IMP

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

Abstract

This is the $\mathbb{K}$ semantic definition of the classic IMP language. IMP is considered a folklore language, without an official inventor, and has been used in many textbooks and papers, often with slight syntactic variations and often without being called IMP. It includes the most basic imperative language constructs, namely basic constructs for arithmetic and Boolean expressions, and variable assignment, conditional, while loop and sequential composition constructs for statements.

MODULE IMP-SYNTAX

Syntax

This module defines the syntax of IMP. Note that `<=` is sequentially strict and has a LaTeX attribute making it display as $\leq$, and that && is strict only in its first argument, because we want to give it a short-circuit semantics.

SYNTAX *AExp* ::= *Int*
 | *Id*
 | *AExp* / *AExp* [strict]
 | *AExp* + *AExp* [strict]
 | (*AExp*) [bracket]

SYNTAX *BExp* ::= *Bool*
 | *AExp* $\leq$ *AExp* [seqstrict]
 | `!` *BExp* [strict]
 | *BExp* && *BExp* [strict(1)]
 | (*BExp*) [bracket]

SYNTAX *Block* ::= {}
 | {*Stmt*}

SYNTAX *Stmt* ::= *Block*
 | *Id* = *AExp* `;` [strict(2)]
 | `if` (*BExp*)*Block* `else` *Block* [strict(1)]
 | `while` (*BExp*)*Block*
 | *Stmt* *Stmt*

An IMP program declares a set of variables and then executes a statement in the state obtained after initializing all those variables to 0. $\mathbb{K}$ provides builtin support for generic syntactic lists: *List*{*Nonterminal*, *terminal*} stands for *terminal*-separated lists of *Nonterminal* elements.

SYNTAX *Pgm* ::= `int` *Ids* `;` *Stmt*

SYNTAX *Ids* ::= *List*{*Id*, "," }

END MODULE

We are done with the definition of IMP's syntax. Make sure that you write and parse several interesting programs before you move to the semantics.

MODULE IMP

Semantics

This module defines the semantics of IMP. Before you start adding semantic rules to a $\mathbb{K}$ definition, you need to define the basic semantic infrastructure consisting of definitions for *results* and the *configuration*.

Values and results

IMP only has two types of values, or results of computations: integers and Booleans. We here use the $\mathbb{K}$ builtin variants for both of them.

SYNTAX *KResult* ::= *Int*
 | *Bool*

Configuration

The configuration of IMP is trivial: it only contains two cells, one for the computation and another for the state. For good encapsulation and clarity, we place the two cells inside another cell, the "top" cell which is labeled T.

CONFIGURATION:

T: [k: $\$PGM{:}Pgm$ | state: $\bullet_{Map}$]

The configuration variable $\$PGM$ tells the $\mathbb{K}$ tool where to place the program. More precisely, the command "`krun program`" parses the program and places the resulting $\mathbb{K}$ abstract syntax tree in the k cell before invoking the semantic rules described in the sequel. The "$\bullet$" in the state cell, written `.Map` in ASCII in the `imp.k` file, is $\mathbb{K}$'s way to say "nothing". Technically, it is a constant which is the unit, or identity, of all maps in $\mathbb{K}$ (similar dot units exist for other $\mathbb{K}$ structures, such as lists, sets, multi-sets, etc.).

Arithmetic expressions

The $\mathbb{K}$ semantics of each arithmetic construct is defined below.

Variable lookup. A program variable X is looked up in the state by matching a binding of the form $X \mapsto I$ in the state cell. If such a binding does not exist, then the rewriting process will get stuck. Thus our semantics of IMP disallows uses of uninitialized variables. Note that the variable to be looked up is the first task in the k cell (the cell is closed to the left and torn to the right), while the binding can be anywhere in the `state` cell (the cell is torn at both sides).

RULE k: $\dfrac{X{:}Id}{I}$ … state: … $X \mapsto I$ …

Arithmetic operators. There is nothing special about these, but recall that $\mathbb{K}$'s configuration abstraction mechanism is at work here! That means that the rewrites in the rules below all happen at the beginning of the k cell.

