KErl: Executable semantics for Erlang

Judit Kőszegi
koszegijudit@elte.hu

ELTE Eötvös Loránd University
Faculty of Informatics
Budapest, Hungary

Abstract

This paper presents KErl, the first executable semantics for a substantial subset of Erlang, based on the official documentation and the open source implementation of the language. The formal language definition is developed bearing in mind the possibility of further extensions. KErl has been tested against our comprehensive test suite developed alongside the project, following the test driven development methodology. The semantics is defined in a framework which supports deductive verification of program properties. As a demonstration of the application of the semantics, we present the proof of a non-trivial property of a recursive program.

1 Introduction

Erlang [CT09] is an impure functional programming language with built-in support for concurrency, distribution and fault tolerance. As the language is used for implementing many widely used systems and applications, there is a need for not only testing, but formally reasoning about Erlang programs. To be able to formally verify program properties it is crucial to have the formal semantics of the language and a proof system into which we can embed the semantics.

Traditional program verifiers are based on either Hoare logic, separation logic or dynamic logic. Semantics definitions in these logics are rather complex and their implementations provide no practical way for the validation of the semantics itself. The recently introduced matching logic [RŞCM13, Roş15] merges the advantages of the verification focused semantics definition methods while it is more intuitive as it closely relates to how the program is executed (similarly to operational semantics). Its implementation, the $\mathbb{K}$ semantics framework [RŞ10] supports formal definition of languages, as well as concrete or symbolic execution, model checking and formal verification of programs.

Erlang does not have any official formal specification, the semantics of Erlang is only implicitly defined by its reference implementation [erlb] and we can find the informal description of language elements in the Erlang Reference Manual User's Guide [erla]. In the related work, there exists an almost complete small-step operational semantics defined by Fredlund [Fre01], improved and extended by Claessen, Svensson and Earle [Cla05, SF07, SFBE10]. This semantic definition is not executable, cannot be tested, so the semantic rules can contain bugs which could mislead the program verification. Moreover, for verification purposes we have to encode the semantic rules and used together with the quite complex modal μ-calculus. The last published version of this Erlang semantics is from 2010, so far the language has been evolved, the semantics of some language

In: E. Vatai (ed.): Proceedings of the 11th Joint Conference on Mathematics and Computer Science, Eger, Hungary, 20th – 22nd of May, 2016, published at http://ceur-ws.org

constructs have modified and also some new features have been added. Due to these reasons we decided to redefine the semantics of the current version (19.0) of Erlang with matching logic in the $\mathbb{K}$ framework, which proves itself to be an all-in-one solution: after specifying the formal semantics of the language, we get an interpreter derived from the semantics, and the formal analysis and verification methods and toolset "for free". The starting point for our definition is the definition composed by Fredlund, but the operational semantic rules have to be rephrased in matching logic, and we also extend it by adding language features missing from the original definition. This paper introduces the challenges in defining matching logic semantics of Erlang, and shows a practical application: we use the semantics for symbolic program verification. In a previous paper [HKT16] we have already shown how to prove correctness of refactoring transformations defined for Erlang programs using matching logic semantics. We focused on the specification language and the verification method of refactorings, so we have only defined a small subset of the formal semantics of Erlang as a proof of concept. Since then, we have modified, refined and extended the formal language definition of Erlang in the $\mathbb{K}$ framework.

The main contribution of this paper is KErl, the first executable semantics of a sequential subset of Erlang. Despite it is not a complete language definition, it is developed bearing in mind the further language improvement possibilities when choosing the semantic framework and defining the semantic rules, thus making the semantics modular and easily extensible. Alongside defining the semantics of Erlang a test suit has been developed for verifying the semantic rules and their combinations. This test suit is applicable for verifying other Erlang implementations as well as formal semantic definitions. At last but not least, we demonstrate the practical relevance of KErl by using it for reasoning about Erlang program properties.

2 Background: Matching Logic and $\mathbb{K}$ Framework

Matching logic is designed by the motivation of having the possibility to express the formal semantics of any programming language and at the same time to state program properties in the same logic and reason about them using the semantic rules unchanged. Its sound and relatively complete proof system [RŞCM13] is language independent: the inputs of a proof are the formulas expressing the language semantics and a program with the property to be verified.

Matching logic is a specialized many-sorted first-order logic, where we always have a distinguished sort `Cfg`, called *configuration.* It plays an important role either expressing formal semantic rules or program properties as it contains the program code together with various semantic data in a nested structure of labelled cells. Concrete program states thus represented as ground terms of sort `Cfg`, while program state specifications are configuration terms with variables and constraints over them, called *patterns.* Intuitively, patterns behave as predicates in a formula: a pattern satisfied those configuration that match it.

With a matching logic formula we can express only static properties of programs. For defining dynamic behaviour or a formal semantic rule we need to compose a *reachability formula* from two matching logic formulas. The reachability rule $\varphi \Rightarrow \varphi'$ states that a configuration matching φ will advance into a configuration matching φ' (with the same variable substitution).

The $\mathbb{K}$ *framework* is an implementation of matching logic. While in a matching logic formula it is allowed to have more than one configuration term, in $\mathbb{K}$ it is restricted: one can only write formulas containing exactly one term of sort `Cfg`. This way we cannot implement e.g. classical small-step operational semantics in the framework, but in turn the developers of $\mathbb{K}$ could focus on yielding efficient interpreters, in addition to generating rewrite systems as needed for semi-automatic matching logic verification.

For the *matching logic semantics* of a language we need to define the semantic domain and a set of reachability rules capturing the operational semantics of the language. In practice, using the $\mathbb{K}$ framework, we need to (1) give the syntax of the programming language in a BNF-like notation annotated with semantic attributes; (2) specify the pattern of program states as configuration represented by labeled, possibly nested multisets (called cells); and (3) define the set of rewrite rules (reachability formulas) over configurations. The framework offers features that improve the modularity and enhance the readability of the semantic definition compared with matching logic. Further details can be found as we describe the semantics of Erlang.

3 KErl: Matching Logic Semantics of Erlang in $\mathbb{K}$

In this paper, we present the formal semantics of a sequential, side-effect free subset of Erlang. A regular Erlang program is composed from modules; modules contain forms defining various program entities like records or functions; and forms consist of series of expressions. Expressions are eagerly evaluated, strongly but dynamically typed.

At the current stage, our goal is not to have a complete language definition, rather to explore the benefits and challenges of the new semantic definition style for defining the semantics of Erlang. Besides, we aimed to have a sub-language in which we are able to write meaningful and complex enough Erlang programs and prove program properties with the proof system of reachability logic. As future work, we plan to make our semantic definition complete. Due to the modular nature of the framework we use, new data types or language constructs can be easily added to the semantic definition without having to significantly modify the existing rules.

In our current semantic definition we omit modules and module attributes, thus an Erlang program here is a series of function declarations. Unlike in case of C or Java, a regular Erlang program does not have any specified entry point, one can execute any exported function of any module. For the sake of simplicity, following the idea of the main function of other languages, here we require to define a function named `main` with no parameters. When we evaluate a program using our semantic rules, after preprocessing the function definitions, we start to execute this function.

An example KErl program that calculates the sum of the first `10` natural numbers:

```
sum(0, S) -> S;
sum(X, S) -> sum(X-1, S+X).

main() -> sum(10, 0).
```

3.1 Syntax

As mentioned, the syntax of a language can be defined with a BNF-like notation annotated with various attributes. Using attributes we can control e.g. the priority and associativity of operations or the evaluation order of expressions and its sub-expressions.

Erlang is an eagerly evaluated language, so all sub-expressions are evaluated before an expression itself is evaluated (unless in case of sort-circuit logic operators). The evaluation order among the sub-expressions is unspecified in Erlang which can results in nondeterministic behaviour. The strictness and context attributes make sense together with the corresponding semantic rules, so we get into the details in the section about the semantics of expressions (Section 3.6).

