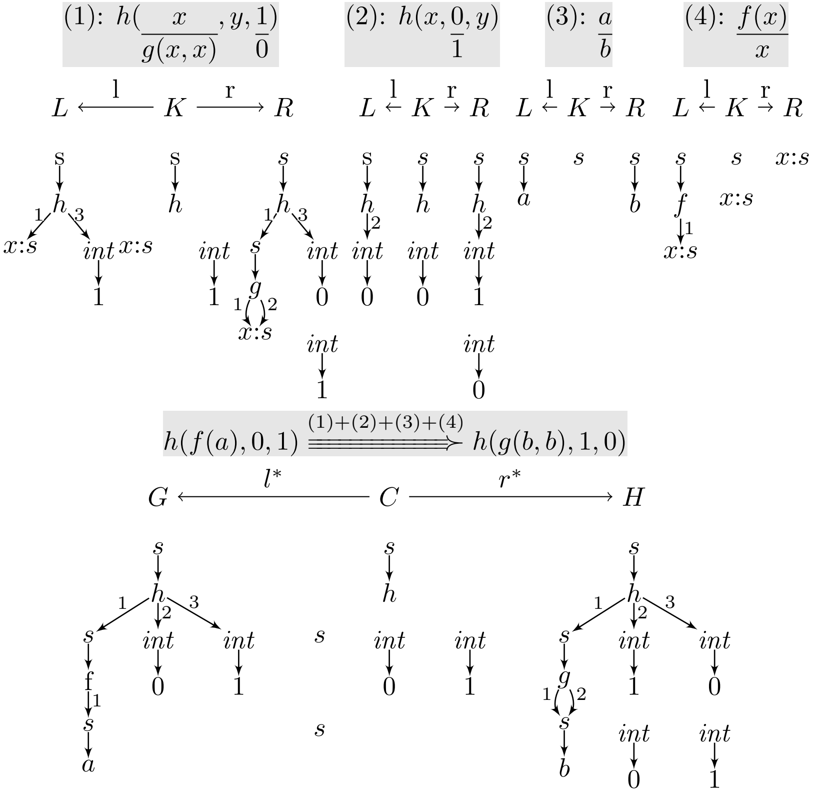


Figure 6.3: Graph representations for the 𝕂 rules (1)–(4) from the motivating example and their concurrent application.

To account for the anonymous variables, the definition of *term* changes as follows:

$$term_G(v_s) = \begin{cases} v_s, \text{ if } v_s \in VAR_G \\ \sigma(t_1, \ldots, t_n), \text{ if } \{v_e\} = \text{target}(\text{source}^{-1}(v_s)), \\ \qquad \text{le}(v_e) = \sigma : s_1 \ldots s_n \to s, \text{ and} \\ \qquad t_i = subterm_G(v_e, i) \text{ for any } 1 \leq i \leq n \end{cases}$$

where $subterm_G$ is defined by on pairs of operation nodes with integers by

$$subterm_G(v_e, i) = \begin{cases} x_{v_e,i}, \text{ if } x_{v_e,i} \in TVARS_G \\ term_G(\text{target}(e)), \text{ if source}(e) = v_e \text{ and le}(e) = i \end{cases}$$

6.3.2 From $\mathbb{K}$ Rules to Graph Rewrite Rules

As we want $\mathbb{K}$ graph rewriting to be a conservative extension of graph jungle evaluation, every 0-sharing $\mathbb{K}$ rule $(\forall\mathcal{X})\ \Box[\underset{right}{\underline{left}}]$ is encoded as the graph jungle evaluation rule corresponding to the rewrite rule $left \to right$—see, for example the encodings of rules (3) and (4) in Figure 6.3. However, if the local context k is non-empty, then the rule is encoded so that the variable-collapsed tree representing k would not be modified by the rule. To be more precise, instead of obtaining K by removing the outgoing edge from the root of L, we will instead only remove the edges connecting the hole variables to their parent operations. Moreover, to further increase concurrency, the variables which appear in the read only pattern k but not in the left substitution are anonymized.

Let us discuss the representation of the $\mathbb{K}$-rule (1) in Figure 6.3, namely $h(\underset{g(x,x)}{\underline{x}}, y, \underset{0}{\underline{1}})$. The left-hand-side is represented as a $\{y\}$-anonymized variable collapsed tree representing $h(x, y, 1)$; variable y is anonymized as only appearing in the pattern k. The interface K is obtained from L by severing (through the removal of edges labeled by 1 and 3) the part of L representing the read-only pattern $h(\Box_1, y, \Box_2)$ (which is the $\{y, \Box_1, \Box_2\}$-anonymized variable collapsed tree representing $h(\Box_1, y, \Box_2)$) from the parts of L representing the left substitution (namely, x and 1). Thus, the l morphism from K to L is clearly an inclusion. R is obtained by taking the disjoint union between K and the variable-collapsed trees corresponding to terms $g(x, x)$ and 0 given by the right substitution, identifying the variables, and ”gluing” them to the part representing the read-only pattern through edges from operation node h labeled 1 and 3, respectively. Similarly as for the l morphism, the morphism r can also be chosen to be an inclusion.

The graph rules in Figure 6.3 are obtained using the definition below. To avoid clutter, we do not depict node or edge names (except for variables). Also, the actual morphisms are not drawn (they are either inclusions or obvious collapsing morphisms).

Definition 4. *Let $\rho :(\forall\mathcal{X})\ k[\ L \Rightarrow R\ ]$ be a $\mathbb{K}$ rewrite rule.*

If ρ is 0-sharing, then the $\mathbb{K}$ graph rewrite rules representing ρ coincide with the graph evaluation rules corresponding to the rewrite rule associated to ρ.

*Otherwise, a $\mathbb{K}$ **graph rewrite rule** representing ρ is a graph rewrite rule $(L_\rho \xleftarrow{l_\rho} K_\rho \xrightarrow{r_\rho} R_\rho)$ such that:*

L_ρ *is a A-anonymized variable collapsed tree representation of $L(k)$, where $A = vars(k) \setminus vars(L)$ are the anonymous variables of ρ;*

K_ρ. *Let K_0 be the subgraph of L_ρ which is a A-anonymized variable collapsed tree representing k; then $K_\rho = (V_{K_\rho}, E_{K_\rho})$ is given by $V_{K_\rho} = V_{L_\rho}$ and $E_{K_\rho} = E_{L_\rho} \setminus \{e \in E_{L_\rho} \mid \mathrm{source}(e) \in V_{K_0}$ and $\mathrm{target}(e) \notin V_{K_0}\}$. l_ρ is the inclusion morphism.*

R_ρ *Let R_0 be an A-anonymized variable collapsed tree representation of $R(k)$ containing K_0 as a subgraph. Then R_ρ is obtained as the pushout between the inclusions of $K_0 \cup VAR_{R_0}$ into K_ρ and R_0, respectively.*

The nodes from K_0 will be called pattern nodes.

Note that the edges removed from L_ρ to obtain K_ρ are those whose target corresponds to the hole variables of k.

Similarly to the graph jungle rules, the (basic) $\mathbb{K}$ graph rules defined above ensure that the gluing conditions are satisfied for any matching morphism. For the remainder of this section, let us fix G to be a term-graph, $\rho_i : (L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i)$, $i = \overline{1,n}$ to be $\mathbb{K}$ graph-rewrite rules, and $m_i : L_i \to G$ to be parallel independent matches. Let $\rho : (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ be the composed rule of $(\rho_i)_{i=\overline{1,n}}$, and let $m : L \to G$ be the composition of the individual matches. It follows that m satisfies the gluing conditions for ρ, and thus (ρ, m) can be applied as a graph transformation. Let us now provide a concrete construction for the derivation of (ρ, m) in **Graph** which will be used in proving the subsequent results.

The pushout complement object of m and l can be defined in **Graph** as $C = G \setminus m(L \setminus K)$ where the difference is taken component-wise. That C is a graph is ensured by the gluing conditions. The standard construction of the pushout object H is to factor the disjoint union of C and R through the equivalence induced by the pushout morphism $\overline{m} : K \to C$ and r. We do this directly, by taking preference for elements in C, and thus choosing representatives from $\overline{m}(K)$ and by choosing as representatives variables for the equivalence classes induced by the parts of r belonging to collapsing rules.

Let us now state some facts given by the structure of $\mathbb{K}$ graph rewrite rules. Let $root_i$ identify the root of L_i in L. Since the lhs cannot be a variable, it follows that L_i has at least one edge and one operation node. K_i is a subgraph of L_i and l_i is the inclusion morphism; moreover K_i contains all nodes of L_i.

We have that $ROOT_L = \{root_i \mid i = \overline{1,n}\}$. Let now J be the set of indexes of collapsing rules. In the following, let i range over $\{1, \ldots, n\}$ and let j range over J.

We define H, together with $r^* : C \to H$ and $m^* : R \to H$, as follows:

- $V_H = (V_C \setminus \{m(root_j) \mid j \in J\}) \uplus (V_R \setminus V_K)$
- $r^*_V(v) = \begin{cases} v,\text{if } v \neq m(root_j), \\ r^*_V(m(r(root_j))),\text{if } v = m(root_j), \end{cases}$
- $m^*_V(v) = \begin{cases} v,\text{if } v \notin V_K \\ r^*_V(m_V(v)),\text{otherwise} \end{cases}$
- $E_H = E_C \uplus (E_R \setminus E_K)$
- $r^*_E(e) = e$ and $m^*_E(e) = \begin{cases} e,\text{if } e \notin E_K \\ m_E(e),\text{otherwise} \end{cases}$
- $\text{source}_H(e) = \begin{cases} r^*(\text{source}_C(e)),\text{if } e \in E_C \\ m^*_V(\text{source}_R(e)),\text{if } e \in E_R \setminus E_K \end{cases}$
- $\text{target}_H(e) = \begin{cases} r^*_V(\text{target}_C(e)),\text{if } e \in E_C \\ m^*_V(\text{target}_R(e)),\text{if } e \in E_R \setminus E_K \end{cases}$

Note that r^*_V is recursively defined. However, it is well defined, because G is acyclic and, since $r_V(root_j) \in VAR_{L_j}$, it must be that $G\restriction_{m_V(r_V(root_j))}$ is a strict subgraph of $G\restriction_{m_V(root_j)}$, implying that the recursion should end because both G and J are finite. It can be easily verified that (H, r^*, m^*) is a pushout of $(\overline{m}, r)$.

Suppose G is a $\mathbb{K}$ graph representation of term t, i.e., that $ROOT_G = \{root_G\}$, $G = G\restriction_{root_G}$, and $term_G(root_G) = t$. When applying a (composed, or not) $\mathbb{K}$ graph rewrite rule to graph G, $root_G$ must be preserved in the context C, because K contains all nodes of L. Therefore, let us define the top of the obtained graph H as being $root_H = r^*(root_G)$. Note that $root_H$ might not be equal to $root_G$, because $root_G$ could be identified with a variable node by a collapsing rule; moreover, $root_H$ might not be the only element of $ROOT_H$, because of the potential "junk" left by the application of the rule. Nevertheless, the term $term_H(root_H)$ would be the one to which $term_G(root_G)$ was rewritten.

6.3.3 Applying $\mathbb{K}$ Rules as Graph Rules

To show that $\mathbf{KGraph_\Sigma}$ admits similar constructions for (composed) $\mathbb{K}$ graph-rewrite rules as **Graph**, that is, that the graphs described above are in fact term-graphs, we need to strengthen the constraints on the matching morphisms.

Indeed, without further constraints, applying $\mathbb{K}$ graph rules on term-graphs can produce cyclic graphs. Take for example, the graph G in Figure 6.4(a), representing the term $f(h(b), h(b))$. Upon applying the $\mathbb{K}$ graph rule corresponding to $f(\underset{a}{\underline{x}}, h(\underset{x}{\underline{b}}))$, we obtain a cyclic graph depicted as graph H in Figure 6.4(b).

One could validly argue that this problem arose because graph G was not a tree; however, the example, depicted in Figure 6.4(b), shows that it is possible that after applying a composed $\mathbb{K}$ graph-rewrite rule on a completely non-collapsed term-graph using a match whose components satisfy the parallel independence property, the graph obtained (we are guaranteed to obtain one) may not be a term-graph. Consider $\mathbb{K}$ rules $f(g(\underset{x}{\underline{a}}), x)$ and $f(y, h(\underset{y}{\underline{b}}))$ discussed at the beginning of this chapter, together with the term to rewrite $f(g(a), h(b))$. Upon formalizing terms as term-graphs and $\mathbb{K}$ rules as $\mathbb{K}$ graph rewrite rules, the result of applying the composed $\mathbb{K}$ graph rewrite rule on the graph representing $f(g(a), h(b))$ is the graph H in Figure 6.4(b), *which has a cycle* and thus it is not a term-graph.

The reason for the cycle being introduced in both examples from Figure 6.4 is that the matches overlap, allowing variable nodes to precede operation nodes in the path order of G, while r reorders the mapping of the variables to create a cycle. In jungle rewriting [68] this issue is prevented by imposing a statically checkable condition on the rules, namely that the path relation between the nodes preserved from L should not be changed by R. Formally, we say that a rule $\rho : (L \xleftarrow{l} K \xrightarrow{r} R)$ is *cycle free* if whenever $v \prec_R x$ with $v \in V_K$ and $x \in VAR_L \cap V_K$, it must be that $v \prec_L x$. This condition is sufficient to prevent the introduction of cycles; however, we find it rather strong in our programming language context—in particular, this condition would disallow rules like the one for reading the value of a variable from the store. In what follows, we give a (semantical) condition on the matching morphism m rather than the rule which is sufficient to avoid the introduction of cycles.

Given a (composed) term-graph rewrite rule $\rho : (L \xleftarrow{l} K \xrightarrow{r} R)$, r induces on K a (partial) replacement order $\prec_r = r^{-1}(\prec_R)$, i.e., $v_1 \prec_r v_2$ in K iff $r(v_1) \prec_R r(v_2)$ (there is a path from $r(v_1)$ to $r(v_2)$ in R). Moreover, given match m of p into G, m induces on K a (partial) matching order $\prec_m = l^{-1}(m^{-1}(\prec_G))$, i.e., $v_1 \prec_r v_2$ in K iff $m(v_1) \prec_G m(v_1)$ (l is an inclusion). Although both these (partial) orders are strict, their combination is not guaranteed to remain strict. We say that the match m is *cycle free* w.r.t. p if the transitive closure of $\prec_m \cup \prec_r$ is also a strict (partial) order.

Proposition 6. *(1) If any matching morphism for a $\mathbb{K}$ graph rewriting rule ρ is cycle free, then ρ is a jungle graph rewriting rule. (2) If ρ is a $\mathbb{K}$ graph rule, G is a term-graph, $G \xRightarrow{(\rho,m)} H$, and m is cycle free w.r.t. ρ, then H is acyclic.*

Proof. Let $\rho : (L \xleftarrow{l} K \xrightarrow{r} R)$ be a $\mathbb{K}$ graph rewriting rule.

(1) Suppose that there exist $v \in V_K$ and $x \in VAR_L$ such that $v \prec_R x$ and $v \not\prec_L x$. Let then G be the graph obtained from L by adding an edge e such that source$(e) = x$ and target$(e) = v$. G is still acyclic, because L is acyclic and because $v \not\prec_L x$. Let $m : L \to G$ be the inclusion morphism. We have that m is not cycle free, since $v \prec_R x$ implies that $v \prec_r x$ and $x \prec_G v$ implies that $x \prec_m v$, contradiction.

(2) Proof by contradiction. Assume that H is not acyclic, and let $e_0, \ldots, e_n$ be a sequence of edges in H exhibiting a cycle. Let then $e_{\alpha_0}, \ldots, e_{\alpha_m}$ be a subsequence of the above sequence with the property that all its elements are edges in C and that the blocks of edges between them (including the one starting

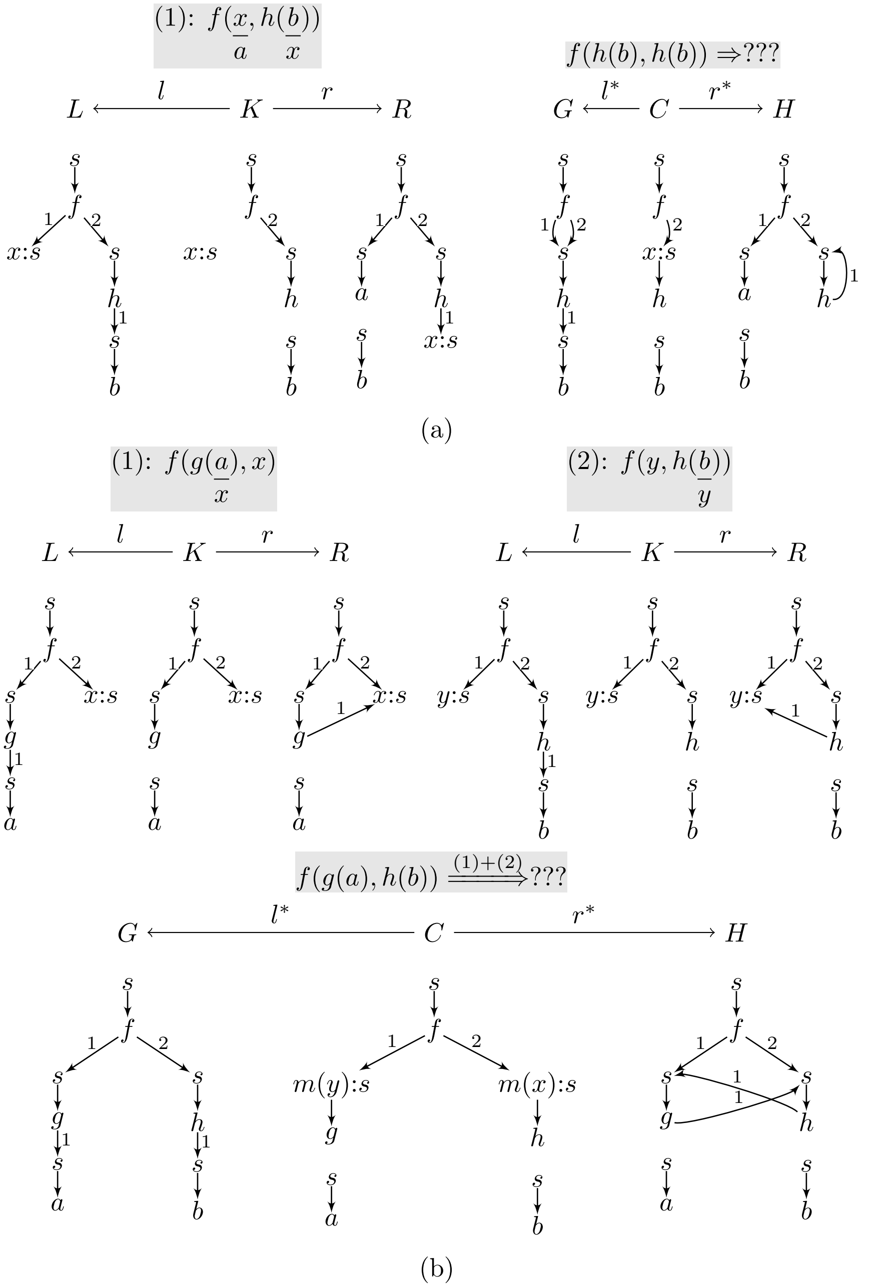

(1): $f(\underline{x}, h(\underline{b}))$
a x
$f(h(b), h(b)) \Rightarrow ???$
$L \xleftarrow{l} K \xrightarrow{r} R$
$G \xleftarrow{l^*} C \xrightarrow{r^*} H$
s
f
$x{:}s$
h
a
b
1
2
(a)
(1): $f(g(\underline{a}), x)$
x
(2): $f(y, h(\underline{b}))$
y
$L \xleftarrow{l} K \xrightarrow{r} R$
g
$y{:}s$
$f(g(a), h(b)) \xRightarrow{(1)+(2)} ???$
$G \xleftarrow{l^*} C \xrightarrow{r^*} H$
$m(y){:}s$
$m(x){:}s$
(b)

Figure 6.4: $\mathbb{K}$ graph rewriting can introduce cycles: (a) on a term-graph with sharing; (b) using parallel reductions.

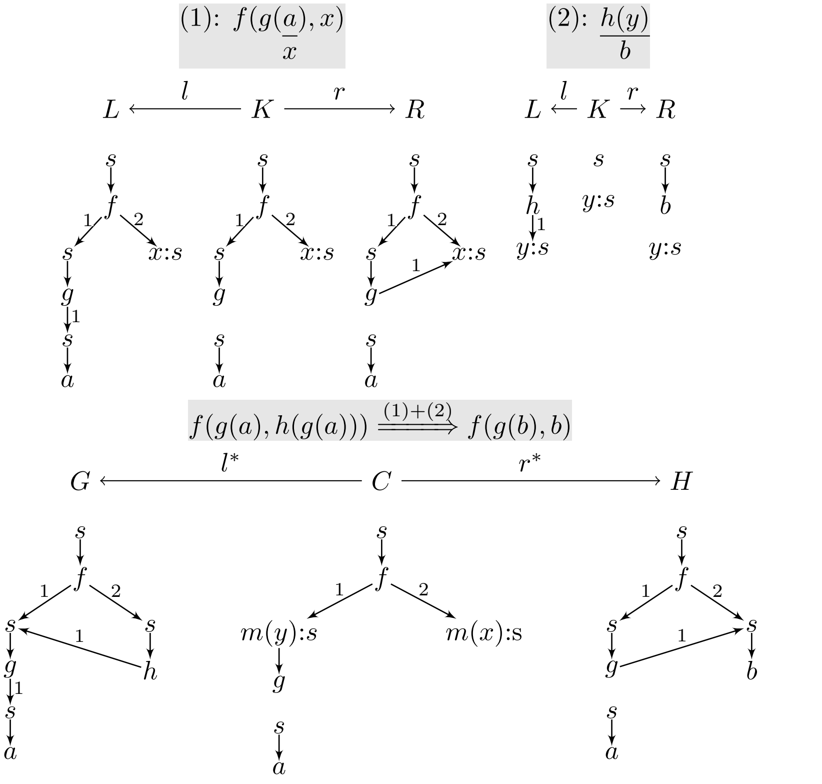

(1): $f(g(\underline{a}), x)$ / x
(2): $h(y)$ / b
$L \xleftarrow{l} K \xrightarrow{r} R$
$L \xleftarrow{l} K \xrightarrow{r} R$
$f(g(a), h(g(a))) \xRightarrow{(1)+(2)} f(g(b), b)$
$G \xleftarrow{l^*} C \xrightarrow{r^*} H$
s
f
g
h
a
b
x:s
y:s
$m(y)$:s
$m(x)$:s
1
2

Figure 6.5: Non-cycle-free match producing non-cycling result

at e_{α_m} and wrapping over to e_{α_0} are alternating between C and $R \setminus K$. Then, if the edges between e_{α_i} and $e_{\alpha_{i+1}}$ are all in C, it must be that both source(e_{α_i}) and target($e_{\alpha_{i+1}}$) are in V_K, and moreover, that source(e_{α_i}) $\prec_m$ target($e_{\alpha_{i+1}}$). Similarly, if the edges between e_{α_i} and $e_{\alpha_{i+1}}$ are in $R \setminus K$, then both target(e_{α_i}) and source($e_{\alpha_{i+1}}$) are in V_K (which we already knew from the previous sentence) and that target(e_{α_i}) $\prec_r$ source($e_{\alpha_{i+1}}$). But this precisely implies that m is not cycle free, contradiction. □

One might be tempted to think that if a morphism is not cycle free, then the resulting graph is bound to be cyclic. However, this is not the case, because when applying a composed rule, the composing rules not involved in the cyclicity condition might break the cycle, and thus produce a valid term-graph, as exhibited by the example in Figure 6.5. The graph to be rewritten is a representation of term $f(g(a), h(g(a)))$ in which the two occurrences of the subterm $g(a)$ have been identified. The match of the composed rule (1) + (2) is not cycle-free because of rule (1). Moreover, if we would only apply rule (1), its application would lead to a cycle. However, when applying both rules together, the cycle is broken, and the resulting graph is indeed corresponding to the term $f(g(b), b)$ which can be obtained from the original term by regular term rewriting. Nevertheless, if the original graph is a tree, then cycle freeness of the matching morphism characterizes acyclicity of the resulting graph.