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

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

Boolean expressions

The rules below are straightforward. Note the short-circuited semantics of &&; this is the reason we annotated the syntax of && with the $\mathbb{K}$ attribute `strict(1)` instead of `strict`.

RULE $\dfrac{I1 \leq I2}{I1 \leq_{Int} I2}$

RULE $\dfrac{!\ T}{\neg_{Bool} T}$

RULE $\dfrac{\mathsf{true}\ \&\&\ B}{B}$

RULE $\dfrac{\mathsf{false}\ \&\&\ —}{\mathsf{false}}$

Blocks and Statements

There is one rule per statement construct except for the conditional, which needs two rules.

Blocks. The empty block `{}` is simply dissolved. The $\bullet$ below is the unit of the computation list structure K, that is, the empty task. Similarly, the non-empty blocks are dissolved and replaced by their statement contents, thus effectively giving them a bracket semantics; we can afford to do this only because we have no block-local variable declarations yet in IMP. Since we tagged the rules below as "structural", the $\mathbb{K}$ tool structurally erases the block constructs from the computation structure, without considering their erasure as computational steps in the resulting transition systems. You can make these rules computational (dropping the attribute `structural`) if you do want these to count as computational steps.

RULE $\dfrac{\{\}}{\bullet_K}$ [structural]

RULE $\dfrac{\{S\}}{S}$ [structural]

Assignment. The assigned variable is updated in the state. The variable is expected to be declared, otherwise the semantics will get stuck. At the same time, the assignment is dissolved.

RULE k: $\dfrac{X = I{:}Int\ ;}{\bullet_K}$ … state: … $X \mapsto \dfrac{—}{I}$ …

Sequential composition. Sequential composition is simply structurally translated into $\mathbb{K}$'s builtin task sequentialization operation. You can make this rule computational (i.e., remove the `structural` attribute) if you want it to count as a computational step. Recall that the semantics of a program in a programming language defined in $\mathbb{K}$ is the transition system obtained from the initial configuration holding that program and counting only the steps corresponding to computational rules as transitions (i.e., hiding the structural rules as unobservable, or internal steps).

RULE $\dfrac{S1{:}Stmt\ \ S2{:}Stmt}{S1 \curvearrowright S2}$ [structural]

Conditional. The conditional statement has two semantic cases, corresponding to when its condition evaluates to `true` or to `false`. Recall that the conditional was annotated with the attribute `strict(1)` in the syntax module above, so only its first argument is allowed to be evaluated.

RULE $\dfrac{\texttt{if}\ (\mathsf{true})S\ \texttt{else}\ —}{S}$

RULE $\dfrac{\texttt{if}\ (\mathsf{false})—\ \texttt{else}\ S}{S}$

While loop. We give the semantics of the `while` loop by unrolling. Note that we preferred to make the rule below structural.

RULE $\dfrac{\texttt{while}\ (B)S}{\texttt{if}\ (B)\{S\ \ \texttt{while}\ (B)S\}\ \texttt{else}\ \{\}}$ [structural]

Programs

The semantics of an IMP program is that its body statement is executed in a state initializing all its global variables to 0. Since $\mathbb{K}$'s syntactic lists are internally interpreted as cons-lists (i.e., lists constructed with a head element followed by a tail list), we need to distinguish two cases, one when the list has at least one element and another when the list is empty. In the first case we initialize the variable to 0 in the state, but only when the variable is not already declared (all variables are global and distinct in IMP). We prefer to make the second rule structural, thinking of dissolving the residual empty `int;` declaration as a structural cleanup rather than as a computational step.

RULE k: $\texttt{int}\ \dfrac{X\ ,\ Xs}{Xs}\ ;\ —$ state: $\rho{:}Map\ \ \dfrac{\bullet_{Map}}{X \mapsto 0}$ requires $\neg_{Bool}(X\ \texttt{in}\ \texttt{keys}\ (\rho))$

RULE $\dfrac{\texttt{int}\ \bullet_{Ids}\ ;\ S}{S}$ [structural]

END MODULE