Tutorial 1 — LAMBDA

Grigore Roşu (grosu@illinois.edu)
University of Illinois at Urbana-Champaign

Abstract

This file defines a simple functional language in $\mathbb{K}$, called LAMBDA, using a substitution style. The explicit objective here is to teach some $\mathbb{K}$ concepts and how they work in the K tool, and not to teach λ-calculus or to argue for one definitional style against another (e.g., some may prefer environment/closure-based definitions of such languages).

Note that the subsequent definition is so simple, that it hardly shows any of the strengths of $\mathbb{K}$. Perhaps the most interesting $\mathbb{K}$ aspect it shows is that substitution can be defined fully generically, and then used to give semantics to various constructs in various languages.

Note: $\mathbb{K}$ follows the literate programming approach. The various semantic features defined in a $\mathbb{K}$ module can be reordered at will and can be commented using normal comments like in C/C++/Java. If those comments start with '`@`' preceded by no space (e.g., '`//@ \section{Variable declarations}`' or '`/*@ \section{Variable declarations} */`') then they are interpreted as formal LaTeX documentation by the `kompile` tool when used with the option `--pdf` (or `--latex`). While comments are useful in general, they can annoy the expert user of $\mathbb{K}$. To turn them off, you can do one of the following (unless you want to remove them manually): (1) Use an editor which can hide or color conventional C-like comments; or (2) Run the $\mathbb{K}$ pre-processor (kpp) on the ASCII .k file, which outputs (to stdout) a variant of the $\mathbb{K}$ definition with no comments.

Substitution

We need the predefined substitution module, so we require it with the command below. Then we should make sure that we import its module called SUBSTITUTION in our LAMBDA module below.

MODULE LAMBDA

Basic Call-by-value λ-Calculus

We first define a conventional call-by-value λ-calculus, making sure that we declare the lambda abstraction construct to be a binder, the lambda application to be strict, and the parentheses used for grouping as a bracket.

Note: Syntax in $\mathbb{K}$ is defined using the familiar BNF notation, with terminals enclosed in quotes and nonterminals starting with capital letters. Currently, $\mathbb{K}$ uses SDF as parsing frontend. Specifically, it extends BNF with several attributes and notations inspired from SDF, plus a few $\mathbb{K}$-specific attributes which will be described in this tutorial. To ease reading, the parsing- or typesetting-specific syntactic notations and attributes that appear in the ASCII semantics, such as the quotes around the terminals and operator precedences and grouping, are not displayed in the generated documentation. We only display the $\mathbb{K}$-specific attributes in the generated documentation, such as `strict`, `binder` and `bracket`, because those have a semantic nature.

Note: The `strict` constructs can evaluate their arguments in any (fully interleaved) orders.

The initial syntax of our λ-calculus:

SYNTAX *Val* ::= *Id*
| λ*Id*.*Exp* [binder]

SYNTAX *Exp* ::= *Val*
| *Exp* *Exp* [strict]
| (*Exp*) [bracket]

SYNTAX *KResult* ::= *Val*

β-reduction

RULE $\dfrac{(\lambda X{:}Id.E{:}Exp)\ \ V{:}Val}{E[V\ /\ X]}$

Integer and Boolean Builtins

The LAMBDA arithmetic and Boolean expression constructs are simply rewritten to their builtin counterparts once their arguments are evaluated. The operations with subscripts in the right-hand sides of the rules below are builtin and come with the corresponding builtin sort; they are actually written like `+Int` in ASCII, but they have LaTeX attributes to be displayed like $+_{Int}$ in the generated document. Note that the variables appearing in these rules have integer sort. That means that these rules will only be applied after the arguments of the arithmetic constructs are fully evaluated to $\mathbb{K}$ results; this will happen thanks to their strictness attributes declared as annotations to their syntax declarations (below).

SYNTAX *Val* ::= *Int*
| *Bool*

SYNTAX *Exp* ::= *Exp* * *Exp* [strict]
| *Exp* / *Exp* [strict]
| *Exp* + *Exp* [strict]
| *Exp* <= *Exp* [strict]

RULE $\dfrac{I1{:}Int * I2{:}Int}{I1 *_{Int} I2}$

RULE $\dfrac{I1{:}Int\ /\ I2{:}Int}{I1 \div_{Int} I2}$ requires $I2 =/=_{Int} 0$

RULE $\dfrac{I1{:}Int + I2{:}Int}{I1 +_{Int} I2}$

RULE $\dfrac{I1{:}Int <= I2{:}Int}{I1 \leq_{Int} I2}$

Conditional

Note that the `if` construct is strict only in its first argument.

SYNTAX *Exp* ::= `if` *Exp* `then` *Exp* `else` *Exp* [strict(1)]

RULE $\dfrac{\texttt{if true then } E \texttt{ else } \_}{E}$

RULE $\dfrac{\texttt{if false then } \_ \texttt{ else } E}{E}$

Let Binder

The let binder is a derived construct, because it can be defined using λ.

SYNTAX *Exp* ::= `let` *Id* = *Exp* `in` *Exp*

RULE $\dfrac{\texttt{let } X = E \texttt{ in } E'{:}Exp}{(\lambda X.E')\ \ E}$ [macro]

Letrec Binder

We prefer a definition based on the μ construct. Note that μ is not really necessary, but it makes the definition of letrec easier to understand and faster to execute.

SYNTAX *Exp* ::= `letrec` *Id* *Id* = *Exp* `in` *Exp*
| μ*Id*.*Exp* [binder]

RULE $\dfrac{\texttt{letrec } F{:}Id\ \ X{:}Id = E \texttt{ in } E'}{\texttt{let } F = \mu F.\lambda X.E \texttt{ in } E'}$ [macro]

RULE $\dfrac{\mu X.E}{E[(\mu X.E)\ /\ X]}$

END MODULE