Validating Formal Semantics by Property-Based Cross-Testing

Péter Bereczky
Dániel Horpácsi
berpeti@inf.elte.hu
daniel-h@elte.hu
ELTE, Eötvös Loránd University
Budapest, Hungary

Judit Kőszegi
Soma Szeier
koszegijudit@elte.hu
szeier529@inf.elte.hu
ELTE, Eötvös Loránd University
Budapest, Hungary

Simon Thompson
S.J.Thompson@kent.ac.uk
University of Kent
Canterbury, UK
ELTE, Eötvös Loránd University
Budapest, Hungary

Abstract

To describe the behaviour of programs in a programming language we can define a formal semantics for the language, and formalise it in a proof assistant. From this semantics we can derive the behaviour of each particular program in the language. But there remains the question of validating the formal semantics: *have we got the formalisation right?*

Our approach is to use property-based cross-testing of formal semantics, which is based on the combination of a number of existing approaches to validation. In particular, we give a concrete implementation of our ideas for a set of formalisations of Erlang and Core Erlang. We describe the adjustments that need to be made to execute these semantics, then we present and evaluate property-based testing in the context of cross-checking semantics, including random program generation and counterexample shrinking.

CCS Concepts: • **Theory of computation → Operational semantics**; **Program verification**; *Functional constructs*; • **General and reference → Validation**.

Keywords: formal semantics, validation, property-based testing, QuickCheck, Coq, K framework

ACM Reference Format:
Péter Bereczky, Dániel Horpácsi, Judit Kőszegi, Soma Szeier, and Simon Thompson. 2020. Validating Formal Semantics by Property-Based Cross-Testing. In *IFL 2020: Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages (IFL '20), September 2–4, 2020, Canterbury, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3462172.3462200

IFL '20, September 2–4, 2020, Canterbury, United Kingdom

ACM ISBN 978-1-4503-8963-1/20/09…$15.00
https://doi.org/10.1145/3462172.3462200

1 Introduction

Our work here is part of a wider project to reason about correctness of refactorings, and that requires a rigorous, formal definition of the programming language under refactoring: in our case, Erlang [9]. In earlier work, we defined and implemented executable formal semantics for the sequential parts of Erlang and Core Erlang [8], including a reduction semantics for a subset of Erlang using the $\mathbb{K}$ framework [24], and a natural semantics for a subset of Core Erlang, implemented in Coq [1, 2]. In this paper we investigate how to validate this work by using property-based testing techniques.

As Core Erlang is an intermediate language between Erlang and BEAM code [37], Erlang can be compiled to both Core Erlang and BEAM, and the semantics of these three languages, and the translations between them, can be compared. There is no complete, up-to-date and precise language specification for any of the above languages. We therefore decided to take the Erlang/OTP compiler and the BEAM interpreter (both in v. 22.0) as the *reference implementations* for reasoning about the correctness of the semantics, due to the high degree of social trust in these standard components stemming from their history and extensive user base. Moreover, if we verify refactorings using a formal semantics consistent with this reference base, we shall preserve program behaviour in the *de facto* implementation.

We therefore say that a formal semantics of Erlang is *correct* if, for every Erlang program `P`, the behaviour of `P` under the formal semantics is the same as the behaviour of the BEAM interpreter running the code obtained from `P` by trusted translation; we define correctness for a formal semantics of Core Erlang in a similar way.

Not only can we test a semantics against the reference implementation, we can also test different formal semantics against each other. This *cross-testing* delivers further benefits:

- If both formal semantics show the same (or similar) incorrect behaviour, that may indicate a generic misconception about the behaviour of a particular language feature, rather than an error in the formalisation,
- If one is correct and the other not, the first can be used to assist the debugging of the second, exploiting the (trusted) transformation from Erlang to Core Erlang.

It is worth noting that the general idea of cross-testing (or differential testing) of (executable) semantics can be generalised for any two languages provided that one can be translated to the other, providing evidence that the definitions are valid *relative to each other*.

In validating the (Core) Erlang semantics, we consider two sources of test programs. Firstly, we run the ErLLVM benchmark suite [16] and check the semantics on the *small* test cases together with our own test suite. Secondly, we use *property-based testing* with randomly generated programs.

The main contributions of this paper are to present:

- A general approach to validation of formal semantic definitions of related languages by property-based cross-testing, addressing shrinking of randomly generated programs and multiple result formats.
- An architecture supporting uniform execution, comparison and testing, built from two formal semantics (given in different styles and implemented with different tools) and a reference implementation.
- Extensive validation of formal semantics for sublanguages of *sequential* Core Erlang and Erlang, implemented in Coq and in the $\mathbb{K}$ framework. These sublanguages contain all salient features from a semantic point of view: extending the semantics to the full languages would be a straightforward exercise.

The rest of the paper is structured as follows. Section 2 discusses the most common approaches to testing formal semantics, then in Section 3 we overview the semantics definitions to be validated. Section 4 explains our testing approach and our implementation for validating Erlang and Core Erlang semantics. Section 5 evaluates our approach and Section 6 summarises future work and concludes.

