IMP++

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

Abstract

This is the $\mathbb{K}$ semantic definition of the IMP++ language. IMP++ extends the IMP language with the features listed below. We strongly recommend you to first familiarize yourself with the IMP language and its $\mathbb{K}$ definition in Tutorial 2 before proceeding.

Strings and concatenation of strings. Strings are useful for the `print` statement, which is discussed below. For string concatenation, we use the same `+` construct that we use for addition (so we overload it).

Variable increment. We only add a pre-increment construct: `++x` increments variable `x` and evaluates to the incremented value. Variable increment makes the evaluation of expressions have side effects, and thus makes the evaluation strategies of the various language constructs have an influence on the set of possible program behaviors.

Input and output. IMP++ adds a `read()` expression construct which reads an integer number and evaluates to it, and a variadic (i.e., it has an arbitrary number of arguments) statement construct `print(e1,e2,...,en)` which evaluates its arguments and then outputs their values. Note that the $\mathbb{K}$ tool allows to connect the input and output cells to the standard input and output buffers, this way compiling the language definition into an interactive interpreter.

Abrupt termination. The `halt` statement simply halts the program. The $\mathbb{K}$ tool shows the resulting configuration, as if the program terminated normally. We therefore assume that an external observer does not care whether the program terminates normally or abruptly, same like with `exit` statements in conventional programming languages like C.

Dynamic threads. The expression construct `spawn s` starts a new concurrent thread that executes statement `s`, which is expected to be a block, and evaluates immediately to a fresh thread identifier that is also assigned to the newly created thread. The new thread is given at creation time the *environment* of its parent, so it can access all its parent's variables. This allows for the parent thread and the child thread to communicate; it also allows for races and "unexpected" behaviors, so be careful. For thread synchronization, IMP++ provides a thread join statement construct "`join t;`", where `t` evaluates to a thread identifier, which stalls the current thread until thread `t` completes its computation. For simplicity, we here assume a sequentially consistent shared memory model. To experiment with other memory models, see the definition of KERNELC.

Blocks and local variables. IMP++ allows blocks enclosed by curly brackets. Also, IMP's global variable declaration construct is generalized to be used anywhere as a statement, not only at the beginning of the program. As expected, the scope of the declared variables is from their declaration point till the end of the most nested enclosing block.

What You Will Learn Here

- How to define a less trivial language in $\mathbb{K}$, as explained above.
- How to use the `superheat` and `supercool` options of the $\mathbb{K}$ tool `kompile` to exhaustively explore the non-determinism due to underspecified evaluation strategies.
- How to use the `transition` option of the $\mathbb{K}$ tool to exhaustively explore the non-determinism due to concurrency.
- How to connect certain cells in the configuration to the standard input and standard output, and thus turn the `krun` tool into an interactive interpreter for the defined language.
- How to exhaustively search for the non-deterministic behaviors of a program using the `search` option of `krun`.

MODULE IMP-SYNTAX

Syntax

IMP++ adds several syntactic constructs to IMP. Also, since the variable declaration construct is generalized to be used anywhere a statement can be used, not only at the beginning of the program, we need to remove the previous global variable declaration of IMP and instead add a variable declaration statement construct

We do not re-discuss the constructs which are taken over from IMP, except when their syntax has been subtly modified (such as, for example, the syntax of the previous "statement" assignment which is now obtained by composing the new assignment expression and the new expression statement constructs); go the last lesson of Tutorial 2 if you are interested in IMP's constructs. For execution purposes, we tag the addition and division operations with the `addition` and `division` tags. These attributes have no theoretical significance, in that they do not affect the semantics of the language in any way. They only have practical relevance, specific to our implementation of the $\mathbb{K}$ tool. Specifically, we can tell the $\mathbb{K}$ tool (using its `superheat` and `supercool` options) that we want to exhaustively explore all the non-deterministic behaviors (due to strictness) of these language constructs. For performance reasons, by default the $\mathbb{K}$ tool chooses an arbitrary but fixed order to evaluate the arguments of the strict language constructs, thus possibly losing behaviors due to missed interleavings. This aspect was irrelevant in IMP, because its expressions had no side effects, but it becomes relevant in IMP++.