3.2 The Program Configuration

The program configuration is an essential part of the language definition: it determines the granularity of the semantics as it stores all information that we consider relevant for the execution of programs. It is an algebraic data structure: bag of labeled, nested cells. For our sub-language, we use the following configuration:

$$\langle\langle \$PGM{:}Pgm \curvearrowright \texttt{main}(\cdot_{Exps})\rangle_{\mathsf{k}}\ \langle\cdot_{Map}\rangle_{\mathsf{defs}}\ \langle\cdot_{Map}\rangle_{\mathsf{funvars}}\rangle_{\mathsf{cfg}}$$

where $\langle\cdots\rangle_{\mathsf{cfg}}$ is a top-level container cell. Following the conventions of $\mathbb{K}$, the $\langle\cdots\rangle_{\mathsf{k}}$ cell contains the code to be executed. In the framework this cell has special roles: a) the program to be executed initially is loaded into the $\$PGM$ variable inside the cell (followed by a call of the `main` function in our case), and b) if we do not mention the name of the cell in a semantic rule, it is automatically applied on the `k` cell. In the $\langle\cdots\rangle_{\mathsf{defs}}$ cell we store the function definitions, while we use the $\langle\cdots\rangle_{\mathsf{funvars}}$ cell when evaluating anonymous functions with internal names: we assign variables to anonymous function definitions (Section 3.6.4). The following is a concrete configuration, the initial configuration of the above shown `sum` program:

$$\langle\langle \texttt{sum}\,(0,\texttt{S})\ \texttt{->}\ \texttt{S};\ \ \texttt{sum}\,(\texttt{X},\texttt{S})\ \texttt{->}\ \texttt{sum}\,(\texttt{S}\,-1,\texttt{S}+\texttt{X}).\ \ \texttt{main}\,()\ \texttt{->}\ \texttt{sum}\,(10,0).\rangle_{\mathsf{k}}\ \langle\cdot_{Map}\rangle_{\mathsf{defs}}\ \langle\cdot_{Map}\rangle_{\mathsf{funvars}}\rangle_{\mathsf{cfg}}$$

While a pattern that is satisfied by the above concrete configuration would be:
$\langle\langle\cdots\ \texttt{main}\,()\ \texttt{->}\ F(P).\rangle_{\mathsf{k}}\ \langle D\rangle_{\mathsf{defs}}\ \langle V\rangle_{\mathsf{funvars}}\rangle_{\mathsf{cfg}} \quad \wedge\ \texttt{length}\,(P) = 2$
Note that `X` and `S` are Erlang program variables represented as constants in matching logic, whereas F, P, D and V are mathematical variables. The ellipsis (three dots) matches anything, and $\cdot_{Map}$ is the empty map.

For implementing the complete Erlang semantics the configuration should be extended by other cells e.g. for storing module information, for tracking side-effects, or for processes and their related message queues. Due to the modular nature of our language definition adding a new element to the configuration has no effect on most of the semantic rules.

3.3 Preprocessing Function Declarations

A KErl program is a series of function definitions, so as the first step of evaluation we preprocess the given program by moving the function definitions from the default `k` cell to the `defs` cell. This cell stores function definitions in a map data structure: function names are assigned to the function bodies (list of match clauses). In the semantics we have different rules for syntactically different function clauses (the last clause is terminated by a dot, while the others have a semicolon at their end); and we also distinguish cases when the first clause of the function is found (the map of the `defs` cell does not contain the name of the current function yet), and when we have already stored a clause/clauses of the same function.

In the following, we show one of these semantic rules. Note that the notation of the $\mathbb{K}$ rules slightly differs from that of reachability formulas as it hides the irrelevant parts and also it groups the rewrites by cells making the semantic rules more compact and readable. Horizontal lines represent reductions, the "$\cdots$" match portions of cells that are irrelevant and thus unchanged. The rule of Figure 1 says that if the next thing to be evaluated is a function clause terminated by a dot and in the map of the `defs` cell we can find a key-value pair with the current function name *Name* in the key, then the application of the rule removes the processed clause from the `k` cell by replacing it with the empty computation unit $\cdot_K$; and adds it to the list of the match clauses assigned to the function name. Meanwhile, the parameter list of the function *Args* is transformed to the tuple $\{Args\}$ in order to simplify future usage for the semantics of function calls (see Section 3.6.4).

$$\left\langle \frac{Name(Args)\ \texttt{->}\ Body\ \texttt{.}}{\cdot_K} \cdots \right\rangle_{\mathsf{k}} \left\langle \cdots\ Name \mapsto \frac{Clauses}{Clauses\,;\,\{Args\}\ \texttt{->}\ Body} \cdots \right\rangle_{\mathsf{defs}}$$

Figure 1: A rule for function declaration

After this phase, the `k` cell contains only the call of the `main` function. For our example program, we get the following configuration after preprocessing:

$$\langle\langle \texttt{main}\ ().\rangle_{\mathsf{k}}\ \ \langle \texttt{main} \mapsto\ \ \{\}\ \texttt{->sum}\ (10,0),\ \ \texttt{sum} \mapsto\ \ \{0,\texttt{S}\}\ \texttt{->S};\ \{\texttt{X},\texttt{S}\} \rightarrow\texttt{sum}\ (\texttt{S}\ -1, \texttt{S} + \texttt{X})\rangle_{\mathsf{defs}}\ \ \langle \cdot_{Map}\rangle_{\mathsf{funvars}}\rangle_{\mathsf{cfg}}$$

3.4 Matching and Substitution

As Erlang is single-assignment, we do not need to store variable environment in the configuration, because once a variable is assigned to a value, we can simply replace all corresponding occurrences of the variable by its value. However, the language has some tricky scoping rules which could be challenging while implementing the semantics. For example, variables can be shadowed inside anonymous functions or list comprehensions, but unexpectedly, an Erlang begin-end block does not open new variable scope. $\mathbb{K}$ has built-in capabilities for matching and substitution, but it cannot handle these special cases, so we have implemented own functions for matching and variable substitution.

Functions in $\mathbb{K}$ are also expressed by reachability rules, but we can use the special `owise` tag for a rule allowing the system to only apply it when none of the other rules of the function matches. Note that if more than one untagged rules matches at the same time, the system choose one of them non-deterministically.

Pattern Matching

In Erlang, variables are bound to values through the *pattern matching* mechanism. Pattern matching can happen in different expressions, but it has a general mechanism which can be implemented regardless of type of the expression. The `getMatching` function has two parameters: a pattern and a value. Intuitively, the matching is successful if we can find a substitution of values for the variables in the pattern such that the pattern and the value becomes identical. If the pattern matches the value, the `getMatching` function returns with a map containing this substitution (variable-value pairs), otherwise with a map containing the special `badmatch` key. This way, unlike the built-in matching mechanism of the framework, we do not have to check and calculate the matching separately.

Unlike in case of classical functional languages (e.g. Haskell), it is allowed to use *non-linear patterns* in Erlang, i.e. patterns that may contain the same variable name multiple times – every variable occurrence should matches to the same value. For example, we can have a three-parameter function with the formal parameter list `(X,Y,X)`.

The actual parameters `(3,2,3)` matches to the formal parameters, while `(3,3,2)` does not match. Technically, we need a helper function for `getMatching` with an extra accumulator parameter initialized to an empty map, used for storing the substitution. The calculation of the matching is recursive, with the following *base cases* for successful matching:

- A basic value (atom or integer) matches to a basic value if the two values are exactly the same. The result is the accumulator map.
- Any value matches to a variable if
    a) the accumulator map does not contain the variable as key. The result is the map extended with the new variable - value assignment.
    b) the accumulator map contains the variable as key and it assigned to the same value. The result is the map.
- Any value matches to the underscore ("do not care" variable). The result is the accumulator map.