2 Related Work

Most programming languages lack a formal definition and are instead defined by a reference implementation. Typically, reference implementations can only be used for interpreting programs, they do not define a formal semantics and therefore cannot be used for constructing formal proofs. Fortunately, there is an increasing effort to equip mainstream languages with formal definitions. For example, C, Java, OCaml, Scheme, Haskell, EVM are being formalised in the $\mathbb{K}$ framework [23], while semantics for C [4], JavaScript [6], R [7] and WebAssembly [22] are being developed in Coq.

Semantics validation. As other authors have pointed out [5, 18, 38], it is crucial to validate the formal definitions against the language specifications and the reference implementations; otherwise, they could not be used to argue about the behaviour of particular programs in the language, or about general properties of the language itself. According to Blazy and Leroy [5], there are five basic methods to validate formal semantics:

M1 Manual review and debugging
M2 Proving properties of the semantics, such as type preservation and determinism
M3 Using verified translations and trusted semantics
M4 Validating executable semantics, e.g. testing against test suites and experimental testing
M5 Using equivalent, alternate versions of the semantics

These methods, or combinations of them, are commonly used when a formal semantics is to be validated. For instance, the semantics of Lolisa [38] was validated with M2, M4 and M5, while CompCert [4, 5] apparently uses all of them.

However, the most common way of validating a formal semantics is method M4: developing an executable version of the semantics and testing it against the reference implementation. This is used for PHP [17], SQL queries [18] and Erlang [24], as well as in the work by Politz et al. on JavaScript [33] and in the work by Roessle et al. [35] on the big-step semantics of x86-64 binaries.

Property-based testing. Case-by-case testing can be significantly improved by partially or fully generalising the correctness checks over the test data. Property-based testing (PBT) [19] is a technique that generalises unit tests into *properties* containing universally quantified variables, testing these properties at randomly generated values for the variables. Applying PBT and its widely used implementation QuickCheck brings the following benefits:

- *General correctness properties are checked with randomly generated values*: universal properties are tested for values defined by *data generators*, which generate possible cases in increasing order of complexity.
- *Automatic shrinking of counterexamples*: the system simplifies the failure cases found, so as to provide locally minimal, more comprehensible counterexamples.
- *Automatic assembly of regression test suites*: the randomly generated test cases, especially those for bugs that have been fixed, can be collected into a set of regression test cases.

In performing property-based testing on semantics definitions, or indeed language processors in general, the main challenges are to build data generators capable of producing well-formed (compilable) programs with non-trivial effects and to define a useful simplification mechanism, called *shrinking*. This problem was already addressed by Pałka et al. [31] in verifying a Haskell compiler, while Perényi and Midtgaard [32] applied property-based testing to verify a C to WebAssembly compiler. The latter reuses the official syntax representation, uses an explicit recursion limit and implments strict shrinking rules; our approach is similar.

For verifying refactoring tools, Horpácsi et al. [14] developed an attribute grammar based generator. In the latter, a subset of Erlang was formalised as an attribute grammar, ultimately synthesising a data generator for random Erlang

programs from the grammar description. We reused Horpácsi's solution in the validation of the formal semantics: we took the grammar-based generator, revised the grammar, and we also added support for shrinking of random programs. It is worth noting that the generator is expected to emit well-formed programs, but not necessarily meaningful algorithms. Some rules restrict the set of generated programs (ensuring static semantics, eliminating infinitely recursion), yet the behaviour is mostly random and it is likely to include unusual, edge cases. We discuss this in more detail in Section 4.

3 Semantics Under Test

In this section we give formal semantics definitions and a reference implementation for the three languages compared in this work. We first introduce the languages, and then we explain some of the issues that arise when the semantics is to be used to interpret particular programs, efficiently.

3.1 Erlang and its Core Language

In the standard Erlang compilation process, the Erlang source code is compiled to Core Erlang (as an intermediate language), which is then compiled to BEAM bytecode (the actual target code). These translations are official, trustworthy pieces of software. In addition, we trust the Erlang/OTP providing a reference implementation for the semantics of the low-level code, BEAM. So the reference interpreter for BEAM can be used as a frame of reference when checking the formal semantics of the higher level languages, Erlang and Core Erlang. The Erlang snippet

```
main() -> (1 + 1) + 2.
```

(with optimisations turned off) translates to

```
'main'/0 = fun() ->
  case <> of
   <> when 'true' -> let <_0> =
        call 'erlang':'+'(1, 1) in
          call 'erlang':'+'(_0, 2)
   <> when 'true' -> primop 'match_fail'
       ({'function_clause'})
  end
```

demonstrating the abstraction gap between these languages. It can be seen that Core Erlang is more explicit about control flow and is lacking some high-level features of Erlang.