The syntax of the IMP++ constructs is self-explanatory. Note that assignment is now an expression construct. Also, `print` is variadic, taking a list of expressions as argument. It is also strict, which means that the entire list of expressions, that is, each expression in the list, will be evaluated. Note also that we have now defined sequential composition of statements as a whitespace-separated list of statements, aliased with the nonterminal `Stmts`, and block as such a (possibly empty) sequence of statements surrounded by curly brackets.

SYNTAX *AExp* ::= *Int*
| *String*
| *Id*
| `++` *Id*
| `read` ()
| *AExp* `/` *AExp* [strict, division]
| *AExp* `+` *AExp* [strict]
| `spawn` *Block*
| *Id* `=` *AExp* [strict(2)]
| (*AExp*) [bracket]

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

SYNTAX *Block* ::= {*Stmts*}

SYNTAX *Stmt* ::= *Block*
| *AExp* `;` [strict]
| `if` (*BExp*)*Block* `else` *Block* [strict(1)]
| `while` (*BExp*)*Block*
| `int` *Ids* `;`
| `print` (*AExps*) `;` [strict]
| `halt ;`
| `join` *AExp* `;` [strict]

SYNTAX *Ids* ::= $List\{Id, \text{“,”}\}$ [strict]

SYNTAX *AExps* ::= $List\{AExp, \text{“,”}\}$ [strict]

SYNTAX *Stmts* ::= $List\{Stmt, \text{“”}\}$

END MODULE

MODULE IMP

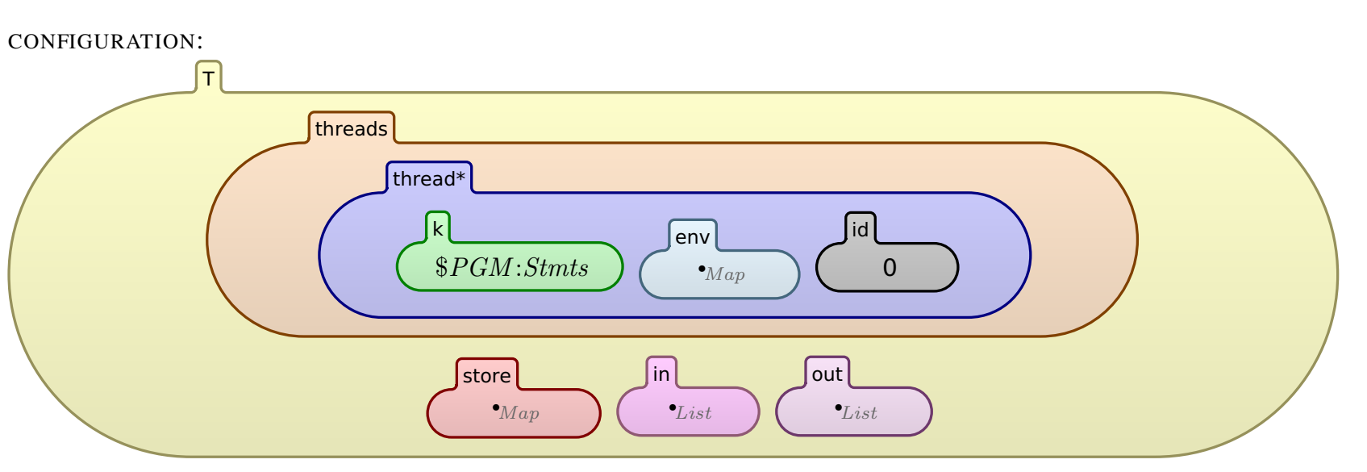
Semantics

We next give the semantics of IMP++. We start by first defining its configuration.

