University of Illinois at Urbana-Champaign
Department of Computer Science Technical Report UIUCDCS-R-2005-2672

K: a Rewrite-based Framework for Modular Language Design, Semantics, Analysis and Implementation

—*Version 1*—

Grigore Roşu*
Department of Computer Science,
University of Illinois at Urbana-Champaign

Abstract

K is an algebraic framework for defining programming languages. It consists of a technique and of a specialized and highly optimized notation. The *K-technique*, which can be best explained in terms of rewriting modulo equations or in terms of rewriting logic, is based on a first-order representation of *continuations* with intensive use of *matching modulo* associativity, commutativity and identity. The *K-notation* consists of a series of high-level conventions that make the programming language definitions intuitive, easy to understand, to read and to teach, compact, modular and scalable. One important notational convention is based on *context transformers*, allowing one to automatically synthesize concrete rewrite rules from more abstract definitions once the concrete structure of the state is provided, by "completing" the contexts in which the rules should apply. The K framework is introduced by defining FUN, a concurrent higher-order programming language with parametric exceptions. A rewrite logic definition of a programming language can be executed on rewrite engines, thus providing an interpreter for the language for free, but also gives an initial model semantics, amenable to formal analysis such as model checking and inductive theorem proving. Rewrite logic definitions in K can lead to automatic, correct-by-construction generation of interpreters, compilers and analysis tools.

Note to readers: *The material presented in this report serves as a basis for programming language design and semantics classes and for several research projects. This report aims at giving a global snapshot of this rapidly evolving domain. Consequently, this work will be published on a version by version basis, each including and improving the previous ones, probably over a period of several years. This version already contains the desired structure of the final version, but not all sections are filled in yet. In this first version I put more emphasis on introducing the K framework and on how to use it. Here I focus less on related work, inductive verification and implementation; these will be approached in next versions of this work. My plan is to eventually transform this material into a book, so your suggestions and criticisms are welcome.*

*Supported by joint NSF grants CCF-0234524, CCF-0448501, and CNS-0509321.

Contents

1 Introduction

Appropriate frameworks for design and analysis of programming languages can not only help in understanding and teaching existing programming languages and paradigms, but can significantly facilitate our efforts and stimulate our desire to define and experiment with novel programming languages and programming language features, as well as programming paradigms or combinations of paradigms. But what makes a programming language definitional framework appropriate? We believe that an *ideal* such framework should satisfy at least some core requirements; the following are a few requirements that guided us in our quest for such a language definitional framework:

1. It should be *generic*, that is, not tied to any particular programming language or paradigm. For example, a framework enforcing object or thread communication via explicit send and receive messages may require artificial encodings of languages that opt for a different communication approach, while a framework enforcing static typing of programs in the defined language may be inconvenient for defining dynamically typed or untyped languages. In general, a framework providing and enforcing *particular* ways to define certain types of language features would lack genericity. Within an ideal framework, one can and should develop and adopt *methodologies* for defining certain types of languages or language features, but these should not be enforced. This genericity requirement is derived from the observation that today's programming languages are so diverse and based on orthogonal, sometimes even conflicting paradigms, that, regardless of how much we believe in the superiority of a particular language paradigm, be it object-oriented, functional or logical, a commitment to any existing paradigm would significantly diminish the strengths of a language definitional framework.

2. It should be *semantics-based*, that is, based on formal definitions of languages rather than on ad-hoc implementations of interpreters/compilers or program analysis tools. Semantics is crucial to such an ideal definitional framework because, without a formal semantics of a language, the problems of program analysis and interpreter or compiler correctness are meaningless. One could use ad-hoc implementations to find errors in programs or compilers, but never to *prove* their correctness. Ideally, we would like to have *one* definition of a language to serve *all* purposes, including parsing, interpretation, compilation, as well as formal analysis and verification of programs, such as static analysis, theorem proving and model checking. Having to annotate or slightly change/augment a language definition for certain purposes is acceptable and unavoidable; for example, certain domain-specific analyses may need domain information which is not present in the language definition; or, to more effectively model-check a concurrent program, one may impose abstractions that cannot be inferred automatically from the formal semantics of the language. However, having to develop an entirely new language definition for each purpose should be unacceptable in an ideal definitional framework.