We have developed formal semantics for both Core Erlang and Erlang, and compare these to the reference semantics of BEAM. The formal definitions are given in different styles, and in different implementation frameworks: the Erlang definition [24] is a reduction-style semantics implemented in the $\mathbb{K}$ framework (v. 3.6), while the more recent Core Erlang definition [1, 2] was specified in the Coq proof assistant (v. 8.12.1).

In the setting of testing the two formal semantics with the "same" input, it is important to ensure that the language features covered by the Erlang definition translate to features covered by the Core Erlang definition. This is an issue to be taken into account as our definitions do not cover the entire languages. As a matter of fact, we made sure that both the Erlang and Core Erlang formal definitions support most sequential constructs, and so does the random program generator. The formalisation of the concurrent language parts is future work. We note that it is also an interesting question whether full coverage of Erlang expressions ensures full coverage of Core Erlang, that is, whether all expressions in Core Erlang can be generated from some Erlang expression. We discuss this topic in Section 5.2.

For historical reasons, these formal definitions were supposed to serve different purposes: the Erlang definition was developed mainly for the precise specification of the language, program execution and simple expression equivalence proofs, whilst the Coq-based Core Erlang definition was created with the need for more advanced proofs in mind. Indeed, the inductive big-step semantics definition in Coq brought the freedom in expressing and proving complex properties of the language semantics, but the automatic execution of such a definition proved to be challenging.

3.2 Execution

We check the validity of the semantics by dynamically testing them against the reference implementation. This requires that the semantics can "run" programs, just like an interpreter does.

Erlang. The (sequential) Erlang definition was given as reduction semantics with evaluation contexts in the $\mathbb{K}$ framework, a language workbench that supports simple and effective syntax and semantics definitions, and generates various execution and analysis tools based on a single definition.

One of the most helpful features of $\mathbb{K}$ for our work is that it has a reasonably effective search technique for finding small-step derivations, basically it synthesises an interpreter for the semantics definition. This means that the small-step semantics of Erlang is inherently executable with the help of $\mathbb{K}$ and does not need any special care in this regard. For the details of this language definition, we refer to previous work by Kőszegi [24].

Core Erlang. The big-step semantics of sequential Core Erlang was formalized in Coq as an inductive relation. Beside the basic language features, it includes exceptions and side effects [1, 2, 27], so it provides a decent coverage of the sequential sublanguage.

As mentioned already, the inductive definition is good for reasoning, but not necessarily for interpretation [5]. In fact, the typical operational semantics is not computable (either because it is not syntax-directed or it is not terminating) and the execution is essentially a proof-search on the transition relation with existential variables. In Erlang and in Core Erlang, both exceptions and divergence are present, thus in our

semantics definitions there can be several derivation rules applicable to a particular configuration. There are two options to mitigate the issue: either implement the proof-search on the inductive definition (presumably with tactics [11]), or (re)define the semantics in a computable style (also in Coq). The latter may be done in the functional big-step semantics style of Owens et al. [30] or as a definitional interpreter ("equivalent alternate semantics" [5]), but in either case, composing the denotational definition and proving it equivalent to the inductive definition requires significant effort. We refer to Bereczky et al. [3] for more details.

On our first attempt we went for the first option and developed a proof search tactic to "execute" programs in the big-step semantics. This tactic [11] uses pattern-matching on the evaluable expression to determine the derivation rules to be used. Since we have exceptions formalised, for one goal there may be multiple rules applicable (i.e. the proof-search is not syntax-driven); however, because our semantics is deterministic[1], we can apply one of the rules, see whether it leads to results, and if not, we can try another matching rule instead. This process can be seen as a backtracking proof-search for the correct evaluation steps. We have implemented this search in Coq's tactic language, Ltac [11].

Unfortunately, such an automatic backtracking proof-search is not efficient enough when run in the Coq interpreter. Using a pretty-big-step style semantics [10] can significantly reduce the number of applicable rules on the concrete goals, but it cannot eliminate all decision points: for instance, executions may terminate either normally or with exceptions, and even if the semantics is deterministic, we cannot tell in advance which branch leads to the normal form (e.g. in [3] the potential exceptions of function applications). Moreover, the proof-search still needs a large amount of time and memory. We tried to make searches faster by adding lemmas about evaluating specific expressions, but this had little effect. In the end, we had to go for the other option of execution and develop a computable variant of the semantics.

3.3 Efficient Execution

The findings described in the previous paragraphs support the suggestion of Blazy and Leroy, so we decided to implement a computable variant of our big-step semantics of Core Erlang, using functional big-step style [30]. The semantics in this style is basically a recursive function equipped with a recursion depth limit. Obviously, we had to prove the equivalence between this computable semantics and the inductive version, so that testing this semantics is sufficient to test the other one. The entire formalisation is open-source and it is available on GitHub [29].

[1]Note that Core Erlang is nondeterministic in theory [8], we followed the footsteps of Neuhäußer and Noll [28] and the reference implementation, and employed a leftmost-innermost evaluation strategy in the formalisation.