Configuration

The original configuration of IMP has been extended to include all the various additional cells needed for IMP++. To facilitate the semantics of threads, more specifically to naturally give them access to their parent's variables, we prefer a (rather conventional) split of the program state into an *environment* and a *store*. An environment maps variable names into *locations*, while a store maps locations into values. Stores are also sometimes called "states", or "heaps", or "memory", in the literature. Like values, locations can be anything. For simplicity, here we assume they are natural numbers. Moreover, each thread has its own environment, so it knows where all the variables that it has access to are located in the store (that includes its locally declared variables as well as the variables of its parent thread), and its own unique identifier. The store is shared by all threads. For simplicity, we assume a sequentially consistent memory model in IMP++. Note that the `thread` cell has multiplicity "*", meaning that there could be zero, one, or more instances of that cell in the configuration at any given time. This multiplicity information is important for $\mathbb{K}$'s *configuration abstraction* process: it tells $\mathbb{K}$ how to complete rules which, in order to increase the modularity of the definition, choose to not mention the entire configuration context. The `in` and `out` cells hold the input and the output buffers as lists of items.

CONFIGURATION:



We can also use configuration variables to initialize the configuration through `krun`. For example, we may want to pass a few list items in the `in` cell when the program makes use of `read()`, so that the semantics does not get stuck. Recall from IMP that configuration variables start with a \$ character when used in the configuration (see, for example, *\$PGM*) and can be initialized with any string by `krun`; or course, the string should parse to a term of the corresponding sort, otherwise errors will be generated. Moreover, $\mathbb{K}$ allows you to connect list cells to the standard input or the standard output. For example, if you add the attribute `stream="stdin"` to the `in` cell, then `krun` will prompt the user to pass input when the `in` cell is empty and any semantic rule needs at least one item to be present there in order to match. Similarly but dually, if you add the attribute `stream="stdout"` to the `out` cell, then any item placed into this cell by any rule will be promptly sent to the standard output. This way, `krun` can be used to obtain interactive interpreters based directly on the $\mathbb{K}$ semantics of the language. For example:

```
bash$ krun sum-io.imp --no-config
Add numbers up to (<= 0 to quit)? 10
Sum = 55
Add numbers up to (<= 0 to quit)? 1000
Sum = 500500
Add numbers up to (<= 0 to quit)? 0
bash$
```

The option `--no-config` instructs `krun` to not display the resulting configuration after the program executes. The input/output streaming works with or without this option, although if you don't use the option then a configuration with empty `in` and `out` cells will be displayed after the program is executed. You can also initialize the configuration using configuration variables and stream the contents of the cells to standard input/output at the same time. For example, if you use a configuration variable in the `in` cell and pass contents to it through `krun`, then that contents will be first consumed and then the user will be prompted to introduce additional input if the program's execution encounters more `read()` constructs.

The old IMP constructs

The semantics of the old IMP constructs is almost identical to their semantics in the original IMP language, except for those constructs making use of the program state and for those whose syntax has slightly changed. Indeed, the rules for variable lookup and assignment in IMP accessed the `state` cell, but that cell is not available in IMP++ anymore. Instead, we have to use the combination of environment and store cells. Thanks to $\mathbb{K}$'s implicit configuration abstraction, we do not have to mention the `thread` and `threads` cells: these are automatically inferred (and added by the $\mathbb{K}$ tool at compile time) from the definition of the configuration above, as there is only one correct way to complete the configuration context of these rules in order to match the configuration declared above. In our case here, "correct way" means that the `k` and `env` cells will be considered as being part of the same `thread` cell, as opposed to each being part of a different thread. Configuration abstraction is crucial for modularity, because it gives us the possibility to write our definitions in a way that may not require us to revisit existing rules when we change the configuration. Changes in the configuration are quite frequent in practice, typically needed in order to accommodate new language features. For example, imagine that we initially did not have threads in IMP++. There would be no need for the `thread` and `threads` cells in the configuration then, the cells `k` and `env` being simply placed at the top level in the `T` cell, together with the already existing cells. Then the rules below would be exactly the same. Thus, configuration abstraction allows you to not have to modify your rules when you make structural changes in your language configuration.

Below we list the semantics of the old IMP constructs, referring the reader to the $\mathbb{K}$ semantics of IMP for their meaning. Like we tagged the addition and the division rules above in the syntax, we also tag the lookup and the assignment rules below (with tags `lookup` and `assignment`), because we want to refer to them when we generate the language model (with the `kompile` tool), basically to allow them to generate (possibly non-deterministic) transitions. Indeed, these two rules, unlike the other rules corresponding to old IMP constructs, can yield non-deterministic behaviors when more threads are executed concurrently. In terms of rewriting, these two rules can "compete" with each other on some program configurations, in the sense that they can both match at the same time and different behaviors may be obtained depending upon which of them is chosen first.