Lists or tuples can be matched *recursively* by matching their first elements using the recursively calculated map resulted from the matching of the rest of elements.

Note that if we extend the semantics with a new data type (either basic or compound), we should not forget to extend the rules of our matching function.

Variable Substitution

Once we successfully matched a value against a pattern, we get new variable assignments as result. The *substitution* function (`subst`) helps us to apply these assignments for any kind of Erlang expression. The function has two parameters: an expression and a map containing the substitution; as result we get back the expression in which the variables are substituted with their assigned values considering the scoping rules. The recursive substitution function has the following *base cases*:

- Any substitution applied to a basic value leaves it unchanged.
- Applying a substitution to a variable
    a) leaves it unchanged if the map does not contain the variable as key.
    b) results the value assigned to the variable, if the map contains the variable as key.
- Any substitution applied to an underscore leaves it unchanged.

For *compound expressions* that do not open new scope we can simply define the substitution by applying the same substitution on all of its sub-expression. Some example rules is shown in Figure 2.

$$\frac{\texttt{subst}\ (\mathit{E1}\ \texttt{+}\ \mathit{E2}, \mathit{Map})}{\texttt{subst}\ (\mathit{E1}, \mathit{Map})\ \texttt{+}\ \texttt{subst}\ (\mathit{E2}, \mathit{Map})} \qquad \frac{\texttt{subst}\ (\ \texttt{case}\ E\ \texttt{of}\ M\ \texttt{end}, \mathit{Map})}{\texttt{case}\ \texttt{subst}\ (E, \mathit{Map})\ \texttt{of}\ \texttt{subst}\ (M, \mathit{Map})\ \texttt{end}}$$

Figure 2: Rules for variable substitution (in all sub-expression)

In case of *fun expression* (anonymous function) variables in the arguments shadow variables of the outer variable scope, thus before we apply the substitution to a clause of the anonymous function, we remove key-value pairs from the map whose key is among the variables of the arguments. The rules in Figure 3 define the substitution for fun expressions.

$$\frac{\texttt{subst}\ (\ \texttt{fun}\ \mathit{Cls}\ \texttt{end}, \mathit{Map})}{\texttt{fun}\ \texttt{substCls}\ (\mathit{Cls}, \mathit{Map})\ \texttt{end}} \qquad \frac{\texttt{substCls}\ (((\mathit{Args})\ \texttt{->}\ E\ \texttt{;}\ \mathit{Cls}), \mathit{Map})}{(\mathit{Args})\ \texttt{->}\ \texttt{substCl}\ (\mathit{Args}, E, \mathit{Map})\texttt{;}\ \texttt{substCls}\ (\mathit{Cls}, \mathit{Map})}$$

$$\frac{\texttt{substCl}\ (\mathit{Args}, \mathit{Exp}, \mathit{Map})}{\texttt{subst}\ (\mathit{Exp},\ \texttt{removeAll}\ (\mathit{Map},\ \texttt{vars}\ (\mathit{Args})))} \qquad \frac{\texttt{substCls}\ ((\mathit{Args})\ \texttt{->}\ E, \mathit{Map})}{(\mathit{Args})\ \texttt{->}\ \texttt{substCl}\ (\mathit{Args}, E, \mathit{Map})}$$

Figure 3: Rules for variable substitution in fun expression

It is possible to define *recursive fun expression* by "naming" the fun with a variable in its head. This variable also shadows the outer variables, so we defined some additional rules that handle this different syntactic occurrence and removing not only the formal arguments, but also this special variable from the enclosing variable environment (Figure 4).

$$\frac{\texttt{substCls}\ ((\mathit{Var}(\mathit{Args})\ \texttt{->}\ E\,\texttt{;}\ \mathit{Cls}), \mathit{Map})}{\mathit{Var}(\mathit{Args})\ \texttt{->}\ \texttt{substCl2}\ (\mathit{Args}, \mathit{Var}, E, \mathit{Map})\texttt{;}\ \texttt{substCls}\ (\mathit{Cls}, \mathit{Map})}$$

$$\frac{\texttt{substCls}\ (\mathit{Var}(\mathit{Args})\ \texttt{->}\ E, \mathit{Map})}{\mathit{Var}(\mathit{Args})\ \texttt{->}\ \texttt{substCl2}\ (\mathit{Args}, \mathit{Var}, E, \mathit{Map})}$$

$$\frac{\texttt{substCl2}\ (\mathit{Args}, \mathit{Var}, \mathit{Exp}, \mathit{Map})}{\texttt{subst}\ (\mathit{Exp},\ \texttt{removeAll}\ (\mathit{Map},\ \texttt{vars}\ (\mathit{Args})\ \ \texttt{SetItem}\ (\mathit{Var})))}$$

Figure 4: Rules for variable substitution in fun expression with internal name

Another language construct that opens new scope is the *list comprehension*, where variables in the generator patterns shadow the variables of the outer scope. For the head expression (denoted by `E` in the first rule of Figure 5) it is similar to the body of the fun expression clause: before applying the substitution we remove key-value pairs where the key is a variable bounded by any of the generators in the list comprehension. The generators and filters should be processed from right to left, because the shadowing of a generator is valid on its left hand side. The second rule of Figure 5 shows the substitution in a generator: the pattern of the generator remains unchanged as all the variables of the pattern is either a new variable of the scope or shadows an outer variable; the current substitution is applied to the list of the generator; then the substitution mechanism continues on the remaining generators and filters. The third rule matches to the empty generator-filter list, while the last one (tagged with `owise`) is applied when the first element of a non-empty generator-filter list is a filter: the substitution is applied to the filter, then is continued on the rest of the generator-filter list.

$$\frac{\texttt{subst}\ ([E\ \texttt{||}\ \mathit{GFs}], \mathit{Map})}{[\,\texttt{subst}\ (E,\ \texttt{removeAll}\ (\mathit{Map},\ \texttt{vars}\ (\mathit{GFs})))\ \texttt{||}\ \texttt{substGFs}\ (\mathit{GFs}, \mathit{Map})]}$$

$$\frac{\texttt{substGFs}\ ((P\ \texttt{<-}\ L\texttt{,}\ \mathit{GFs}), \mathit{Map})}{P\ \texttt{<-}\ \texttt{subst}\ (L, \mathit{Map})\texttt{,}\ \texttt{substGFs}\ (\mathit{GFs},\ \texttt{removeAll}\ (\mathit{Map},\ \texttt{vars}\ (P)))}$$

$$\frac{\texttt{substGFs}\ ((\cdot_{GFs}), \_)}{(\cdot_{GFs})}$$

$$\frac{\texttt{substGFs}\ ((F\texttt{,}\ \mathit{GFs}), \mathit{Map})}{\texttt{subst}\ (F, \mathit{Map})\texttt{,}\ \texttt{substGFs}\ (\mathit{GFs}, \mathit{Map})}\quad [\text{owise}]$$

Figure 5: Rules for variable substitution in list comprehension

Note that if we extend the semantics with new language construct, we should not forget to extend the rules of the substitution function.

3.5 Data Types

For our sub-language definition we chose those Erlang data types that are essential and most commonly used in Erlang programs.

3.5.1 Primitive Types

In KErl we can use two primitive types: *integer number* (with the conventional notation) and the Erlang-specific *atom* type. An atom is a literal, a constant with name. There is no boolean data type in Erlang, instead the atoms `true` and `false` are used to denote boolean values. In the syntax we have to explicitly define these literals, because they used in semantic rules of expressions connected with logical operators.