The functional big-step definition shows an order of magnitude better performance in terms of execution time, which is a notable improvement over the evaluation tactic. Even though in this sense the functional big-step style semantics seems to be superior to the classic big-step, the inductive big-step semantics can be more suitable when it comes to proving properties of programs by induction.

Extracting Haskell from Coq. Once we developed the computable variant of the big-step semantics in Coq, there was one obvious option to try: whether extracting Haskell from Coq brings further improvements in speed and reliability. Coq has not been developed as a general purpose functional language, it has been designed to write formal specifications and develop mathematical proofs. The introduction to Coq [12] also highlights this possibility of extracting executable programs (Haskell or OCaml) from the specifications.

This means that the execution of the specifications with the Coq interpreter is not yet the optimal option for semantics execution. Just to mention a concrete example, in our case, we need a recursion depth limit to execute the functional big-step semantics. This limit is basically a natural number in Coq, but in practice the Coq interpreter can only compute small natural numbers, which limits the size of the evaluable Core Erlang programs. To overcome these limitations, we decided to implement the option to extract the functional big-step semantics to Haskell (GHC 8.10.4), and execute this extracted semantics to obtain the results. We have obtained improvements this way, especially for evaluating large programs which was impossible in Coq due to the limitations of natural numbers.

4 Property-Based Cross-Testing

After reviewing the semantics to be checked, we discuss in detail the property-based cross-testing approach we developed. In particular, Section 4.1 gives an overview of the approach as a derivative of the methods introduced in Section 2, then Section 4.2 explains the program generator and the counterexample shrinking mechanism, and in Section 4.3 we provide some technical details about the implementation.

4.1 Overview of the Method

Our testing approach is a combination of the fundamental semantics validation techniques outlined by Blazy and Leroy [5], which have been discussed in Section 2. In our particular case of cross-testing Erlang and Core Erlang semantics,

- We adapt method M3 by using verified translation (i.e. the official Erlang/OTP compiler) from Erlang to Core Erlang, and from Core Erlang to BEAM. Our trusted semantics component is the executable definition of BEAM (i.e. the official Erlang/OTP interpreter).

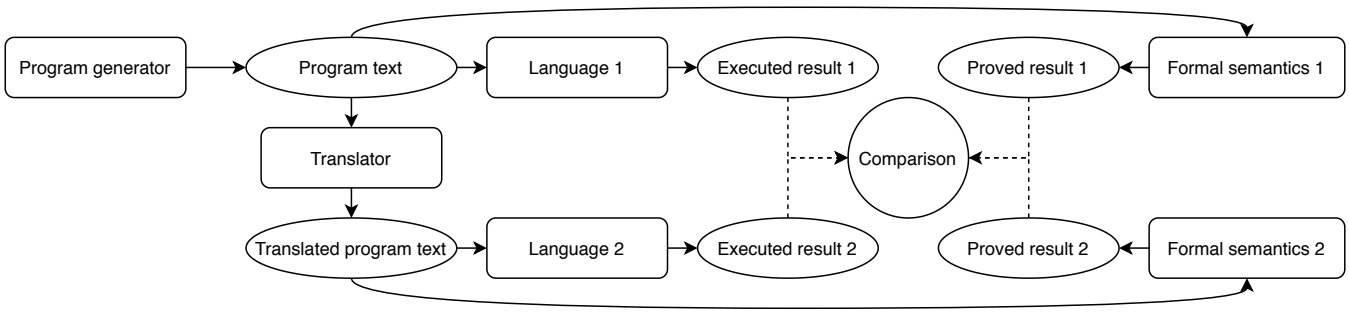

Figure 1. The general design of our approach

- We adapt method M4 by using a test suite as well as randomly generated programs to test our semantics against the reference implementation (i.e. the official Erlang/OTP interpreter). For this, we needed to make both the small-step semantics for Erlang and the big-step semantics for Core Erlang executable. We sought to gather execution information from the inductively defined big-step semantics, namely the final configurations and the corresponding proofs in the operational semantics, and we also developed an equivalent, functional version [30] of this semantics to speed up the testing process.
- Finally, we adapt method M5 in two different ways:
  - Firstly, by having semantics in two different styles (even though for two slightly different languages): the Erlang semantics is in small-step (reduction style with evaluation contexts), while the Core Erlang semantics is given as a (functional) big-step semantics.
  - Secondly, for Core Erlang, we developed inductive and functional big-step semantics which we also proved equivalent, so that testing the functional definition is sufficient to verify the inductive version.

We believe that this *combination* (as opposed to simple composition) of methods results in an even more effective formal semantics validation technique.

As seen in Figure 1, our method can be summarised as follows. We consider two programming languages with reference implementations and executable formal semantics (possibly in different semantics frameworks), as well as a translator between the two languages. We use a data generator to sample random programs in the first language, and we translate each program into the second language. Then we feed the original and translated programs into the corresponding implementations and semantics, and finally we compare the results. This latter step is of interest mainly from the technical point of view; in general, it is a structural equality check on the resulting values. The comparison is implemented within a QuickCheck property, thus failing tests are shrunk automatically in a local search for related but simpler failing tests.