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

Variable lookup.

RULE $\left\langle \frac{X:Id}{I} \cdots \right\rangle_{\mathsf{k}} \left\langle \cdots X \mapsto N \cdots \right\rangle_{\mathsf{env}} \left\langle \cdots N \mapsto I \cdots \right\rangle_{\mathsf{store}}$ [lookup]

Arithmetic constructs.

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

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

Boolean constructs.

RULE $\frac{I1:Int \leq I2:Int}{I1 \leq_{Int} I2}$

RULE $\frac{!\; T:Bool}{\neg_{Bool} T}$

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

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

Variable assignment. Note that the old IMP assignment statement "X = I;" is now composed of two constructs: an assignment expression construct "X = I", followed by a semicolon ";" turning the expression into a statement. The rationale behind this syntactic restructuring has been explained in Lesson 7. Here is the semantics of the two constructs:

RULE $\frac{—:\mathrm{Int}\ ;}{\bullet_K}$

RULE $\left\langle \frac{X = I:Int}{I} \cdots \right\rangle_{\mathsf{k}} \left\langle \cdots X \mapsto N \cdots \right\rangle_{\mathsf{env}} \left\langle \cdots N \mapsto \frac{—}{I} \cdots \right\rangle_{\mathsf{store}}$ [assignment]

Sequential composition. Sequential composition has been defined as a whitespace-separated syntactic list of statements. Recall that syntactic lists are actually syntactic sugar for cons-lists. Therefore, the following two rules eventually sequentialize a syntactic list of statements "s1 s2 ... sn.. into the corresponding computation "s1 > s2 > ... > sn".

RULE $\frac{\bullet_{Stmts}}{\bullet_K}$

RULE $\frac{S\ Ss}{S \curvearrowright Ss}$ [structural]

Conditional statement.

RULE $\frac{\mathtt{if}\ (\mathsf{true}) S\ \mathtt{else}\ —}{S}$

RULE $\frac{\mathtt{if}\ (\mathsf{false}) —\ \mathtt{else}\ S}{S}$

While loop. The only thing to notice here is that the empty block has been replaced with the block holding the explicit empty sequence. That's because in the semantics all empty lists become explicit corresponding dots (to avoid parsing ambiguities)

RULE $\frac{\mathtt{while}\ (B) S}{\mathtt{if}\ (B)\{S\ \mathtt{while}\ (B) S\}\ \mathtt{else}\ \{\bullet_{Stmts}\}}$ [structural]

The new IMP++ constructs

We next discuss the semantics of the new IMP++ constructs.

Strings. First, we have to state that strings are also results. Second, we give the semantics of IMP++ string concatenation (which uses the already existing addition symbol `+` from IMP) by reduction to the built-in string concatenation operation.

SYNTAX *KResult* ::= *String*

RULE $\frac{Str1:String + Str2:String}{Str1 +_{String} Str2}$

Variable increment. Like variable lookup, this is also meant to be a supercool transition: we want it to count both in the non-determinism due to strict operations above it in the computation and in the non-determinism due to thread interleavings. This rule also relies on $\mathbb{K}$'s configuration abstraction. Without abstraction, you would have to also include the `thread` and `threads` cells.

RULE $\left\langle \frac{\mathtt{++}\ X}{I +_{Int} 1} \cdots \right\rangle_{\mathsf{k}} \left\langle \cdots X \mapsto N \cdots \right\rangle_{\mathsf{env}} \left\langle \cdots N \mapsto \frac{I}{I +_{Int} 1} \cdots \right\rangle_{\mathsf{store}}$ [increment]

Read. The `read()` construct evaluates to the first integer in the input buffer, which it consumes. Note that this rule is tagged `increment`. This is because we will include it in the set of potentially non-deterministic transitions when we kompile the definition; we want to do that because two or more threads can "compete" on reading the next integer from the input buffer, and different choices for the next transition can lead to different behaviors.