3.5.2 Compound Types

A *tuple* is a compound, heterogeneous data with fixed number of elements enclosed by braces.

SYNTAX $\mathit{TupleExp} ::= \{\mathit{Exps}\}$ [strict]

SYNTAX $\mathit{Exps} ::= \mathit{List}\{\mathit{Exp}, \text{“,”}\}$ [strict]

Unlike traditional, statically typed functional languages, an Erlang *list* is also a heterogeneous data structure. Syntactically, we can construct lists from comma-separated items enclosed by square brackets, or with the head-tail notation as shown in the following syntax rule:

SYNTAX $\mathit{ListExp} ::= [\mathit{Exps}]$ [strict]
$\qquad\qquad\qquad\mid\ [\mathit{Exps} \mid \mathit{Exp}]$ [seqstrict]

In the $\mathbb{K}$ framework rules tagged with `macro` label applied everywhere it matches before the application of any other non-macro semantic rules. With the following rules we can transform every list to its head-tail normal form easing the definition of list-related semantic rules, as we only have to deal with one kind of syntactic occurrence of lists.

$\dfrac{[\mathit{Es}{:}\mathit{Exps}]}{[\mathit{Es} \mid [\cdot_{\mathit{Exps}}]]} \wedge \mathit{Es} \neq_K \cdot_{\mathit{Exps}}$ [macro] $\qquad \dfrac{[E{:}\mathit{Exp}\,\mathbf{,}\ \mathit{Es} \mid T]}{[E \mid [\mathit{Es} \mid T]]} \wedge \mathit{Es} \neq_K \cdot_{\mathit{Exps}}$ [macro]

Figure 6: Rules for transforming lists to their normal form

3.5.3 Functions

Functional objects in Erlang called *funs*. Funs make it possible to create an anonymous function and can be used everywhere, where other types of data, e.g. it can be assigned to a variable or passed as argument of an other function.

SYNTAX $\mathit{FunExp} ::=$ `fun` $\mathit{Clauses}$ `end`

3.6 Semantics of Expressions

This section gives an overview of the executable semantics of Erlang expressions and explain some selected semantic rules more detailed. The semantics of each non-trivial language construct is defined by several rules. We can distinguish two type of rules: a *structural* rule rearranges the current program state so that other rules can be applied; while a *computational* rule captures the intuition of a computational step in the execution. We recall that rules without cell context information are applied, by default, on the content of the `k` cell. This cell, by convention, represents a stack of computations waiting to be performed. Initially, the program to be executed is loaded into the `k` cell, and the whole program is in the top-level. Later on, we can split and sequentialize the computations with the $\curvearrowright$ sequential composition (“followed by”) operator, which is automatically added to any language defined in the framework.

3.6.1 Arithmetic and Logic Expressions

Conventional *arithmetic operations* with integers like addition, subtraction, multiplication, integer division or remainder can be implemented by built-in operators of the framework, like $+_{\mathit{Int}}$ or $\%_{\mathit{Int}}$ in the example rules of Figure 7.

$\dfrac{\mathit{I1} + \mathit{I2}}{\mathit{I1} +_{\mathit{Int}} \mathit{I2}} \qquad \dfrac{\mathit{I1}\ \texttt{rem}\ \mathit{I2}}{\mathit{I1}\ \%_{\mathit{Int}}\ \mathit{I2}} \wedge \mathit{I2} =/=_{\mathit{Int}} 0$

Figure 7: Rules for some arithmetic expression

As mentioned, evaluation order of sub-expressions can be defined by strictness attributes of the corresponding syntax rule. For addition, we have the following syntax:

SYNTAX $\mathit{Exp} ::= \mathit{Exp}$ `+` Exp [strict]

meaning that the two operands of the addition will be evaluated (in a non-deterministic order) to values before we apply the semantic rule of addition. In general, `strict` attribute causes the generation of rules that bring the non-terminals of the annotated syntax to the top of computation, and plug them back when they are evaluated to a final value. Note that in the syntax we have to define what kind of terms count as (terminal) value in our language.

In Erlang, each type of data can be checked for *equality* or compared by *comparison* operators with any other type of data, so in the semantics we have to take care every combination of the operands of the binary equality and comparison operators. Also, in case of a future extension of data types, we should not forget to extend the set of semantic rules of these operators. As each comparison operator can be expressed using the $<$ operator and the logical `not` operator, we only define computational rules for $<$ and structural rules for the others ($>$, $=<$, $>=$). Tuples and lists are compared by their length, in case of same length their elements are compared right-to-left. A fragment of rules is shown in Figure 8.

$$\frac{(\mathit{E1}:\mathit{Value})\ \texttt{>=}\ (\mathit{E2}:\mathit{Value})}{\texttt{not}\ (\mathit{E1}\ \texttt{<}\ \mathit{E2})} \qquad \frac{(\mathit{E1}:\mathit{Int})\ \texttt{<}\ (\mathit{E2}:\mathit{Int})}{(\mathit{E1} <_{\mathit{Int}} \mathit{E2})} \qquad \frac{(\mathit{E1}:\mathit{Int})\ \texttt{<}\ (\mathit{E2}:\mathit{Atom})}{\mathsf{true}}$$

$$\frac{\{X\ \texttt{,}\ \mathit{Xs}\}\ \texttt{<}\ \{Y\ \texttt{,}\ \mathit{Ys}\}}{\texttt{length}\ (\mathit{Xs}) <_{\mathit{Int}} \texttt{length}\ (\mathit{Ys})} \quad \wedge\ \texttt{length}\ (\mathit{Xs}) =/=_{\mathit{Int}} \texttt{length}\ (\mathit{Ys})$$

$$\frac{\{X\ \texttt{,}\ \mathit{Xs}\}\ \texttt{<}\ \{Y\ \texttt{,}\ \mathit{Ys}\}}{(X\ \texttt{<}\ Y)\ \texttt{orelse}\ (X\ \texttt{==}\ Y\ \texttt{andalso}\ \{\mathit{Xs}\}\ \texttt{<}\ \{\mathit{Ys}\})} \quad \wedge\ \texttt{length}\ (\mathit{Xs}) ==_{\mathit{Int}} \texttt{length}\ (\mathit{Ys})$$

Figure 8: Rules for comparison operators

We have two *sort-circuit binary logic operators*: `andalso` and `orelse`. The second operand is evaluated only if necessary. That is, the second operand is evaluated only if the first evaluates to false in an `orelse` expression or to true in an `andalso` expression. Returns either the value of first expression (that is, `true` or `false`) or the value of the second - if it is evaluated. This behaviour is unusual, because the evaluated value can be different from `true` or `false` atoms. The rules of Figure 9 defines formally the semantics specified informally above.

$$\frac{\mathsf{true}\ \texttt{andalso}\ B}{B} \qquad \frac{\mathsf{true}\ \texttt{orelse}\ \_}{\mathsf{true}}$$

$$\frac{\mathsf{false}\ \texttt{andalso}\ \_}{\mathsf{false}} \qquad \frac{\mathsf{false}\ \texttt{orelse}\ B}{B}$$

Figure 9: Rules for sort-circuit logic operators

3.6.2 Branching Expressions

The most commonly-used branching expression is the *case expression*, whose syntax is defined as follows (`GuardSeq` is explained later):

SYNTAX $\mathit{CaseExp} ::=$ `case` Exp `of` Match `end` [strict(1)]

SYNTAX $\mathit{Match1} ::= \mathit{Exp}$ `->` Exp

$\quad\mid \mathit{Exp}$ `when` $\mathit{GuardSeq}$ `->` Exp

SYNTAX $\mathit{Match} ::= \mathit{Match1}$

$\quad\mid \mathit{Match}\ \mathit{Match}$

where `strict(1)` attribute means that the first non-terminal is evaluated to a value before any of the semantic rules of case expression is applied. With the semantic rules we process the match clauses right-to-left and rewrite the case expression according the currently matching rule. Rules of Figure 10 applicable when there is no guard sequence in the currently processed match clause.