4.2 Program Generator

As stated in Section 2, to apply property-based testing on Erlang semantics definitions, we need a data generator for random programs. Composing a syntax tree generator from the basic generator functions is cumbersome and error-prone, although there exist implementations [26, 34] that rely on the lower-level description of the data type of well-formed abstract syntax trees. The canonical way of defining the language is specifying it with a formal grammar that can be used for sampling programs [20, 25, 34]. Since programming languages are typically context-sensitive, the notation has to be an enhanced variant of context-free grammars.

In our previous work on validating refactoring tools [14, 21], we already defined part of Erlang with an attribute grammar and developed a translator that turns attribute grammars into equivalent generators, where a grammar and a generator are equivalent if they generate the same language. This solution allows us to easily define and refine the (weighted) set of (syntactically and semantically valid) Erlang abstract syntax trees, and then it synthesises a QuickCheck generator from the grammar automatically.

However, in this work we made two important improvements to the existing generator. On one hand, we had to tailor the grammar to the language coverage of the semantics, so that we generate programs that we can evaluate in the formal definition. More importantly, we added support for proper shrinking of programs. None of the existing Erlang program generators, including our previous work, addresses how counterexample programs are shrunk. There are default mechanisms for shrinking, but those may not be suitable for syntax tree generators.

What illustrates the complexity of the problem very well is that QuviQ's program generator[2] [34] crashes with "out of memory" every time a program of *generator size* larger than 3 (about a hundred lines of code) needs to be shrunk. Furthermore, if a tiny program is generated with a smaller generator size, it is shrunk to `-module([]).` which is not even a well-formed program. We provide a fairly generic solution by means of the grammar-based generator.

[2] `eqc_erlang_program:module/1`

Syntax tree format. What is an abstract syntax tree in Erlang? In the Erlang/OTP Syntax Tools [15], abstract syntax trees are represented by so-called tagged tuples, where the subtrees of a node are given in a list. For example, the following verbatim shows the syntax tree that represents the concrete expression `{1, 2 + 3}`:

```
{tree,tuple,{attr,0,[],none},
  [{tree,integer,{attr,0,[],none},1},
   {tree,infix_expr,{attr,0,[],none},
     {infix_expr,'+',
       {tree,integer,{attr,0,[],none},2},
       {tree,integer,{attr,0,[],none},3}}}]}
```

In our experience, generating these tuples directly may be error-prone (accidentally constructing tuples that do not represent valid syntax trees), and on the other hand, built-in shrinking may work improperly for them. We can overcome the first issue by generating symbolic calls to the abstract syntax constructors defined in the *erl_syntax* module in the Syntax Tools. Symbolic calls are of the form `{call, M, F, As}`, where `M` and `F` determine the module and function names, while `As` is the argument list. The second issue, shrinking, will be addressed in the following paragraphs.

This snippet shows the symbolic call that reduces to the previous syntax tree:

```
{call, erl_syntax,tuple,
  [[{call, erl_syntax,integer, [1]},
    {call, erl_syntax,infix_expr,
      [{call, erl_syntax,integer, [2]}, '+',
       {call, erl_syntax,integer, [3]}]}]]}
```

Syntax tree generator. To sample programs for testing, we create a generator that produces random symbolic calls building syntax trees. Both the syntactic constructors and the subtrees can be randomized, yielding test programs varying in the used language features and in their structural complexity.

In particular, expressions may be either simple or compound (this alternative can be achieved by using the *oneof*[3] combinator), and compound expressions with arbitrary number of subexpressions can be generated with the *list* combinator: the tuple expression will have an arbitrary number of elements, and the infix expression will contain random operator and operands. With this, we can generalise the previous concrete syntax tree into the following generator function for expressions:

```
expr() ->
  oneof([
    ?LET(I, int(),
         {call, erl_syntax, integer, [I]}),
    ?LET(E1, expr(),
      ?LET(Op, oneof(['+', '-', '*', '/']),
        ?LET(E2, expr(),
             {call, erl_syntax,
                    infix_expr,
                    [E1, Op, E2]}))),
    ?LET(Es, list(expr()),
         {call, erl_syntax, tuple, [Es]})]).
```

In the grammar-based approach, we obtain such a generator from the following piece of grammar (and all other language elements can be formalized with similar notations):

```
expr ->
  int :: {call,erl_syntax,integer, ['$1']})
| {expr} :: {call,erl_syntax,tuple, ['$1']}
| expr infix_op expr ::
  {call,erl_syntax,infix_expr,
                  ['$1','$2','$3']}
```

The grammar description we actually use in our implementation is more sophisticated in a number of aspects: recursive generators are equipped with an explicit recursion limit, all symbols are extended with a set of attributes, and the attributes can be automatically inherited, split and aggregated. Last but not least, the grammar-based generator treats recursive and repeated symbols in a special way from the shrinking point of view.