Proposition 7. *Let G be a tree term-graph.*

1. *If ρ is a simple $\mathbb{K}$ graph rule and m is a match for ρ into G, then m is cycle free.*
2. *If ρ is a composed $\mathbb{K}$ graph rule and $G \overset{(\rho,m)}{\Longrightarrow} H$, then H is acyclic iff m is cycle free w.r.t. ρ.*

Proof. Observation 1: Since G is a tree, $v_1 \prec_G v$ and $v_2 \prec_G v$ implies that either $v_1 \prec_G v_2$ or $v_2 \prec_G v_1$.

Observation 2: Assuming m is not cycle free, since both $\prec_m$ and $\prec_r$ are acyclic, it must be that the cycle is obtained by an alternating sequence $v_1 \prec_r x_1 \prec_m v_2 \ldots \prec_r x_{n-1} \prec_m v_n = v_1$, where x_i is a variable node and v_i is a pattern node for all $1 \leq i < n$.

(1) Let us show that is impossible to have $x \prec_m v$ where x is a variable node and v is a pattern node, whence m must be cycle free. Indeed, $x \prec_m v$ means that $m(x) \prec_G m(v)$, which would lead to $x \prec_L v$ (since $m(L)$ is a subtree of G), which is not possible, as x is a leaf in L.

(2) We only need to prove that if m is not cycle-free, then H has cycles, as the converse was proven in the general case by Proposition 6. Assume m is not cycle free, and consider a minimal sequence exhibiting a cycle as in Observation 2. We want to show that this sequence is also valid if we replace $\prec_m$ with $\prec_{\overline{m}} = \overline{m}^{-1}(\prec_C)$, which would necessarily lead to a cycle in H, as H is obtained as the pushout between C and R identifying K. We again reason by contradiction and assume that this is not the case, that is, there exists $1 \leq i < n$ such that $x_i \prec_m v_{i+1}$ but $x_i \not\prec_{\overline{m}} v_{i+1}$. However, this can only happen if an edge between $m(x_i)$ and $m(v_{i+1})$ in G is removed by another rule. Therefore, there must exist a pattern node v and a variable node x such that $x_i \prec_m v$, $v \prec_L x$, $x \prec_m v_{i+1}$, and $v \not\prec_K x$. From $v_{i+1} \prec_r x_{i+1}$ we deduce that $v_{i+1} \prec_r x_{i+1}$ are part of the same rule, and therefore there must be some $v' \in K$ such that $v' \prec_L v_{i+1}$ and $v' \prec_L x_{i+1}$. Using Observation 1, $m(v) \prec_G m(x) \prec_G m(v_{i+1})$ and $m(v') \prec_G m(v_{i+1})$ implies that either $m(v) \prec_G m(v')$ or $m(v') \prec_G m(v)$. Using the parallel independence condition we deduce that $m(v) \prec_G m(v')$, whence $x_i \prec_m v \prec_m v' \prec_m x_{i+1} \prec_m v_{i+2}$. However, $x_i \prec_m v_{i+2}$ is in contradiction with our original assumption that the cycle was minimal. □

Next result shows that, under cycle-freeness conditions, $\mathbf{KGraph_\Sigma}$ is closed under (parallel) derivations using $\mathbb{K}$ graph rewrite rules.

Theorem 6. *Let G, $(\rho_i)_{i=\overline{1,n}}$, $(m_i)_{i=\overline{1,n}}$, ρ, m, C, and H be defined as above. If m is cycle-free w.r.t. p then the following hold:*

(Parallel) Derivation: $G \xRightarrow[\mathbf{KGraph_\Sigma}]{\rho,m} H$;

Serialization: *There exist $(G_i)_{i=\overline{0,n}}$ such that $G_0 = G$, $G_n = H$, and $G_{i-1} \xRightarrow[\mathbf{KGraph_\Sigma}]{\rho_i} G_i$ for each $1 \leq i \leq n$.*

Proof. From the parallel independence condition, there exists a derivation $G \overset{\rho,m}{\Longrightarrow} H$ in **Graph**, and, H must be acyclic (Proposition 6). To prove the Derivation claim we only need to show that the graphs produced by the derivation, C and H, are indeed term-graphs.

Assuming that we have proved the Derivation claim, we can use the serializability result for the category of graphs iteratively, the first step being the following: From $G \xRightarrow{p_1+\cdots+p_n,m} H$ we deduce that $G \xRightarrow{p_1+\cdots+p_{n-1},m'} H' \overset{p_n}{\Longrightarrow} H$, where m' is the composition of $(m_i)_{i=\overline{1,n-1}}$; however, by the derivation claim, H' is also a term-graph, and, therefore, we can iterate to obtain the serialization result in $\mathbf{KGraph}_{\boldsymbol{\Sigma}}$.

To prove the derivation part of the theorem, we only need to show that the graphs C and H defined above are term-graphs. First, let us show that C is a term-graph. Conditions (0)—C is bipartite, (1.ii) at most n consistently labeled outward edges for each operation node, (2)—at most one outward edge for each sort node, and (3)—C is acyclic are obviously satisfied, since we only remove nodes and edges. For (1.i) we only need to notice that whenever $e \in E_L \setminus E_K$ such that source(e) is a sort node then target$(e) \in V_L \setminus V_K$ since it is the root operation node corresponding to a 0-sharing rule. Let $l^* : C \to G$ and $\overline{m} : K \to C$ be the morphisms completing the pushout diagram. We have that l^* is an inclusion and $\overline{m}$ is the restriction and co-restriction of m to K and C, respectively.

Let us now additionally verify that H is a term-graph.

(0)—H is bipartite. This is ensured by the fact that R is bipartite and r only identifies nodes of the same kind.

(1.i)—each operation node has exactly one inward edge. Proof by contradiction. Suppose there exists distinct edges e, e' in E_H such that target$_H(e) =$ target$_H(e)$ and it is an operation node. Since $\top_i$ and $r_V(\top_i)$ are sort nodes, we can assume, as above that $e \in E_C$, $e' \in E_R \setminus E_K$, target$_R(e') \in V_K$, and target$_C(e) = m_V($target$_R(e'))$. However, $e' \in E_R \setminus E_K$, target$_R(e') \in V_K$, and target$_R(e')$ operation node constitute a contradiction with the fact that R satisfies $(1.i)$, since there should be another edge in E_K with the same target as e'.

(1.ii)—each operation node's outward edges are consistent. Since both C and R are term-graphs, the labels of outward edges of operation sorts, as well as the labels of their targets must be consistent in H. To complete our proof we only need to additionally show that no duplicates are introduced by the merging. Proof by contradiction. Suppose there exists distinct edges e and e' in E_H, such that source$_H(e) =$ source$_H(e')$ is an operation node, and le$_H(e) =$ le$_H(e')$. Then we can assume that $e' \in E_C$, $e \in E_R \setminus E_K$, and source$_R(e) \in V_K$, inducing that source$_C(e') = m_V($source$_R(e))$. From $e \in E_R \setminus E_K$ and source$_R(e) \in V_K$

we infer that there exists i such that $e \in E_{R_i} \setminus E_{K_i}$ and $s_{R_i}(e) \in V_K$ is an operation node. Therefore, $x_{\mathrm{source}_{R_i}(e),\mathrm{le}_{R_i}(e)}$ cannot be a (term) variable of R_i, and therefore, it cannot be a term variable of L_i, as well. Moreover, since $\mathrm{source}_{R_i}(e) \in V_K$, it must be that $\mathrm{source}_{R_i}(e) \in V_{L_i}$, and hence there exists $e_i \in E_{L_i}$ such that $\mathrm{source}_{L_i}(e_i) = \mathrm{source}_{R_i}(e)$ and $\mathrm{le}_{L_i}(e_i) = \mathrm{le}_{R_i}(e)$. But this implies that $e_i \in E_{L_i} \setminus E_{K_i}$, which contradicts with the fact that $e' \in E_C$ (since e' has the same source and label).

(2)—each sort node has at most one outward edge. Proof by contradiction. Suppose there exist distinct edges e and e' in E_H such that $\mathrm{source}_H(e) = \mathrm{source}_H(e') = v$, and v is a sort node. We can then suppose (without loss of generality) that $e \in E_C$ and $e' \in E_R \setminus E_K$. Then $\mathrm{source}_R(e') \in V_K$ and $v = \mathrm{source}_H(e) = \mathrm{source}_C(E) = m_V(\mathrm{source}_R(e'))$. Reusing a previous argument, from $\mathrm{source}_r(e') \in V_K$, $e' \in E_R \setminus E_K$ and $\mathrm{source}_R(e')$ sort node we deduce that $\mathrm{source}_R(r') \in ROOT_L$. Therefore, there exists i such that $e' \in E_{R_i} \setminus E_{K_i}$ and $\mathrm{source}_{R_i}(e') = \top_i$. However, this implies that $\mathrm{source}_C^{-1}(m_V(\mathrm{source}_{R_i}(e'))) = \emptyset$, which contradicts with $e \in E_C$.

(3)—*H* is acyclic. This is ensured by the hypothesis that m is cycle-free w.r.t. p. □

6.4 𝕂 Rewriting—Semantics

Theorem 6 allows us to capture the serializable fragment of 𝕂 concurrent rewriting as the relation $\Rrightarrow$ defined below:

Definition 5. *Let t be a Σ-term and let $\rho_1, \cdots, \rho_n$ be $\mathbb{K}$ rules (not necessarily distinct). Then $t \overset{\rho_1+\cdots+\rho_n}{\Rrightarrow} t'$ iff there is a term-graph H such that $G \xRightarrow[\mathbf{KGraph}_{\mathbf{\Sigma}}]{K2G(\rho_1)+\cdots+K2G(\rho_n)} H$ and $term_H(\top_H) = t'$, where G is the tree term-graph representing t. We say that $t \Rrightarrow t'$ iff there is a (composed) $\mathbb{K}$-rule ρ such that $t \overset{\rho}{\Rrightarrow} t'$.*

We next show that the 𝕂 concurrent rewriting above is a conservative extension of the standard term rewriting relation.

6.4.1 Soundness and Completeness w.r.t. Term Rewriting

We can give a straightforward definition for what it means for a 𝕂-rule to match a term: *one* 𝕂-rule $\rho : (\forall \mathcal{X})\ k[\ L \Rightarrow R\ ]$ matches a term t using context C and substitution θ iff its corresponding rewrite rule $\mathit{K2R}(\rho) : (\forall X) L(k) \to R(k)$ matches t using the same C and θ, that is, iff $t = C[\theta(L(k))]$. This conforms to the intuition that, when applied sequentially, 𝕂 rules behave exactly as their corresponding rewrite rules. We next show that the rewrite relation induced

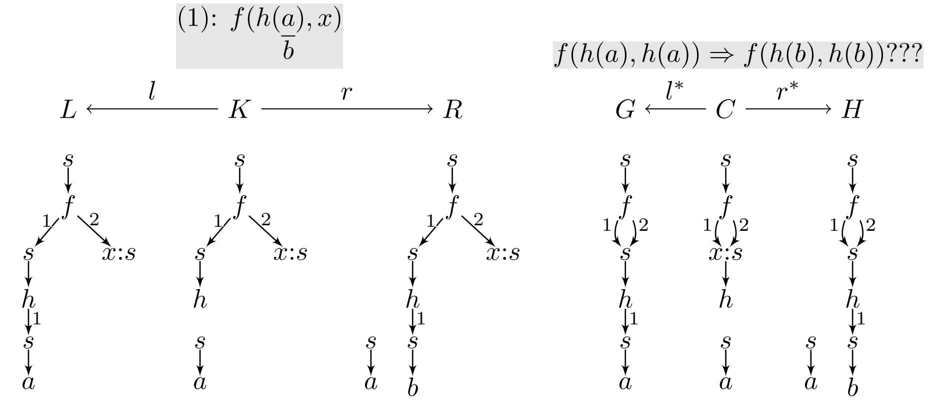

(1): $f(h(\underset{b}{\underline{a}}),x)$
$f(h(a),h(a)) \Rightarrow f(h(b),h(b))$???
L
l
K
r
R
G
l^*
C
r^*
H
s
f
h
a
b
$x{:}s$
1
2

Figure 6.6: Subterm sharing might lead to unsound $\mathbb{K}$ graph rewriting.

by $\mathbb{K}$ rules indeed captures the standard term rewrite relation. We will do that by reducing rewriting using $\mathbb{K}$ graph rules to rewriting using 0-sharing $\mathbb{K}$ graph rules, which, as we previously mentioned is actually an instance of jungle evaluation in the graph world. Then, we can use the soundness and completeness of jungle evaluation w.r.t. term rewriting to obtain that $\mathbb{K}$ term rewriting is sound and complete w.r.t regular term rewriting.

However, it turns out that, although preserving the term-graph structure (under cycle-freeness assumptions, $\mathbb{K}$ rewriting on graphs might not be sound w.r.t. term rewriting in the presence of subterm sharing. Consider the example in Figure 6.6. We want to apply rule $f(h(\underset{b}{\underline{a}}),x)$, corresponding to the regular rewrite rule $f(h(a),x) \to f(h(b),x)$, to the term $f(h(a),h(a))$. If we would represent $f(h(a),h(a))$ as a tree, then the $\mathbb{K}$ graph rewriting step would be sound, leading to a graph depicting $f(h(b),h(a))$; however, if we decide to collapse the tree representing $h(a)$ then we obtain $f(h(b),h(b))$, as depicted in Figure 6.6 which cannot be obtained through regular rewriting. The reason for this unsound rewriting is that part of the read-only patten of the rule is shared. To overcome this, we will restrict the read-only pattern of the rule to only match against a tree in the graph to be rewritten. We say that a match $m : L \to G$ of a $\mathbb{K}$ graph rewrite rule $\rho : (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ is *safe* if $m(K\restriction_{root_L})$ is a tree in G, that is, if $indegree_G(m_V(v)) = 1$ for any $v \in V_{K\restriction_{root_L}} \setminus \{root_L\}$. Note that, if G is a tree then all matching morphism on G are safe.

Proposition 8. *Let ρ be a proper $\mathbb{K}$ rewrite rule, let ρ_0 be its associated* 0*-sharing $\mathbb{K}$ rewrite rule, and let m be a cycle free safe matching morphism for $K2G(\rho)$ in G. Let H be such that $G \xRightarrow[\mathbf{KGraph}_\Sigma]{K2G(\rho),m} H$, and let H' be such that $G \xRightarrow[\mathbf{KGraph}_\Sigma]{K2G(\rho_0),m} H'$. Then for any $v \in ROOT_G$, $term_H(v) = term_H(v)$.*

Proof. First, cycle freeness ensures the existence of H; moreover, any 0-sharing $\mathbb{K}$-rule generates the graph representation of a jungle evaluation rule, and thus the existence of H' is ensured.

Second, since ρ is proper, neither ρ nor ρ_0 is collapsing, and therefore $\mathit{ROOT}_G \subseteq \mathit{ROOT}_H$ and $\mathit{ROOT}_G \subseteq \mathit{ROOT}_{H'}$, so the final claim is also defined.

Let $\mathit{K2G}(\rho) : (L_\rho \xleftarrow{l_\rho} K_\rho \xrightarrow{r_\rho} R_\rho)$ and $\mathit{K2G}(\rho_0) : (L_\rho \xleftarrow{l_{\rho_0}} K_{\rho_0} \xrightarrow{r_{\rho_0}} R_{\rho_0})$ be the complete descriptions of $\mathit{K2G}(\rho)$ and $\mathit{K2G}(\rho_0)$, and let C, C' be the corresponding context graphs obtained in the process of applying the rules to G.

We have that $C = G \setminus m(L_\rho \setminus K_\rho)$, whence $V_C = V_G$ and $E_C = E_G \setminus \{m(e_{\Box_i}) \mid e_{\Box_i} \in E_{L_\rho}, \text{target}(e_{\Box_i}) \text{ corresponds to } \Box_i \in \mathcal{W}\}$. Also $C' = G \setminus m(L_\rho \setminus K_{\rho_0})$, whence $V_{C'} = V_G \setminus m_V(v_0)$ and $E_{C'} = E_G \setminus (\text{source}^{-1}(v_0) \cup \text{target}^{-1}(v_0))$, where $v_0 = \text{target}(\text{source}^{-1}(\mathit{root}_{L_\rho}))$.

H' is obtained by "gluing" on C' $R_{\rho_0} \setminus K_{\rho_0}$, that is the variable collapsed tree representation of $R(k)$ in which the root and the variable nodes have been removed. This gluing is done by setting the source of the topmost edge to be $m_V(\mathit{root}_L)$ and the target of any edge whose target is variable node x in R_{ρ_0} to be $m_V(x)$.

H is obtained by "gluing" on C $R_\rho \setminus K_\rho$, that is the variable collapsed tree representation of $\{R(\mathit{Hole}) \mid \Box \in \mathcal{W}\}$ in which the variable nodes have been removed, and an edge $e'_{\Box_i}$ for each $\Box_i$ has been added having as target the node representing the root of $R(\Box_i)$. The gluing is done by setting the target of any edge whose target is variable node x in R_ρ to be $m_V(x)$ and by setting the source of $e'_{\Box_i}$ to be $\text{source}_G(e_{\Box_i})$.

We can define a morphism $f : H \to H'$, as follows: For $C \setminus m(K_0)$ it is the identity: $f_V(v) = v if v \in V_C \setminus m(V_{L_\rho}) = V_G \setminus m(V_{L_\rho})$. $f_E(e) = e if e \in E_C \setminus m(E_{L_\rho}) = E_G \setminus m(E_{L_\rho})$. For the root of L_ρ and for its variables, it is also the identity: $f_V(m(\mathit{root}_{L_\rho})) = m(\mathit{root}_{L_\rho})$; $f_V(m(x)) = m(x)$. Now, for $K_0 = K_\rho\!\restriction_{\mathit{root}_{L_\rho}}$, it yields the copy of K_0 in R: $f_V(m_V(v)) = v$ for any $v \in V_{K_\rho\restriction_{\mathit{root}_{L_\rho}}}$, $v \neq \mathit{root}_{L_\rho}$ and $f_V(m_V(e)) = e$ for any $e \in E_{K_\rho\restriction_{\mathit{root}_{L_\rho}}}$. Finally, the mapping $R_\rho \setminus K_\rho$ it is already determined by the mapping of the elements coming from K_0, ad basically says that the variable collapsed trees corresponding to $R(\Box_i)$ are mapped to their corresponding (variable collapsed) subtrees coming from R_{ρ_0}.

It is relatively easy to verify that f is an injective morphism. Moreover the nodes and edges which are not in its image are part of the graph $m(K_0)$ (excluding root_L and the topmost operation node as well as its adjacent edges), which, by being required to be a tree in G, has no incoming edge, and thus is not part of $\overline{H'} = H'\!\restriction_{\mathit{ROOT}_G}$. Hence, the restriction and co-restriction of f to $\overline{H} = H\!\restriction_{\mathit{ROOT}_G}$ and $\overline{H'}$, respectively, is a bijection, and, therefore for any $v \in \mathit{ROOT}_G$, $H\!\restriction_v$ is isomorphic with $H'\!\restriction_v$, whence $\mathit{term}_H(v) = \mathit{term}_{H'}(v)$. $\Box$

Since the $\mathbb{K}$ graph representation of a term t without anonymous variables is a graph jungle representing the same term, and since the $\mathbb{K}$ term-graph representation of a 0-sharing $\mathbb{K}$ rewrite rule is a graph jungle rule representing the rewrite rule associated to it, we can use the soundness and completeness of jungle rewriting w.r.t. standard term rewriting [68, 38] to prove the sequential soundness and completeness of $\mathbb{K}$ graph rewriting w.r.t. standard term rewriting, and, by combining that with Theorem 6, to prove the serializability result for $\mathbb{K}$ concurrent rewriting.

Theorem 7. *Let ρ, ρ_1, $\ldots$, ρ_n be $\mathbb{K}$ rules. The following hold:*

Completeness: *If $t \xRightarrow{K2R(\rho)} t'$ then $t \overset{\rho}{\Rrightarrow} t'$.*

Soundness: *If $t \overset{\rho}{\Rrightarrow} t'$ then $t \xRightarrow{K2R(\rho)^*} t'$.*

Serializability: *If $t \overset{\rho_1+\cdots+\rho_n}{\Rrightarrow} t'$, then there exists a sequence of terms $t_0, \cdots, t_n$, such that $t_0 = t$, $t_n = t'$, and $t_{i-1} \overset{\rho_i^*}{\Rrightarrow} t_i$.*

Proof. Let G be the tree term-graph representation of t.

Completeness. From the completeness of jungle evaluation, we infer that there exists H such that $G \xRightarrow{m,K2G(\rho_0)} H$ and $term_H(m^*(root_G)) = t'$. Since G is a tree, m must be both cycle-free and safe for $K2G(\rho)$. From Proposition 8 we then infer that $G \xRightarrow{m,K2G(\rho)} H'$ and that $term_{H'}(m^*(root_G)) = t'$, whence $t \overset{\rho}{\Rrightarrow} t'$.

Soundness. From $t \overset{\rho}{\Rrightarrow} t'$ it follows that there exists a rewrite sequence $G \xRightarrow{m,K2G(\rho)} H'$ such that $term_{H'}(m^*(root_G)) = t'$. Again, since G is a tree, m must be both cycle free and safe, whence, by Proposition 8, $G \xRightarrow{m,K2G(\rho_0)} H$ such that $term_H(m^*(root_G)) = t'$, and by the soundness of jungle evaluation, $t \xRightarrow{K2R(\rho)^*} t'$.

Serializability. $t \overset{\rho_1+\cdots+\rho_n}{\Rrightarrow} t'$ implies that $G \xRightarrow{K2G(\rho_1)+\ldots+K2G(\rho_n)} H$ such that $term_H(m^*(root_G)) = t'$. Applying Theorem 6, we deduce that there exist $G_0, G_1, \ldots, G_n$ such that $G_0 = G$, $G_n = H$, and $G_{i-1} K2G(\rho_1) G_i$. Since G is a tree and all rules satisfy the parallel independence property, we can deduce that the matching morphism for each of the steps is safe, and thus also cycle free. Therefore we can for each step apply Proposition 8, and the the soundness of jungle evaluation w.r.t. rewriting, to obtain the desired answer. □

Therefore, $\mathbb{K}$ concurrent rewriting is sound and complete for term rewriting, while providing a higher degree of concurrency in one step than existing approaches, be them either through term-graph rewriting or through term rewriting.

6.5 Capturing 𝕂 Rewriting within Rewriting Logic

As previously mentioned, the direct representation of 𝕂 into rewriting logic (by replacing 𝕂 rules with their corresponding rewrite rules), loses part of the concurrency intended for 𝕂 definitions. However, at a theoretical level, rewriting logic can capture the read-only sharing of resources by using equational abstraction to create multiple copies of the resources being shared [104]. Starting from the theory of concurrent objects and their representation in rewriting logic proposed by Meseguer [104], in this section we sketch a formalization of 𝕂 graph rewriting into rewriting logic which (although non-executable) captures the concurrency of 𝕂 rewriting.

To do so, we first define graphs as a theory, then identify term-graphs as being graphs over special labels (defined by the signature) and containing a special node identified as root. To formalize the relation between terms and graphs, we also define a theory for terms, as well as back and forth transformations between terms and term graphs. Finally, we show how 𝕂 graph rewrite rules can be represented as rewrite rules over concurrent objects and their read-only copies. To avoid getting into intricate technical details, we here only consider the fragment of 𝕂 graph rewriting obtained by removing collapsing rules, as these amount to non-injective graph transformations, requiring nodes from the original graph to be identified, which would add an additional layer of complexity to the representation.