$$\frac{\texttt{case } V \texttt{ of } P \texttt{ -> } E\texttt{; \_ end}}{\texttt{subst}\ (E,\ \texttt{getMatching}\ (V,P))} \wedge \texttt{isMatching}\ (V,P) \qquad [1]$$

$$\frac{\texttt{case } V \texttt{ of } P \texttt{ -> } E \texttt{ end}}{\texttt{subst}\ (E,\ \texttt{getMatching}\ (V,P))} \wedge \texttt{isMatching}\ (V,P) \qquad [2]$$

$$\frac{\texttt{case } V \texttt{ of } P \texttt{ -> } E\texttt{; } \mathit{Ms} \texttt{ end}}{\texttt{case } V \texttt{ of } \mathit{Ms} \texttt{ end}} \wedge \neg_{Bool}(\ \texttt{isMatching}\ (V,P)) \qquad [3]$$

$$\frac{\texttt{case } V \texttt{ of } P \texttt{ -> } E \texttt{ end}}{\texttt{badmatch}} \wedge \neg_{Bool}(\ \texttt{isMatching}\ (V,P)) \qquad [4]$$

Figure 10: Rules for case expression

The first two rules can be applied when the evaluated value (`V`) of the head of the case matches the pattern (`P`) of the first unprocessed match clause. We rewrite the case expression to the body expression of the first match clause for which we apply the substitution resulted from the successful matching. If the match does not succeeds, we remove the current first match clause and recursively try the matching with the remaining clauses (rule 3); or if it is the last (or only) clause, then we returns the **`badmatch`** atom (rule 4). In Erlang, the latter causes an exception, but in the current stage of KErl we do not have exceptions, instead, the evaluation stucks, or the result is a special atom denoting the type of the exception (as here).

As the syntax rules show, a match clause can have a *guard sequence.* Here, we do not get into the details, just overview the idea of defining the semantics for this special case. A guard sequence is a sequence of guards separated by semicolons. A guard is a sequence of guard expression (restricted, side-effect free Erlang expressions) separated by colons. Semicolons and colons in fact are syntactic sugars: semicolons can be replaced by **`orelse`** logical operator, while colons can be replaced by **`andalso`**. Thus, for processing guards we only need to define structural rules replacing the connectors. The semantic rules for match clauses containing guard sequence are similar to the above four rules, but in case of successful matching we have to force the evaluation of the guard sequence first, and if it is **`true`**, then we get the same result as without the guard (rule 1,2), otherwise we get the same result as in case of failed matching (rule 3,4).

The other branching expression of Erlang is the *if expression.* This kind of expression can be transformed to case expression with structural rules, as it can be regarded as case expression with clauses containing patterns that matches to any value (e.g. the "underscore" pattern). The following example code snippet shows an if expression and the semantically equivalent case expression:

```
if
   A > B -> A;
   A < B -> B;
   true  -> eq
end
```

```
case foo of
   _ when A > B -> A;
   _ when A < B -> B;
   _ when true  -> eq
end
```

3.6.3 Block and Match Expressions (Assignment)

In KErl, expression sequence only allowed inside a begin-end block. The expression sequence is tranformed to embedded case expressions depending whether the first expression is a match expression or not. This idea appears in the doctoral thesis of Fredlund [Fre01].

$$\frac{\texttt{begin } P \texttt{ = } E\texttt{, } \mathit{Es} \texttt{ end}}{\texttt{case } E \texttt{ of } P \texttt{ -> begin } \mathit{Es} \texttt{ end end}}$$

$$\frac{\texttt{begin } E\texttt{, } \mathit{Es} \texttt{ end}}{\texttt{(case } E \texttt{ of \_ -> begin } \mathit{Es} \texttt{ end end)}} \wedge \neg_{Bool}\ (\texttt{isMatchExp}\ (E))$$

Figure 11: Rules for block and match expressions

3.6.4 Function Call

As we do not have module structure in KErl, module-qualified calls are missing from our current semantic definition. It is allowed to call a function with the following syntax:

$$\textsc{syntax}\quad \mathit{CallExp} ::= \mathit{Exp}(\mathit{Exps})\ [\text{seqstrict}]$$

where the first expression before the opening parenthesis should be an atom or an anonymous function. If it is an atom, the expression is a call to a function stored in the `defs` cell. With the rule of Figure 12 we look up the match clauses of the function and rewrite the function call to a case expression with a tuple in its head composed by the actual parameters and with the match clauses of the function in its body (recall that we also have transformed the formal parameters of the function definition to a tuple in the preprocessing phase). Note that we need the condition of the rule because we do not store the implementation of the built-in functions (BIF) in the `defs` cell, but rather we define their semantics directly in KErl. So we only try to look up the name of the function, if it is not defined as BIF.

$$\left\langle \frac{\mathit{Name}{:}\mathit{Atom}(\mathit{Args})}{\texttt{case}\ \{\mathit{Args}\}\ \texttt{of}\ \mathit{Match}\ \texttt{end}} \ \cdots \right\rangle_{\mathsf{k}} \ \langle \cdots\ \mathit{Name} \mapsto \mathit{Match}\ \cdots \rangle_{\mathsf{defs}} \ \wedge \neg_{\mathit{Bool}}\ \texttt{isBIF}\ (\mathit{Name})$$

Figure 12: Rule for function call

Anonymous Function Call

In the section about the substitution (Section 3.4) we highlighted fun expressions, because it is one of the expressions which causes variable shadowing, thus special substitution rules needed. In this section we explain the formal semantics of the call to an anonymous function. Starting with version 17.0, the language supports anonymous functions with an internal name (which is a variable) easing the definition of recursive anonymous functions, but making the formal semantics of fun expressions more challenging. As introduction, we show an example code snippet with a call to a recursive anonymous function.

```
fibonacci(N) ->
  fun Fib(0, A, _) -> A;
      Fib(N, A, B) -> Fib(N-1, B, A+B)
  end(N, 0, 1).
```

The function `fibonacci` calculates the `N`. fibonacci number using a three-parameter recursive anonymous function. The parameter of the `fibonacci` function (`N`) is shadowed in the second clause of the anonymous function by the first formal parameter. The variable `Fib` is used as the internal name of the anonymous function, so the `Fib(N-1, B, A+B)` expression is a recursive call to the fun. We can evaluate the call to an anonymous function with the rules shown in Figure 13.

$$\left\langle \frac{\texttt{fun}\ \mathit{Cls}\ \texttt{end}(\mathit{Args})}{\texttt{case}\ \{\mathit{Args}\}\ \texttt{of}\ \ \texttt{getMatchFromFun}\ (\mathit{Cls})\ \texttt{end} \curvearrowright\ \texttt{restoreFunvars}\ (\mathit{Funvars})} \ \cdots \right\rangle_{\mathsf{k}}$$

$$\left\langle \frac{\mathit{Funvars}}{(\mathit{Funvars} -_{\mathit{Map}}\ (\ \texttt{getFunvar}\ (\ \texttt{fun}\ \mathit{Cls}\ \texttt{end})))\ \ \ \texttt{getFunvar}\ (\ \texttt{fun}\ \mathit{Cls}\ \texttt{end})} \right\rangle_{\mathsf{funvars}}$$

$$\left\langle \frac{\mathit{Var}}{\mathit{Fun}} \ \cdots \right\rangle_{\mathsf{k}} \ \langle \cdots\ \mathit{Var} \mapsto \mathit{Fun}\ \cdots \rangle_{\mathsf{funvars}}$$

Figure 13: Rules for anonymous function call

In the `k` cell we rewrite the function call to a case expression similarly as in case of a call to a named function. For this, using the `getMatchFromFun` helper function we transform the clauses of the anonymous function for the body of the case expression by tupling the formal parameters and by removing the variable denoting the name of the fun (if the clause has one). After the newly constructed case expression we insert the computation restoreFunvars (*Funvars*) with a sequential composition in order to restore the content of the funvars cell after evaluated the anonymous call, because during the evaluation we store the variable denoting its name assigned to the fun expression itself in the map of the funvars cell, but this variable only valid inside the fun expression. Anonymous functions can be embedded into each other, thus variable shadowing can be happen also in case of variables denoting fun names. In the $\mathbb{K}$ framework, we do not have the possibility to update the value of a key-value pair in a map, instead, we have to remove the old key-value pair (using the -Map operation), and then insert the new key-value pair (using simple concatenation). The second rule shows the variable lookup from the funvars cell: the variable is replaced by its assigned anonymous function.