3. It should be *executable*. There is not much one can do about the correctness of a definition, except to accumulate confidence in it. In the case of a programming language, one can accumulate confidence by proving desired properties of the language (but how many, when to stop?), or by proving equivalence to other formal definitions of the same language if they exist (but what if these definitions all have the same error?), or most practical of all, by having the possibility to execute programs *directly* using the semantic definition of the language. In our experience, executing hundreds of programs exercising various features of a language helps to not only find and fix errors in that language definition, but also stimulates the desire to

experiment with new features. A *computational logic framework* with efficient executability and a spectrum of meta-tools can serve as a basis to define executable formal semantics of languages, and also as a basis to develop generic formal analysis techniques and tools.

4. It should be able to naturally support *concurrency*. Due to the strong trend in parallelizing computing architectures for increased performance, probably most future programming languages will be concurrent. To properly define and reason about concurrent languages, the semantics of the underlying definitional framework should be inherently concurrent, rather than artificially graft support for concurrency on an essentially sequential paradigm, for example, by defining or simulating a process/thread scheduler. A semantics for *true concurrency* would be preferred to one based on *interleavings*, because executions of concurrent programs on parallel architectures are *not* interleaved.

5. It should be *modular*, to facilitate reuse of language features. In this context, modularity of a programming language definitional framework means more than just allowing grouping language feature definitions in modules. What it means is the ability to add or remove language features without having to modify any definitions of other, unrelated features. For example, if one adds parametric exceptions to one's language, then one should just include the corresponding module and change no other definition of any other language feature. In a modular framework, one can therefore define languages by adding features in a "plug-and-play" manner. A typical structural operational semantics (SOS) [22] definition of a language lacks modularity because one needs to "update" all the SOS rules whenever the structure of the state, or configuration, changes (like in the case of adding support for exceptions).

6. It should allow one to define any desired level of *computation granularity*, to capture the various degrees of abstraction of computation encountered in different programming languages. For example, some languages provide references as first class value citizens (e.g., C and C++); in order to get the value v that a reference variable x points to, one wants to perform *two semantic steps* in such languages: first get the value l of x, which is a location, and then read the value v at location l. However, other languages (e.g., Java) prefer not to make references visible to programmers, but only to use them internally as an efficient means to refer to complex data-structures; in such languages, one thinks of grabbing the (value stored in a) data-structure in *one semantic step*, because its location is not visible. An ideal framework should allow one to flexibly define any of these computation granularities. Another example would be the definition of functions with multiple arguments: an ideal framework should *not* enforce one to eliminate multiple arguments by currying; from a programming language design perspective, the fact that the underlying framework supports only one-argument functions or abstractions is regarded as a limitation of the framework. Other examples are mentioned later in the paper. Having the possibility to define different computation granularities for different languages gives a language designer several important benefits: better understanding of the language by not having to bother with low level implementation-like or "encoding" details, more freedom in how to implement the language, more appropriate abstraction of the language for formal analysis of programs, such as theorem proving and model checking, etc. From a computation granularity perspective, an extreme worst case of a definitional framework would provide a fixed computation granularity for all programming languages, for example one based on encodings of language features in λ-calculus or on Turing machines.

There are additional desirable, yet more subjective and thus harder to quantify, requirements of an ideal language definitional framework, including: it should be simple, easy to understand and teach; it should have good data representation capabilities; it should scale well, to apply to arbitrarily large languages; it should allow proofs of theorems about programming languages that are easy to comprehend; etc. The six requirements above are nevertheless ambitious. Moreover, there are subtle tensions among them, making it hard, if not impossible, to find an ideal language definitional framework. Some 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, sometimes most, of these ideal features (we discuss several such frameworks and their limitations in Section 3). 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 get a dynamic semantics of a language, but its (model-based) denotational semantics, proving properties about programs, model checking, etc., are "something else".