Concurrent objects and emulsifying axioms. We will base our encoding of graphs within rewriting on the specification of concurrent objects in rewriting logic [104]. Below is the syntax for concurrent objects, as existent in Maude [34]:

```
mod CONFIGURATION is
  sorts Attribute AttributeSet .
  sorts Oid Cid Object Msg Configuration .
  subsorts Attribute <  AttributeSet .
  subsorts Object Msg <Configuration .
  op none : →  AttributeSet .
  op _,_  :  AttributeSet AttributeSet  →  AttributeSet [assoc comm id: none] .
  op ⟨_:_|_⟩  :  Oid Cid AttributeSet → Object .
  op none : →  Configuration .
  op __  :  Configuration Configuration  →  Configuration [assoc comm id: none] .
endm
```

In this specification, objects are viewed as floating in a multi-set "molecule soup", in which they can re-arrange themselves to be matched by the rules. To address the problem of concurrent read access to objects, Meseguer [104] proposes that "emulsifying axioms" be used to create multiple (potentially unbounded) *read only* copies of an object. The original object is transformed into a read-only copies generator, which generates copies while increasing a counter to ensure that, in order to recover the original object, all copies must be packed back into the generator. The (non-executable) axioms for emulsification are presented below:

```
mod EMULATING−READ−ONLY−OBJECTS is including CONFIGURATION .
  protecting NAT .
  op {_:_|_|_} : Oid Cid AttributeSet Nat → Object .
  op [_:_|_] : Oid Cid AttributeSet → Object .
  var O : Oid . var C : Cid . var AS : AttributeSet . var N : Nat .
  eq ⟨ O : C | AS ⟩ = { O : C | AS | 0 } .
  eq { O : C | AS | N } = { O : C | AS | s(N) } [ O : C | AS ] .
endm
```

Standard rewriting modulo the concurrency enhancing axioms presented above can now be used to effectively capture concurrent rewriting of objects in which objects which are only read can be shared by different instances of the rewrite rules, by rewriting the rules to use the syntax of read-only object to specify which objects are read, but kept unchanged by the rule. Take, for example,the rewrite rule

```
rl ⟨ O : C | active: false ⟩ ⟨ O : Visitor | none ⟩
   ⇒ ⟨ O : C | active: true ⟩ ⟨ O : Visitor | none ⟩
```

in which an object of a Visitor class is used to activate objects of the C class. If this rule is applied to an initial multiset containing

```
⟨ o1 : C | active: false ⟩ ⟨ o2 : C | active: false ⟩ ⟨ o3 : Visitor | none ⟩,
```

it will need two steps to activate both objects. However, by transforming the rule above (to make the sharing of the `Visitor` object explicit) into

```
rl ⟨ O : C | active: false ⟩ [O : Visitor | none]
   ⇒ ⟨ O : C | active: true ⟩ [O : Visitor | none]
```

and using rewriting modulo the concurrency enhancing axioms specified above (in addition to the multiset axioms), all objects can now be activated in one rewriting step, by using the axioms to heat and cool the state, as follows:

```
  ⟨ o1 : C | active: false ⟩ ⟨ o2 : C | active: false ⟩ ⟨ o3 : Visitor | none ⟩
= ⟨ o1 : C | active: false ⟩ ⟨ o2 : C | active: false ⟩
  {o3 : Visitor | none | 0}
= ⟨ o1 : C | active: false ⟩ ⟨ o2 : C | active: false ⟩
  { o3 : Visitor | none | 1} [o3 : Visitor | none]
= ⟨ o1 : C | active: false ⟩ ⟨ o2 : C | active: false ⟩
  {o3 : Visitor | none | 2} [o3 : Visitor | none] [o3 : Visitor | none]
= (⟨ o1 : C | active: false ⟩ [o3 : Visitor | none])
  (⟨ o2 : C | active: false ⟩ [o3 : Visitor | none]) {o3 : Visitor | none | 2}
⇒ (⟨ o1 : C | active: true ⟩ [o3 : Visitor | none])
  (⟨ o2 : C | active: true ⟩ [o3 : Visitor | none]) {o3 : Visitor | none | 2}
= ⟨ o1 : C | active: true ⟩ ⟨ o2 : C | active: true ⟩
  { o3 : Visitor | none | 2 } [o3 : Visitor | none] [o3 : Visitor | none]
= ⟨ o1 : C | active: true ⟩ ⟨ o2 : C | active: true ⟩
```

```
  {o3 : Visitor | none | 1} [o3 : Visitor | none]
= ⟨ o1 : C | active: true ⟩ ⟨ o2 : C | active: true ⟩
  {o3 : Visitor | none | 0}
= ⟨ o1 : C | active: true ⟩ ⟨ o2 : C | active: true ⟩ ⟨ o3 : Visitor | none ⟩
```

In the following encoding of term graphs and graph rewriting rules we will do precisely that, encoding the part of the graph belonging to the interface graph using read-only constructs for its objects.

Terms and term graphs. Upon analyzing multiple possible representations for graphs as algebraic structures, we have chosen to represent graphs as collections of objects belonging to two classes: *Node* and *Edge*. Nodes have only one attribute, their *label*, while edges have three attributes: *source*, *target*, and *label*.

```
mod GRAPH is including CONFIGURATION .
  sorts NodeId EdgeId .
  sorts NodeLabel EdgeLabel Label .
  subsorts NodeId EdgeId < Oid .
  subsorts NodeLabel EdgeLabel <Label .
  ops Node Edge : → Cid .
  ops source:_ target:_ : NodeId → Attribute .
  op label:_ : Label → Attribute .
endm
```

To represent terms (and term graphs) we introduce the following description of signatures:

```
fmod SIGNATURE is
  sorts Sort Operation SortList .
  subsort Sort < SortList .
  op sort : Operation → Sort .

  op none : → SortList .
  op _x_ : SortList SortList → SortList [assoc id: none] .

  op arity : Operation → SortList .
endfm
```

Within this frame, we represent proper terms as either constants (i.e., operation symbols), or applications of operation symbols to other terms. We additionally identify two types of variables, normal and anonymous, which are terms, but not proper ones.

```
fmod TERM is including SIGNATURE .
  sorts Variable ProperTerm Term TermList NeTermList .
  subsort Operation < ProperTerm < Term < NeTermList < TermList .
  subsort Variable < Term .
  op Any : → Term .
  op empty : → TermList .
  op _,_ : TermList TermList → TermList [assoc id: empty] .
  op _,_ : NeTermList TermList →NeTermList [ditto] .
  op _,_ : TermList NeTermList →NeTermList [ditto] .
  op _(_) : Operation NeTermList →ProperTerm .

  op sort : Variable → Sort .
endfm
```

A loose semantics for term graphs is obtained from the graph specification above by customizing node labels to be either operations or sorts and the edge labels to be natural numbers. Moreover, a term graph needs to have its root node distinguished among the others. For this, we will use a special message *root*.

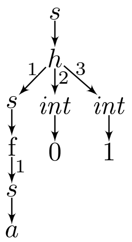
```
mod TERM−GRAPH is including SIGNATURE . extending GRAPH .
  protecting NAT .
  subsorts Operation Sort < NodeLabel .
  subsort Nat < EdgeLabel .
  op root : NodeId → Msg .
endm
```

Relating terms and term graphs. To formalize the correspondence between terms and their graph representation, we define two transformations, one from terms to graphs, and another one from graphs to terms.

The terms-to-graphs transformation follows the intuitive description of representing terms as graphs we used in the previous sections, which is the same as one introduced in Chapter 2. We choose to use sequences of integers (representing positions in the terms) as identifiers for both node and edge objects, while variables are identified as themselves. Also, to simplify the presentation we allow two objects to have the same identifier as long as they belong to different classes.

```
mod TERM−TO−GRAPH is including TERM . including TERM−GRAPH .
  protecting LIST{Int} .
  subsort List{Int} < NodeId EdgeId .
  subsort Variable < NodeId .

  op term2graph_ : Term →Configuration .
  eq term2graph(T) = t2g(T,0) root(0) .

  op t2g : Term NodeId →Configuration .
  eq t2g(O,V 0) = ⟨V 0 : Node | label: sort(O) ⟩
                  ⟨ V : Node | label: O ⟩
                  ⟨ V 0 : Edge | source: (V 0), target: V ⟩ .
  eq t2g(Var, N) = ⟨ Var : Node | label: sort(Var) ⟩ .


  eq t2g(_‘(_‘)( O,NeTl),(V 0)) = t2g(O,(V 0)) t2gL(NeTl, (V 1)) .
  op t2gL : NeTermList NodeId →Configuration .
  eq t2gL((T,NeTl),V N) = t2gL(T,V N) t2gL(NeTl,V s(N)) .
  eq t2gL(T, V N)
   = ⟨ (V N) : Edge | source: V, target: (V N 0), label: N ⟩ t2g(T,(V N 0)) .
  eq t2gL(Var, V N)
   = ⟨ (V N) : Edge | source: V, target: Var, label: N ⟩ t2g(Var, 0) .
  eq t2gL(Any, V N) = none .

  var T : Term . var NeTl : NeTermList . var V : List{Int} . var N : Nat .
  var O : Operation . var S : Sort .
endm
```

For example, by evaluating the `term2graph` function defined in the above module on the term to be rewritten from our running example, $h(f(a), 0, 1)$, whose graph representation is



we obtain the following concurrent objects representation of the graph:

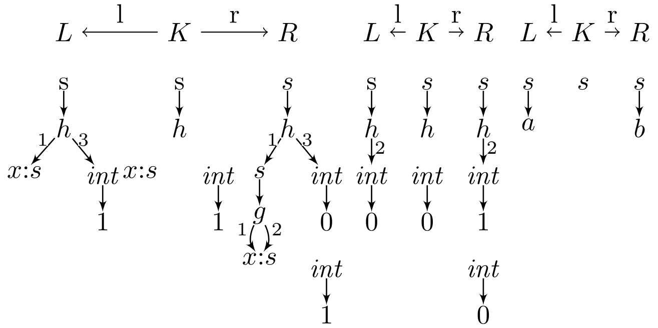
```
root(0)
⟨ 0 : Node | label: s ⟩ ⟨ 0 : Edge | source : 0,target : nil ⟩
⟨ nil : Node | label: h ⟩
⟨ 1 0 : Node | label: s ⟩ ⟨ 1 0 : Edge | source: 1 0,target: 1 ⟩
⟨ 1 : Node | label: f ⟩ ⟨ 1 : Edge | source: nil, target: 1 0,label: 1 ⟩
⟨ 2 0 : Node | label: int ⟩ ⟨ 2 0 : Edge | source: 2 0,target: 2 ⟩
⟨ 2 : Node | label: 0 ⟩ ⟨ 2 : Edge | source: nil,target: 2 0,label: 2 ⟩
⟨ 3 0 : Node | label: int ⟩ ⟨ 3 0 : Edge | source: 3 0,target: 3 ⟩
⟨ 3 : Node | label: 1 ⟩ ⟨ 3 : Edge | source: nil,target: 3 0, label: 3 ⟩
⟨ 1 1 0 : Node | label: s ⟩ ⟨ 1 1 0 : Edge | source: 1 1 0,target: 1 1 ⟩
⟨ 1 1 : Node | label: a ⟩ ⟨ 1 1 : Edge | source: 1,target: 1 1 0,label: 1 ⟩
```

To transform a (ground) term-graph into its corresponding term, all we need to do is to start with its root, which is specified by the root message, and to follow the links and their specified positions until we arrive at terminal nodes (which are constants). The module below defines such a transformation in Maude.

```
mod GRAPH−TO−TERM is including TERM . including TERM−GRAPH .

  op graph2term : Configuration → Term .
  eq graph2term(root(V0) G) = g2t(root(V0) G, V0) .

  op g2t : Configuration NodeId → Term .
  eq g2t(G ⟨ V0 : Node | label: S ⟩ ⟨ V : Node | label: O ⟩
          ⟨ E : Edge | source: V0, target: V ⟩, V0)
   = g2tHelp(G ⟨V0 : Node | label: S ⟩ ⟨ V : Node | label: O ⟩
          ⟨ E : Edge | source: V0, target: V ⟩,V,O, arity(O)) .

  op g2tHelp : Configuration NodeId Operation SortList → TermList .
  eq g2tHelp(G, V, O, none) = O .
  eq g2tHelp(G, V, O, S x Sl) = _‘(_‘)( O,g2tL(G,V,S x Sl, 1)) .

  op g2tL : Configuration NodeId SortList Nat → TermList .
  eq g2tL(G, V, none, N) = empty .
  eq g2tL(⟨ E : Edge | source: V, target: V0, label: N ⟩
         ⟨ V0 : Node | label: S ⟩ G,V,S x Sl,N)
   = g2t(⟨ E : Edge | source: V, target: V0, label: N ⟩
        ⟨ V0 : Node | label: S ⟩ G, V0),
     g2tL(⟨ E : Edge | source: V, target: V0, label: N ⟩
         ⟨ V0 : Node | label: S ⟩ G,V, Sl,s(N)) .

  var G : Configuration . var V V0 : NodeId . var S : Sort .
  var O : Operation . var Sl : SortList . var N : Nat . var E : EdgeId .
endm
```

Representing 𝕂 rules. Let us intuitively describe here how 𝕂 rules can be encoded as rewrite rules on concurrent objects representing graphs by encoding rules (1)–(3) from our running example. Recall the three rules and their graph representation:

(1): $h(\underset{g(x,x)}{\underline{x}}, y, \underset{0}{\underline{1}})$ (2): $h(x, \underset{1}{\underline{0}}, y)$ (3): $\underset{b}{\underline{a}}$



A 𝕂 rule is represented by encoding its graph representation ($L \hookleftarrow K \hookrightarrow R$) as a rewrite rule. To do that, we represent L and R as the left-hand-side l and the right-hand-side r of a rewrite rule $l \to r$, respectively, as explained next. Let k be the term obtained from representing K as a multiset of objects specifying nodes and edges, where identifiers of objects are chosen to be distinct variables, and all objects are specified using their read-only variant. For example, for rule (1) above, k may look as follows:

$[V_0$: Node | label: s] $[V$: Node | label: h]

$[V_3$: Node | label: 1] $[V_{10}$: Node | label: s] $[V_{30}$: Node | label: int]
$[E_0$: Edge | source: V_0, target: $V]$ $[E_{30}$: Edge | source: V_{30}, target: $V_3]$

Then the set of concurrent objects l representing L can be obtained by adding to k the concurrent objects representation of the nodes and edges which are in L but not in K, again using distinct variables as their id. For rule (1) above, l can be obtained by adding the following objects to k:

$\langle$ E_1 : Edge | source: V, target: V_{10}, label: 1 $\rangle$
$\langle$ E_3 : Edge | source: V, target: V_{30}, label: 3 $\rangle$

To generate r we have to add to k the concurrent objects representation of nodes and edges which are in R but not in K. However, while using fresh variables as identifiers worked for the other concurrent objects appearing in the rule, and was even necessary as they are supposed to match existing nodes, this approach does not work anymore for the new objects introduced by R. Nevertheless, there is an easy solution for this [104]. Given the nature of $\mathbb{K}$ rules, there are always edges (and sometimes even nodes) which must be removed by the rule application. As their identifier will disappear, we can use it in generating identifiers for the objects introduced by R. Based on our choice to use lists of natural numbers encoding positions in the original term as identifiers for nodes, let us now show how the new identifiers can be obtained from them. For each hole $\square \in \mathcal{W}$, let $r^0_\square = \mathrm{term2graph}(\mathcal{R}(\square))$, where $\mathcal{R}(\square)$ is the replacement term for $\square$ in the $\mathbb{K}$ rule. If the $\mathbb{K}$ rule is a zero-sharing rule, then we obtain r from $r^0_\square$ by removing the `root` message and replacing its top sort node object with the top sort node object of L (and updating the unique edge having it as a source to reflect that) followed by prefixing all remaining identifiers with X `-1`, where X is the variable standing for the identifier of the top operation node of L (which is deleted by the rule), and -1 is used as a separator (as it is not a natural number); an exception from this rule are the nodes representing variables which are replaced with the corresponding nodes from l, while links to them are updated to point to the new nodes. For example, rule (3) can be represented as:

rl $[V_0$: Node | label: s] $\langle$ V : Node | label: a $\rangle$
$\langle$ E_0 : Edge | source: V_0, target: V $\rangle$
$\Rightarrow$ $[V_0$: Node | label: s] $\langle$ V -1 : Node | label: b $\rangle$
$\langle$ V -1 0 : Edge | source: V_0, target: V -1 $\rangle$

If the $\mathbb{K}$ rule is proper, then for each hole $\square$ of its pattern let

$\langle$ $X_\square$: Edge | from: X, to: Y, label: N $\rangle\}$

be the edge corresponding to $\square$ in l that must be deleted by the rule. Then $r_\square$ is obtained from $r^0_\square$ by removing the `root` message, prefixing all identifiers with $X_\square - 1$, and adding the object

$\langle$ $X_\square$ -1 : Edge | from: X to: $X_\square$ -1, label: N $\rangle$

representing the edge which must replace the deleted one; again, as an exception from this rule, variables are handled as described above. Then r is obtained by simply concatenating k with all instances of $r_{\square}$. For example, rule (1) can be represented by

$$\begin{array}{l}
\text{rl } [V_0 : \text{Node} \mid \text{label: s}]\ [V : \text{Node} \mid \text{label: h}] \\
\quad [V_3 : \text{Node} \mid \text{label: 1}]\ [V_{10} : \text{Node} \mid \text{label: s}]\ [V_{30} : \text{Node} \mid \text{label: int}] \\
\quad [E_0 : \text{Edge} \mid \text{source: } V_0\text{, target: } V]\ [E_{30} : \text{Edge} \mid \text{source: } V_{30}\text{, target: } V_3] \\
\quad \langle\ E_1 : \text{Edge} \mid \text{source: } V\text{, target: } V_{10}\text{, label: 1}\ \rangle \\
\quad \langle\ E_3 : \text{Edge} \mid \text{source: } V\text{, target: } V_{30}\text{, label: 3}\ \rangle \\
\Rightarrow\ [V_0 : \text{Node} \mid \text{label: s}]\ [V : \text{Node} \mid \text{label: h}] \\
\quad [V_3 : \text{Node} \mid \text{label: 1}]\ [V_{10} : \text{Node} \mid \text{label: s}]\ [V_{30} : \text{Node} \mid \text{label: int}] \\
\quad [E_0 : \text{Edge} \mid \text{source: } V_0\text{, target: } V]\ [E_{30} : \text{Edge} \mid \text{source: } V_{30}\text{, target: } V_3] \\
\quad \langle\ E_1\ {-1} : \text{Edge} \mid \text{source: } V\text{, target: } E_1\ {-1}\ 0\text{, label: 1}\ \rangle \\
\quad \langle\ E_1 - 1\ 0 : \text{Node} \mid \text{label: s}\ \rangle\ \langle\ E_1\ {-1} : \text{Node} \mid \text{label: g}\ \rangle \\
\quad \langle\ E_1\ {-1}\ 0 : \text{Edge} \mid \text{source: } E_1\ {-1}\ 0\text{, target: } E_1\ {-1}\ \rangle \\
\quad \langle\ E_1\ {-1}\ 1 : \text{Edge} \mid \text{source: } E_1\ {-1}\text{, target: } V_{10}\text{, label: 1}\ \rangle \\
\quad \langle\ E_1\ {-1}\ 2 : \text{Edge} \mid \text{source: } E_1\ {-1}\text{, target: } V_{10}\text{, label: 2}\ \rangle \\
\quad \langle\ E_3\ {-1} : \text{Edge} \mid \text{source: } V\text{, target: } E_3\ {-1}\ 0\text{, label: 3}\ \rangle \\
\quad \langle\ E_3\ {-1}\ 0 : \text{Node} \mid \text{label: int}\ \rangle\ \langle\ E_3\ {-1} : \text{Node} \mid \text{label: 1}\ \rangle \\
\quad \langle\ E_3\ {-1}\ 0 : \text{Edge} \mid \text{source: } E_3\ {-1}\ 0\text{, target: } E_3\ {-1}\ \rangle
\end{array}$$

Given a $\mathbb{K}$ system R, let $RWL_{Graph}(R)$ be its embedding into rewriting logic following the approach sketched above. Based on the results proved in the sections above relating $\mathbb{K}$ rewriting to graph rewriting, and on the results showing how the concurrency of graph rewriting can be captured in rewriting logic [104], we claim the following (without proof):

Conjecture 1. *The following are equivalent:*

1. $t \Rrightarrow_R t'$ *in one concurrent* $\mathbb{K}$ *step;*
2. *term2graph*$(t) \Rightarrow_{RWL_{Graph}(R)} G$ *in one concurrent step such that*

$$\text{graph2term}(G) = t'.$$

Applying the rules in parallel. To help exploring the amount of concurrency available in one $\mathbb{K}$ rewriting step, we have implemented a prototype based on the representation of terms and rules specified above. However, as the emulsifying axioms are not executable, we altered the rules so we can achieve the desired result on a different path. In essence, what we did was to change rules' behavior so that they saturate the configuration. That is, the L graph is expressed normally, without marking the parts from K as read only, while the R graph copies the parts from K unchanged, but turns all newly introduced objects into a saturated form (we use the same syntax as for read-only versions of objects) which, consequently, cannot be matched by subsequent rules. Additionally, we add another rule which can arbitrarily choose to saturate any object. In this new encoding, rule (3) is expressed as

rl $\langle\ V_0$: Node | label: s $\rangle$
$\quad\langle\ V$: Node | label: a $\rangle$ $\langle\ E_0$: Edge | source: V_0, target: $V\ \rangle$
$\Rightarrow\ \langle\ V_0$: Node | label: s $\rangle$
$\quad[V\ {-1}$: Node | label: b] $[V\ {-1}\ 0$: Edge | source: V_0, target: $V\ {-1}]$

while the saturation rule is

rl $\langle$ Oid : Cid | AS $\rangle \Rightarrow$ [Oid : Cid | AS] .

where `Oid`, `Cid`, and `AS` are variables of the corresponding sorts.

Let us call the rewrite system obtained from R by applying these transformations $\text{Maude}_{\mathit{Graph}}(R)$. We make an additional claim completing the one above (again, without proof):

Conjecture 2. *The following are equivalent:*

1. $t \Rrightarrow_R t'$ *in one concurrent* $\mathbb{K}$ *step;*

2. $\text{term2graph}(t) \Rightarrow^{+} G$*, and* $\text{graph2term}(\text{unsaturate}(G)) = t'$*, where the* `unsaturate` *function turns all saturated objects into unsaturated ones.*

Hence, assuming the acyclicity condition is satisfied, the normal forms of the graph representation of a term t obtained by rewriting using $\text{Maude}_{\mathit{Graph}}(R)$ are precisely the saturated versions of the graph representation of terms obtainable from t in one step of concurrent rewriting. When rewriting the term above using the graph rules, and then recovering the term from the graph, we can obtain the following 7 possibilities of advancing in one step: $h(g(f(b), f(b)), 1, 0)$, $h(g(f(a), f(a)), 1, 0)$, $h(g(f(b), f(b)), 0, 0)$, $h(g(f(a), f(a)), 0, 0)$, $h(f(b), 1, 1)$, $h(f(a), 1, 1)$, and $h(f(b), 0, 1)$, yielding two more terms, i.e., $h(g(f(b), f(b)), 1, 0)$, and $h(g(f(a), f(a)), 1, 0)$ in addition to those which could be obtainable through sideways and nested parallel applications of the corresponding rewrite rules.

6.6 Discussion

We have shown how the desired concurrency for $\mathbb{K}$ rewriting can be formalized and captured precisely through an embedding into graph rewriting. Moreover, we have proven that the concurrent rewriting relation is serializable and sound w.r.t. standard rewriting, implying that execution of $\mathbb{K}$ systems can be stuttering simulated by the regular rewrite systems obtained by flattening them. This result ensures that the K-Maude tool (described in the next chapter) which implements $\mathbb{K}$ on top of the Maude rewrite engine can be safely and soundly used to execute and analyze executions of $\mathbb{K}$ definitions, as long as one does not try to prove properties using the next operator of LTL, or similar constructs. Finally, we have sketched how $\mathbb{K}$ concurrent rewriting can itself be captured theoretically and explored practically using rewriting logic.