RULE $\left\langle \frac{\mathtt{read}\ ()}{I} \cdots \right\rangle_{\mathsf{k}} \left\langle \frac{I:Int}{\bullet_{List}} \cdots \right\rangle_{\mathsf{in}}$ [read]

Print. The `print` statement is strict, so all its arguments are eventually evaluated (recall that `print` is variadic). We append each of its evaluated arguments, in order, to the output buffer, and structurally discard the residual `print` statement with an empty list of arguments. We only want to allow printing integers and strings, so we define a *Printable* syntactic category including only these and define the `print` statement to only print *Printable* elements. Alternatively, we could have had two similar rules, one for integers and one for strings. Recall that, currently, $\mathbb{K}$'s lists are cons-lists, so we cannot simply rewrite the head of a list (P) into a list ($\bullet$). The first rule below is tagged, because we want to include it in the list of transitions when we kompile; different threads may compete on the output buffer and we want to capture all behaviors. The second rule is structural because we do not want it to count as a computational step.

SYNTAX *Printable* ::= *Int*
| *String*

SYNTAX *AExp* ::= *Printable*

RULE $\left\langle \mathtt{print}\ (\frac{P:Printable\ ,\ AEs}{AEs})\ ; \cdots \right\rangle_{\mathsf{k}} \left\langle \cdots \frac{\bullet_{List}}{P} \right\rangle_{\mathsf{out}}$ [print]

RULE $\frac{\mathtt{print}\ (\bullet_{AExps})\ ;}{\bullet_K}$ [structural]

Halt. The `halt` statement empties the computation, so the rewriting process simply terminates as if the program terminated normally. Interestingly, once we add threads to the language, the `halt` statement as defined below will terminate the current thread only. If you want an abrupt termination statement that halts the entire program, then you need to discard the entire contents of the `threads` cell, so the entire computation abruptly terminates the entire program, no matter how many concurrent threads it has, because there is nothing else to rewrite.

RULE $\left\langle \frac{\mathtt{halt}\ ; \curvearrowright —}{\bullet_K} \right\rangle_{\mathsf{k}}$

Spawn thread. A spawned thread is passed its parent's environment at creation time. The `spawn` expression in the parent thread is immediately replaced by the unique identifier of the newly created thread, so the parent thread can continue its execution. We only consider a sequentially consistent shared memory model for IMP++, but other memory models can also be defined in $\mathbb{K}$; see, for example, the definition of KERNELC. Note that the rule below does not need to be tagged in order to make it a transition when we kompile, because the creation of the thread itself does not interfere with the execution of other threads. Also, note that $\mathbb{K}$'s configuration abstraction is at heavy work here, in two different places. First, the parent thread's `k` and `env` cells are wrapped within a `thread` cell. Second, the child thread's `k`, `env` and `id` cells are also wrapped within a `thread` cell. Why that way and not putting all these four cells together within the same thread, or even create an additional `threads` cell at top holding a `thread` cell with the new `k`, `env` and `id`? Because in the original configuration we declared the multiplicity of the `thread` cell to be "$*$", which effectively tells the $\mathbb{K}$ tool that zero, one or more such cells can co-exist in a configuration at any moment. The other cells have the default multiplicity "one", so they are not allowed to multiply. Thus, the only way to complete the rule below in a way consistent with the declared configuration is to wrap the first two cells in a `thread` cell, and the latter two cells under the "$\bullet$" also in a `thread` cell. Once the rule applies, the spawning thread cell will add a new thread cell next to it, which is consistent with the declared configuration cell multiplicity. The unique identifier of the new thread is generated using the "fresh" side condition.

RULE $\left\langle \frac{\mathtt{spawn}\ S}{T} \cdots \right\rangle_{\mathsf{k}} \left\langle \rho \right\rangle_{\mathsf{env}} \frac{\bullet_{Bag}}{\left\langle \left\langle S \right\rangle_{\mathsf{k}} \left\langle \rho \right\rangle_{\mathsf{env}} \left\langle T \right\rangle_{\mathsf{id}} \cdots \right\rangle_{\mathsf{thread}}}$ requires `fresh` $(T:Int)$

Join thread. A thread who wants to join another thread `T` has to wait until the computation of `T` becomes empty. When that happens, the join statement is simply dissolved. The terminated thread is not removed, because we want to allow possible other join statements to also dissolve.

