WRLA'06, ENTCS 176(4), pp 215-231, 2007 WRLA 2006

A Rewrite Framework for Language Definitions and for Generation of Efficient Interpreters ⋆

Mark Hills [a,1] Traian Şerbănuţă [a,2] Grigore Roşu [a,3]

[a] *Department of Computer Science*
University of Illinois at Urbana-Champaign, USA
201 N Goodwin Ave, Urbana, IL 61801

Abstract

A rewrite logic semantic definitional framework for programming languages is introduced, called K, together with partially automated translations of K language definitions into rewriting logic and into C. The framework is exemplified by defining SILF, a simple imperative language with functions. The translation of K definitions into rewriting logic enables the use of the various analysis tools developed for rewrite logic specifications, while the translation into C allows for very efficient interpreters. A suite of tests show the performance of interpreters compiled from *K* definitions.

Key words: programming languages, rewriting logic, language interpreters.

1 Introduction

The K language definition framework [9] is a rewrite logic based framework for specifying programming languages. It includes both a notation, the *K-notation*, consisting of a series of domain-specific syntactic-sugar conventions aiming at simplifying and enhancing readability of language definitions, and a language definition technique, the *K-technique*, based on a first-order representation of continuations. As part of our ongoing research, we are developing a number of tools around K to assist in defining and analyzing programming languages.

Here, we show two pieces of this work. First, we show the semantics of a simple programming language with functions defined using K. This language has standard imperative features, including a controlled jump in the form of

⋆ Supported by NSF grants CCR 0234524, CCF-0448501, and CNS-0509321.
[1] Email: mhills@cs.uiuc.edu
[2] Email: tserban2@cs.uiuc.edu
[3] Email: grosu@cs.uiuc.edu

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

a function return. Second, we provide some details of a translation from our notation in K to an interpreter for the language, written in C. We are actively working on providing for the automated construction of interpreters from K definitions of languages, and currently have a semi-automated translation.

In Section 2, we present an overview of the K notation together with details of how it can be translated into rewrite logic. In Section 3 we show K at work by defining the Simple Imperative Language with Functions, or SILF. In Section 4 we provide details of our translation from K to C, including some initial performance figures of comparisons with equivalent programs written in other languages. Section 4 discusses related work, while Section 5 discusses future work and concludes the paper.

2 The K Language Definition Framework

Here we briefly recall the *K-framework* [9], useful to compactly, modularly and intuitively define languages in rewrite logic. It consists of the *K-notation*, i.e., a series of notational conventions for matching modulo axioms, for eliding unnecessary variables, for sort inference, and for *context transformers*, and of the *K-technique*, which is a *continuation-based* technique to define languages algebraically. The K-framework is described in detail in [9].

Matching Modulo. Despite its general intractability [3], matching modulo *Associativity*, *Commutativity*, and *Identity*, or *ACI-matching*, tends to be relatively efficient in practice. Many rewrite engines support it in its full generality. ACI-matching leads to compact and elegant, yet efficiently executable specifications. Different languages have different ways to state that binary operations are associative and/or commutative and/or have identities; to keep the discussion generic, we assume that all ACI operations are written using the *mixfix* concatenation notation "_ _" and have identity "·", while all but one [4] of the AI operations use the comma notation "_ , _" and have identity written also "·". In particular implementations of K specifications, to avoid confusion one may want to use different names for the different ACI or AI operations. ACI operations correspond to multi-sets, while the AI operations correspond to lists. Therefore, for any sort *Sort*, we tacitly add supersorts "*SortSet*", "*SortNeSet*", "*SortList*", and "*SortNeList*" of *Sort* (with the "*Ne*" versions being *non-empty*), constant operations "$\cdot : \rightarrow SortSet$" and "$\cdot : \rightarrow SortList$", and ACI operation "$\_\ \_ : SortSet \times SortSet \rightarrow SortSet$" and AI operation "$\_\,,\_ : SortList \times SortList \rightarrow SortList$" both with identities "·".

ACI operations will be used to define states as "soups" of attributes; e.g., the state of a language can be a "soup" containing a store, locks which are busy, input/output buffers, etc., as well as a set of threads. Soups can be nested; for example, a thread may contain itself a soup of thread attributes, such as an

[4] The exception to the comma notation for AI operations will be the "continuation"; defined later, it will follow, just for ease of reading, the notation $\_ \curvearrowright \_$.

environment, a set of locks that it holds, several stacks (for functions, exceptions, loops, etc.); an environment is further a soup of pairs (name,location), etc. Lists will be used to specify structures where the order of the attributes matters, such as buffers (for input/output), parameters of functions, etc.