To conclude this chapter, we identify here some of the current limitations of the approach presented above and describe potential future work aimed at addressing them.

Cycle freeness. Although we have found a sufficient condition to obtain sound and serializable concurrent executions, the condition obtained is rather semantical, and might be non-trivial to check. However, as we already pointed out, it seems that all of the rule combinations in our current definitions would generate cycle free executions. An interesting research problem not addressed here is finding generic enough syntactic conditions which would guarantee that cycle freeness is satisfied for all possible combinations of matches.

Obtaining unserializability. This chapter only discussed the serializable fragment of $\mathbb{K}$, which, although it provides more concurrency in one step, does not introduce more behaviors in the transitive closure of the transition relation. However, further research is needed to understand whether having unserializable executions is desirable in certain cases, and what would be the way to model it within $\mathbb{K}$ rewriting. For example, one way of obtaining unserializable executions would be through the use of side-conditions, assuming that the constraints imposed by the side conditions are only validating the matching process and pose no restriction on the rewriting semantics. In this spirit, one could allow rules like "$f(\underset{b}{\underline{a}}, x)$ when $x = c$" and "$f(x, \underset{d}{\underline{c}})$ when $x = a$" to concurrently apply on $f(a, c)$ to obtain $f(b, d)$ which would not be obtainable through normal rewriting. However, such a treatment would require that rules with side conditions are not merely rule schemata, as they could add yet another layer to the rules, namely what part of the matching pattern is "freed" by the rule.

Appropriate graph representations for lists and sets. Graphs are free from the constraints of terms of having a specific order between the arguments of an operator; that is why, for example, we had to encode the positions of arguments as labels on edges. For this reason, graphs seem to be the perfect environment for representing associative and commutative (AC) constructs such as sets, bags, and maps: have one node standing for the AC constructor and then link each subgraph representing an element of the set/bag/map to the constructor node through an unlabeled edge. This representation would make matching modulo AC much easier and direct. One could also imagine similar representations for lists. It would be interesting to study the relation between term graphs enriched with structures like these and the terms modulo AC they abstract, and whether the use of such structures would allow representing $\mathbb{K}$ rules directly (without adapting them in order to match) as graph rewrite rules, while still maintaining the same rewriting relation on terms modulo AC.

Direct semantics for $\mathbb{K}$ rewriting. The concurrent semantics we have obtained for $\mathbb{K}$ models the desired intuition, which was easier to capture through graph rewriting due to the existing similar concurrency results available for graph rewriting. Furthermore, this chapter can be seen as a contribution to the field of term-graph rewriting in general, by introducing sharing to the rules. However, there are some limitations imposed by this semantics through embedding into graph rewriting, including going back and forth between terms and graphs, and problems with modeling non-left-linearity of the rules. Therefore a concurrent semantics for $\mathbb{K}$ directly using terms, variables, and substitutions, as the one of rewriting logic, remains a desirable goal.

Chapter 7

K-Maude—A Rewriting Logic Implementation of the $\mathbb{K}$ Framework

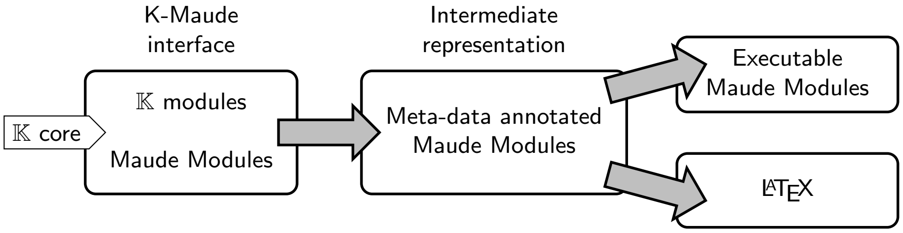


Figure 7.1: K-Maude overview. Grayed arrows correspond to translating tools.

The $\mathbb{K}$ technique has been manually (without automated tool support) used in the context of rewriting logic and Maude for more than five years, for teaching programming language and program verification courses as well as for several research projects. Such of $\mathbb{K}$ in Maude turned our to be verbose and error prone, because Maude is a general purpose rewrite engine not specifically optimized for programming languages. Thus, the idea of developing a $\mathbb{K}$ specialized layer on top of Maude came naturally. The resulting integrated toolkit is called K-Maude and is the subject of this chapter.

As rewriting logic relies on the Maude language and rewrite engine to be able to write, execute, and analyze rewrite theories, the $\mathbb{K}$ Framework relies on the K-Maude tool for writing, executing and analyzing $\mathbb{K}$ definitions of programming languages and analysis tools. The K-Maude language is an ASCII representation of the $\mathbb{K}$ mathematical language and K-Maude definitions are translated into Maude rewrite theories for execution and analysis purposes, or in LaTeX definitions for inclusions in research papers and presentations. Most $\mathbb{K}$ definitions presented in this dissertation were generated by massaging the LaTeX output of the K-Maude tool.

Figure 7.1 shows the architecture of K-Maude. The gray arrows represent translators implemented as part of the toolkit. The $\mathbb{K}$ core contains the ingredients of the $\mathbb{K}$ technique, that are handy in most language definitions, such as ones for defining computations, configurations, environments, stores, etc. The K-Maude interface is what the user typically sees: besides usual Maude modules (K-Maude fully extends Maude), one can also include K-Maude files (with extension `.k`) containing modules using the $\mathbb{K}$ specialized notation.

A first component of K-Maude translates $\mathbb{K}$ modules to Maude modules. The resulting Maude modules encode $\mathbb{K}$-specific features as meta-data attributes and serve as an intermediate representation of K-Maude definitions. Since this representation is just an artifact of using Maude, we will refrain from describing it and we will identify it with the $\mathbb{K}$ module it stands for. This intermediate representation can be further translated to different back-ends. We provide two such translators, one to executable/analyzable Maude and one to LaTeX. The former yields actual executable language definitions in Maude which can serve as interpreters for the defined languages or as a basis for formal analysis.

The K-Maude to LaTeX translator is meant to serve for documentation purposes. Indeed, we believe that $\mathbb{K}$ can be used by ordinary language designers as a formal notation for rigorously specifying the semantics of their languages, the same way context-free grammars are used for formally specifying syntax, so a user-friendly LaTeX notation may be preferred.

The remainder of this chapter is organized as follows. Section 7.1 briefly introduces the ASCII notation for the $\mathbb{K}$ definitional framework. Section 7.2 gives a user perspective of K-Maude, both w.r.t. its built-in features and how it can be used. Section 7.3 describes how K-Maude is translated to Maude, so that language designers can execute and formally analyze their $\mathbb{K}$ language definitions using Maude, and Section 7.4 describes how K-Maude is translated to LaTeX, so that language designers can visualize their language definitions.

7.1 Writing $\mathbb{K}$ in ASCII

As the $\mathbb{K}$ framework [146, 143] was discussed at length in Chapter 4, we will here only briefly recall its features, while focusing on their current representation in the K-Maude tool.

The idea underlying language semantics in $\mathbb{K}$ is to represent the program configuration as a "nested soup" structure, which contains the context needed for the computation, with elements of the context represented as multisets or lists each *wrapped* inside a corresponding *cell*; a cell may also contain other cells. Objects wrapped by cells generally include standard items such as environments, stores, etc, as well as items specific to the given semantics. Mathematically, cells are written using the notation $\langle\ldots\rangle_{\mathsf{env}}$; here '$\mathsf{env}$' denotes the *cell label* and '...' will represent the contents of the cell. When written in ASCII, such as in K-Maude, we prefer to use the XML-like notation ⟨env⟩ ... ⟨/env⟩. One regularly used cell, labeled by 'k', represents the current *computations structure* of sort K, or simply the *computation*, which is a $\curvearrowright$-separated list of tasks, such as $t_1 \curvearrowright t_2 \curvearrowright \ldots \curvearrowright t_n$. Another, labeled by $\top$, represents the entire configuration structure.

Figure 7.2 presents the definition of IMP++, a concurrent imperative language using the $\mathbb{K}$ framework, as written in the K-Maude extension of Maude. The definition bares a resemblance to the general Maude module syntax, but, in

```
kmod IMPPP−SYNTAX is
  including PL−INT + PL−ID
  syntax AExp ::= Int | Id
      | AExp + AExp
        [gather(E e) prec 33 strict]
      | AExp / AExp
        [gather(E e) prec 31 strict]
      | ++ Id [prec 0]
      | read
  syntax BExp ::= Bool
      | AExp <= AExp
        [prec 37 latex "{#1}\leq{#2}"
         seqstrict]
      | not BExp [prec 53 strict]
      | BExp and BExp
        [gather(E e) prec 55 strict (1)]
  syntax Stmt ::= { }
      | AExp ; [prec 90 strict]
      | Stmt Stmt [prec 100 gather(e E)]
      | Id = AExp ;
        [prec 80 gather (e E) strict (2)]
      | if BExp then Stmt else Stmt
        [strict (1)]
      | while BExp do Stmt
      | print AExp ; [strict]
      | spawn Stmt [prec 90]
      | haltThread ;
      | var Id ;  [prec 2]
      | { Stmt } [gather(&)]
endkm

kmod IMPPP−CONFIGURATION is
 including IMPPP−SYNTAX + K
 syntax K ::= AExp | BExp | Stmt
 syntax KResult ::= Bool | Int
 configuration
  ⟨T⟩
   ⟨threads⟩⟨thread∗⟩
    ⟨k⟩.K⟨/k⟩ ⟨env⟩.Map⟨/env⟩
   ⟨/thread∗⟩⟨/threads⟩
   ⟨ store ⟩.Map⟨/store⟩
   ⟨nextLoc⟩0⟨/nextLoc⟩
   ⟨in⟩.List⟨/in⟩ ⟨out⟩.List⟨/out⟩
  ⟨/T⟩
endkm
```

```
kmod IMPPP−SEMANTICS is
 including IMPPP−CONFIGURATION
 rule ⟨k⟩X:Id ⇒ I:Int⟨_/k⟩
        ⟨env_⟩X ↦ N:Nat⟨_/env⟩
        ⟨ store_⟩N ↦ I⟨_/ store ⟩
 rule I1:Int + I2:Int ⇒ I1 +Int I2
 rule I1 / I2 ⇒ I1 /Int I2 if I2 ! =Int 0
 rule ⟨k⟩++ X ⇒I +Int 1⟨_/k⟩
        ⟨env_⟩X ↦ N⟨_/env⟩
        ⟨ store_⟩N ↦(I ⇒ I +Int 1)⟨_/ store ⟩
 rule ⟨k⟩read ⇒ I⟨_/k⟩
        ⟨ in ⟩ListItem(I) ⇒ .⟨_/ in ⟩
 rule I1 <= I2 ⇒ I1 <=Int I2
 rule not T:Bool ⇒ notBool T
 rule true and B:BExp ⇒B
 rule false and _ ⇒ false
 rule {} ⇒ .    rule I ; ⇒ .
 rule S1:Stmt S2:Stmt ⇒ S1 ↷ S2
 rule ⟨k⟩X = I ; ⇒ .⟨_/k⟩
        ⟨env_⟩X ↦ N⟨_/env⟩
        ⟨ store_⟩N ↦(_ ⇒ I)⟨_/ store ⟩
 rule if true then S1 else _ ⇒ S1
 rule if false then _else S2 ⇒ S2
 rule ⟨k⟩while B do S:Stmt ⇒
  if B then S while B do S else {}⟨_/k⟩
 rule ⟨k⟩print I ; ⇒ .⟨_/k⟩
        ⟨out_⟩. ⇒ ListItem(I)⟨/out⟩
 rule ⟨k⟩spawn S ⇒ .⟨_/k⟩
        ⟨env⟩Env:Map⟨/env⟩
        (. ⇒ ⟨thread_⟩
                 ⟨k⟩S⟨/k⟩
                 ⟨env⟩Env⟨/env⟩
               ⟨_/thread⟩)
 rule ⟨thread_⟩⟨k⟩.K⟨/k⟩ ⟨_/thread⟩ ⇒ .
 rule ⟨k⟩haltThread ; ↷ _ ⇒ .⟨/k⟩
 rule ⟨k⟩var X ; ⇒ .⟨_/k⟩
        ⟨env⟩Env ⇒ Env[N / X]⟨/env⟩
        ⟨ store_⟩. ⇒ N ↦ 0⟨_/ store ⟩
        ⟨nextLoc⟩N ⇒ N +Nat 1⟨/nextLoc⟩
 rule ⟨k⟩{S} ⇒ S ↷ env(Env)⟨_/k⟩
        ⟨env⟩Env⟨/env⟩
 syntax K ::= env ( Map )
 rule env(_) ↷ env(Env) ⇒ env(Env)
 rule ⟨k⟩env(Env) ⇒ .⟨_/k⟩
        ⟨env⟩ _ ⇒ Env⟨/env⟩
endkm
```

Figure 7.2: Full definition of IMP++ in K-Maude

addition to Maude syntax, it uses several $\mathbb{K}$ specific constructs which will be detailed in the sequel. Although the definition is written using pure ASCII in the K-Maude tool, we have replaced some of the ASCII symbols with mathematical symbols when typesetting, to improve readability. That is, we have replaced, here, and everywhere else in this chapter where code is presented the following: the mapping construct '|->' by '↦', the $\mathbb{K}$ arrow '~>' by '↷', the operation definition keyword '->' by '→', and the rewrititng construct '=>' by '⇒'.

Although not enforced by the K-Maude tool, the Imp++ presented below is divided into three modules: syntax, configuration, and semantics. To point out that, although they bear similarities to Maude modules, the $\mathbb{K}$ modules are quite different, we introduce them by the special keyword **kmod_is_endkm**. The left column presents the syntax of Imp++ (lines 1–31) and the default configuration of a running program (lines 33–46), while the right column presents the executable semantics of Imp++. Let us start describing Imp++ from its configuration module (lines 33–46). Imp++'s computations consist of expressions(arithmetic and booelan), and statements (line 35). Among them, Boolean and integers are distinguished as results, that is, finished computations (line 36). Execution wise, Imp++ is an environment-based multi-threaded language (lines 39–41); The $\ast$ postfixed to the name of the thread cell indicates its multiplicity. All threads are grouped in a threads cell and share a common store (line 42), as well as an input and an output stream (line 44). The entire configuration is contained in a top cell T.

An particularly important (in the context of programming language definitions) extension of Maude employed by K-Maude is the ability to use the Backus-Naur Form (BNF) of declaring syntax, introduced by the **syntax** keyword, in addition to the algebraic mix-fix operation declaration syntax provided by Maude. Although this syntax is equivalent with the algebraic one using sort for non-terminals and subsorts and mixfix operations for productions, and in fact it currently is simply syntactic sugar for that, we believe that it is more appealing for people already familiar with BNF from parsing/compiling programming languages.

Arithmetic expressions are constructed from variables and integers with addition and division (lines 2–7, 49–53), but additionally include variable increment (lines 8, 54–56), to exhibit side effects, and external input (lines 9,57–58). Boolean expressions (lines 10–16, 59–62) are constructed from comparing arithmetic expressions, with conjunction and negation as connectives. Statements consists of standard constructs as the empty statement, the expression statement, sequential composition, assignment, conditional, and loop (lines 17–24, 63–71), to which were added the following: output (lines 25, 72–73), thread creation and dissolution (lines 26, 74–80), abrupt thread termination (lines 27, 81), as well as local variable declarations and blocks (lines 28–29, 82–91). To exhibit features of K-Maude not shown by the Imp++ definition, we will also discuss features from the Agent definition presented in Section 4.3.

Due to $\mathbb{K}$'s own use of the '.' symbol (for generic unit), we do not use '.' as terminator in $\mathbb{K}$ modules, as mandatory in Maude modules; instead, we rely on reserved keywords such as **including**, **syntax**, **configuration**, **context**, and **rule**, to disambiguate declarations. The special BNF syntax declarations described above can be annotated with Maude operator attributes to ease disambiguation (such as precedence and gathering), or to specify semantic attributes (such as associativity). However, in addition to Maude's attributes, $\mathbb{K}$ specific attributes can be added, such as **strict**, which is used to specify that (certain) arguments of a language construct need to be evaluated first (and their effects on the global state be propagated) before giving semantics to the construct itself.

$\mathbb{K}$ Rewrite Rules

A $\mathbb{K}$ definition consists of two types of sentences: structural rules (often reversible, like equations) and computational rules (typically non-reversible).

Structural rules carry no computational meaning; instead, borrowing a concept from Chams, structural rules can *heat* and *cool* computations. When a computation is heated, it breaks into smaller pieces, exposing subexpressions of more complex expressions for evaluation. Cooling reverses this process, reassembling the (potentially modified) pieces into a computation with the same "shape". The following are examples of structural rules:

$$a_1 + a_2 \rightleftharpoons a_2 \curvearrowright a_1 + \square$$

$$a_1 + a_2 \rightleftharpoons a_1 \curvearrowright \square + a_2$$

$$\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \rightleftharpoons b \curvearrowright \texttt{if}\ \square\ \texttt{then}\ s_1\ \texttt{else}\ s_2$$

Language syntax is completely abstract in $\mathbb{K}$, in the sense that each language construct is a 'KLabel' which is applied to other computations, i.e., terms of sort K; for convenience, and also supported by the K-Maude tool, we continue to use the mix-fix notation for syntax, like above. Unlike in evaluation contexts, $\square$ is not a "hole", but rather part of a KLabel, carrying the obvious "plug" intuition; e.g., the KLabels involving $\square$ above are '$\square + \_$', '$\_ + \square$', and '`if` $\square$ `then` _ `else` _'.

Many structural rules can be automatically generated by annotating constructs in the language syntax with '**strict**' attributes: a '**strict**' attribute generates the appropriate structural rules for each strict argument. If an operator is intended to be strict in only some of its arguments, then the positions of the strict arguments are listed as arguments of the '**strict**' attribute; for example, the first two equations above directly correspond to the attribute '**strict**' for addition in Imp++ (line 5), i.e., strict in all arguments, while the last one corresponds to **strict**(1) attribute used for the Imp++ conditional (line 24). One can also define evaluation contexts in $\mathbb{K}$, by indicating the "hole" where evaluation should take place; for example, assuming an extension of Imp++ with pointers and a C-like dereferencing operator ∗_ (like the Agent definition presented in Section 4.3), a context declaration ‘**context** ∗ [HOLE] = _;’ says that the argument of ∗_ needs to be evaluated before the assignment can be defined.

Computational rules represent actual steps of computation. However, to account for the differences between $\mathbb{K}$ rules and regular rewrite rules, we chose to introduce $\mathbb{K}$ rules with the **rule** keyword, even when they have the form of regular rewrite rules, e.g., the rules for addition and conditional (line 52, and lines 68–69, respectively). **rule** can also be used to express structural rules, by adding the **structural** attribute to the end of the rule.

In-place rewriting. In addition to regular rewrite rules, of the form ‘**rule** l ⇒ r’, $\mathbb{K}$ allows one to also write rules using the following contextual notation:

$$C[\frac{t_1}{t'_1}, \frac{t_2}{t'_2}, ..., \frac{t_n}{t'_n}]$$

which says that in (multi-)context C (that is a term with multiple, ordered holes), each pattern t_i rewrites to t'_i for each $i \in \{1, ..., n\}$. An n-hole context could formally be described as a term over the set of variables $\{\square_i\}_{1 \leq i \leq n}$ containing exactly one occurrence of each variable $\square_i$. An instantiation of an n-context C with terms $t_1, \ldots, t_n$, written $C[t_1, \ldots, t_n]$ is obtained by applying on C the substitution yielding t_i for each $\square_i$.

One motivation for in-place rewriting rules is that they allow for a more compact and less error-prone representation for rules matching large configurations but effecting only small changes. Another motivation is that the context C can be concurrently shared by various rules, which can apply concurrently provided that none of them changes C (C is “read only”). If one ignores this concurrency aspect, then one can translate each $\mathbb{K}$ contextual rule into a rule $C[t_1, t_2, ..., t_n] \rightarrow C[t'_1, t'_2, ..., t'_n]$; this is precisely what K-Maude does. In K-Maude, the mathematical in-place rewriting $\frac{l}{r}$ is injected in-place as ‘l ⇒ r’. By default, the in-place rewriting construct ‘_ ⇒ _’ is greedy; parenthesis can be used for disambiguation purposes (see, e.g., lines 56, and 76–79).

Anonymous variables. Another advantage emerging from the single-term representation of rules induced by in-place rewriting is that, variables occurring only once in the rule can now be “anonymized”, that is, replaced by the anonymous variable symbol ‘ _ ’, since they are only used for matching purposes. This is especially true in the case of matching inside cells, of which we usually use/replace only one object in a rule, but we need to match the contents of the entire cell. A special notation for cells is used to help the intuition that the cells might be “open” at one end, or both. For example, in the rule for reading a variable from the store (lines 49–51), one can either use $\langle\text{k}\rangle\text{X} \Rightarrow \text{I}\langle\_/\text{k}\rangle$ or $\langle\text{k}\rangle(\text{X} \Rightarrow \text{I}) \curvearrowright \_\langle/\text{k}\rangle$ to specify that X is to be matched and replaced at the beginning of computation, and use either $\langle\text{env}\_\rangle\text{X}{\mapsto}\text{N}\langle\_/\text{env}\rangle$ or $\langle\text{env}\rangle\_\ \text{X} \mapsto \text{N}\ \langle/\text{env}\rangle$ to specify that $\text{X} \mapsto \text{N}$ is to be matched in the middle of the environment; also, one can use either $\langle\text{out}\_\rangle\texttt{.} \Rightarrow \text{ListItem(I)}\langle/\text{out}\rangle$ or $\langle\text{out}\rangle\_(\texttt{.} \Rightarrow \text{ListItem(I)})\langle/\text{out}\rangle$ to add an integer to the end of the output list (line 73). One could think of this notation as having the following intuition: ⟨⟩ is a membrane delimiter, which can carry inside attributes such as name (e.g., ‘out’), visual information specifying that a membrane is closing the cell (‘/’), as well as information specifying whether a part of the cell (the left, the right, or both) is subsumed by the pattern (‘_’). Therefore one can read the “tag” $\langle\_/\text{k}\rangle$ as: the membrane closing the k cell while subsuming the final part of the cell. By convention we will always attach ‘/’ to the name of the cell, and ‘_’ to the membrane wall closest to the cell contents.

As previously mentioned, the unit elements for the List, Set, Map, and even $\mathbb{K}$ sorts is ‘.’; however, since cells are not typed, whenever disambiguation is needed, one can postfix the name of the sort to the ‘.’; e.g., the initial values in the configuration term (lines 37–45), or the empty computation ‘.K’ used in the rule for dissolving a thread (line 80). The preprocessor transform them to the right constants used in the Maude represenation. Also, the preprocessor allows one to avoid variable declarations by declaring variables inline in the rules, using the Maude syntax, e.g., I:Int. However, once a variable was declared inline, subsequent apparitions of the variable need not be sorted anymore, as they would assume the alread declared sort.

The notation used by the K-Maude tool is a one dimensional ASCII rendition of the $\mathbb{K}$ mathematical notation presented in Chapter 4. For example, the $\mathbb{K}$ mathematical rendering of the IMP++ assignment rule (lines 65–67) is:

$$\langle \underset{\cdot}{\underline{X\ =\ I;}}\ \_\rangle_{\mathsf{k}}\ \langle\_\ \ X \mapsto N\ \ \_\rangle_{\mathsf{env}}\ \langle\_\ \ N \mapsto \underset{I}{\underline{\_}}\ \ \_\rangle_{\mathsf{store}}$$

The above rule says that if the assignment $X = I$ is the first computational task, and if X is at location N in the environment, then replace whatever is at location N in the store by I and discard the assignment. Note that in the mathematical notation the membranes wrapping the cells, e.g., $\langle\text{env}\_\rangle$ $\langle\_/\text{env}\rangle$ are replaced by “thinner” membranes, e.g., $\langle\_\quad\_\rangle_{\mathsf{env}}$, but still maintain all relevant information.