Following up recent work in rewriting logic semantics [19, 18], in this paper we argue that *rewriting logic* [17] can be a reasonable starting point towards the development of such an ideal framework. We call it a "starting point" because we believe that without appropriate language-specific front-ends (notations, conventions, etc.) and without appropriate definitional techniques, rewriting logic is simply too general, like the machine code of a processor or a Turing machine or a λ-calculus. In a nutshell, one can think of rewriting logic as a framework that gives complete semantics (that is, models that make the expected rewrite relation, or "deduction", complete) to the otherwise usual and standard *term rewriting modulo equations*. If one is specifically *not* interested in the model-theoretical semantic dimension of the proposed framework, but only in its operational (including proof theoretical, model checking and, in general, formal verification) aspects, then one can safely think of it as a framework to define programming languages as standard term rewrite systems modulo equations. The semantic counterpart is achieved at no additional cost (neither conceptual not notational), by just regarding rewrite systems as rewrite logic specifications.

A rewrite logic theory consists of a set of uninterpreted operations constrained equationally, together with a set of rewrite rules meant to define the concurrent evolution of the defined system. The distinction between equations and rewrite rules is only semantic. They are both executed as rewrite rules $l \to r$ by rewrite engines, following the simple, uniform and parallelizable *match-and-apply* principle of term rewriting: find a subterm matching l, say with a substitution θ, then replace it by $\theta(r)$. Therefore, if one is interested in just a dynamic semantics of a language, then, with few exceptions, one needs to make no distinction between equations and rewrite rules; the exceptions are some special equations, such as associativity and commutativity, enabling specialized algorithms in rewrite engines, which are used for non-computational purposes, namely for keeping structures, such as states, in convenient canonical forms.

Rewriting logic admits an *initial model semantics*, where equations form equivalence classes on terms and rewrite rules define transitions between such equivalence classes. Operationally, rewrite rules can be applied concurrently, thus making rewrite logic a very simple, generic and universal framework for concurrency; indeed, many other theoretical frameworks for concurrency, including π-calculus, process algebra, actors, etc., have been seamlessly defined in rewriting logic [16]. In our context of programming languages, a language definition is a rewrite logic theory in which (at least) the concurrent features of the language are defined using rewrite rules. A program together

with its initial state are given as an uninterpreted term, whose denotation in the initial model is its corresponding transition system. Depending on the desired type of analysis, one can, using existing tool support, generate anywhere from one path in that transition system (e.g., when "executing" the program) to all paths (e.g., for model checking).

One must, nevertheless, treat the simplicity and generality of rewriting logic with caution; "general" and "universal" need not necessarily mean "better" or "easier to use", for the same reason that machine code is not better or easier to use than higher level programming languages that translate into it. In our context of defining programming languages in rewriting logic, the right questions to ask are whether rewriting logic provides a natural framework for this task or not, and whether we get any benefit from using it. In spite of its simplicity and generality, rewriting logic does *not* give us any immediate recipe for *how* to define languages as rewrite logic theories. Appropriate *definitional techniques* and *methodologies* are necessary in order to make rewriting logic an effective computational framework for programming language definition and formal analysis.

In this paper we propose K, a domain-specific language front-end to rewriting logic that allows for compact, modular, executable, expressive and easy to understand and change semantic definitions of programming languages, concurrent or not. K could be explained and presented orthogonally to rewriting logic, as a standalone language definitional framework (same as, e.g., SOS or reduction semantics), but we prefer to regard it as a language-specific front-end to rewrite logic to reflect from the very beginning the fact that it inherits all the good properties and techniques of rewriting logic, a well established formalism with many uses and powerful tool support.

As discussed in [19, 18, 4] and in Section 3 of this paper, other definitional frameworks, such as SOS [22], MSOS [21] and reduction semantics [8], can also be easily translated into rewriting logic. More precisely, for a particular language formalized using any of these formalisms, say F, one can devise a rewrite logic specification, say R_{F}, which is *precisely the intended original language definition F, not an artificial encoding of it*; in other words, there is a one-to-one correspondence between derivations using F and rewrite logic derivations using R_{F}, obviously modulo a different but minor and ultimately irrelevant syntactic notation. This way, R_{F} has all the properties, good or bad, of F. However, in order to achieve such a bijective correspondence and thus the faithful translation of F into rewriting logic, one typically has to significantly restrain the strength of rewriting logic, reducing it to a limited computational framework, just like F. For example, the faithful translation of SOS definitions requires one conditional rewrite rule per SOS rule, but the resulting rewrite logic definition can apply rewrites only at the top, just like SOS, thus enforcing always an interleaving semantics for concurrent languages, just like SOS. Therefore, the fact that these formalisms translate into rewriting logic does *not* necessarily mean that they inherit all the good properties of rewriting logic. However, K *extends* rewriting logic with features that are meaningful for programming language formal definitions but which can be translated automatically back into rewriting logic, so K *is* rewriting logic.