RULE $\left\langle \frac{\mathtt{join}\ (T)\ ;}{\bullet_K} \cdots \right\rangle_{\mathsf{k}} \left\langle \cdots \left\langle \bullet_K \right\rangle_{\mathsf{k}} \left\langle T \right\rangle_{\mathsf{id}} \cdots \right\rangle_{\mathsf{thread}}$

Blocks. The body statement of a block is executed normally, making sure that the environment at the block entry point is saved in the computation, in order to be recovered after the block body statement. This step is necessary because blocks can declare new variables having the same name as variables which already exist in the environment, and our semantics of variable declarations is to update the environment map in the declared variable with a fresh location. Thus, variables which are shadowed lose their original binding, which is why we take a snapshot of the environment at block entrance and place it after the block body (see the semantics of environment recovery at the end of this module). Note that any store updates through variables which are not declared locally are kept at the end of the block, since the store is not saved/restored. An alternative to this environment save/restore approach is to actually maintain a stack of environments and to push a new layer at block entrance and pop it at block exit. The variable lookup/assign/increment operations then also need to change, so we do not prefer that non-modular approach. Compilers solve this problem by statically renaming all local variables into fresh ones, to completely eliminate shadowing and thus environment saving/restoring. The rule below can be structural, because what it effectively does is to take a snapshot of the current environment; this operation is arguably not a computational step.

RULE $\left\langle \frac{\{Ss\}}{Ss \curvearrowright \mathtt{env}\ (\rho)} \cdots \right\rangle_{\mathsf{k}} \left\langle \rho \right\rangle_{\mathsf{env}}$ [structural]

Variable declaration. We allocate a fresh location for each newly declared variable and initialize it with 0.

RULE $\left\langle \mathtt{int}\ \frac{X:Id\ ,\ Xs}{Xs}\ ; \cdots \right\rangle_{\mathsf{k}} \left\langle \frac{\rho}{\rho[N\ /\ X]} \right\rangle_{\mathsf{env}} \left\langle \cdots \frac{\bullet_{Map}}{N \mapsto 0} \cdots \right\rangle_{\mathsf{store}}$ requires `fresh` $(N:Nat)$

RULE $\frac{\mathtt{int}\ \bullet_{Ids}\ ;}{\bullet_K}$ [structural]

Auxiliary operations. We only have one auxiliary operation in IMP++, the environment recovery. Its role is to discard the current environment in the `env` cell and replace it with the environment that it holds. This rule is structural: we do not want them to count as computational steps in the transition system of a program.

SYNTAX K ::= `env` (*Map*)

RULE $\left\langle \frac{\mathtt{env}\ (\rho)}{\bullet_K} \cdots \right\rangle_{\mathsf{k}} \left\langle \frac{—}{\rho} \right\rangle_{\mathsf{env}}$ [structural]

If you want to avoid useless environment recovery steps and keep the size of the computation structure smaller, then you can also add the rule

```
rule (env(_) => .) ~> env(_)  [structural]
```

This rule acts like a "tail recursion" optimization, but for blocks.

END MODULE

On Kompilation Options

We are done with the IMP++ semantics. The next step is to kompile the definition using the `kompile` tool, this way generating a language model. Depending upon for what you want to use the generated language model, you may need to kompile the definition using various options. We here discuss these options.

To tell the $\mathbb{K}$ tool to exhaustively explore all the behaviors due to the non-determinism of addition, division, and threads, we have to kompile with the command:

```
kompile imp.k --superheat="addition division" --supercool="lookup increment"
              --transition="lookup assignment increment read print"
```

As already mentioned, the syntax and rule tags play no theoretical or foundational role in $\mathbb{K}$. They are only a means to allow `kompile` to refer to them in its options, like we did above. By default, `kompile`'s options are all empty, because this yields the fastest language model when executed. Nonempty options may slow down the execution, but they instrument the language model to allow for formal analysis of program behaviors, even for exhaustive analysis.