7.2 K-Maude Interface

For the purpose of this chapter, $\mathbb{K}$ can be regarded as a notational layer on top of rewriting logic, specialized and optimized for writing definitions of complex programming languages and models. Since our aim for K-Maude is to fully support rewriting logic and Maude, we implemented it as an extension of Maude.

Consequently, one is free to use or not the $\mathbb{K}$ notation when writing language definitions. An extreme approach, which could be convenient for existing Maude users who want to gradually get exposed to $\mathbb{K}$, is to only include the provided K-Maude core and then follow the $\mathbb{K}$ language definitional technique but use plain Maude, the same way one can give SOS or other semantic definitions using plain Maude, as depicted in Chapter 3. This section presents the ingredients of both the $\mathbb{K}$ technique and the specific $\mathbb{K}$ notation used by the K-Maude tool. However, we will insist more on notation here, and refer the interested reader to the specific sections from Chapter 4 discussing in depth the corresponding concepts within the $\mathbb{K}$ technique.

7.2.1 K-Maude Core

The core syntax of K-Maude can be found in file 'k-prelude.maude', module `K-TECHNIQUE`. It starts by providing means to build computations, as sequences of abstract syntax trees, as well as distinguishing among them the result computations. Then it defines lists, bags, sets, and maps, as sorts and a way to inject computations as their elements. Finally, the core provides minimal support for describing configurations as "nested soups" of cells, along with two default cell names, 'k' and 'T'.

Basic $\mathbb{K}$ syntax.

A computation is a term of a specific sort K, and is defined as a list of tasks with identity '$\cdot$' and constructor '$\_ \curvearrowright \_$' as well as a way of building structured computations by applying labels (one per language construct, and additional, as needed) on top of lists of computations:

$$\textbf{op}\ \_\curvearrowright\_\ :\ K\ K \rightarrow\ K\ [\text{prec } 100\ \ \text{assoc id}:\ .]\ \ .$$
$$\textbf{op}\ \_(\_)\ :\ \mathit{KLabel}\ \mathit{List}\{K\}\ \rightarrow\ K\ [\text{prec } 0 \text{ gather}(\&\ \&)]\ .$$

In K-Maude, the sort List{K} is built from $\mathbb{K}$ using '$\_\,,,\,\_$' as a constructor (to allow user to use the single ',' in definitions) and '$.$List{K}' as unit (to disambiguate from the unit of computations). Finished computations are distinguished to allow for a computational treatment of strictness rules. Sort KResult, is meant to describe *results*, or computations which need no further evaluation, and sort List{KResult} is the subsort of ListK built only from results. We additionally introduce KResultLabel and KHybridLabel, as subsorts of KLabel, together with their corresponding application constructors:

$$\textbf{op}\ \_(\_)\ :\ \text{KResultLabel}\ \mathit{List}\{K\} \rightarrow \mathit{KResult}\ [\text{ditto}]\ \ .$$
$$\textbf{op}\ \_(\_)\ :\ \text{KHybridLabel}\ \mathit{List}\{\mathit{KResult}\} \rightarrow \mathit{KResult}\ [\text{ditto}]\ \ .$$

The distinction between the two is that, while the first encapsulates the entire list of computations below into a result, the second is "hybrid", that is, it only becomes a result when all the computations it "wraps" become results.

Lists, bags, sets, and maps.

K-Maude provides generic sorts List, Set, Bag and Map, constructed from their corresponding element sorts ListItem, SetItem, BagItem and MapItem, with construtor ‘__’ and having unit ‘.’. Moreover, the following injections of 𝕂 into the elemnt sorts are provided: ListItem, SetItem, BagItem, and ‘_↦_’ (to inject a pair of 𝕂’s into MapItem).

Labelled Cells.

The configuration is defined as a structured “soup” of cells. Therefore, we use the already defined sort Bag to hold such a collection of cells. Having a unique sort for cells makes cell nesting easy; a cell holding other cells simply needs to take a Bag as an argument. To declare a cell, one only needs to specify its label, as a ‘CellLabel’.

```
sort CellLabel .

op ⟨_⟩_⟨/_⟩ : CellLabel K CellLabel → BagItem [prec 0] .
op ⟨_⟩_⟨/_⟩ : CellLabel List CellLabel → BagItem [prec 0] .
op ⟨_⟩_⟨/_⟩ : CellLabel Bag CellLabel → BagItem [prec 0] .
op ⟨_⟩_⟨/_⟩ : CellLabel Set CellLabel → BagItem [prec 0] .
op ⟨_⟩_⟨/_⟩ : CellLabel Map CellLabel → BagItem [prec 0] .

ops k T : → CellLabel .
```

K-Maude currently allows five kind of cells, i.e., containing either a computation, a list, a bag, a set, or a map. The syntax for the cells is defined as that of an XML element, with an opening, and closing tag, which must match (i.e., the CellLabels must be equal).

7.2.2 K-Maude Specific Modules

The K-Maude modules are introduced by a specific keyword **kmod_is_endkm**, to distinguish them from usual Maude modules; however, this is purely a syntactic sugar, as the internal representation of a 𝕂 module in Maude would be a proper system module. A script, external to Maude, is used to preprocess the 𝕂 modules into a form which can be parsed by Maude, for example by a adding the terminator ‘.’ at the end of declarations, by changing **kmod** into ‘**mod**’ and **endkm** into ‘**endm**’, and by wrapping specific 𝕂 attributes in ‘metadata’ strings.

7.2.3 Language Syntax and Annotations

As already mentioned, the syntax of the language is defined using BNF rules; however, the preprocessor transforms each such BNF rule to a Maude declaration of a mix-fix algebraic operation, following the equivalence between CFG grammars and mix-fix algebraic signatures. For example, the algebraic operation associated to the BNF rule for conditional (line 22) is translated into the operation declaration **op** if_then_else_ : BExp Stmt Stmt →Stmt.

Syntax Attributes.

In addition to the existing operation declaration attributes provided by Maude, K-Maude introduces several new attributes:
– **strict** specifies what arguments need to be evaluated before evaluating the language construct itself. For example, the '**strict**(2)' attribute of the assignment declaration in Imp++ (line 22), states that the semantic rule for assignment can assume that the second argument is evaluated.
– **seqstrict** is similar, but also states the evaluation *sequence* of the arguments.
– **hybrid** specifies that the language construct would become a value once all its arguments have been fully evaluated.

7.2.4 Defining the Program Configuration

Currently, the program configuration is specified by providing a term which should stand for the initial configuration, introduced by the **configuration** keyword. The cell labels present in the configuration are then inferred and declared as CellLabel constants by the preprocessing tool. Moreover, to keep closer to the XML notation we do not write spaces around the cell labels; these are also added by the preprocessing tool.

Specifying the structure of the configuration serves not only for documenting purposes, but has consequences in both modularity and compactness of definitions, since the semantics rules need to mention only the context required for them to apply, as detailed in next section.

7.2.5 Defining Language Semantics

𝕂-specific semantics constructs which are supported by K-Maude are contexts, and (structural or computational) rules. Contexts can be though of as evaluation contexts, specifying the order of evaluation, while 𝕂 rules provide notational shortcuts to make definitions more compact.

Context Strictness.

𝕂-contexts are usually used for specifying strictness constraints which depend on a context rather than on a single construct. For example, the context strictness declaration '**context** $*$ [HOLE] = _' (see the Agent definition in Section 4.3) specifies the evaluation to an L-value of a pointer in the assignment construct, allowing the rule for pointer-assignment to assume it has a value in place of the first argument (the hole); the fact that it would also have a value instead of the second argument was specified by the strictness annotation for '_=_'.

Actually, all strictness annotations are turned by the tool into context strictness ones during the compilation process, before the actual heating and cooling equations are being generated.

$\mathbb{K}$ Rules.

$\mathbb{K}$ rules are introduced by the **rule** keyword, and basically describe a special pattern term enriched with syntax for expressing the K-specific features described below. There are two flavors, structural and computational, distinguished by adding the attribute **structural** to the former. The intuition is that the structural rules prepare the program state for a computational step. Therefore, the K-Maude tool translates the former to equations and the latter to rules.

In-place rewriting. A $\mathbb{K}$ rule is a term which should contain at least one occurrence of the T1 $\Rightarrow$ T2 construct, which is used as a textual representation for the $\mathbb{K}$ visual replacement pattern: $\frac{T1}{T2}$. For example, the increment rule of Imp++ (lines 54–56) contains two non-trivial in-place replacements, while also sharing quite a bit of the context: if the construct '++ X' is found on top of the computation, and the environment contains the mapping of X to a location N, and the store maps N to an integer I, then '++ X' is locally replaced by the value associated to $I + 1$, and I is locally replaced by $I + 1$ in the store. The '$\_ \Rightarrow \_$' arrow is greedy, i.e., it will expand to the nearest enclosing boundaries. Therefore, one might sometimes need to use parenthesis to clearly fix those boundaries for parsing reasons, e.g., changing the value at a location in the store in line 56, but also semantic reasons, e.g., new thread creation in lines 76–79.

Anonymous variables. Specified by '_', anonymous variables can be used to replace all variables whose name is not needed in the match-and-replace process. For example, the Imp++ 'haltThread' rule (line 81) uses an anonymous variable to abstract the remainder of the computation, since it will be discarded by the rule.

Cell comprehension. K-Maude allows partial specification of the contents of a cell, by adding '_' as an attribute inside the membrane delimiter '$\langle\rangle$' on the side of the cell which should be abstracted away. For example, the Imp++ rule for output (lines 72–73), saying that the integer argument of a 'print' statement is appended to the contents of the 'output' cell, abstracts away both the rest of the computation, and the existing output list, as not being changed by the rule.

Context Abstraction and Context Transformation.

The main reason for specifying the structure of the configuration is that one does not need to mention the full context required for the application of a rule, but only the parts which are relevant. Within a rigid configuration structure in which the path to each cell is unambiguous, it becomes straight-forward to infer what context needs to be added to a rule to adapt it to the running configuration. A simple instance of using context abstraction is the Imp++ assignment rule (lines 65–67). The rule for assignment should be the same in any definition containing an environment and a store. Although in our definition the store is not at the same level with the computation and the environment, we can still use this rule in the specification, because it can be easily inferred which store the rule refers to.

The following rule could be used to define a rendez-vous synchronization construct as follows:

rule ⟨k⟩ rv i ⇒ . ⟨_/k⟩ ⟨k⟩ rv i ⇒ . ⟨_/k⟩ .

Note that, although the two computation cells need to be in two different threads, there is no danger of confusion, since the multiplicity of the 'k' cell is one, so the only way to make sense of this rule is to have each computation in its own thread, since the multiplicity of the 'thread' cell may vary.

Default contexts. Another aspect of the context abstraction with impact on modularity is filling the context with default values on the right-hand-side of an (in-place) rewriting pattern. One such example is the Imp++ rule for thread creation (lines 74–79). Note that we have specified the thread cell as being incomplete in both sides. This is used as a notation to specify that the thread cell is incompletely specified, and thus it should be context-transformed, filling all gaps with default values. For this specific configuration, this notation was not necessary, but this allows for modular changes of configuration when adding cells having constant initial values when a thread is started, such as a function call stack or a set of locks hold by the thread.

It is arguable that context abstraction could have undesirable effects on badly written specifications. However, due to its deterministic nature, we believe it to be rather useful and intuitive. Besides saving the need for providing additional context (which could get quite large and tedious to write), and thus providing brevity to specification, it also enables reusing, since now a rule specifies only the minimal needed context. Moreover, K-Maude desugars 𝕂 rules to pure rewriting logic rules and equations, so one could always inspect the resulting rules to ensure no unexpected behaviors are introduced when resolving the context abstraction.

Rewrite Rules.

One can additionally use regular rewrite rules and equations when giving semantics to the language constructs. The 𝕂-specific syntactic conventions presented above also apply to them; e.g., one can use '_' as an anonymous variable in the left-hand-side of a rule, and even context abstraction in the right-hand-side, which is useful, for example, to set up the initial configuration when starting the execution of a program.

7.3 From K-Maude to Maude

This section describes the technical part of the K-Maude tool. As the semantics of the $\mathbb{K}$ framework itself is given using rewriting logic, it comes natural that the executable semantics of $\mathbb{K}$, as given by the K-Maude tool, is given by reduction to pure Maude (executable) rewrite theories. That is, each of the K-specific features is transformed into its rewriting logic representation.

Syntax.

As mentioned in Sec. 7.2.3, the K-Maude interface allows the definition of syntax as an algebraic signature, using subsorting and mixfix operations to emulate CFG grammar descriptions. This allows the programs to look more natural, but also, more important, it improves the readiness of the semantic rules. Nevertheless, as previously mentioned, the $\mathbb{K}$ framework takes a fairly abstract view on syntax, that is, a tree built as labels applied to (possibly empty) lists of subtrees. To achieve that, the K-Maude tool transforms all syntax into labels. With syntax being just labels and with the distinction between value (of sort KResult) and non-value computations, the strictness attributes are easily desugared as heating only on non-value computations and cooling only on values.

Semantics.

The semantic part of a $\mathbb{K}$ definition is gradually transformed into an executable Maude module as follows: First, the configuration term is used to resolve context abstraction. Next, cell comprehension is resolved by adding anonymous variables, which, in their turn are replaced with proper (fresh) variables of the right sort. Then, $\mathbb{K}$ rules are transformed into rewriting logic equations and rules, by resolving the in-place rewriting. Finally, all computation terms (including the test programs specified by the user) are transformed into ASTs.

Besides the original preprocessor, which wraps the $\mathbb{K}$ definitions so that they can be recognized and parsed by Maude, all syntax and semantic transformations are entirely defined within Maude, taking advantage of its reflective capabilities and of the predefined Maude modules used to represent and transform meta-terms and meta-modules. In the sequel we present some highlights of the process described above.

7.3.1 From Syntax to $\mathbb{K}$ Syntax and $\mathbb{K}$ Representation

When specifying the syntax we want to take advantage of the full power of specification, to obtain a syntax as close as possible to the desired language syntax, which also makes the semantics rules look more natural and easy to read. To do that, and to reduce parsing conflicts, we allow specification features as multiple sorts, mixfix operators, and so on. However, we want to keep computations to a minimal structure to facilitate easy and generic traversal functions, which are crucial for advanced reflective features such as code generation (see, e.g., the Agent defintion 4.3). To achieve this, the K-Maude tool automatically generates the labels for the abstract (running) syntax from the input (user) syntax.

Abstract Syntax.

The 𝕂 running syntax only consists of K labels, as defined in the core syntax of computations presented above. Since the semantic rules mix the syntax with semantics-specific contructs, and use them in contexts where computations are required, the user has to subsort all syntactic categories to computation sorts K and KResult, depending on whether they represent proper computations or values, respectively. The tool uses this information to generate the appropriate labels, i.e., constants of the 'KLabel' or the 'KResultLabel' sorts, for each operation symbol. For example, for the conditional construct, its corresponding K label declaration is '**op** 'if_then_else_ : → KLabel' To avoid label symbol conflicts, the K label symbols are generated by simply quoting the identifier used to declare the mixfix syntactic construct.

Handling Data Types.

There are certain sorts, such as integers, booleans, and identifiers, which need to be handled in a special way, to be able to identify them when giving the semantics. To address that, we allow certain sorts to be identified as builtins at the user level, by introducing a new computation sort Builtins and subsorting all such sorts to it. These sorts will be injected into labels in an appropriate manner, following the subsorting chain to either KResult or K. For example, integers are injected into KResultLabel as '**op** Int_ : Int → KResultLabel', since they are subsorted to KResult, while identifiers are injected to KLabel through '**op** Id_ : Id → KLabel'.

Translating Terms.

The constant labels and constant injections defined above are used to completely replace the original syntax. For example, the program fragment

```
if a <= 2 then a = 2; else {}
```

would be translated to:

```
'if_then_else('_<=_(Id a(.List{K}),,Int 2(.List{K})),,
                 '_=_;(Id a(.List{K}),,Int 2(.List{K })),,'‘{‘}(. List{K}))
```

7.3.2 Strictness

Strictness declarations provided as attributes to operator declarations are translated into 𝕂 contexts declarations, one for each position in which the operation should be strict. Then, each context is transformed into two equations: one which pulls the strict argument (represented by the hole) out of the context for evaluation, and another one which, once the argument becomes a value, plugs it back into its original context.

Strict Operator Attributes.

For each position declared as strict for an operation, a context declaration is generated, containing a hole. For example, the **strict**(2) declaration for the assignment operation in IMP++ (line 22) would generate the following context declaration: **context** '_=_;(K1:K,,[HOLE]), while the **seqstrict** declaration for _<=_ is desugared into two context declarations, **context** '_<=_([HOLE],,K1:K) and **context** '_<=_(K1:KResult,,[HOLE]). Sequential strictness in esured by requiring the first argument of the last context above to be an evaluated computation.

Strict Contexts.

Although we could identify proper computations by a side condition testing that they are not of sort KResult, we prefer to introduce a new category of computations, KProper, with the intuition that KProper and KResult form a partition of the K sort. Since all computations are built by applying labels on other lists of computation, we therefore also introduce the sort KProperLabel, and change all existing label definitions such that any K label which is not a result label will be a proper label. For example, the label associated to the conditional would now have KProperLabel as its resulting sort; the same holds for the 'Id_' injection. Having KProper computations, the strict contexts are desugared as follows: two equations are generated for each context, one for pulling out the *proper* computation for evaluation and the other for plugging in the *result* computation. For the assignment operation declared strict in the second argument, the generated equations are:

```
eq ⟨ k ⟩ K1:K = Kcxt:KProper ↷Rest:K ⟨/ k ⟩
 = ⟨ k ⟩ Kcxt:KProper ↷ freezer ("'_=_;(K1:K,,'[HOLE']:K)")(
                      freezeVar("K1:K")(K1:K)) ↷Rest:K ⟨/ k ⟩ .
eq ⟨ k ⟩ Kcxt:KResult ↷ freezer ("'_=_;(K1:K,,'[HOLE']:K)")(
                      freezeVar("K1:K")(K1:K)) ↷Rest:K ⟨/ k ⟩ .
 = ⟨ k ⟩ K1:K = Kcxt:KResult ↷Rest:K ⟨/ k ⟩
```

These equations apply only at the top of the continuation, because they should only affect the current evaluation redex. Again, as a way to generate unique and meaningful identifiers, we have chosen to have a generic wrapper freezer which takes the printed form of an entire context, represented as a string, and returns a K label. Moreover, all the variable arguments are wrapped by a label obtained from appling the special freezeVar constructor over the string representation of the variable name. This serves not only to easily identify variables visually, but also to prevent variable contents from mixing in the case of variables of sort List{K}.

7.3.3 $\mathbb{K}$ Semantics

This section describes and exemplifies the process of translating the $\mathbb{K}$ semantic constructs to Maude constructs, obtaining an executable definition as a result.

Applying Context Transformers.

Although K-Maude allows the specification to omit the configuration context (for modularity and compactness purposes), this context needs to be filled in by the tool as a first step towards obtaining a runnable definition. To do that, we use the tree associated to the **configuration** declaration and iteratively match the cells having the maximal level in the tree, and to wrap them (if not already wrapped) by their corresponding parent cell in the configuration tree, and then continue. Let us present how the context transformers algorithm works on the examples discussed in Sec. 7.2.5.

The assignment rule. For this rule, the 'k' and 'env' cells are the deepest in the configuration tree; they both are subcells of the 'thread' cell. Since the 'store' cell corresponds to a higher level in the configuration tree, the 'k' and 'env' cells are wrapped by a 'thread' cell in the first iteration of the algorithm:

rule $\langle\mathsf{thread}\_\rangle\ \langle\mathsf{k}\rangle\ \mathrm{X} = \mathrm{I}\ ;\ \Rightarrow\ .\ \langle\_/\mathsf{k}\rangle\ \langle\mathsf{env}\_\rangle\ \mathrm{X} \mapsto \mathrm{N}\ \langle\_/\mathsf{env}\rangle\ \langle\_/\mathsf{thread}\rangle$
$\langle\mathsf{store}\,\_\rangle\ \mathrm{N} \mapsto (\_\ \Rightarrow\ \mathrm{I})\ \langle\_/\mathsf{store}\rangle$

However, the 'store' cell is still higher in the configuration than the 'thread' cell, so the 'thread' cell itself needs to be wrapped by the 'threads' cell:

rule $\langle\mathsf{threads}\_\rangle\ \langle\mathsf{thread}\_\rangle\ \langle\mathsf{k}\rangle\ \mathrm{X} = \mathrm{I}\ ;\ \Rightarrow\ .\ \langle\_/\mathsf{k}\rangle\ \langle\mathsf{env}\_\rangle\ \mathrm{X} \mapsto \mathrm{N}\ \langle\_/\mathsf{env}\rangle$
$\langle/\_\mathsf{thread}\rangle\ \langle/\_\mathsf{threads}\rangle\ \ \langle\mathsf{store}\,\_\rangle\ \mathrm{N} \mapsto (\_\ \Rightarrow\ \mathrm{I})\ \langle\_/\mathsf{store}\rangle$

The levels of the cells in the new term correspond to their levels in the configuration term; therefore the algorithm concludes successfully.

The rendez-vous rule. **rule** $\langle\mathrm{k}\rangle\ \mathrm{rv\ I}\ \Rightarrow\ .\ \langle\_/\mathrm{k}\rangle\ \langle\mathrm{k}\rangle\ \mathrm{rv\ I}\ \Rightarrow\ .\ \langle\_/\mathrm{k}\rangle$
Although the two computations are here at the same level, their multiplicity does not correspond to the one declared in the configuration term. Therefore the context transformers will wrap each of them in their container 'thread' cell:

rule $\langle\mathsf{thread}\_\rangle\ \langle\mathsf{k}\rangle\ \mathtt{rv}\ \mathrm{I}\ \Rightarrow\ .\ \langle\_/\mathsf{k}\rangle\ \langle\_/\mathsf{thread}\rangle$
$\langle\mathsf{thread}\_\rangle\ \langle\mathsf{k}\rangle\ \mathtt{rv}\ \mathrm{I}\ \Rightarrow\ .\ \langle\_/\mathsf{k}\rangle\ \langle\_/\mathsf{thread}\rangle$

Since the thread cell has variable multiplicity, the process is complete.

Default cell values. Consider a simple 'run' construct, which given the program to be run and a list of input values creates an initial configuration for running the program with the given input. As only the k cell and the in cell would have non-default values in the initial configuration, we can write the rule for initiating the computation as:

```
rule run(P,L) ⇒ ⟨T_⟩ ⟨k⟩ P ⟨/k⟩ ⟨in⟩ L ⟨/in⟩ ⟨_T⟩
```

Since an incomplete cell appears in the right-hand-side, it will be replaced by the corresponding default configuration (sub)term in which the user-specified cells substitute their corresponding cell in the configuration. Moreover, a cell havein multiplicity zero or more is only included if one of its sub-cells was specified by the user. For our example, the generated rule would be:

```
rule run(P,L) ⇒ ⟨T⟩
  ⟨threads⟩ ⟨thread⟩ ⟨k⟩ P ⟨/k⟩ ⟨env⟩.Map⟨/env⟩ ⟨/thread⟩⟨/threads⟩
  ⟨store⟩.Map⟨/store⟩ ⟨nextLoc⟩0⟨/nextLoc⟩ ⟨in⟩ L ⟨/in⟩ ⟨out⟩.List⟨/out⟩
⟨/T⟩
```

Resolving Variables.

Once the context transformers have been applied (taking advantage of the cell comprehension feature), the next step towards obtaining a standard rewriting theory is to resolve cell comprehension and anonymous variables by replacing them with variables of the right sort. To do that, the 𝕂 definition is traversed, and each term is recursively visited. The visitor uses contextual information to infer the constructor and the variables needed to resolve cell comprehension, and then it uses the full signature to resolve the anonymous variables. For example, the assignment rule presented above will look as follows after this step:

```
rule ⟨threads⟩ ?1:Bag ⟨thread⟩ ?2:Bag
        ⟨k⟩ (X = I ; ⇒ .) ↷ ?3:K ⟨/k⟩ ⟨env⟩ ?4:Map X ↦N ⟨/env⟩
     ⟨/thread⟩ ⟨/threads⟩ ⟨store⟩ ?5:Map N ↦(?6:Int ⇒ I) ⟨/store⟩
```

Note that although set comprehension uses ellipses on both sides of the cell, we only need one variable, since the constructor is associative and commutative. The names for the replacement variables start with '?' and have appended numbers for disambiguation.