An Overview of K

To give the reader an early feel for how K works, we next define a very simple untyped language, that we call λ_{K}, including booleans and integers together with the usual operations on them, λ-expressions and λ-application, conditionals, references and assignments, and a halt statement. To emphasize the modularity aspect and the strength of *context transformers*, we then show how one can extend λ_{K} with threads. Appendix D shows a translation of the K definition of sequential λ_{K} below into Maude, while Appendix E a translation of its concurrent extension with threads.

Consider the following syntax of non-concurrent λ_{K}:

$$
\begin{array}{rcl}
\mathit{Var} & ::= & \text{identifier} \\
\mathit{Bool} & ::= & \text{assumed defined, together with basic bool operations } \mathsf{not}_{\mathit{Bool}}\_ : \mathit{Bool} \to \mathit{Bool}, \text{etc.} \\
\mathit{Int} & ::= & \text{assumed, together with } \_ +_{\mathit{Int}} \_ : \mathit{Int} \times \mathit{Int} \to \mathit{Int},\ \_ <_{\mathit{Int}} \_ : \mathit{Int} \times \mathit{Int} \to \mathit{Bool}, \text{etc.} \\
\mathit{Exp} & ::= & \mathit{Var} \mid \mathit{Bool} \mid \mathit{Int} \mid \mathsf{not}\ \mathit{Exp} \mid \mathit{Exp} + \mathit{Exp} \mid \mathit{Exp} < \mathit{Exp} \mid ... \\
& & \mid \lambda\, \mathit{VarList}^{[,]}.\mathit{Exp} \mid \mathit{Exp}\ \mathit{ExpList}^{[,]} \\
& & \mid \mathsf{if}\ \mathit{Exp}\ \mathsf{then}\ \mathit{Exp}\ \mathsf{else}\ \mathit{Exp} \\
& & \mid \mathsf{ref}\ \mathit{Exp} \mid\ *\ \mathit{Exp} \mid \mathit{Exp} := \mathit{Exp} \\
& & \mid \mathsf{halt}\ \mathit{Exp}
\end{array}
$$

Boolean and integer expressions come with standard operations, which are indexed for clarity; note that we use the infix notation for some of these operations. Expressions of λ_{K} extend variables, booleans, integers, as well as all the "builtin" operations. The λ-abstraction and λ-application are standard, except that we assume by default that λ-abstractions take any number of arguments; here, $\mathit{VarList}^{[,]}$ and $\mathit{ExpList}^{[,]}$ stay for comma separated lists of variables and expressions, respectively. We deliberately decided *not* to follow the standard approach in which λ-abstractions are defined with only one argument and then multiple arguments are eliminated via currying, because that would change the *granularity level* of the language definition (see requirement number 6 above); additionally, most language implementations treat multiple arguments of functions together, as a block. In our view, from a language definitional and design perspective, a framework imposing such changes of granularity in a language definition just for the purpose of "reducing every language feature to a basic set of well-chosen constructs" is rather limited and falls into the same category with a framework translating any language construct into a sequence of Turing machine operations. We will later see that rewriting logic, and implicitly K, allow us to tune the granularity of computation also via *equational abstraction*: if certain rules are not intended to generate computation steps in the semantics of a language, then we make them equations; the more equations versus rules in a language definition, the fewer computational steps (and the larger the equivalence classes of terms/expressions/programs in the initial model of that language's definition).