Now let us see the semantic rules in action. Assuming that the fibonacci function was called with 10 in its parameter, after some steps, we have the following code in the k cell.

```
fun Fib(0, A, _) -> A;
    Fib(N, A, B) -> Fib(N-1, B, A+B)
end(10, 0, 1).
```

After applying the rule that processes anonymous function calls (first rule of Figure 13), the content of the k cell is:

```
case {10, 0, 1} of
  {0, A, _} -> A;
  {N, A, B} -> Fib(N-1, B, A+B)
end.
```

At the same time, the following assignment is inserted into the map of the funvars cell:

```
Fib ↦ fun Fib(0,A,_) -> A; Fib(N,A,B) -> Fib(N-1, B, A+B) end
```

In the current code, the tuple in the head of the case expression matches the second clause, so applying the proper semantic rules for case expressions (third and second rule of Figure 10), we reaches the state where the code in the k cell is:

```
Fib(10-1, 1, 0+1)
```

After evaluating the actual parameters of the function call (caused by the strictness attribute of the corresponding syntactic rule), the variable lookup rule (second rule of Figure 13) results the following code:

```
fun Fib(0, A, _) -> A;
    Fib(N, A, B) -> Fib(N-1, B, A+B)
end(9, 1, 1).
```

Thereafter, we can continue with rewriting the call to a case expression, an so on, until we finally have 0 in the first parameter and the evaluation terminates with the value of the second parameter.

3.6.5 Built-in Functions

Built-in functions (BIFs) are built-in to the Erlang virtual machine mainly because they implement a functionality that is impossible or inefficient to implement in Erlang. Most of these functions are belonging to the erlang module, and not just in KErl, but in the normal Erlang setting they can be called using the function name only (because the erlang module is auto-imported). Moreover, a subset of BIFs has a special role in the language, because in guard sequences we cannot use user-defined functions, but we can call one of these collected BIFs. Because BIFs are commonly used in Erlang programs, we decided to implement several of them in our semantics. As mentioned in Section 3.6.4, we do not insert the BIFs into the defs cell; we implement their semantics explicitly with rules. In Figure 14 we show rules for two BIFs: the hd unary function returns the head element (first item) of the list given as parameter; the setelement three-parameter function returns a tuple which is a copy of the second parameter with the element given by the index of the first parameter replaced by the value of the third parameter.

$$\frac{\texttt{hd}([X \mid \_])}{X} \qquad \frac{\texttt{setelement2}\ (1, (E\texttt{,}\ Es), V)}{V\texttt{,}\ Es}$$

$$\frac{\texttt{setelement}(I\texttt{,}\ \{Es\}\texttt{,}\ V)}{\{\ \texttt{setelement2}\ (I, Es, V)\}} \qquad \frac{\texttt{setelement2}\ (I, (E\texttt{,}\ Es), V)}{E\texttt{,}\ \texttt{setelement2}\ (I -_{Int} 1, Es, V)}$$

Figure 14: Rules for BIFs (hd and setelement)

3.6.6 List Comprehension

List comprehension, which is analogous to set comprehension in Zermelo-Frankel set theory, is one of the most complex language constructs in Erlang. The result of a list comprehension is a list of elements produced by evaluating the head expression of the list comprehension in environments of each combinations of generator list elements for which all filters are true. As mentioned in Section 3.4, variables in the generator patterns shadow variables surrounding the list comprehension. Besides that, pattern matching and variable binding in generators occur according to the normal pattern matching rules, and if a match fails, then that element of the generator list is skipped. In case of successful matching the substitution of the newly assigned variables applied to the head expression and to the other generators and filters in the left hand side of the current generator.

Many modern functional programming languages have this feature, however, in our knowledge, no publication about formal semantics of a language mentions the formal semantics of list comprehension. One exception is The Haskell 98 Report [P$^+$03] where they show how list comprehension can be translated into the Haskell kernel. In KErl, we apply a similar idea, and with the three structural rules of Figure 15 we recursively rewrite the list comprehension by processing its generators and filters left-to-right:

[1] After processing each generators and filters, the generator-filter list in the list comprehension becomes empty, so we return the head expression of the list comprehension in a one-element list as result.

[2] If the current first element of the generator-filter list is a generator, we compose a function call, where the called function is recursive anonymous function and the parameter of the call is the generator list of the current generator. In the fun we matches the elements of the list one-by-one to the pattern of the generator. If the element matches the pattern, we recursively evaluate the list comprehension with the remaining generators and filters.

[3] If the current first element of the generator-filter list is not a generator, then it should be a filter. We have already processed the generators in its right hand side, we have substituted all variables with a value in the filter expression, so if we use this filter expression as the head of the newly constructed `case` expression, because of the strictness rule of the `case`, this expression is evaluated to a value. According the this value, we either continue to process the remaining generators and filters (in case of `true`), or we return with an empty list.

$$\frac{[E \texttt{ || } \cdot_{GFs}]}{[E]} \qquad [1]$$

$$\frac{[E \texttt{ || } P \texttt{ <- } L\texttt{,}\ GFs]}{\texttt{fun } F([\,]) \texttt{ -> } [\,]\texttt{; } F([X \mid Xs]) \texttt{ -> case } X \texttt{ of } P \texttt{ -> } [E \texttt{ || } GFs] \texttt{ ++ } F(Xs)\texttt{; \_ -> } [\,] \texttt{ end end}(L)} \qquad [2]$$

$$\frac{[E \texttt{ || } Bool\texttt{,}\ GFs]}{\texttt{case } Bool \texttt{ of true -> } [E \texttt{ || } GFs]\texttt{; \_ -> } [\,] \texttt{ end}} \wedge \neg_{Bool} \texttt{ isGenerator } (Bool) \qquad [3]$$

Figure 15: Rules for list comprehension

4 Testing the Semantics

This section overviews the validation of KErl, the executable formal semantics of Erlang. As mentioned, the language has neither any formal documentation, nor an official test suite. Therefore we decided to develop an own test suite which can be used for testing our semantic definition as well as applicable to validate any other formal semantics or alternative compiler/interpreter of Erlang.

During the development of the KErl semantics, we follow the test-driven development methodology: before adding a new language feature to the semantics, we wrote test cases, which have been validated against the open-source reference implementation. First, we tried to cover all behaviour and corner cases of the new feature in isolation, then specify tests for its interaction with the previously defined features. We use these tests also for regression testing: when we implemented a language construct which not only requires new rules for its semantics, but existing functions have to be modified (e.g. `getMatching` or the `subst`), we verified whether the previously developed and tested semantic rules still performs correctly.