Resolving In-place Rewriting.

Transforming 𝕂 rules into rewrite rules and equations becomes relatively simple upon the completion of the previous steps. The two terms *l* and *r* of the rewrite rule (l ⇒ r) or equation (l = r), corresponding to the 𝕂 rule C[l1 ⇒ r1 ,..., ln ⇒ rn], can be inferred as being l = C[l1 ,..., ln], and r = C[r1 ,..., rn]. This inference process is defined by building the two terms *l* and *r* together while traversing the 𝕂 rules. If the rule has the structural attribute, then it would be transformed into an equation; otherwise, into a rewrite rule. At the completion of this step, the assignment rule is:

```
rl ⟨threads⟩ ?1:Bag ⟨thread⟩ ?2:Bag
      ⟨k⟩ X = I ; ↷ ?3:K ⟨/k⟩ ⟨env⟩ ?4:Map X ↦N ⟨/env⟩
   ⟨/thread⟩ ⟨/threads⟩ ⟨store⟩ ?5:Map N ↦?6:Int ⟨/store⟩
⇒ ⟨threads⟩ ?1:Bag ⟨thread⟩ ?2:Bag
```

$$\begin{array}{l}
\quad\langle\mathsf{k}\rangle\ .\ \curvearrowright\ ?3{:}K\ \langle/\mathsf{k}\rangle\ \langle\mathsf{env}\rangle\ ?4{:}\mathit{Map}\ \mathrm{X}\mapsto\mathrm{N}\ \langle/\mathsf{env}\rangle\\
\langle/\mathsf{thread}\rangle\ \langle/\mathsf{threads}\rangle\ \langle\,\mathsf{store}\,\rangle\ ?5{:}\mathit{Map}\ \mathrm{N}\mapsto\mathrm{I}\ \langle/\mathsf{store}\rangle
\end{array}$$

Reduction to the 𝕂 Abstract Syntax.

After all previos transformation have applied, the rule is transformed to the AST form. Additionally, this step reduces the compositions of constructors with their identities (due to the use of · in rules) which were introduced at the previous step. The final running version of the assignment rule would thus be:

$$\begin{array}{l}
\mathrm{rl}\ \ \langle\mathsf{threads}\rangle\ ?1{:}\mathit{Bag}\ \langle\mathsf{thread}\rangle\ ?2{:}\mathit{Bag}\\
\qquad\langle\mathsf{k}\rangle\ \text{'\_=\_;}(\mathit{Id}\ \mathrm{X}(.\mathit{List}\{K\}),,\mathit{Int}\ \mathrm{I}\,(.\,\mathit{List}\{K\}))\ \curvearrowright\ ?3{:}K\ \langle/\mathsf{k}\rangle\\
\qquad\langle\mathsf{env}\rangle\ ?4{:}\mathit{Map}\ \mathit{Id}\ \mathrm{X}(.\mathit{List}\{K\})\mapsto\mathit{Int}\ \mathrm{N}(.\mathit{List}\{K\})\ \langle/\mathsf{env}\rangle\\
\quad\ \langle/\mathsf{thread}\rangle\ \langle/\mathsf{threads}\rangle\\
\quad\ \langle\,\mathsf{store}\,\rangle\ ?5{:}\mathit{Map}\ \mathit{Int}\ \mathrm{N}(.\mathit{List}\{K\})\mapsto\mathit{Int}\ ?6{:}\mathit{Int}(.\,\mathit{List}\{K\})\ \langle/\mathsf{store}\rangle\\
\Rightarrow\ \langle\mathsf{threads}\rangle\ ?1{:}\mathit{Bag}\ \langle\mathsf{thread}\rangle\ ?2{:}\mathit{Bag}\\
\qquad\langle\mathsf{k}\rangle\ ?3{:}K\ \langle/\mathsf{k}\rangle\ \langle\mathsf{env}\rangle\ ?4{:}\mathit{Map}\ \mathit{Id}\ \mathrm{X}(.\mathit{List}\{K\})\mapsto\mathit{Int}\ \mathrm{N}(.\mathit{List}\{K\})\ \langle/\mathsf{env}\rangle\\
\quad\ \langle/\mathsf{thread}\rangle\ \langle/\mathsf{threads}\rangle\\
\quad\ \langle\,\mathsf{store}\,\rangle\ ?5{:}\mathit{Map}\ \mathit{Int}\ \mathrm{N}(.\mathit{List}\{K\})\mapsto\mathit{Int}\ \mathrm{I}\,(.\,\mathit{List}\{K\}))\ \langle/\mathsf{store}\rangle
\end{array}$$

7.4 From K-Maude to LaTeX

To facilitate the visualization, undestanding, and debugging of 𝕂 definitions, as well as their inclusion in research papers and presentations, K-Maude allows for annotations (as special attributes) specifying how various constructs should be represented in LaTeX, and provides a tool (written in Maude, as well) which automatically generates a LaTeX document from a provided K-Maude definition. The LaTeX-specific attributes (currently only renameTo) are wrapped in the **latex** attribute. For example, the following environment cell definition requires that '⟨=' be typeset as '≤'.

op _<=_ : *AExp AExp* →*BExp* [**latex**(renameTo _\ensuremath\leq_)]

Typesetting Styles.

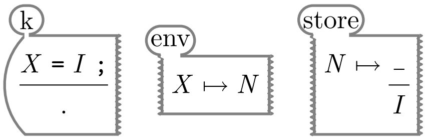
The LaTeX generated from 𝕂 modules is fully configurable, as each specific part of a definition is enclosed in LaTeX macros. The compilation script then takes the output produced by Maude and includes a style file in the preamble, containing definitions for all the macros. Moreover, it allows for the used to provide its own style file which is loaded after the main one, and can customize part of the macros. K-Maude currently provides two such main styles, differing only in the way they typeset cells. One of them typesets rules using only the mathematical 𝕂 notation, producing rules as the one at the end of Sec. 7.1, or the ones in Chapters 4 and 6. The other, used in Chapter 1 and presented below, employs a more graphical notation for cells, and it is thus better for visualizing definitions.

Formatted Output.

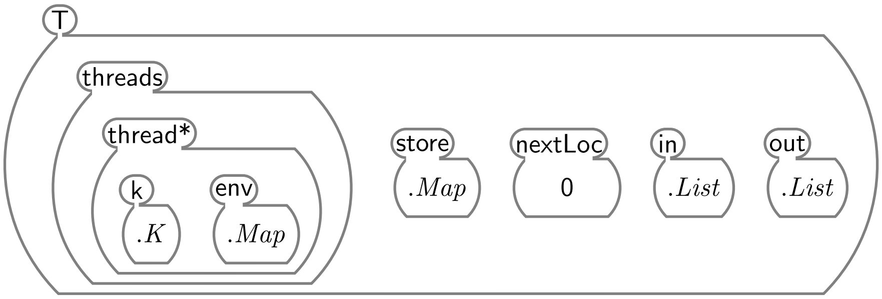
Sort, subsort, and operation declarations obtained after desugaring the original BNF notation are converted back to their equivalent BNF notation. For example, the Imp++ syntax for arithmetic expressions (lines 11–22) is automatically typeset to:

$$\begin{array}{lll} \mathit{AExp} & ::= & \mathit{Int} \mid \mathit{Id} \\ & \mid & \mathit{AExp}\ \texttt{+}\ \mathit{AExp}\ \ [\text{strict}] \\ & \mid & \mathit{AExp}\ \texttt{/}\ \mathit{AExp}\ \ [\text{strict}] \\ & \mid & \texttt{++}\ \mathit{Id} \\ & \mid & \texttt{read} \end{array}$$

𝕂 cells are represented using the `tikz` package as rectangles with rounded sides and with the cell label attached to the top. Completely specified cells have both sides rounded. Incomplete cells, on either side, have the corresponding side "jagged". For example, the Imp++ assignment rule (lines 39–41) is typesetted as:



The configuration configuration term for Imp++ (lines 38–45) is typesetted to:



To ensure that the definition is typsetted in the order it was written in, and that the cells inside a rule are typesetted in order specified by the user, we use modified versions of the `K-TECHNIQUE` module and of the Maude `META-MODULE` module for the 𝕂 to LaTeX transformation. More precisely, both modules are altered by removing all commutativity attributes. This basically means that, for the purpose of this transformation, bags and sets of (meta-) rules, equations, membership axioms, operation declarations, subsorts, and sorts, are all regarded as lists.

7.5 Discussion

The current K-Maude implementation is reasonably stable and is successfully used in teaching Programming Languages Design at UIUC and in Iași, Romania.

Nevertheless, there is still plenty of place for improvement in areas such as module system support, compilation of definitions, interface and error messages, or the possibility of exploring all behaviors for non-deterministic/concurrent executions. A more detailed exposition of the future work proposed for the K-Maude tool can be found in Section 9.2.

Chapter 8

Related Work

In this chapter we review the research literature which is closest to the material presented in the dissertation. In the sections below we will focus on three main areas of research which we believe to be mostly related to our topic.

First we will review the existing efforts within the rewriting logic semantics project, with a special emphasis on the work based on the $\mathbb{K}$ framework and technique in Section 8.1. Then, we will review the major existing techniques and frameworks for programming language design in Section 8.2. Finally, we conclude with some related rewriting-based formalisms in Section 8.3.

8.1 Rewriting Logic Semantics

RLS is a collective international project. Through the efforts of various researchers, there is by now a substantial body of work demonstrating the usefulness of this approach [21, 174, 170, 168, 106, 173, 31, 147, 175, 58, 57, 87, 22, 108, 109, 27, 26, 55, 35, 153, 3, 166, 40, 152, 91, 80, 61, 6, 56, 7]. A first snapshot of the RLS project was given in [109], and a second in [111]. In particular, a substantial body of experience in giving programming language definitions, and using those definitions both for execution and for analysis purposes has already been gathered. For example, Java 1.4 (see also [29] for a complete formal semantics) and the JVM (see [58, 56]) have been specified in Maude this way, with the Maude rewriting logic semantics being used as the basis of Java and JVM program analysis tools that for some examples outperform well-known Java analysis tools [58, 57]. A semantics of a Caml-like language with threads was discussed in detail in [109], and a modular rewriting logic semantics of a subset of CML has been given in [27] using the Maude MSOS tool [28]. Other language case studies, all specified in Maude, include BC [22], CCS [174, 22], CIAO [166], Creol [87], ELOTOS [173], MSR [25, 169], PLAN [168, 166], the ABEL hardware description language [91], SILF [80], FUN [143], Orc [6, 7], and the π-calculus [170].

$\mathbb{K}$ Related Work

Although introduced relatively recently, $\mathbb{K}$ has already generated a consistent body of research projects and publications. Even from its incipient stages, $\mathbb{K}$

aimed at scalability: to define and analyze real life programming languages. For example, a comprehensive definition of Java 1.4 in Maude was specified following the $\mathbb{K}$ technique and used to derive a state-of-art competitive model-checker for Java [57]. More recently, the same definition was adapted and used to verify security-related properties for Java programs [4, 5]. Besides analyzing Java programs, the $\mathbb{K}$ technique was also used to define a complete static semantics for *C* which can support checking pluggable domain specific policies, such as units of measurement [81], which can be used to analyze real C programs. Additionally, [149] uses $\mathbb{K}$ to define a symbolic semantics for pointer allocation in a C-like language, aiming at runtime-verifying memory safety. A $\mathbb{K}$ semantic definition of C is given in [51], one for Scheme R5RS in [100], and one for the Beta language in [79].

There are also several tools and techniques based on $\mathbb{K}$. For example, Chapter 7 describes the K-Maude prototype, a Maude implementation of the $\mathbb{K}$ framework which fully supports the $\mathbb{K}$ definitional style as portrayed here. A module system for $\mathbb{K}$ aiming at maximizing modularity and code-reuse is discussed in [77]. The potential for efficient executability of $\mathbb{K}$ definitions has been first empirically noted in [80]. An experimental object-oriented programming language is defined using $\mathbb{K}$ in [76, 74, 75], together with several formal analyses and optimizations based on it. The $\mathbb{K}$ framework is compared in [161] with P-systems [131] and shown that it can be systematically used to define, execute and analyze a large body of P-systems. Matching logic is a new axiomatic semantics extending the benefits of both Hoare logic and separation logics, which fundamentally relies on the $\mathbb{K}$ framework [145, 148]. As shown in [52], $\mathbb{K}$ can also be effectively used to define type inferencers and prove their soundness w.r.t. the language semantics.

8.2 Approaches to Programming Language Design

As $\mathbb{K}$ builds upon the existing operational semantics techniques and frameworks, we here summarize the observations from Chapter 3, insisting on the relationship these techniques have with $\mathbb{K}$.

8.2.1 Structural Operational Semantics (SOS) Frameworks

We here discuss the most common SOS approaches, namely big-step and small-step SOS, their modular variant MSOS, and reduction semantics with evaluation contexts.

Big-Step SOS Introduced as natural semantics [89], also named relational/evaluation semantics [116], big-step semantics is "the most denotational" of the operational semantics. One can view big-step definitions as definitions of relations

interpreting each language construct in an appropriate domain. Deterministic (or functional) big-step operational semantics can be easily and efficiently interpreted/implemented. It is particularly useful for defining type systems.
Limitations: Due to its monolithic, single-step evaluation, it is hard to debug or trace big-step semantic definitions. If the program is wrong, no information is given about where the failure occurred. Divergence is not observable in the evaluation relation. It may be hard or impossible to model concurrent features. It is not modular, e.g., to add side effects to expressions, one must redefine the rules to allow expressions to evaluate to pairs (value-store). It is inconvenient (and non-modular) to define complex control statements.
Relationship to $\mathbb{K}$: The transitive closure of the $\mathbb{K}$ rewrite relation can be seen as a big-step relation.

Small-step SOS Introduced by Plotkin [136], also called transition (or reduction) semantics, small-step semantics captures the notion of one computational step. Therefore, it stops at errors, pointing them out. It is easy to trace and debug. It gives interleaving semantics for concurrency.
Limitations: Like big-step, it is non-modular. It does not give a "true concurrency" semantics: one has to choose a certain interleaving (no two rules can be applied on the same term at the same time), mainly because reduction is forced to occur only at the top. It is still hard to deal with control — one has to add corner cases (additional rules) to each statement to propagate control changing information. Each small step traverses the entire program to find the next redex; since the size of the program may grow unbounded (e.g., through loop/fixed-point unrolling), each small step may take unbounded resources in the worst case, making it difficult to interpret efficiently.
Relationship to $\mathbb{K}$: $\mathbb{K}$'s rewriting is small-step in spirit; however, $\mathbb{K}$ rewrite rules can apply anywhere and concurrently, while SOS transitions only apply at the top of the term, interleaved.

Modular SOS (MSOS) Mosses [122, 123] introduced MSOS to deal with the non-modularity of existing SOS styles. The MSOS solution involves moving the non-syntactic state components into the labels on transitions, plus a discipline of only selecting needed attributes from the states.
Limitations: Being inherently an SOS, MSOS can still only define interleaving semantics to concurrent languages. While the use of labels gives MSOS the ability to modularly deal with some forms of control, such as abrupt termination, at our knowledge it still cannot support the definition of arbitrarily complex control-intensive features such call/cc. Also, MSOS is driven by the structure of the syntax and appears to have no escape mechanism to reflectively traverse, mark, modify and/or store arbitrary syntax in labels, which makes it hard or impossible to modularly define language constructs for code generation (e.g., `quote`/`unquote`/`eval` —see Section 4.3).

Relationship to $\mathbb{K}$*:* Both MSOS and $\mathbb{K}$ make use of labeled information to achieve modularity. MSOS uses labels as record fields on the transition relation, while $\mathbb{K}$ uses labels as cell names in the configuration. In both MSOS and $\mathbb{K}$ one can use the labels in semantic rules only to refer to configuration items of interest, which is crucial for modularity. MSOS' labels have an additional role, to yield labeled transition systems, while $\mathbb{K}$'s cell names are not intended to be used for that. One can label $\mathbb{K}$ rules as we did with the rules ρ_r and ρ_w in Section 6.2, and, in principle, one can incorporate in $\mathbb{K}$ the same complex rule labeling of rewrite logic [103]. However, we have not done that. So far, we used additional cells in the configuration to represent/store the emitted signals.

Reduction semantics with evaluation contexts Introduced by Felleisen and colleagues (see, e.g., [180]), the evaluation contexts style improves over small-step SOS in two ways: (1) it gives a more compact semantics to context-sensitive reduction, by using parsing (rather than small-step SOS rules) to find the next redex; and (2) it provides the possibility to also modify the context in which a reduction occurs, making it much easier to deal with control-intensive features, in particular to define constructs like call/cc. Additionally, one can also incorporate the configuration as part of the evaluation context, and thus have full access to semantic information "by need".
Limitations: It still only allows "interleaving semantics" for concurrency. It is too "rigid to syntax", in that it is hard or impossible to define semantics in which values are more than plain syntax; for example, one cannot give a syntactic semantics to a functional language based on closures for functions (instead of substitution), because one needs special, non-syntactic means to handle and recover environments (as we do in $\mathbb{K}$, see Section 4.3). Although context-sensitive rewriting might seem to be easily implementable by rewriting, in fact one has to perform an amount of "parsing" polynomial (linear best case, quadratic worst-case) in the size of the program for each computational step.However, one might obtain efficient implementations for restricted forms of context-reduction definitions by applying refocusing techniques [41].
Relationship to $\mathbb{K}$*:* Both reduction semantics and $\mathbb{K}$ make use of evaluation contexts and can store or modify them. Reduction semantics "splits/plugs" syntax into contexts, while $\mathbb{K}$ "heats/cools" syntax into computations. The former is achieved by an implicit "advanced" parsing of syntax into a context and a redex,while the latter is achieved using rewrite rules.

The Chemical Abstract Machine (Cham) Berry and Boudol [15] introduced Cham to give language semantics within the Gamma model [10], a multiset-rewriting model of computation. The Cham views a distributed state as a "solution" where "molecules" float, and understands concurrent transitions as "reactions" that can occur simultaneously in many points of the solution.

Limitations: While chemistry as a model of computation sounds attractive, technically speaking the Cham is in fact a restricted case of rewriting logic [103]. Moreover, some of the chemical "intuitions", such as the airlock operation which imposes a particular chemical "discipline" to access molecules inside a solution, inhibit the potential for the now well-understood and efficient matching and rewriting *modulo associativity and commutativity.* In fact, to our knowledge, there is no competitive implementation of the Cham. Although this solution-molecule paradigm seems to work for languages with simple state structure, it is not clear how one could represent the state for complex languages with threads, locks, environments, etc. Finally, Chams provide no mechanism to freeze the current molecular structure as a "value", and then to store or retrieve it, as we would need in order to to define language features like call/cc. It would therefore seem hard to define complex control-intensive language features in Cham.
Relationship to $\mathbb{K}$: Like the Cham, $\mathbb{K}$ also organizes the configuration of the program or system as a potentially nested structure of molecules (called cells and potentially labeled in $\mathbb{K}$). Like in Cham, the configuration is also rewritten until it "stabilizes". Unlike in Cham, the $\mathbb{K}$ rules can match and write inside and across multiple cells in one parallel step. Also, unlike in Cham (and in rewriting logic), $\mathbb{K}$ rewrite rules can apply concurrently even in cases when they overlap.

8.2.2 Other Approaches

Algebraic denotational semantics. This approach, (see [179, 66, 23, 121] for early papers and [64, 172] for two more recent books), is a special case of RLS, namely, the case in which the rewrite theory $\mathcal{R}_\mathcal{L}$ defining language $\mathcal{L}$ is an equational theory. While algebraic semantics shares a number of advantages with RLS, its main limitation is that it is well-suited for giving semantics to *deterministic* languages, but not well-suited for concurrent language definitions. At the model-theoretic level, initial algebra semantics, pioneered by Joseph Goguen, is the preferred approach (see, for example, [66, 64]), but other approaches, based on loose semantics or on final algebras, are also possible.

Higher-order approaches. The most classic higher-order approach, although not exactly operational, is *denotational semantics* [155, 156, 154, 120]. Denotational semantics has some similarities with its first-order algebraic cousin mentioned above, since both are based on semantic equations. Two differences are: (i) the use of first-order equations in the algebraic case versus the higher-order ones in traditional denotational semantics; and (ii) the kinds of models used in each case. A related class of higher-order approaches uses higher-order functional languages or higher-order theorem provers to give operational semantics to programming languages. Without trying to be comprehensive, we can mention, for example, the use of Scheme in [60], the use of ML in [134], and the use of Common LISP within the ACL2 prover in [92]. There is also a body of work on using monads [117, 178, 96] to implement language interpreters in higher-order functional languages; the monadic approach has better modularity characteristics than standard SOS. A third class of higher-order approaches are based on the use of higher-order abstract syntax (HOAS) [133, 71] and higher-order logical frameworks, such as LF [71] or λ-Prolog [124], to encode programming languages as formal logical systems. For a good example of recent work in this direction see [113] and references there.

Logic-programming-based approaches. Going back to the Centaur project [18, 37], logic programming has been used as a framework for SOS language definitions. Note that λ-Prolog [124] belongs both in this category and in the higher-order one. For a recent textbook giving logic-programming-based language definitions, see [164].

Abstract state machines. Abstract State Machine (ASM) [67] can encode any computation and have a rigorous semantics, so any programming language can be defined as an ASM and thus implicitly be given a semantics. Both big- and small-step ASM semantics have been investigated. The semantics of various programming languages, including, for example, Java [165], has been given using ASMs.

8.3 Other Rewriting Frameworks

We here discuss some rewriting approaches which have been used or have the potential to be used for programming language semantics.

Graph Rewriting Graph rewriting [44] extends the intuitions of term rewriting to arbitrary graphs, by developing a match-and-apply mechanism which can work directly on graphs.
Limitations: Graph rewriting is rather complex and, although it has been used to model (mostly concurrent) languages and systems [9], with a few notable recent exceptions [90, 139], most such definitions remained at the purely theoretical level. Part of the reason could be the notorious difficulty of graph rewriting in dealing with structure copying and equality testing, operations which are crucial for programming language semantics and relatively easy to provide in term rewriting systems. Finally, in spite of decades of research, to our knowledge there are still no graph rewriting engines comparable in performance to term rewrite engines. Nevertheless, graph rewriting has received increased attention recently as a means to formalize and analyze modeling languages such as UML [63], to which it can be easier related through the graph-like structure of UML specifications.
Relationship to $\mathbb{K}$: The $\mathbb{K}$ rules, like those in graph-rewriting, have both read-only components, which could be shared among rule instances to maximize concurrency, and read-write components.

P-Systems Păun's membrane systems (or P-systems) [131] are computing devices abstracted from the structure and the functioning of the living cell. In classical *transition P-systems*, the main ingredients of such a system are the *membrane structure*, in the compartments of which *multisets* of symbol-objects evolve according to given *evolution rules*. The rules are localized, associated with the membranes and they are used in a *nondeterministic maximally parallel* manner. *Limitations:* When looked at from a programming language semantics perspective, the P-systems have a series of limitations that have been addressed in $\mathbb{K}$, such as: (1) lack of structure for non-membrane terms, which are plain constants; and (2) strict membrane locality (even stricter than in the Cham), which allows a very limited kind of rules (symport/antiport [130]) to match and rewrite within multiple membranes at the same time. Regarding (1), programming languages often handle complex data-structures or make use of complex semantic structures, such as environments mapping variables to values, or closures holding code as well as environments, etc., which would be hard or impossible to properly encode using just constants. Regarding (2), strict membrane locality would require one to write many low-level rules and introduce and implement by local rules artificial "cross-membrane communication protocols". For example, the semantics of variable lookup involves acquiring the location of the variable from the environment cell (which holds a map), then the value at that location in the store cell (which holds another map), and finally the rewrite of the variable in the code cell into that value. All these require the introduction of encoding constants and rules in several membranes and many computation steps.
Relationship to $\mathbb{K}$*:* $\mathbb{K}$-systems share with P-systems the ideas of cell structures and the aim to maximize concurrency; for example the read-only parts of a $\mathbb{K}$ rule play a similar role to promotors/inhibitors [19] from P-systems. However, $\mathbb{K}$ rules can span across multiple cells at a time, and the objects contained in a cell can have a rich algebraic structure (as opposed to being constants, as in P-systems). An in-depth comparison between the two formalisms, showing how one can use $\mathbb{K}$ to model P-systems can be found in [161].