Theoretically, the heating/cooling rules in $\mathbb{K}$ are fully reversible and unconstrained by side conditions as we showed in the semantics of IMP. For example, the theoretical heating/cooling rules corresponding to the `strict` attribute of division are the following:

$$\begin{array}{l} E_1/E_2 \;\Rightarrow\; E_1 \curvearrowright \square/E_2 \\ E_1 \curvearrowright \square/E_2 \;\Rightarrow\; E_1/E_2 \\ E_1/E_2 \;\Rightarrow\; E_2 \curvearrowright E_1/\square \\ E_2 \curvearrowright E_1/\square \;\Rightarrow\; E_1/E_2 \end{array}$$

The other semantic rules apply *modulo* such structural rules. For example, using heating rules we can bring a redex (a subterm which can be reduced with semantic rules) to the front of the computation, then reduce it, then use cooling rules to reconstruct a term over the original syntax of the language, then heat again and non-deterministically pick another redex, and so on and so forth without losing any opportunities to apply semantic rules. Nevertheless, these unrestricted heating/cooling rules may create an immense, often unfeasibly large space of possibilities to analyze. Super-heating/cooling implement an optimization which works well with other implementation choices made in the current $\mathbb{K}$ tool. Recall from the detailed description of the IMP language semantics that (theoretical) reversible rules like above are restricted to complementary conditional rules of the form

$$\begin{array}{l} E_1/E_2 \;\Rightarrow\; E_1 \curvearrowright \square/E_2 \ \text{ if } E_1 \notin \mathit{KResult} \\ E_1 \curvearrowright \square/E_2 \;\Rightarrow\; E_1/E_2 \ \text{ if } E_1 \in \mathit{KResult} \\ E_1/E_2 \;\Rightarrow\; E_2 \curvearrowright E_1/\square \ \text{ if } E_2 \notin \mathit{KResult} \\ E_2 \curvearrowright E_1/\square \;\Rightarrow\; E_1/E_2 \ \text{ if } E_2 \in \mathit{KResult} \end{array}$$

Therefore, our tool eagerly heats and lazily cools the computation. In other words, heating rules apply until a redex gets placed on the top of the computation, then some semantic rule applies and rewrites that into a result, then a cooling rule is applied to plug the obtained result back into its context, then another argument may be chosen and completely heated, and so on. This leads to efficient execution, but it may and typically does hide program behaviors. Super-heating/cooling allows you to interfere with this process. More precisely, whenever a rule tag is included in the `supercool` category, the $\mathbb{K}$ tool will continue to apply (unrestricted) cooling rules on the resulting computation disregarding the membership to *KResult* side condition, thus plugging everything back into its context, until no construct tagged `superheat` is left uncooled. This way, the heating rules now have the possibility to pick another evaluation order for the cooled fragment of computation. This way, we can think of super-heating/cooling as marking fragments of computation in which exhaustive analysis of the evaluation order is performed. Used carefully, this mechanism allows us to explore more non-deterministic behaviors of a program, even all of them like here, still efficiently. For example, with the semantics of IMP++ given below, the `krun` command with the `--search` option detects all five behaviors of the following IMP++ program (`x` can be 0, 1, 2, 3, or undefined due to division-by-zero):

```
  int x,y;
  x = 1;
  y = ++x / (++x / x);
```

Besides non-determinism due to underspecified argument evaluation orders, which the current $\mathbb{K}$ tool addresses by means of superheating and supercooling as explained above, there is another important source of non-determinism in programming languages: non-determinism due to concurrency/parallelism. For example, when two or more threads are about to access the same location in the store and at least one of these accesses is a write (i.e., an instance of the variable assignment rule), there is a high chance that different choices for the next transition lead to different program behaviors. While in the theory of $\mathbb{K}$ all the non-structural rules count as computational steps and hereby as transitions in the transition system associated to the program, in practice that may yield a tremendous number of step interleavings to consider. Most of these interleavings are behaviorally equivalent for most purposes. For example, the fact that a thread computes a step 8+3 ⇒ 11 is likely irrelevant for the other threads, so one may not want to consider it as an observable transition in the space of interleavings. Since the K tool cannot know without help which transitions need to be explored and which do not, our approach is to let the user say so explicitly using the `transition` option of `kompile`.