Shrinking. During property based testing, QuickCheck controls the size of the generated data. It starts with small terms and keeps increasing the size progressively, until it finds a counterexample. Even though it is not desired for us to generate large test programs (which are expensive to run and hard to comprehend), some bugs in the semantics might only be revealed with complex combinations of language features rendered as long program texts. One of the prominent features of property-based testing is the generators' ability to automatically simplify their sampled value and present (locally) minimal counterexamples.

For built-in generators, shrinking is straightforward: for instance, natural numbers are shrunk by decreasing their value toward zero, intervals shrink toward the empty interval, while the built-in list generator starts dropping elements. However, for generators as complex as defining a programming language, this built-in mechanism needs to be be overridden by using the *shrink* and *letshrink* combinators. As Pałka et al. [31] point out, the main problem is to make sure the programs are shrunk structurally whilst maintaining well-formedness. We apply the following shrinking methods in our grammar-based generation:

- *Recursive symbols: shrinking to subtrees.* By default, the *?LET* combinator shrinks bottom-up; in the generator `?LET(Pat, G1, G2)`, "the result is shrunk by first shrinking the value generated by `G1` while the test still fails, then shrinking the value generated by `G2`" [34]. For instance, if an infix expression is defined

[3] In fact, we use the *frequency* combinator to weight the base and recursive cases, and the ratio is a parameter to the testing tool.

like above, the shrinking will try to shrink the left subexpression, then the right subexpression, but will never try to simplify the entire expression into one of its subexpressions, preventing it to become structurally simpler. In our solution, the recursive rules are automatically reordered so that the generator can use the *letshrink* combinator to allow shrinking to any of the recursive subexpressions. This can reduce the structural complexity of the generated tree in a very intuitive way. For instance, the following are valid simplification steps:

```
1 + 2   →   1
1 + 2   →   2
```

- *Recursive symbols: shrinking to base cases.* In some cases, it may prove useful to shrink compound expressions into non-recursive, base cases (by using the *shrink* combinator). This, unlike the previous method, does not keep any of the original subtrees, but generates a new, simpler subtree. Like the above method, this shrinking results in decreasing the structural complexity of the generated program. Ultimately, it simplifies any compound expression toward the smallest literal of its type. For instance:

```
1 + 2   →   0
```

- *Repeated symbols.* Our grammar description supports EBNF-like notations of repetition. If a grammar symbol is enclosed in curly braces (e.g. `{expr}`), it is generated repeatedly in a list — if the symbol is preceded by a value, it is the list size, otherwise we generate a list of an arbitrary size. Actually, these constructs could be generated by using the built-in QuickCheck combinators *list* and *vector*, respectively, but the shrinking of *list* only "drops elements from the list" [34], while the shrinking of the *vector* combinator is not specified in the documentation [34] at all. Thus, we defined our own list generators, which, when shrunk, first simplify the list items as much as possible (preserving the counterexample), and then start dropping the elements. For instance:

```
{1 + 2, true}   →   {1, true}
{1 + 2, true}   →   {2, true}
{1 + 2, true}   →   {0, true}
{1 + 2, true}   →   {0, false}
{1 + 2, true}   →   {0}
{1 + 2, true}   →   {false}
{1 + 2, true}   →   {}
```

Our current implementation can be configured to either shrink the simplified list to single-element lists (e.g. tuples shrink to singleton tuples of their elements and functions are simplified to one of their clauses), or to shrink lists to any of their sublists. We plan to make it configurable whether the elements are shrunk first or they are dropped from the list first.

An example shrinking. Let us discuss a simple example briefly. In one of our experiments, there was a 150LOC program generated, containing three functions and compound expressions with lots of clauses. Like in many similar cases, the shrinking was able to reduce it to just 4 lines of code. The example contained the following *match expression*, surrounded by a number of different expressions:

```
main() ->
  ...
  Q = list_to_tuple([go ||
    YwTEHsy <- [] ++
               [pain, [], [], 'PRESENT'],
    length(YwTEHsy) < 0,
    false or false or not false,
        not (true or true),
        length(YwTEHsy) /= 17]),
  ...
```

This was a counterexample indeed, which uncovered an interesting bug in one of the semantics. The evaluation of the above expression yields the value `list_to_tuple([])` as the guards of the list comprehension are never satisfiable. The tricky part is that one of the guard expressions throws an exception ("bad argument"), but in Erlang that should be treated as *false*. However, the Erlang semantics evaluated this expression to the "bad argument" exception. It is far from trivial to tell the cause of the bug by only looking at the 150LOC program.

Fortunately, once the counterexample was found, shrinking kicked in gear and did a whole lot of simplifications. On one hand, it was able to simplify the function to this single expression that caused the bug, as well as it dropped the rest of the generated functions. The `list_to_tuple` and `length` calls remained while the list generator got simplified to a singleton list with the `'PRESENT'` atom, some of the guards were dropped, only one of the length checks remained, which was just enough to keep the bug in the program. From the shrunk code, we can see that the cause of the error was that the Erlang semantics does not handle guards evaluating to exceptions as *false* guards.

```
main() ->
    Q = list_to_tuple([go ||
      YwTEHsy <- ['PRESENT'],
      length(YwTEHsy) /= 0]),
    0.
```

Why did `0` remain in the function if it is not needed to reproduce the error? In the current grammar definition, each function clause has a number of *statements* and a final *expression* (which is not a matching). This restriction could be easily relaxed by changing the attribute grammar specification.