There are several other biology-inspired computational frameworks, such as the Calculus of Looping Sequences [11] and MGS [62], which share with $\mathbb{K}$ the aims for concurrency and use one form or another of rewriting as their evolution mechanism.

Chapter 9

Conclusions

The purpose of this dissertation is to show that definitions of programming languages are not just on-paper theoretical endeavors only accessible to the well-trained mathematician, but that instead they can be quite practical and accessible to most programming language enthusiasts.

By practical we mean both that one can define real (and thus complex) languages, with emphasis on control and concurrency features, which are known to be problematic to define. Moreover, the definitions obtained do not only serve as a documentation of the language and as a way to understand it better, but also they can be used directly, or slightly adapted, to execute, debug, and analyze programs written in those languages.

All these are made possible by choosing rewriting as a paradigm of computation and rewriting logic as a paradigm of modeling transition systems, following the ideas put forward in the rewriting logic semantics project [110, 111], which led to the development of $\mathbb{K}$, a specialized programming language design framework based on rewriting.

To convince ourselves that the RLS project is well founded, we started by showing how RLS can be used as a logical framework for operational semantics definitions of programming languages. In particular, by showing in detail how it can faithfully capture big-step and small-step SOS, MSOS, context reduction, continuation-based semantics, and the Cham, we hope to have illustrated what might be called its *ecumenical* character; that is, its flexible support for a wide range of definitional styles, without forcing or pre-imposing any given style.

It is precisely this flexibility which makes RLS useful as a way of exploring *new* definitional styles; however, flexibility comes at the price of being over qualified for the task of defining languages, in the sense of not committing or advancing any style in particular. The development of the $\mathbb{K}$ framework was motivated by the quest for the best definitional framework within RLS. The $\mathbb{K}$ semantic framework, consisting of a general-purpose concurrent rewriting approach together with a definitional technique specialized for concurrent programming languages and systems, brings together the advantages of the existing language definitional frameworks while avoiding their limitations. In spite of its youth, the $\mathbb{K}$ framework already proved practical as it was used with relatively little effort (in comparison with similar attempts within other frameworks) to define complex languages like

Java, Scheme, Verilog, or C, and to use those definitions for analyzing programs written in those languages. We believe that all the above make $\mathbb{K}$ a strong candidate for the next generation of language definitional frameworks.

To show that one can actually make these definitions useful, we have chosen from the areas to which $\mathbb{K}$ was applied (static semantics, type soundness, model checking, program verification) the area of runtime verification, demonstrating how, through relatively minor alterations or extensions, one can turn $\mathbb{K}$ definitions in runtime verification tools for memory safety, datarace detection, or monitoring.

On a more theoretical note, although initially introduced as a mere simplifying notation, $\mathbb{K}$ rules actually provide valuable information about parts of the term which only need to be read for the a rule to apply. Deepening this intuition, we envisioned a concurrent semantics for $\mathbb{K}$ which would allow parallelism with sharing of data in addition to the sideways and nested parallelism already available for term rewriting. The concurrent semantics of $\mathbb{K}$ presented in Chapter 6 formalizes these intuitions by exploiting the connections between $\mathbb{K}$ rules and graph rewriting rules. Although the potential for concurrent rewriting is increased by identifying parts of a rule which can be shared with other rules, this may also lead to inconsistencies if not used properly. Nevertheless, we showed that, under reasonable conditions, the $\mathbb{K}$ concurrent semantics is actually sound, complete, and serializable w.r.t. term rewriting.

The success of a programming language definitional framework, and especially of an operational one which concretely describes the execution of the programs, is largely depending on the available tool support to allow actual execution, exploration and analysis of executions of programs in the defined language. Although the $\mathbb{K}$ technique has been used on top of the Maude rewrite engine since its early development stages, its direct use in Maude proved to be sometimes difficult and verbose, leading to modularity and copy-pasting errors. To this aim, K-Maude was introduced as an implementation of the $\mathbb{K}$ language definitional framework in Maude, which allows for an almost zero representational distance between on-paper $\mathbb{K}$ definitions, and their textual representations. The K-Maude interface comes as an extension of Maude, allowing users to define a language using $\mathbb{K}$ modules with specific $\mathbb{K}$ syntax in addition to the existing Maude modules. The K-specific modules extend the Maude module syntax with constructs aimed at simplifying the language definition task by abstracting away irrelevant details. This multi-layered abstractions allow for concise language definitions with a high potential for reusing language features. K-Maude defines several meta-transformations which gradually translate $\mathbb{K}$ modules into either executable Maude modules, to obtain interpreters and analysis tools, or into LaTeX, to obtain formal language semantics documentation.

We believe all the material presented in this dissertation and summarized above brings enough evidence that rewriting, and in particular the $\mathbb{K}$ framework, is a natural environment to formally define the semantics of real-life (concurrent) programming languages and to test and analyze programs written in those languages. Nevertheless, although we believe $\mathbb{K}$ to be reasonably stable by now, there is still place for improvements at both theoretical and tool support level.

9.1 Future Work—Theory

Regarding the theoretical contributions to the $\mathbb{K}$ framework, we point below two directions of further research which we believe to be most relevant.

Deepening $\mathbb{K}$'s concurrent semantics. The concurrency semantics for $\mathbb{K}$ presented in this dissertation sets the ground for additional interesting research questions. One of them is related to serializability: would it make sense to have unserializable executions, and if so, how should they look like and how could they be captured, as the graph rewriting approach seems unable to cope with them directly. In particular, we are interested in exploring the relation between side-conditions, concurrency, and unserializability.

Relating $\mathbb{K}$ with other frameworks. Although $\mathbb{K}$ was developed on the shoulders of the existing operational frameworks, further study is required to determine precise relations with these formalisms. Moreover, we are interested in exploring the possibility of relating $\mathbb{K}$ with powerful theorem proving frameworks like Isabelle [125], PVS [129], or Coq [13], to allow proving properties about the language itself, not only about the executions of programs written in that language.

9.2 Future Work—Tool Support

Although quite expressive and powerful already, the K-Maude tool is still work in progress. We list below some areas of future work relevant to the K-Maude tool in particular and to implementations of the $\mathbb{K}$ framework in general.

Modularization. The current implementation, although quite modular, has still limited support for modularization, being only partially able to make use of Maude's module system. A desirable fix in this direction would be to develop a programming language specific module system for $\mathbb{K}$ in the K-Maude tool, following the proposal in [77, 73].

Interface&Error Support. K-Maude kept improving in key aspects regarding interface (including most recently the addition of the BNF layer and a better lexer in the pre-processor stage), but there is still work to be done. Specifically, error reporting is an aspect which still requires attention. Although the current architecture of the tool splits the transformation from $\mathbb{K}$ specifications to executable Maude definitions in multiple small stages, and thus it allows to easily identify in which stage of the compilation an error occurred, and although specific checks are performed to catch early common specification mistakes, it is sometimes hard to concretely isolate a fault. The main reason for this is that our compilation stages are written in a functional-like style and, since Maude does not have a global state, one has to propagate errors until the top, much like in a SOS definition of a programming language. And, similar to them, this makes the addition of error reporting tedious and error-prone. One possible solution would be to rewrite K-Maude in a 𝕂 like style. Another would be to analyze the ”core dumps”, i.e., stuck compilation terms for the source of the errors.

True Concurrency. Currently, K-Maude simply translates 𝕂 rules into Maude rules and equations, and therefore the additional potential for concurrency information brought by the 𝕂 rules (proposed in Section 6.2 and developed in Chapter 6) is lost; moreover, Maude itself applies only one rewrite step at a time, so one cannot directly visualize even the amount of concurrency allowed in one step by RWL. It would be beneficial to develop an exploration mode for the tool which would allow one to precisely see the amount of concurrency achievable in one 𝕂 concurrent step.

Compilation. The K-Maude implementation is very effective in providing for free an execution and debugging environment for testing the definitions, as well as an execution analysis platform containing a model checker and an inductive theorem prover. However, it would be beneficial to target other languages, either programming languages with limited support for rewriting but with increased performance w.r.t. execution time (such as Haskell and OCaml), as well as dedicated theorem provers (as Isabelle an Coq). This goal of compiling 𝕂 definitions has accompanied the development of the tool since its inception (see, e.g., [80]), and the initial steps towards automatically compiling 𝕂 definitions in OCaml [84] seem promising.

Bibliography

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 31–46, New York, NY, USA, 1990. ACM Press. doi: 10.1145/96709.96712.

[2] Gul A. Agha, José Meseguer, and Koushik Sen. PMaude: Rewrite-based specification language for probabilistic object systems. In *3rd Workshop on Quantitative Aspects of Programming Languages (QAPL 05)*, volume 153(2) of *Electronic Notes in Theoretical Computer Science*, pages 213–239, 2006. doi: 10.1016/j.entcs.2005.10.040.

[3] Wolfgang Ahrendt, Andreas Roth, and Ralf Sasse. Automatic validation of transformation rules for java verification against a rewriting semantics. In Geoff Sutcliffe and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005, Montego Bay, Jamaica, December 2-6, 2005, Proceedings*, volume 3835 of *Lecture Notes in Computer Science*, pages 412–426. Springer, 2005. doi: 10.1007/11591191_29.

[4] Mauricio Alba-Castro, María Alpuente, and Santiago Escobar. Automatic certification of Java source code in rewriting logic. In Stefan Leue and Pedro Merino, editors, *FMICS*, volume 4916 of *Lecture Notes in Computer Science*, pages 200–217. Springer, 2007. ISBN 978-3-540-79706-7. doi: 10.1007/978-3-540-79707-4_15.

[5] Mauricio Alba-Castro, María Alpuente, Santiago Escobar, Pedro Ojeda, and Daniel Romero. A tool for automated certification of Java source code in Maude. In *Proceedings of the Eighth Spanish Conference on Programming and Computer Languages (PROLE 2008)*, volume 248 of *Electronic Notes in Theoretical Computer Science*, pages 19–29, 2009. doi: 10.1016/j.entcs.2009.07.056.

[6] Musab AlTurki. A rewriting logic approach to the semantics of Orc. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, December 2005.

[7] Musab AlTurki and José Meseguer. Real-time rewriting semantics of Orc. In Michael Leuschel and Andreas Podelski, editors, *Proceedings of the 9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 14-16, 2007, Wroclaw, Poland*, pages 131–142. ACM Press, 2007. ISBN 978-1-59593-769-8. doi: 10.1145/1273920.1273938.

[8] Irina Mariuca Asavoae and Mihail Asavoae. Collecting semantics under predicate abstraction in the k framework. In Ölveczky [126], pages 123–139. ISBN 978-3-642-16309-8. doi: 10.1007/978-3-642-16310-4_9.

[9] Paolo Baldan, Fabio Gadducci, and Ugo Montanari. Modelling calculi with name mobility using graphs with equivalences. In *TERMGRAPH'06*, volume 176(1) of *Electronic Notes in Theoretical Computer Science*, pages 85–97, 2007. doi: 10.1016/j.entcs.2006.10.028.

[10] Jean-Pierre Banâtre and Daniel Le Métayer. The GAMMA model and its discipline of programming. *Science of Computer Programming*, 15(1): 55–77, 1990. doi: 10.1016/0167-6423(90)90044-E.

[11] Roberto Barbuti, Andrea Maggiolo-Schettini, Paolo Milazzo, and Angelo Troina. A calculus of looping sequences for modelling microbiological systems. *Fundamenta Informaticae*, 72(1–3):21–35, 2006.

[12] Hendrik Pieter Barendregt, Marko C. J. D. van Eekelen, John R. W. Glauert, Richard Kennaway, Marinus J. Plasmeijer, and M. Ronan Sleep. Term graph rewriting. In *PARLE'87*, volume 259 of *Lecture Notes in Computer Science*, pages 141–158. Springer, 1987. doi: 10.1007/3-540-17945-3_8.

[13] B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq proof assistant reference manual – version 7.4. 2003. URL http://coq.inria.fr/doc/main.html.

[14] Zine-El-Abidine Benaissa, Daniel Briaud, Pierre Lescanne, and Jocelyne Rouyer-Degli. λ-ν, a calculus of explicit substitutions which preserves strong normalisation. *The Journal of Functional Programming*, 6(5):699–722, 1996. doi: 10.1017/S0956796800001945.

[15] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, 1992. doi: 10.1145/96709.96717.

[16] Peter Borovanský, Horatiu Cirstea, Hubert Dubois, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. ELAN *V 3.4 User Manual*. LORIA, Nancy (France), fourth edition, January 2000.

[17] Peter Borovanský, Claude Kirchner, Hélène Kirchner, and Pierre-Etienne Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285(2):155–185, 2002. doi: 10.1016/S0304-3975(01)00358-9.

[18] Patrick Borras, Dominique Clément, Th. Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and V. Pascual. CENTAUR: The system. In *Software Development Environments (SDE)*, pages 14–24, 1988. doi: 10.1145/64137.65005.

[19] Paolo Bottoni, Carlos Martín-Vide, Gheorghe Păun, and Grzegorz Rozenberg. Membrane systems with promoters/inhibitors. *Acta Informatica*, 38 (10):695–720, 2002. doi: 10.1007/s00236-002-0090-7.

[20] Gérard Boudol. Some chemical abstract machines. In *REX School/Symp.*, volume 803 of *Lecture Notes in Computer Science*, pages 92–123. Springer, 1993. doi: 10.1007/3-540-58043-3_18.

[21] Christiano Braga. *Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics*. PhD thesis, Departamento de Informática, Pontificia Universidade Católica de Rio de Janeiro, Brasil, 2001.

[22] Christiano Braga and José Meseguer. Modular rewriting semantics in practice. In *Proceedings of the Fifth International Workshop on Rewriting Logic and Its Applications (WRLA 2004)*, volume 117 of *Electronic Notes in Theoretical Computer Science*, pages 393–416. Elsevier, 2005. doi: 10.1016/j.entcs.2004.06.019.

[23] Manfred Broy, Martin Wirsing, and Peter Pepper. On the algebraic definition of programming languages. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(1):54–99, 1987. doi: 10.1145/9758.10501.

[24] Roberto Bruni and José Meseguer. Semantic foundations for generalized rewrite theories. *Theoretical Computer Science*, 360(1-3):386–414, 2006. doi: 10.1016/j.tcs.2006.04.012.

[25] Iliano Cervesato and Mark-Oliver Stehr. Representing the MSR cryptoprotocol specification language in an extension of rewriting logic with dependent types. In P. Degano, editor, *Proceedings of the Fifth International Workshop on Rewriting Logic and Its Applications (WRLA 2004)*, volume 117 of *Electronic Notes in Theoretical Computer Science*, pages 183–207. Elsevier, 2005. doi: 10.1016/j.entcs.2004.06.023.

[26] Fabricio Chalub. An implementation of Modular SOS in Maude. Master's thesis, Universidade Federal Fluminense, May 2005. URL `http://www.ic.uff.br/~frosario/dissertation.pdf`.

[27] Fabricio Chalub and Christiano Braga. A modular rewriting semantics for CML. *The Journal of Universal Computer Science*, 10(7):789–807, 2004. doi: 10.3217/jucs-010-07-0789.

[28] Fabricio Chalub and Christiano Braga. Maude MSOS tool. In Grit Denker and Carolyn Talcott, editors, *Proceedings of the Sixth International Workshop on Rewriting Logic and its Applications (WRLA 2006)*, volume 176(4) of *Electronic Notes in Theoretical Computer Science*, pages 133–146. Elsevier, 2007. doi: 10.1016/j.entcs.2007.06.012.

[29] Feng Chen and Grigore Roșu. Rewriting Logic Semantics of Java 1.4, 2004. URL `http://fsl.cs.uiuc.edu/java`.

[30] Feng Chen and Grigore Roșu. Parametric and sliced causality. In *Computer Aided Verification (CAV'07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 240–253, 2007. doi: 10.1007/978-3-540-73368-3_27.

[31] Feng Chen, Grigore Roșu, and Ram Prasad Venkatesan. Rule-based analysis of dimensional safety. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 9-11, 2003, Proceedings*, volume 2706 of *Lecture Notes in Computer Science*, pages 197–207. Springer, 2003. doi: 10.1007/3-540-44881-0_15.

[32] Feng Chen, Traian Florin Șerbănuță, and Grigore Roșu. jPredictor: a predictive runtime analysis tool for Java. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 221–230, New York, NY, USA, 2008. ACM. doi: 10.1145/1368088.1368119.

[33] M. Clavel, M. Palomino, and A. Riesco. Introducing the ITP tool: a tutorial. *Journal of Universal Computer Science*, 12(11):1618–1650, 2006. doi: 10.3217/jucs-012-11-1618.

[34] M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude, A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007. doi: 10.1007/978-3-540-71999-1.

[35] Manuel Clavel and Juan Santa-Cruz. ASIP + ITP: A verification tool based on algebraic semantics. In *PROLE 2005: V Jornadas sobre Programación y Lenguajes*, pages 149–158. Thomson, 2005.

[36] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285 (2):187–243, 2002. doi: 10.1016/S0304-3975(01)00359-0.

[37] Dominique Clément, Joëlle Despeyroux, Laurent Hascoet, and Gilles Kahn. Natural semantics on the computer. In Kazuhiru Fuchi and Maurice Nivat, editors, *Proceedings of the France-Japan AI and CS Symposium*, pages 49–89. ICOT, Japan, 1986. Also, Information Processing Society of Japan, Technical Memorandum PL-86-6 and Rapport de recherche #0416, INRIA.

[38] Andrea Corradini and Francesca Rossi. Hyperedge replacement jungle rewriting for term-rewriting systems and logic programming. *Theoretical Computer Science*, 109(1&2):7–48, 1993. doi: 10.1016/0304-3975(93) 90063-Y.

[39] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic approaches to graph transformation: Basic concepts and double pushout approach. In *Handbook of graph grammars and computing by graph transformations*, volume 1, pages 163–246. World Scientific, 1997.

[40] Marcelo d'Amorim and Grigore Roșu. An equational specification for the Scheme language. *The Journal of Universal Computer Science*, 11(7): 1327–1348, 2005. doi: 10.3217/jucs-011-07-1327. Selected papers from the 9th Brazilian Symposium on Programming Languages (SBLP'05). Also Technical Report No. UIUCDCS-R-2005-2567, April 2005.

[41] Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. RS RS-04-26, BRICS, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, November 2004. This report supersedes BRICS report RS-02-04. A preliminary version appears in the informal proceedings of the *Second International Workshop on Rule-Based Programming*, RULE 2001, Electronic Notes in Theoretical Computer Science, Vol. 59.4.

[42] Răzvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report. The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, volume 6 of *AMAST Series in Computing*. World Scientific, 1998.

[43] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965. doi: 10.1145/365559.365617.

[44] Hartmut Ehrig. Introduction to the algebraic theory of graph grammars (a survey). In Volker Claus, Hartmut Ehrig, and Grzegorz Rozenberg, editors, *Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 1–69. Springer, 1978. ISBN 3-540-09525-X.

[45] Hartmut Ehrig and Hans-Jörg Kreowski. Parallelism of manipulations in multidimensional information structures. In *MFCS'76*, volume 45 of *Lecture Notes in Computer Science*, pages 284–293. Springer, 1976. doi: 10.1007/3-540-07854-1_188.

[46] Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. Graph-grammars: An algebraic approach. In *FOCS*, pages 167–180. IEEE, 1973. doi: 10.1109/SWAT.1973.11.

[47] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of algebraic graph transformation.* Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006.

[48] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The Maude LTL model checker and its implementation. In *Proc. 10th International SPIN Workshop (10th SPIN)*, Lecture Notes in Computer Science, pages 230–234. Springer, 2003. doi: 10.1007/3-540-44829-2_16.

[49] Steven Eker, Narcis Martí-Oliet, Josè Meseguer, and Albert Verdejo. Deduction, strategies, and rewriting. In T. Boy de la Tour M. Archer and C. Muñoz, editors, *Proceedings of the 6th International Workshop on Strategies in Automated Deduction (STRATEGIES 2006)*, volume 174(11) of *Electronic Notes in Theoretical Computer Science*, pages 3–25. Elsevier, 2007. doi: 10.1016/j.entcs.2006.03.017.

[50] Chucky Ellison. A rewriting logic approach to defining type systems. Master's thesis, University of Illinos at Urbana Champaign, 2008.

[51] Chucky Ellison and Grigore Roşu. A formal semantics of C with applications. Technical Report http://hdl.handle.net/2142/17414, University of Illinois, November 2010. URL http://fsl.cs.uiuc.edu/index.php/A_Formal_Semantics_of_C_with_Applications.

[52] Chucky Ellison, Traian Florin Șerbănuță, and Grigore Roșu. A rewriting logic approach to type inference. In *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. doi: 10.1007/978-3-642-03429-9_10.

[53] David Espinosa. *Semantic Lego.* PhD thesis, Columbia University, 1995.

[54] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent bug patterns and how to test them. In *IPDPS*, page 286. IEEE Computer Society, 2003. ISBN 0-7695-1926-1. doi: 10.1109/IPDPS.2003.1213511.

[55] A. Farzan and J. Meseguer. Partial order reduction for rewriting semantics of programming languages. In Grit Denker and Carolyn Talcott, editors, *Proceedings of the Sixth International Workshop on Rewriting Logic and its Applications (WRLA 2006)*, volume 176(4) of *Electronic Notes in Theoretical Computer Science*, pages 61–78. Elsevier, 2007. doi: 10.1016/j.entcs.2007.06.008.

[56] Azadeh Farzan. *Static and dynamic formal analysis of concurrent systems and languages: a semantics-based approach*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2007.

[57] Azadeh Farzan, Feng Chen, José Meseguer, and Grigore Roșu. Formal analysis of Java programs in JavaFAN. In Rajeev Alur and Doron 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. doi: 10.1016/j.entcs.2007.06.008.

[58] Azadeh Farzan, José Meseguer, and Grigore Roșu. Formal JVM code analysis in JavaFAN. In Charles Rattray, Savi Maharaj, and Carron Shankland, editors, *Algebraic Methodology and Software Technology, 10th International Conference, AMAST 2004, Stirling, Scotland, UK, July 12-16, 2004, Proceedings*, volume 3116 of *Lecture Notes in Computer Science*, pages 132–147. Springer, 2004. doi: 10.1007/978-3-540-27815-3_14.

[59] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD-machine, and the lambda-calculus. In *3rd Working Conference on the Formal Description of Programming Concepts*, pages 193–219, Ebberup, Denmark, August 1986.

[60] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press, Cambridge, MA, 2nd edition, 2001. ISBN 0-262-06217-8. URL http://www.cs.indiana.edu/eopl/.

[61] A. Garrido, J. Meseguer, and R. Johnson. Algebraic semantics of the C preprocessor and correctness of its refactorings. Technical Report UIUCDCS-R-2006-2688, Department of Computer Science, University of Illinois at Urbana-Champaign, February 2006.

[62] Jean-Louis Giavitto and Olivier Michel. MGS: a rule-based programming language for complex objects and collections. In Mark van den Brand and Rakesh Verma, editors, *RULE'01*, volume 59(4) of *Electronic Notes in Theoretical Computer Science*, pages 286–304. Elsevier Science Publishers, 2001. doi: 10.1016/S1571-0661(04)00293-2.

[63] Martin Gogolla, Paul Ziemann, and Sabine Kuske. Towards an integrated graph based semantics for UML. In *GT'02*, volume 72(3) of *Electronic Notes in Theoretical Computer Science*, pages 160–175, 2003. doi: 10.1007/s10270-008-0101-4.