For simplicity, we here assume a call-by-value evaluation strategy in λ_{K}; the other evaluation strategies for languages defined in K present no difficulty and are taught on a regular basis to undergraduate students [24]. The conditional expression expects its first argument to evaluate to a boolean and then, depending on its truth value, evaluates to either its second or its third expression argument; note that for the conditional we used the "mix-fix" syntactic notation, rather then a prefix one. The expression $\mathsf{ref}\ E$ evaluates E to some value, stores it at some location or *reference*, and then returns that reference as a result of its evaluation; therefore, in λ_{K} we (semantically) assume that references are first-class values in the language even though the programmer cannot use them in programs explicitly, since the syntax of λ_{K} does not define references (it actually does not define values, either). The expression $* R$, called *dereferencing* of R, evaluates R to a reference and then returns the value stored at that reference. The expression $R := E$, called *assignment*, first evaluates R to an (existing) reference r and E to a value v, and then writes v at r; the value v is also the result of the evaluation of the assignment. Finally, the expression $\mathsf{halt}\ E$ evaluates E to some value v and then aborts the computation and returns v as a result.

We next define the usual let bindings and sequential composition as *syntactic sugar* over λ_{K}, that is, $\mathsf{let}\ X = E\ \mathsf{in}\ E'$ is $(\lambda X.E')E$, $\mathsf{let}\ F(X) = E\ \mathsf{in}\ E'$ is $(\lambda F.E')(\lambda X.E)$, $\mathsf{let}\ F(X,Y) = E\ \mathsf{in}\ E'$ is $(\lambda F.E')(\lambda X,Y.E)$, etc., and $E; E'$ is $(\lambda D.E')E$, where D is a fresh "dummy" variable. If these

constructs were intended to be part of the language, then these definitions are clearly *not* suitable from a computational granularity point of view, because they change the intended granularity of these language constructs. In Section 5, we show how statements like these and many others can be defined *directly* (without translating them to other constructs), in the context of the more complex FUN language. For now, we can regard them as "notational conventions" for the more complex, equivalent expressions. With these conventions, the following is a λ_K expression that calculates factorial of n (here, n is regarded as a "macro"; also, note that this code cannot be used as part of a function to return the result of the factorial, because halt terminates the program — one could do it if one defines parametric exceptions as we do later in the paper, and then throw an exception instead of halt):

$$\begin{array}{l}\mathsf{let}\ r = \mathsf{ref}\ n\\ \mathsf{in\ let}\ g(m,h) = \ \mathsf{if}\ m > 1\ \mathsf{then}\ (r := (*r) * m;\ h(m-1,h))\ \mathsf{else\ halt}\ (*r)\\ \quad \mathsf{in}\ g(n-1,g)\end{array}$$

While λ_K is clearly a very simple toy language, we believe that it contains some canonical features that any ideal language definitional framework should be able to support naturally. Indeed, if a framework cannot define λ-expressions naturally than that framework is inappropriate to define most functional languages and most likely many other non-trivial languages. The conditional statement has been chosen for two reasons: first, most languages have conditional statements as a means to allow various control flows in programs, and, second, conditionals have an interesting evaluation strategy, namely strict in the first argument and lazy in the second and third; therefore, a naive use of a purely "lazy" or a purely "strict" definitional framework may lead to inconvenient encodings of conditionals. References are either directly or indirectly part of several mainstream languages, so they must present no difficulty in any language definitional framework. Finally, halt is one of the simplest control-intensive language constructs; if a language definitional framework cannot define halt naturally, then it most likely cannot define any control-intensive statements, including break and continue of loops, exceptions, and call/cc.

On the other hand, if a language definitional framework can define the λ_K language above easily, then it most likely can define many other programming language constructs of interest. Our purpose for introducing λ_K here is not to highlight specific features that languages could or should have, but instead to highlight that a language definitional framework should be flexible enough to support a rapid, yet formal, investigation of language features, lowering the boundary between new ideas and executable systems for trying these ideas. An important language feature that we have deliberately *not* added yet to λ_K is concurrency; that is because many existing frameworks were designed for sequential languages and one may therefore argue that a comparison of those with K on a concurrent language would not be fair. However, we will add threads to λ_K shortly and show how they can be given a semantics in K in a straightforward manner. Concurrent λ_K shows most of the subtleties of our framework.

The language definitional framework K consists of two important components:

- The *K-notation*, which can be regarded as a programming language specific *front-end* to rewriting logic, allows users to focus fully on the actual semantics of language features, rather than on distracting details or artifacts of rewriting logic, such as complete sort and operation declarations even though these can be trivially inferred from context, or adding conceptually unrelated state context just for the purpose of well formedness of the rewrite logic