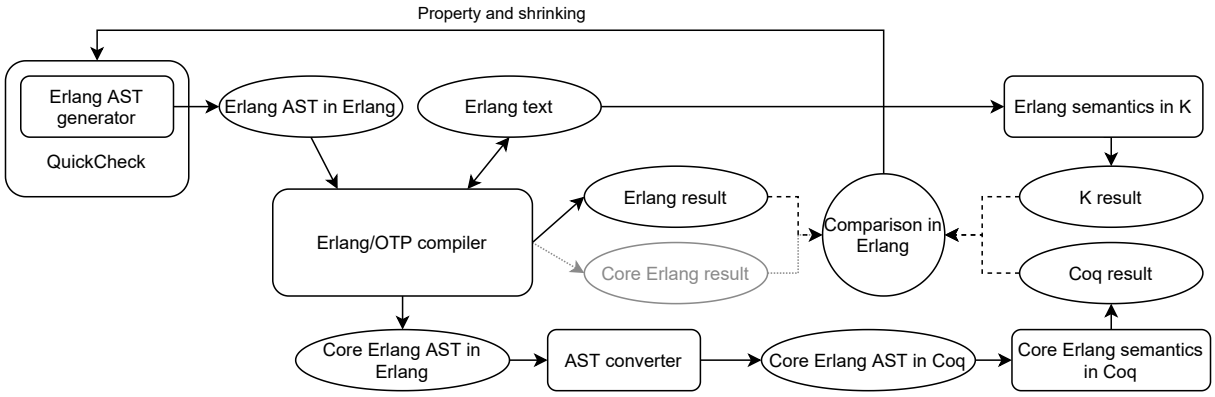

Figure 2. The components of our testing framework

Shrinking strategies. Our current prototype implements an expensive but rather exhaustive shrinking strategy: it allows expressions to shrink to any if their subexpressions, and if none of them breaks the property, compound expressions are replaced with simple base cases. Note that these rules apply to each and every expression in the tree, thousands of syntax nodes in any average size program, resulting in the shrinking time (along with the branches in the tree of possible simplifications) being exponential to the number of syntax nodes. This strategy is likely to find significantly reduced counterexamples, but it is very computation-heavy.

We have found that if shrinking is defined as above, the sampled programs' size needs to be kept down by the generator manually, otherwise the shrinking may take up unreasonable amounts of time. If the programs get larger, shrinking is better be made more aggressive: for instance, one can disable shrinking to subexpressions and reduce expressions to the default value of their type immediately, cutting the shrinking tree and therefore the average time for shrinking significantly. Apparently, this weakened strategy is less expensive, but in some cases it yields considerably more complex counterexamples.

4.3 Implementation

In this section, we give an overview of the structure and the behaviour of our implementation of the semantics validation system (see Figure 2). It is an open-source project and it is available on Github [13]. Basically, it compares the behaviour of the above-mentioned small-step semantics of Erlang implemented in the $\mathbb{K}$ framework and the big-step semantics of Core Erlang implemented in Coq (or the Haskell extraction, see Section 3.3) with each other and with the behaviour of the reference implementation. In property-based testing mode, it relies on QuickCheck (we use QuickCheck Mini 2.01) to sample random programs, check the equivalence property and do counterexample shrinking.

The Erlang/OTP Compiler. The reference implementation Erlang/OTP compiler and interpreter 22.0 is a trusted component and reference for reasoning in our solution. It plays different roles in the testing process:

- Pretty-prints randomly generated Erlang syntax trees
- Preprocesses Erlang code for the Erlang semantics in $\mathbb{K}$ (this step was necessary to unfold macros)
- Translates Erlang to Core Erlang and emits the abstract syntax tree (AST)
- Translates Erlang to BEAM and interprets the bytecode (i.e. executes the program to be tested and provides the result expected from the semantics definitions)
- Compares the outcomes emitted by the semantics with the expected (BEAM) result
- Feeds back the result of the test case into QuickCheck

In the Erlang to Core Erlang translation, we disable optimisation in order not to reduce the original code complexity. We plan to refine this solution and perform the validation with both the optimised and the unoptimised versions of the Core Erlang object code.

Conversions. We also adapted a glue component that helps feeding the Core Erlang program into the Coq implementation of the semantics. Unlike the Erlang semantics in $\mathbb{K}$, the Core Erlang formalisation in Coq does not implement a parser and therefore it takes trees rather than text as input. As we ought to avoid developing a Core Erlang parser in Coq, we opted for pretty-printing the Core Erlang AST into Coq text defining the same AST within Coq (in case if the extracted Haskell code is in use, the converter emits the AST in Haskell).