One of our motivations of choosing to specify the Erlang semantics in $\mathbb{K}$ was to have an executable semantics: at each stage of our work we had a working interpreter for the fragment of Erlang we defined up to that point. As a consequence, we had the possibility to test all of the meaningful set of semantic rules as they were being developed. In the current stage, we have a comprehensive test suite of more than 70 tests, and our language definition pass each of them.

5 Verifying Program Properties

In this section we show an application of the KErl semantics: stating a program property by a reachability logic formula and prove it with a semi-automatic verification tool.

Symbolic Circular Coinduction.

A sound and relatively complete 7-rule proof system is available [RŞCM13] for matching logic, which is theoretically suitable for proving program properties expressed with reachability formulas. However, this 7-rule proof system is rather complex, there is no practical strategy published for building the proofs. A not complete, but semi-automatic system is implemented as an extension of the $\mathbb{K}$ framework, so we have chosen this simplified version of the above mentioned proof system introduced in a related technical report [ALR15]. *Symbolic Circular Coinduction* (SCC) is coinduction-based extension of symbolic execution that can be used for deductive verification of program properties specified by RL formulas. The proof system consists of 3 inference rules, and can be easily implemented with a straightforward tactic. The report presents a prototype tool that can automatically build proofs if we give a language definition and an RL formula expressing some correctness property. Note that both the 7-rule and 3-rule proof systems are sound only on deterministic languages. In KErl, we have some non-determinism caused by the under-specified evaluation order of the sub-expression, but because in the current stage we do not have side-effects in the language, it does not introduce outside observable difference in the behaviour of the programs. For non-deterministic languages they introduced *all-path reachability logic* [ŞCM$^+$14], but we only need to use it if we have side-effects or parallelism in the language.

Now let us see our example program from the introduction of Section 3 and the property to be verified. We implemented the classical `sum` program recursively. In the paper about reachability logic [RŞCM13] a similar example is shown for an imperative language, so it is interesting to see how it works in case of a functional language. In the imperative language they used a `while` loop and compose the property using the values of program variables, here we use recursion and state the property using a constraint on the final value of the program.

```
sum(0, S) -> S;
sum(X, S) -> sum(X-1, S+X).

main() -> sum(N:Int, S0:Int)
```

Note that it is not a concrete, but a symbolic program, because we do not call the `sum` function with concrete values, but with symbolic ones: N and $S0$ are symbolic values (mathematical variables), while `X` and `S` are regular Erlang variables. We would like to prove, that for any integer values N and $S0$, the result of the call will equal to $N * (N + 1)$ `div` $2 + S0$. In order to be able to apply the circularity rule during the proof (see later), we make some evaluation steps on our initial program: put the function definition into the `defs` cell and replace the call to the `main` function with its body. The reachability formula expressing the initial goal of the proof is the following:

$\langle\langle \mathtt{sum}\,(N{:}Int, S0{:}Int)\rangle_{\mathsf{k}}\ \langle \mathtt{sum} \mapsto [\{\mathtt{0,S}\}\ \mathtt{->S},\ \{\mathtt{X,S}\}\ \mathtt{->sum}\,(\mathtt{X}-1, \mathtt{S}+\mathtt{X})]\rangle_{\mathsf{defs}}\ \langle \cdot_{Map}\rangle_{\mathsf{funvars}}\rangle_{\mathsf{cfg}} \quad \wedge N \geq 0$
$\Rightarrow$
$\langle\langle V{:}Value\rangle_{\mathsf{k}}\ \langle D{:}Map\rangle_{\mathsf{defs}}\ \langle \cdot_{Map}\rangle_{\mathsf{funvars}}\rangle_{\mathsf{cfg}} \quad \wedge\ V == N * (N+1)\ \mathtt{div}\ 2 + S0$

The input of the SCC proof system is a $\mathbb{K}$ definition of the language and the property to prove. The proof is successful, if every branch of the proof ends with a reachability formula, where the right hand side is the logical implication of the left hand side: this is the axiom rule. During the proof, we can use two inference rules: we can generate the derivative(s) of the configuration in the left hand side of the formula either by applying a rule of the given semantics, or by using an already proved previous goal (circularity). Formal definition of the proof system and other details is presented in the above mentioned technical report [ALR15].

In the followings we highlight the main steps of the proof for our example property. The `defs` and the `funvars` cells remain unchanged during the proof, so we replace them with ellipses. As first step, we apply the semantic rule of function call that rewrites the call to a case expression:

$\langle\langle \mathtt{case}\ \{N{:}Int, S0{:}Int\}\ \mathtt{of}\ \{\mathtt{0,S}\}\ \mathtt{->}\ \mathtt{S};\ \{\mathtt{X,S}\}\ \mathtt{->}\ \mathtt{sum}\,(\mathtt{X}-1, \mathtt{S}+\mathtt{X})\ \mathtt{end}\ \rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \quad \wedge N \geq 0$
$\Rightarrow$
$\langle\langle V{:}Value\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \quad \wedge\ V == N * (N+1)\ \mathtt{div}\ 2 + S0$

Because we try to match the tuple of symbolic values N and $S0$ to the clause patterns of the case expression, the proof branches and two different constraints are generated. In the first branch let the constraint $N == 0$. In this case, the first clause of the case expression matches, we substitute Erlang variable `S` for the symbolic value $S0$ in the body of the first clause, and finally we acquire the following formula:

$\langle\langle S0{:}Int\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \quad \wedge N == 0$
$\Rightarrow$
$\langle\langle V{:}Value\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \quad \wedge\ V == N * (N+1)\ \mathtt{div}\ 2 + S0$

Here, the left hand side implies the right hand side in the sense of matching logic $(0*(0+1)\ \mathtt{div}\ 2 + S0\ ==\ S0)$, which is the axiom in the proof system, so this branch of the proof is successfully terminated. Now let us see the other branch of the proof with the constraint $N/=0$. In this case the second pattern matches and we substitute Erlang variables `X` and `S` for the symbolic values N and $S0$ in the body of the second clause of the case expression.

$\langle\langle \mathtt{sum}\,(N{:}Int - 1, N + S0{:}Int)\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \quad \wedge N/=0$
$\Rightarrow$
$\langle\langle V{:}Value\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \quad \wedge\ V == N * (N+1)\ \mathtt{div}\ 2 + S0$

In our initial formula, in the `k` cell we had $sum(N, S0)$, so a similar structure, but different symbolic values. In this case, we can use the initial formula as circularity rule, and rewrite the current formula according. This way we get the following formula:

$\langle\langle V0{:}Value\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \quad \wedge\ V0 == (N-1) * (N-1+1)\ \mathtt{div}\ 2 + (N + S0) \wedge N > 0$
$\Rightarrow$
$\langle\langle V{:}Value\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \quad \wedge\ V == N * (N+1)\ \mathtt{div}\ 2 + S0$

where $(N-1) * (N-1+1)\ \mathtt{div}\ 2 + (N + S0) == N * (N+1)\ \mathtt{div}\ 2 + N$ is mathematically valid, so again, we get the axiom of the proof system, and both branches of the proof terminate with success.