[64] Joseph Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs*. MIT Press, 1996.

[65] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen, editor, *Applications of Algebraic Specification using OBJ*. Cambridge, 1993.

[66] Joseph A. Goguen and Kamran Parsaye-Ghomi. Algebraic denotational semantics using parameterized abstract modules. In Josep Díaz and Isidro Ramos, editors, *Formalization of Programming Concepts, International Colloquium, Peniscola, Spain, April 19-25, 1981, Proceedings*, volume 107 of *Lecture Notes in Computer Science*, pages 292–309. Springer, 1981. doi: 10.1007/3-540-10699-5_106.

[67] Yuri Gurevich. Evolving algebras 1993: Lipari Guide. In Egon Börger, editor, *Specification and Validation Methods*, pages 9–37. Oxford University Press, 1994.

[68] Annegret Habel, Hans-Jörg Kreowski, and Detlef Plump. Jungle evaluation. In *ADT'87*, volume 332 of *Lecture Notes in Computer Science*, pages 92–112. Springer, 1987. doi: 10.1007/3-540-50325-0_5.

[69] Annegret Habel, Jürgen Müller, and Detlef Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001. doi: 10.1017/S0960129501003425.

[70] Samuel P. Harbison and Guy L. Steele. *C: A Reference Manual (5th Edition)*. Prentice Hall, 2002.

[71] Robert Harper, Furio Honsell, and Gordon D. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993. doi: 10.1145/138027.138060.

[72] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *STTT*, 2(4):366–381, 2000. doi: 10.1007/s100090050043.

[73] Mark Hills. *A Modular Rewriting Approach to Language Design, Evolution and Analysis*. PhD thesis, University of Illinois at Urbana-Champaign, 2009. URL http://hdl.handle.net/2142/14600.

[74] Mark Hills and Grigore Roșu. KOOL: An application of rewriting logic to language prototyping and analysis. In Franz Baader, editor, *RTA*, volume 4533 of *Lecture Notes in Computer Science*, pages 246–256. Springer, 2007. ISBN 978-3-540-73447-5. doi: 10.1007/978-3-540-73449-9_19.

[75] Mark Hills and Grigore Roșu. On formal analysis of OO languages using rewriting logic: Designing for performance. In *Proceedings of the 9th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'07)*, volume 4468 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2007. doi: 10.1007/978-3-540-72952-5_7. also appeared as Technical Report UIUCDCS-R-2007-2809, January 2007.

[76] Mark Hills and Grigore Roșu. A rewriting approach to the design and evolution of object-oriented languages. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 827–828, New York, NY, USA, 2007. ACM. doi: 10.1145/1297846.1297908.

[77] Mark Hills and Grigore Roșu. Towards a module system for K. In Andrea Corradini and Ugo Montanari, editors, *WADT*, volume 5486 of *Lecture Notes in Computer Science*, pages 187–205. Springer, 2008. ISBN 978-3-642-03428-2. doi: 10.1007/978-3-642-03429-9_13.

[78] Mark Hills and Grigore Rosu. A rewriting logic semantics approach to modular program analysis. In Christopher 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. ISBN 978-3-939897-18-7. doi: 10.4230/LIPIcs.RTA.2010.151.

[79] Mark Hills, T. Barış Aktemur, and Grigore Roșu. An executable semantic definition of the Beta language using rewriting logic. Technical Report UIUCDCS-R-2005-2650, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[80] Mark Hills, Traian Florin Șerbănuță, and Grigore Roșu. A rewrite framework for language definitions and for generation of efficient interpreters. In Grit Denker and Carolyn Talcott, editors, *Proceedings of the 6th International Workshop on Rewriting Logic and its Applications (WRLA'06)*, volume 176(4) of *Electronic Notes in Theoretical Computer Science*, pages 215–231. Elsevier Science, July 2007. doi: 10.1016/j.entcs.2007.06.017. also appeared as Technical Report UIUCDCS-R-2005-2667, December 2005.

[81] Mark Hills, Feng Chen, and Grigore Roșu. A rewriting logic approach to static checking of units of measurement in C. In *Proceedings of the 9th International Workshop on Rule-Based Programming (RULE'08)*, volume To Appear of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2008.

[82] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–15, 1962.

[83] Berthold Hoffmann and Detlef Plump. Implementing term rewriting by jungle evaluation. *ITA*, 25:445–472, 1991.

[84] Michael Ilseman, Chucky Ellison, and Grigore Rosu. On compiling rewriting logic language definitions into competitive interpreters. Technical report, University of Illinos at Urbana Champaign, 2010.

[85] ISO/IEC JTC 1, SC 22, WG 14. Rationale for international standard—programming languages—C. Technical Report 5.10, International Organization for Standardization, April 2003.

[86] ISO/IEC JTC 1, SC 22, WG 14. ISO/IEC 9899:201x: Programming languages—C. Committee draft, International Organization for Standardization, August 2008.

[87] Einar Broch Johnsen, Olaf Owe, and Eyvind W. Axelsen. A run-time environment for concurrent objects with asynchronous method calls. In N. Martí-Oliet, editor, *Proceedings of the Fifth International Workshop on Rewriting Logic and its Applications (WRLA 2004)*, volume 117 of *Electronic Notes in Theoretical Computer Science*, pages 375–392. Elsevier, 2005. doi: 10.1016/j.entcs.2004.06.012.

[88] K. The K Framework, 2010. URL http://k-framework.googlecode.com.

[89] Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987. doi: 10.1007/BFb0039592.

[90] Harmen Kastenberg, Anneke Kleppe, and Arend Rensink. Defining object-oriented execution semantics using graph transformations. In Roberto Gorrieri and Heike Wehrheim, editors, *FMOODS*, volume 4037 of *Lecture Notes in Computer Science*, pages 186–201. Springer, 2006. ISBN 3-540-34893-X. doi: 10.1007/11768869_15.

[91] M. Katelman and J. Meseguer. A rewriting semantics for ABEL with applications to hardware/software co-design and analysis. In Grit Denker and Carolyn Talcott, editors, *Proceedings of the Sixth International Workshop on Rewriting Logic and its Applications (WRLA 2006)*, volume 176(4) of *Electronic Notes in Theoretical Computer Science*, pages 47–60. Elsevier, 2007. doi: 10.1016/j.entcs.2007.06.007.

[92] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Press, 2000.

[93] Hans-Jörg Kreowski. Transformations of derivation sequences in graph grammars. In *FCT'77*, pages 275–286, 1977. doi: 10.1007/3-540-08442-8_94.

[94] Masahito Kurihara and Azuma Ohuchi. Modularity in noncopying term rewriting. *Theoretical Computer Science*, 152(1):139–169, 1995. doi: 10.1016/0304-3975(94)00248-3.

[95] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Transactions on Computers*, 28(9):690–691, 1979. ISSN 0018-9340. doi: 10.1109/TC.1979.1675439.

[96] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343, New York, NY, USA, 1995. ACM Press. doi: 10.1145/199448.199528.

[97] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 378–391, New York, NY, USA, 2005. ACM. doi: 10.1145/1040305.1040336.

[98] Narciso Martí-Oliet and José Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285(2):121–154, 2002. doi: 10.1016/S0304-3975(01)00357-7.

[99] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, 2nd. Edition*, pages 1–87. Kluwer Academic Publishers, 2002. First published as SRI Tech. Report SRI-CSL-93-05, August 1993.

[100] Patrick Meredith, Mark Hills, and Grigore Roșu. An executable rewriting logic semantics of k-scheme. In Danny 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.

[101] Patrick O'Neil Meredith, Michael Katelman, José Meseguer, and Grigore Roşu. A formal executable semantics of Verilog. In *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'10)*, pages 179–188. IEEE, 2010. doi: doi:10.1109/MEMCOD.2010.555863.

[102] José Meseguer. Rewriting as a unified model of concurrency. In Jos C. M. Baeten and Jan Willem Klop, editors, *CONCUR'90*, volume 458 of *Lecture Notes in Computer Science*, pages 384–400. Springer, 1990. ISBN 3-540-53048-7. doi: 10.1007/BFb0039072.

[103] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992. doi: 10.1016/0304-3975(92)90182-F.

[104] José Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR'96*, volume 1119 of *Lecture Notes in Computer Science*, pages 331–372. Springer, 1996. ISBN 3-540-61604-7. doi: 10.1007/3-540-61604-7_64.

[105] José Meseguer. Membership algebra as a logical framework for equational specification. In Francesco Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1997. doi: 10.1007/3-540-64299-4_26.

[106] José Meseguer. Software specification and verification in rewriting logic. In M. Broy and M. Pizka, editors, *Models, Algebras, and Logic of Engineering Software, NATO Advanced Study Institute, Marktoberdorf, Germany, July 30 – August 11, 2002*, pages 133–193. IOS Press, 2003.

[107] José Meseguer. A rewriting logic sampler. In Dang Van Hung and Martin Wirsing, editors, *Theoretical Aspects of Computing - ICTAC 2005, Second International Colloquium, Hanoi, Vietnam, October 17-21, 2005, Proceedings*, volume 3722 of *Lecture Notes in Computer Science*, pages 1–28. Springer, 2005. doi: 10.1007/11560647_1.

[108] José Meseguer and Christiano Braga. Modular rewriting semantics of programming languages. In Charles Rattray, Savi Maharaj, and Carron Shankland, editors, *Algebraic Methodology and Software Technology, 10th International Conference, AMAST 2004, Stirling, Scotland, UK, July 12-16, 2004, Proceedings*, volume 3116 of *Lecture Notes in Computer Science*, pages 364–378. Springer, 2004. doi: 10.1007/978-3-540-27815-3_29.

[109] José Meseguer and Grigore Roșu. Rewriting logic semantics: From language specifications to formal analysis tools. In David A. Basin and Michaël Rusinowitch, editors, *Automated Reasoning - Second International Joint Conference, IJCAR 2004, Cork, Ireland, July 4-8, 2004, Proceedings*, volume 3097 of *Lecture Notes in Computer Science*, pages 1–44. Springer, 2004. doi: 10.1007/978-3-540-25984-8_1.

[110] José Meseguer and Grigore Roșu. The rewriting logic semantics project. In *Proceedings of the Second Workshop on Structural Operational Semantics (SOS 2005)*, volume 156(1) of *Electronic Notes in Theoretical Computer Science*, pages 27–56. Elsevier, 2006. doi: 10.1016/j.entcs.2005.10.027.

[111] José Meseguer and Grigore Roșu. The rewriting logic semantics project. *Theoretical Computer Science*, 373(3):213–237, 2007. doi: 10.1016/j.tcs.2006.12.018.

[112] José Meseguer, Miguel Palomino, and Narciso Martí-Oliet. Equational abstractions. *Theoretical Computer Science*, 403(2-3):239–264, 2008. doi: 10.1016/j.tcs.2008.04.040.

[113] Dale Miller. Representing and reasoning with operational semantics. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Computer Science*, pages 4–20. Springer, 2006. doi: 10.1007/11814771_3.

[114] Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992. doi: 10.1007/BFb0032030.

[115] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–77, 1992. doi: 10.1016/0890-5401(92)90008-4.

[116] Robin Milner, Mads Tofte, Robert Harper, and David Macqueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, USA, 1997.

[117] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Edinburgh University, Department of Computer Science, June 1989.

[118] Eugenio Moggi. A modular approach to denotational semantics. In David H. Pitt, Pierre-Louis Curien, Samson Abramsky, Andrew M. Pitts, Axel Poigné, and David E. Rydeheard, editors, *Category Theory and Computer Science*, volume 530 of *Lecture Notes in Computer Science*, pages 138–139. Springer, 1991. ISBN 3-540-54495-X. doi: 10.1007/BFb0013462.

[119] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[120] P. D. Mosses. Denotational semantics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B, Chapter 11*. North-Holland, 1990.

[121] Peter D. Mosses. Unified algebras and action semantics. In Burkhard Monien and Robert Cori, editors, *STACS 89, 6th Annual Symposium on Theoretical Aspects of Computer Science, Paderborn, FRG, February 16-18, 1989, Proceedings*, volume 349 of *Lecture Notes in Computer Science*, pages 17–35. Springer, 1989. doi: 10.1007/BFb0028970.

[122] Peter D. Mosses. Pragmatics of modular SOS. In Hélène Kirchner and Christophe Ringeissen, editors, *Algebraic Methodology and Software Technology, 9th International Conference, AMAST 2002, Saint-Gilles-les-Bains, Reunion Island, France, September 9-13, 2002, Proceedings*, volume 2422 of *Lecture Notes in Computer Science*, pages 21–40. Springer, 2002. doi: 10.1007/3-540-45719-4_3.

[123] Peter D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:195–228, 2004. doi: 10.1016/j.jlap.2004.03.008.

[124] Gopalan Nadathur and Dale Miller. An overview of Lambda-PROLOG. In Kenneth A. Bowen Robert A. Kowalski, editor, *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988, Proceedings*, pages 810–827. MIT Press, 1988.

[125] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* Springer, 2002. LNCS Tutorial 2283.

[126] Peter Csaba 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*, 2010. Springer. ISBN 978-3-642-16309-8. doi: 10.1007/978-3-642-16310-4.

[127] Peter Csaba Ölveczky and José Meseguer. Real-Time Maude 2.1. In N. Martí-Oliet, editor, *Proceedings of the Fifth International Workshop on Rewriting Logic and its Applications (WRLA 2004)*, volume 117 of *Electronic Notes in Theoretical Computer Science*, pages 285–314. Elsevier, 2005. doi: 10.1016/j.entcs.2004.06.015.

[128] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 391–407. Springer, 2009. ISBN 978-3-642-03358-2. doi: 10.1007/978-3-642-03359-9_27.

[129] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference.* Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.

[130] Andrei Păun and Gheorghe Păun. The power of communication: P systems with symport/antiport. *New Generation Computing*, 20(3):295–305, 2002. ISSN 0288-3635. doi: 10.1007/BF03037362.

[131] Gheorghe Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61:108–143, 2000. doi: 10.1006/jcss.1999.1693.

[132] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981. ISSN 0020-0190.

[133] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 199–208, New York, NY, USA, 1988. ACM Press. doi: 10.1145/53990.54010.

[134] B. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[135] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi calculus. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.

[136] Gordon D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004. doi: 10.1016/j.jlap.2004.05.001. Original version: University of Aarhus Technical Report DAIMI FN-19, 1981.

[137] Detlef Plump. Hypergraph rewriting: critical pairs and undecidability of confluence. In *Term graph rewriting: theory and practice*, pages 201–213. John Wiley and Sons Ltd., Chichester, UK, 1993. ISBN 0-471-93567-0.

[138] Detlef Plump. Term graph rewriting. In *Handbook of graph grammars and computing by graph transformation*, volume 2, pages 3–61. World Scientific, 1999.

[139] Arend Rensink and Eduardo Zambon. A type graph model for java programs. In David Lee, Antónia Lopes, and Arnd Poetzsch-Heffter, editors, *FMOODS/FORTE*, volume 5522 of *Lecture Notes in Computer Science*, pages 237–242. Springer, 2009. ISBN 978-3-642-02137-4. doi: 10.1007/978-3-642-02138-1_18.

[140] John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3-4):233–248, 1993. doi: 10.1007/BF01019459.

[141] Grigore Roşu, Chucky Ellison, and Wolfram Schulte. Matching logic: An alternative to Hoare/Floyd logic. In *Thirteenth International Conference on Algebraic Methodology And Software Technology (AMAST '10)*. LNCS, 2010. to appear.

[142] Grigore Roșu. CS322, Fall 2003 - Programming Language Design: Lecture Notes. Technical Report UIUCDCS-R-2003-2897, University of Illinos at Urbana Champaign, December 2003. Lecture notes of a course taught at UIUC.

[143] Grigore Roșu. 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.

[144] Grigore Roșu. Programming languages—a rewriting approach—. draft. URL http://fsl.cs.uiuc.edu/pub/pl.pdf.

[145] Grigore Roșu and Wolfram Schulte. Matching logic — extended report. Technical Report Department of Computer Science UIUCDCS-R-2009-3026, University of Illinois at Urbana-Champaign, January 2009.

[146] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010. doi: 10.1016/j.jlap.2010.03.012.

[147] Grigore Roșu, Ram Prasad Venkatesan, Jon Whittle, and Laurentiu Leustean. Certifying optimality of state estimation programs. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 301–314. Springer, 2003. doi: 10.1007/978-3-540-45069-6_30.

[148] Grigore Roșu, Chucky Ellison, and Wolfram Schulte. From rewriting logic executable semantics to matching logic program verification. Technical report, University of Illinois, July 2009. URL http://hdl.handle.net/2142/13159.

[149] Grigore Roşu, Wolfram Schulte, and Traian Florin Şerbănuţă. Runtime verification of C memory safety. In *Runtime Verification (RV'09)*, volume 5779 of *Lecture Notes in Computer Science*, pages 132–152, 2009. doi: 10.1007/978-3-642-04694-0_10.

[150] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3-4):289–360, 1993. doi: 10.1007/BF01019462.

[151] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The semantics of x86-cc multiprocessor machine code. In Zhong Shao and Benjamin C. Pierce, editors, *POPL*, pages 379–391. ACM, 2009. ISBN 978-1-60558-379-2. doi: 10.1145/1480881.1480929.

[152] Ralf Sasse and José Meseguer. Java+ITP: A verification tool based on Hoare logic and algebraic semantics. In Grit Denker and Carolyn L. Talcott, editors, *Proceedings of the 6th International Workshop on Rewriting Logic and its Applications (WRLA 2006)*, volume 176(4) of *Electronic Notes in Theoretical Computer Science*, pages 29–46, 2007. doi: 10.1016/j.entcs.2007.06.006.

[153] Ralph Sasse. Taclets vs. rewriting logic – relating semantics of Java. Master's thesis, Fakultät für Informatik, Universität Karlsruhe, Germany, May 2005. Technical Report in Computing Science No. 2005-16.

[154] David A. Schmidt. *Denotational Semantics – A Methodology for Language Development*. Allyn and Bacon, Boston, MA, 1986.

[155] Dana Scott. Outline of a mathematical theory of computation. In *Proceedings, Fourth Annual Princeton Conference on Information Sciences and Systems*, pages 169–176. Princeton University, 1970. Also appeared as Technical Monograph PRG 2, Oxford University, Programming Research Group.

[156] Dana Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. In *Microwave Research Institute Symposia Series, Vol. 21: Proc. Symp. on Computers and Automata*. Polytechnical Institute of Brooklyn, 1971.

[157] Koushik Sen, Grigore Roşu, and Gul Agha. Runtime safety analysis of multithreaded programs. In *FSE*, pages 337–346, 2003. doi: 10.1145/940071.940116.

[158] Koushik Sen, Grigore Roşu, and Gul Agha. Detecting errors in multithreaded programs by generalized predictive analysis. In *Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'05)*, volume 3535 of *Lecture Notes in Computer Science*, pages 211–226, 2005. doi: 10.1007/11494881_14.

[159] Traian Florin Şerbănuţă and Grigore Rosu. K-Maude: A rewriting based tool for semantics of programming languages. In Ölveczky [126], pages 104–122. ISBN 978-3-642-16309-8. doi: 10.1007/978-3-642-16310-4_8.

[160] Traian Florin Şerbănuţă, Grigore Roşu, and José Meseguer. A rewriting logic approach to operational semantics. *Information and Computation*, 207:305–340, 2009. doi: 10.1016/j.ic.2008.03.026.

[161] Traian Florin Șerbănuță, Gheorghe Stefanescu, and Grigore Roșu. Defining and executing P systems with structured data in K. In David W. Corne, Pierluigi Frisco, Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Workshop on Membrane Computing (WMC'08)*, volume 5391 of *Lecture Notes in Computer Science*, pages 374–393. Springer, 2009. ISBN 978-3-540-95884-0. doi: 10.1007/978-3-540-95885-7_26.

[162] Traian Florin Şerbănuţă, Feng Chen, and Grigore Roşu. Maximal causal models for sequentially consistent multithreaded systems. Technical report, University of Illinois, 2010. URL http://hdl.handle.net/2142/17336.

[163] Jaroslav Ševčík and David Aspinall. On validity of program transformations in the Java memory model. In Jan Vitek, editor, *ECOOP*, volume 5142 of *Lecture Notes in Computer Science*, pages 27–51. Springer, 2008. ISBN 978-3-540-70591-8. doi: 10.1007/978-3-540-70592-5_3.

[164] Kenneth Slonneger and Barry L. Kurtz. *Formal Syntax and Semantics of Programming Languages*. Addison-Wesley, 1995.

[165] Robert F. Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer, 2001.

[166] M.-O. Stehr and C. L. Talcott. Practical techniques for language design and prototyping. In J. L. Fiadeiro, U. Montanari, and M. Wirsing, editors, *Abstracts Collection of the Dagstuhl Seminar 05081 on Foundations of Global Computing. February 20 – 25, 2005. Schloss Dagstuhl, Wadern, Germany*, 2005.

[167] Mark-Oliver Stehr. CINNI - a generic calculus of explicit substitutions and its application to λ-, σ- and π- calculi. In K. Futatsugi, editor, *Proceedings of the Third International Workshop on Rewriting Logic and its Applications (WRLA 2000)*, volume 36 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2000. doi: 10.1016/S1571-0661(05)80125-2.

[168] Mark-Oliver Stehr and Carolyn L. Talcott. Plan in Maude: Specifying an active network programming language. In F. Gadducci and U. Montanari, editors, *Proceedings of the Forth International Workshop on Rewriting Logic and its Applications (WRLA 2002)*, volume 71 of *Electronic Notes in Theoretical Computer Science*, pages 240–260. Elsevier, 2002. doi: 10.1016/S1571-0661(05)82538-1.

[169] Mark-Oliver Stehr, Iliano Cervesato, and Stefan Reich. An execution environment for the MSR cryptoprotocol specification language. 2004. URL http://formal.cs.uiuc.edu/stehr/msr.html.

[170] Prasanna Thati, Koushik Sen, and Narciso Martí-Oliet. An executable specification of asynchronous Pi-Calculus semantics and may testing in Maude 2.0. In F. Gadducci and U. Montanari, editors, *Proceedings of the Forth International Workshop on Rewriting Logic and its Applications (WRLA 2002)*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002. doi: 10.1016/S1571-0661(05)82539-3.

[171] Mark van den Brand, Jan Heering, Paul Klint, and Pieter A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(4):334–368, 2002. doi: 10.1145/567097.567099.

[172] A. van Deursen, J. Heering, and P. Klint. *Language Prototyping: An Algebraic Specification Approach*. World Scientific, 1996.

[173] A. Verdejo. *Maude como marco semántico ejecutable*. PhD thesis, Facultad de Informática, Universidad Complutense, Madrid, Spain, 2003.

[174] Alberto Verdejo and Narciso Martí-Oliet. Implementing CCS in Maude 2. In F. Gadducci and U. Montanari, editors, *Proceedings of the Forth International Workshop on Rewriting Logic and its Applications (WRLA 2002)*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002. doi: 10.1016/S1571-0661(05)82540-X.

[175] Alberto Verdejo and Narciso Martí-Oliet. Executable structural operational semantics in Maude. *Journal of Logic and Algebraic Programming*, 67(1-2): 226–293, 2006. doi: 10.1016/j.jlap.2005.09.008.

[176] Patrick Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285(2):487–517, 2002. doi: 10.1016/S0304-3975(01)00366-8.

[177] Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in Stratego/XT 0.9. In Christian Lengauer, Don S. Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer, 2003. doi: 10.1007/978-3-540-25935-0_13.

[178] Philip Wadler. The essence of functional programming. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, New York, NY, USA, 1992. ACM Press. doi: 10.1145/143165.143169.

[179] Mitchell Wand. First-order identities as a defining language. *Acta Informatica*, 14:337–357, 1980. doi: 10.1007/BF00286491.

[180] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994. doi: 10.1006/inco.1994.1093.

[181] Yong Xiao, Zena M. Ariola, and Michel Mauny. From syntactic theories to interpreters: A specification language and its compilation. *The Computing Research Repository (CoRR)*, cs.PL/0009030, September 2000.