CS422 - Programming Language Design

K: A Rewrite Logic Framework for Programming Language Design

Grigore Roșu

Department of Computer Science
University of Illinois at Urbana-Champaign
What is K

K is an algebraic framework for defining programming languages or type systems for them. It consists of

- The **K-technique**, which is based on rewriting *modulo* associativity, commutativity and identity, and uses a first-order representation of *continuations*;

- The **K-notation**, which consists of a series of conventions that make the language definitions intuitive, easy to understand, to read and to teach, compact, modular and scalable.

The K framework is introduced by defining $\lambda_K$, a simple higher-order programming language. From here on in the class we’ll use K to define languages or language features.
Why \textbf{K}

As seen in previous lectures, the \textit{SOS definitional styles were problematic} when we wanted to define more complex features of languages, such as a halt statement, regardless of whether they are used on paper or formalized and automated using rewriting.

We have also seen that the rewrite-logic-based \textit{functional style had the same limitations as big-step SOS}. 
The *continuation style appears to be more appropriate* to define complex languages, because one has the control context of the program to evaluate explicit in the state, so one could easily define statements that change that control.

However, *the continuation style looked rather heavy* and hard to read and understand, at least in its Maude representation.

*K builds upon the continuation style*, providing a highly optimized notation and several intuitive conventions.

*K can be mechanically translated into rewriting logic*, so our K-language definitions can be executed in Maude after translation.
We exemplify the K framework by defining a simple functional language; yet, this language would be difficult to define if one does not use a good definitional formalism. Basic principles of functional programming were explained in the previous lecture notes.
Syntax of $\lambda_K$

$\text{Var} ::= \text{identifier}$

$\text{Bool} ::= \text{assumed defined, together with } \text{not}_{\text{Bool}} ::= \text{Bool} \rightarrow \text{Bool}, \text{etc.}$

$\text{Int} ::= \text{assumed, together with basic operations such as}$

$\_ +_{\text{Int}} \_ : \text{Int} \times \text{Int} \rightarrow \text{Int}, \_ <_{\text{Int}} \_ : \text{Int} \times \text{Int} \rightarrow \text{Bool}, \text{etc.}$

$\text{Exp} ::= \text{Var} | \text{Bool} | \text{Int} | \text{not} \text{ Exp} | \text{Exp} + \text{Exp} | \text{Exp} < \text{Exp} | \ldots$

$| \lambda \text{VarList}[^1].\text{Exp} | \text{Exp ExpList}[^1]$

$| \text{if} \ \text{Exp} \ \text{then} \ \text{Exp} \ \text{else} \ \text{Exp}$

$| \text{ref} \ \text{Exp} | \text{* Exp} | \text{Exp ::= Exp}$

$| \text{halt Exp}$
Other common functional language constructs can be easily defined using the constructs above, as *syntactic sugar*:

- let $X = E$ in $E'$ is $(\lambda X. E')E$,
- let $F(X) = E$ in $E'$ is $(\lambda F. E')(\lambda X. E)$,
- let $F(X, Y) = E$ 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.
With these, if \( n \) is for example 3, then

\[
\text{let } r = \text{ref } n \\
\text{in let } g(m, h) = \begin{cases} 
(r := (*)r \cdot m; h(m - 1, h)) & \text{if } m > 1 \\
\text{else halt } (*r) \end{cases} \\
\text{in } g(n - 1, g)
\]

translates into

\[
(\lambda r . \\
(\lambda g . g(3 - 1, g)) \\
(\lambda m, h . \begin{cases} 
(\lambda d . h(m - 1, h) (r := (*)r \cdot m)) & \text{if } m > 1 \\
\text{else halt } (*r) \end{cases}) \\
) (\text{ref } 3)
\]
Definition of $\lambda_K$ in $K$

The complete $K$ definition of $\lambda_K$ is shown in Figure 1 in the $K$-report (a link to it is provided). We discussed this definition and $K$ in class; a more detailed discussion can be found in the report.

Exercise 1 Read the first 22 pages (the introduction) of the $K$-report.