6 Related work

Formal Semantics of Erlang

There exists several approaches for defining the formal semantics of Erlang, we mention here the most relevant ones. Huch [Huc99] defines the operational semantics of a small subset of Erlang (he refers it as "Core Erlang") and presents a verification technique using abstract interpretation and LTL model checking. The author talks about a prototype implementation of the method, but to our knowledge the tool have never released, and there is no further reference to its usage in the literature. Fredlund [Fre01] defines a bigger subset of the language together with a theorem prover based on μ-calculus and first-order predicate logic. With his research group, they implemented a reasoning framework called Erlang Verification Tool [FGN+03], but it had no support for proof automatization, the logic it used is quite complex, so the user of the tool had to be an expert in theorem proving. Except their own use cases, we have not found any usage of this tool. A more successful utilization of this semantics is the McErlang model checker [FS07], which is successfully used for proving non-trivial properties of various distributed Erlang programs. The tool is still available and is maintained, however, the last news are from 2011 and it is not clear, which is the supported version of Erlang now. While Fredlund's semantics valid only for Erlang programs run on a single node, Svensson and Claessen [Cla05] modified some rules and added an extra layer to the semantics to properly model distributed behaviour. Svennson and Fredlund [SF07] then refined it by defining the behaviour of disconnected nodes. Finally, the previous authors presents a unified semantics for future Erlang [SFBE10] trying to make the semantics more simpler and clearer and to stimulate discussions about the future of the language. According to this overview we can state that there does not exist an up-to-date formal semantics of Erlang. The old ones can be refreshed not at all or only with difficulty, since they either not modular enough or the implementation of the semantics is not open or even not available any more. Moreover, none of them is executable, therefore we should rely their validity without having the possibility to test them.

Language Definitions in $\mathbb{K}$

The $\mathbb{K}$ semantic framework has been successfully used for formally defining a number of real-word, complex programming languages. The most complete formal semantics of C [HER15], Java [BR15] and JavaScript[PcR15] was developed by the research team of the framework, but also several independent usage can be found, like $\mathbb{K}$PHP [FM14]. These projects demonstrates the strengths and main benefits of the framework: the development of the semantics is effective as the language features can be added and immediately tested one by one; and it provides language-independent tools like symbolic execution engine, semantic debugger, model checker or deductive program verifier once a semantics is given to that language. The C language formalization of Ellison and Roşu [HER15] shows how to discover program flaws or how to enumerate nondeterministic behaviour using the built-in capabilities of $\mathbb{K}$. K-Java [BR15] is the first complete formal semantics of Java. A comprehensive test suite has been developed for the language, and some applications are presented in the paper: detecting deadlocks with spate space exploration and proving a correctness property with LTL model checking. The KJS [PcR15] semantic definition is passed a widely-used conformance test suit, and by tracing the coverage of semantic rules the tests exercised they also extend the test suite. They used the semantics for verifying program properties and for finding security vulnerabilities.

7 Conclusion and Future Work

We have discussed KErl, which to our knowledge is the first executable formal semantics of Erlang. In a previous work [HKT16] we defined a smaller subset of the language in the $\mathbb{K}$ framework as proof of concept to our formal verification method aiming to prove the correctness of behaviour-preserving refactoring transformations. The primary motivation of refining and extending the formal language definition was to make this verification method applicable for real-word, widely used refactorings. Thereafter, we realized that the semantic definition style and framework is very user-friendly, and at the same time expressive enough to define the formal semantics of no matter complex language. A great advantage, that the semantics can be tested by running programs of the defined language with the interpreter generated from the language definition; and also many other language-independent, useful tools provided for formal program analysis and verification.

As we cannot find an up-to-date, complete or easily expandable, verification-focused formal semantics for the Erlang programming language, we decided to continue our previous work and define the matching logic semantics of Erlang in the $\mathbb{K}$ framework. We utilized the generated interpreter and developed a test suite in parallel with the development of the semantics. We explored the available formal verification methods for matching logic and chose to use the automated SCC proof system, which is a simplification of the one-path reachability logic proof system. It works only with deterministic languages, so as first step we have implemented a side-effect free, sequential sub-language. In the near future, we plan to make the semantics of the sequential language constructs complete, e.g. by extending KErl with modules, records or with the missing data types, like floating point number or map. Later, we also plan to define the semantics of Erlang features connected to parallel and distributed execution, like processes and message passing; and to examine the proof system of the all-path reachability logic for reasoning about parallel and distributed programs.

References

[ALR15] Andrei Arusoaie, Dorel Lucanu, and Vlad Rusu. A Generic Framework for Symbolic Execution: Theory and Applications. Research Report RR-8189, Inria, September 2015.

[BR15] Denis Bogdănaş and Grigore Roşu. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*, pages 445–456. ACM, January 2015.

[Cla05] Koen Claessen. A semantics for distributed erlang. In *In Proceedings of the ACM SIPGLAN 2005 Erlang Workshop*, pages 78–87. ACM Press, 2005.

[CT09] Francesco Cesarini and Simon Thompson. *Erlang Programming.* O'Reilly Media, Inc., 2009.

[erla] Erlang Reference Manual User's Guide. `http://erlang.org/doc/reference_manual/users_guide.html`. Accessed July, 2016.

[erlb] Erlang website. `http://www.erlang.org`. Accessed July, 2016.

[FGN+03] Lars-Åke Fredlund, Dilian Gurov, Thomas Noll, Mads Dam, Thomas Arts, and Gennady Chugunov. A verification tool for ERLANG. *International Journal on Software Tools for Technology Transfer*, 4(4):405–420, 2003.

[FM14] Daniele Filaretti and Sergio Maffeis. *An Executable Formal Semantics of PHP*, pages 567–592. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[Fre01] Lars-Åke Fredlund. *A Framework for Reasoning about Erlang code.* PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2001.

[FS07] Lars-Åke Fredlund and Hans Svensson. McErlang: A Model Checker for a Distributed Functional Programming Language. *SIGPLAN Not.*, 42(9):125–136, October 2007.

[HER15] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. Defining the Undefinedness of C. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, pages 336–345. ACM, June 2015.

[HKT16] Dániel Horpácsi, Judit Kőszegi, and Simon Thompson. Towards Trustworthy Refactoring in Erlang. In *Proceedings of VPT'16,* Eindhoven, The Netherlands, 2nd April 2016, volume 216 of *Electronic Proceedings in Theoretical Computer Science*, pages 83–103. Open Publishing Association, 2016.

[Huc99] Frank Huch. Verification of Erlang Programs Using Abstract Interpretation and Model Checking. *SIGPLAN Not.*, 34(9):261–272, September 1999.

[P+03] Simon Peyton Jones et al. The Haskell 98 Language and Libraries: The Revised Report. *Journal of Functional Programming*, 13(1):1–255, Jan 2003. `http://www.haskell.org/definition/`.

[PcR15] Daejun Park, Andrei Ştefănescu, and Grigore Roşu. KJS: A Complete Formal Semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, pages 346–356. ACM, June 2015.

[Roş15] Grigore Roşu. Matching Logic - Extended Abstract. In *Proceedings of RTA'15*, volume 36 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5–21, Dagstuhl, Germany, July 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[RŞ10] Grigore Roşu and Traian Florin Şerbănuţă. An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[RŞCM13] Grigore Roşu, Andrei Ştefănescu, Ştefan Ciobâcă, and Brandon M. Moore. One-Path Reachability Logic. In *Logic in Computer Science (LICS), 2013 28th Annual IEEE/ACM Symposium on*, pages 358–367, 2013.

[ŞCM$^+$14] Andrei Ştefănescu, Ştefan Ciobâcă, Radu Mereuta, Brandon M. Moore, Traian Florin Şerbănută, and Grigore Roşu. All-Path Reachability Logic. In *Proceedings of RTA-TLCA'14*, volume 8560 of *LNCS*, pages 425–440. Springer, July 2014.

[SF07] Hans Svensson and Lars-Åke Fredlund. A more accurate semantics for distributed erlang. In *Proceedings of the 2007 SIGPLAN Workshop on ERLANG Workshop*, ERLANG '07, pages 43–54, New York, NY, USA, 2007. ACM.

[SFBE10] Hans Svensson, Lars-Åke Fredlund, and Clara Benac Earle. A Unified Semantics for Future Erlang. In *Proceedings of the 9th ACM SIGPLAN Workshop on Erlang*, Erlang '10, pages 23–32, New York, NY, USA, 2010. ACM.