Tutorial 3— LAMBDA++

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

Abstract

This file contains an environment-based $\mathbb{K}$ semantic definition of LAMBDA++, an extension of the LAMBDA language (defined in Tutorial 1) with a `callcc` construct. The objective here is to further disseminate some of the features of the K framework, in particular to illustrate how popular environment-based and closure-based semantics can be defined in K.

For notational/kompilation/krun simplicity and to avoid OS errors, we continue to write LAMBDA and lambda as names for modules and program extensions, respectively, in the sequel.

To restrict the default program parser invoked by krun, namely kast, to only parse proper LAMBDA++ programs no matter what other syntactic constructs we add to Exp later on in the semantics, we put the actual program syntax in a module with the suffix -SYNTAX. This issue was discussed in more detail in Lesson 2 of this tutorial. In short, the parser generated by kompile to be used by kast will be by default built only based on the syntax in this module. Type kompile –help to see how to tell the parser which syntax to use.

MODULE LAMBDA-SYNTAX

Syntax

We move all the LAMBDA++ syntax here.

SYNTAX $\mathit{Val} ::= \mathit{Int}$
$\quad\mid \mathit{Bool}$

SYNTAX $\mathit{Exp} ::= \mathit{Val}$
$\quad\mid \mathit{Id}$
$\quad\mid \lambda \mathit{Id}.\mathit{Exp}$
$\quad\mid \mathit{Exp}\ \ \mathit{Exp}$ [strict]
$\quad\mid (\mathit{Exp})$ [bracket]
$\quad\mid \mathit{Exp} * \mathit{Exp}$ [strict]
$\quad\mid \mathit{Exp}$ / Exp [strict]
$\quad\mid \mathit{Exp}$ + Exp [strict]
$\quad\mid \mathit{Exp}$ <= Exp [strict]

SYNTAX $\mathit{Exp} ::=$ `if` Exp `then` Exp `else` Exp [strict(1)]
$\quad\mid$ `let` Id = Exp `in` Exp
$\quad\mid$ `letrec` $\mathit{Id}\ \ \mathit{Id}$ = Exp `in` Exp
$\quad\mid \mu \mathit{Id}.\mathit{Exp}$
$\quad\mid$ `callcc` Exp [strict]

One thing you may want to do, now that the entire syntax is in one place, is to play with precedences. This way, you can make kompile generate the parser you want for your programs, so that you won't have to put lots of parentheses in your programs.

END MODULE

MODULE LAMBDA

Semantics

The next module contains the semantics of all the LAMBDA++ constructs, in the order in which their syntax was declared above.

The K Results

We should not forget to define the results of our computations. Here is a rule of thumb: whenever you have any strictness attributes, your should also define some K results. Or even simpler: *always define your results!* (unless you define a theoretical semantics, for analysis but not for execution purposes, you will need to define your results)

SYNTAX $\mathit{KResult} ::= \mathit{Val}$

Configuration

Since LAMBDA++ is such a simple language, its configuration is minimal for an environment-based semantics: it only contains the k cell, an environment cell, and a store cell. An environment binds variable names to locations, and a store binds locations to values.

CONFIGURATION:

$$\left\langle \langle \$PGM{:}\mathit{Exp} \rangle_{\mathsf{k}}\ \ \langle {}^{\bullet}\mathit{Map} \rangle_{\mathsf{env}}\ \ \langle {}^{\bullet}\mathit{Map} \rangle_{\mathsf{store}} \right\rangle_{\mathsf{T}}$$

Recall that $PGM is where the program is placed by krun after parsing.

Closures

In environment-based definitions of lambda-calculi, λ-abstractions evaluate to *closures*. A closure is like a λ-abstraction, but it also holds the environment in which it was declared. This way, when invoked, a closure knows where to find in the store the values of all the variables that its body expression refers to. To invoke a closure, we need to switch to closure's environment, then create a new binding for closure's parameter, then evaluate the closure's body, and then switch back to caller's environment.