For example, let us define an operation $update : Environment \times Name \times Location \to Environment$, where *Environment* is the set sort *NameLocationSet* associated to a pairing sort *NameLocation* with one constructor pairing operation $(\_,\_) : Name \times Location \to NameLocation$. $update(Env, X, L)$ is the same as *Env* except in the location of X, which should be replaced by L:

$$(\forall X : Name;\ L, L' : Location;\ Env : Environment)$$
$$update((X, L')\ Env,\ X,\ L) = (X, L)\ Env.$$

The ACI-matching algorithm "knows" that the first argument of *update* has an ACI constructor, so it will be able to match the lhs of this equation even though the pair (X, L') does *not* appear on the first position in the environment.

Sort Inference. Surprisingly, the variable declarations part of the equation of *update* takes almost half the size of the sentence. It is often the case in our experiments with defining languages in Maude that variable declarations take a significant amount of space, sometimes more than half the entire language specification. However, in most cases *the sorts of variables can be automatically inferred from the context*. To simplify this process, we assume that all variable names start with a capital letter. Consider, e.g., the two terms of the equation above, $update((X, L')\ Env,\ X,\ L)$ and $(X, L)\ Env$. Since the arity of *update* is $Environment \times Name \times Location \to Environment$, one can immediately infer that the sorts of X and L are *Name* and *Location*, respectively. Further, since the first argument of *update* has the sort *Environment* and since environments are constructed using the operation $\_\ \_ : Environment \times Environment \to Environment$, one can infer that the sort of *Env* is *Environment*.

Because of subsorting, a variable occurring on a position in a term may have multiple sorts. For example, the variable *Env* above can have both the sort *Environment* (which aliases *NameLocationSet*) and the sort *NameLocation*. The report [9] discusses in more depth the subtleties of sort inference in the presence of subsorting. Here we only recall that if an occurrence of a variable can have multiple sorts, we assume by default, or by convention, that that variable occurrence has *the largest* sort among those that it can have; this convention corresponds to the intuition that we assume the "least" information about each variable occurrence. If the same variable appears on multiple positions then we infer for that variable the "most concrete" sort that it can have among them. Technically, this is the intersection of all the largest sorts inferred for that variable on the different positions where it appears. If the variable sort-inference process is ambiguous, or if one is not sure, or if one really wants a different sort than the inferred one, or even simply for clarity, one is given the possibility to sort variables "on-the-fly": we append the

sort to the variable using ":", e.g., $X : Sort$. For example, from the term $update(Env, X, L)$ one can only infer that the sort of Env is $Environment$, the most general possible under the circumstances. If for any reason one wants to refer to a "special" environment of just one pair, then one can write $update(Env{:}NameLocation, X, L)$.

Underscore Variables and Tuples. With the sort inference conventions, the equation defining the operation *update* can be therefore written as

$$update((X, L')\ Env,\ X,\ L) = (X, L)\ Env.$$

Note that the location L' that occurs in the lhs is not needed; it is only used for "structural" purposes, i.e., it is there only to say that the name X is allocated at some location, but we do not care what that location is (we change it anyway). Since this will be a common phenomenon in our language definitions, we take the liberty to replace unnecessary letter variables by underscores, like in Prolog. Therefore, the equation above can be written

$$update((X, \_)\ Env,\ X,\ L) = (X, L)\ Env.$$

Like we need to pair names and locations to create environments, we will often need to tuple two or more terms in order to "save" current information for later processing. In K, by convention we allow all tupling operations without defining them explicitly. Like the sorts of variables, their arities can also be inferred from the context. Concretely, if the term $(X_1 : Sort1,\ X_2 : Sort2,\ \ldots,\ X_n : Sortn)$ appears in some context (the variable sorts may be inferred), then we implicitly add to the signature the sort $Sort1Sort2...Sortn$ and the operation $(\_, \_, \ldots, \_) : Sort1 \times Sort2 \times \cdots \times Sortn \to Sort1Sort2...Sortn$.

Contextual Notation for Rewrite Rules. All the subsequent rewrite rules will apply on just one (large) term, encoding the state of the program. Specifically, most of them will apply on subterms selected via matching, but only if the structure of the state permits it. In other words, most of our rules will be of the form $C[t_1] \cdots [t_n] \to C[t'_1] \cdots [t'_n]$, where C is some context term with $n \geq 0$ "holes" and t_1, ..., t_n are subterms that need to be replaced by t'_1, ..., t'_n in that context. C needs not match the entire state, but nevertheless sometimes it can be quite large. To simplify notation and ease reading, in K we write rules as $C[\underline{t_1}] \cdots [\underline{t_n}]$. This notation follows a natural intuition: first

$$\overline{t'_1} \qquad \overline{t'_n}$$

write the state context in which the transformation is intended to take place, then underline what needs to change, then write the changes under the line. Our contextual notation above proves to be particularly useful when combined with the "_" variables: if "_" appears in a context C, then it means that we do not care what is there but that we do not change it either.

Matching Prefixes, Suffixes and Fragments. We here introduce one more piece of notation that will help us further compact our language definitions by eliminating the need to mention unnecessary underscore variables. Many state attribute "soups" will be wrapped with specific operators