SYNTAX $\mathit{Val} ::= \mathsf{closure}_{\lambda}(\mathit{Map}, \mathit{Id}, \mathit{Exp})$ [klabel('closure)]

RULE $\left\langle \dfrac{\lambda X{:}\mathit{Id}.E}{\mathsf{closure}_{\lambda}(\rho, X, E)} \right\rangle_{\mathsf{k}} \ \langle \rho \rangle_{\mathsf{env}}$ [structural]

RULE $\left\langle \dfrac{\mathsf{closure}_{\lambda}(\rho, X, E)\ \ V{:}\mathit{Val}}{E \curvearrowright \rho'} \right\rangle_{\mathsf{k}} \ \left\langle \dfrac{\rho'}{\rho[X \texttt{ <- } N]} \right\rangle_{\mathsf{env}} \ \left\langle \dfrac{{}^{\bullet}\mathit{Map}}{(N{:}\mathit{Int} \mapsto V)} \right\rangle_{\mathsf{store}}$

RULE $\left\langle \dfrac{X}{V} \right\rangle_{\mathsf{k}} \ \langle X \mapsto N \rangle_{\mathsf{env}} \ \langle N \mapsto V \rangle_{\mathsf{store}}$

Environment Recovery

The environment-recovery computation item defined below is useful in many semantics, like it was above. It is so useful, that there are discussions in the $\mathbb{K}$ team to add it to the set of pre-defined $\mathbb{K}$ features.

RULE $\left\langle \text{—:Val} \curvearrowright \dfrac{\rho}{{}^{\bullet}K} \right\rangle_{\mathsf{k}} \ \left\langle \dfrac{\text{—}}{\rho} \right\rangle_{\mathsf{env}}$ [structural]

Arithmetic Constructs

Not much to say here. They have exactly the same semantics as in LAMBDA and IMP. Note that we let it in programmer's hands to check that the denominator of a division is different from zero. If a division-by-zero is issued, then completely non-deterministic result can happen depending upon what back-end one uses for the K tool. Currently, Maude is used and Maude gets stuck with a term of the form `I /Int 0`, but one should not rely on that. If you want to catch division-by-zero in the semantics, instead of letting the back-end do whatever it wants, you should add a side condition to the division rule.

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

RULE $\dfrac{I1 \texttt{ / } I2}{I1 \div_{\mathit{Int}} I2}$

RULE $\dfrac{I1 \texttt{ + } I2}{I1 +_{\mathit{Int}} I2}$

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

Conditional

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

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

Let Binder

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

Letrec Binder

We define `letrec` in term of `mu`, whose semantics is below.

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

Mu

To save the number of locations needed to evaluate $\mu X.E$, we replace it with a special closure which binds X to a fresh location holding the closure itself. This has the same effect as binding X to a reference that points back to the fixed-point.

SYNTAX $\mathit{Exp} ::= \mathsf{closure}_{\mu}(\mathit{Map}, \mathit{Exp})$ [klabel('muclosure)]

RULE $\left\langle \dfrac{\mu X.E}{\mathsf{closure}_{\mu}(\rho[X \texttt{ <- } N], E)} \right\rangle_{\mathsf{k}} \ \langle \rho \rangle_{\mathsf{env}} \ \left\langle \dfrac{{}^{\bullet}\mathit{Map}}{(N{:}\mathit{Int} \mapsto \mathsf{closure}_{\mu}(\rho[X \texttt{ <- } N], E))} \right\rangle_{\mathsf{store}}$ [structural]

RULE $\left\langle \dfrac{\mathsf{closure}_{\mu}(\rho, E)}{E \curvearrowright \rho'} \right\rangle_{\mathsf{k}} \ \left\langle \dfrac{\rho'}{\rho} \right\rangle_{\mathsf{env}}$

Callcc

For `callcc`, we need to create a new closure-like value which wraps both the remaining computation and the environment in which it is supposed to be executed. Forget the environment and you get a wrong `callcc`.

SYNTAX $\mathit{Val} ::= \mathsf{closure}_{\texttt{callcc}}(\mathit{Map}, K)$ [klabel('cc)]

RULE $\left\langle \dfrac{\texttt{callcc } V{:}\mathit{Val}}{V\ \ \mathsf{closure}_{\texttt{callcc}}(\rho, K)} \curvearrowright K \right\rangle_{\mathsf{k}} \ \langle \rho \rangle_{\mathsf{env}}$

RULE $\left\langle \dfrac{\mathsf{closure}_{\texttt{callcc}}(\rho, K)\ \ V{:}\mathit{Val} \curvearrowright \text{—}}{V \curvearrowright K} \right\rangle_{\mathsf{k}} \ \left\langle \dfrac{\text{—}}{\rho} \right\rangle_{\mathsf{env}}$

END MODULE