Orchestration. In our implementation, the validation process is controlled by an Erlang script that coordinates the rest of the components. In particular, it uses the QuickCheck generator to synthesise random programs, invokes Erlang/OTP to obtain the expected result, does the conversions to obtain representations to be fed into the formal semantics in $\mathbb{K}$ and Coq, invokes the semantics, and finally, compares the results.

The script can run either unit test suites or random tests, and it can provide coverage information both for the semantics rules and for the generator grammar. The user can adjust the complexity of the randomly generated programs, and can turn on or off shrinking of counterexamples.

5 Evaluation

In this section, we evaluate our approach from different points of view: we present some faults we found in the semantics and we measure the mean time to failure, which characterises how quickly these faults can be found. We also measure the coverage of the generators and the semantics, and the shrinking time for programs of different sizes. For our testing, we used an average performance laptop (8GB RAM, Intel i5 8th-gen 8-core 1.6GHz processor).

5.1 Bugs found

Naturally, the main goal of our testing was to find faults in the semantics which may be hard to spot by only using hand written test cases. The key errors found (and fixed) include:

- Both semantics handled list append operations incorrectly, but in the same way: neither of them handled improper lists initially (e.g. `[1,2] ++ 3`). However, both semantics were corrected in the same way.
- Value lists and `try` expressions were only partly supported in the Core Erlang semantics; to overcome this issue, a major rework of the semantics was needed.
- The Core Erlang semantics used Coq's `div` function instead of `quot` for the formalisation of Erlang's `div` operator, which carried a slightly different semantics.
- Comparison operators were not implemented for all kinds of values in the Erlang semantics. We could use the comparison formalised in the Core Erlang semantics as a guide to supplement the missing cases.
- The generator expression of the list comprehension in the Erlang semantics was assumed to be a list literal (while it should evaluate to one).
- List normalisation (e.g. transforming the list of `[1,2,3]` to `[1|[2|[3|[]]]]` syntactically) rewrite rules did not apply anywhere in the configuration in the Erlang semantics, which in some cases prevented progress in the small-step derivation.

We also found two potential internal faults in the $\mathbb{K}$ framework (deprecated 3.6 version). In an operator chain, `andalso` (and similarly `orelse`) operators could have been also parsed as the `and` operator and `also` atom. We avoided this parsing ambiguity by marking the syntax of `andalso` preferred, that is the parsing rule for `andalso` will be used whenever there are other matching rules; this error is fixed in version 5.0. Secondly, a chain of infix operators and operands (without parentheses), where some of the operands in the middle of the chain are function calls, resulted in parsing errors.

5.2 Coverage

To measure the effectiveness of our testing, first we present the coverage of it, that is the coverage of the generator (i.e. what kind of expressions are generated) and rule coverage for both semantics.

Generator coverage. Our attribute grammar-based program synthesiser supports the generation of a representative subset of Erlang. It generates a module with a number of top-level functions that use the following language constructs: literals and compound expressions for integers, booleans, atoms, tuples and lists (including list comprehensions), local variables, *match* and case expressions, function application. In addition, a representative set of built-in functions (e.g. arithmetic, boolean and cast operators) are generated. In fact, these are the language elements that are implemented by both of the semantics, and we keep the generator and the semantics in sync in terms of language coverage.

Semantics coverage. For both semantics, we measured rule coverage, that is how many rules of the semantics were used compared to the number of all rules. These rules are the rewrite rules in case of the Erlang semantics, while the different branches of the recursive functional semantics are considered as derivation rules in case of Core Erlang (moreover, the use of built-in functions is also included in both cases). We used an additional cell in the configuration, to log how many times different derivation rules were used. The statistics are described in Table 1.

	Erlang semantics	**Core Erlang semantics**
Number of rules	69	70
Coverage	75.36%	75.71%
Number of exception-free rules	51	53
Exception-free coverage	86.28%	92.45%
Test suite coverage	100%	80%

Table 1. Semantics rule coverage after 1000 tests

We note that the generated programs are static semantically valid, which reduces (for some expressions it eliminates) the probability of generating expressions that yield exceptions. This is one of the reasons why we do not get full rule coverage (only 75.36%) with the random testing. The other reason is that some expressions (e.g. nameless functions and their application) are supported in the Core Erlang semantics, but have not been generated in the random testing yet.

After finding and correcting the faults and misconceptions in the semantics, we assembled a test suite (including well-known sequential Erlang benchmarks [16, 36]) that provides maximal rule coverage on the Erlang semantics. The advantage of using official benchmarks is that they contain small but meaningful, pragmatic examples. This test suite uncovered that the Erlang semantics lacked the formalisation of some exceptions. In this case, the semantics of Core Erlang was useful again to supplement these.

Semantics	Error type	Time (s)	Number of tests
Core Erlang	Wrong addition	148.523	38.4
Core Erlang	Missing rem	17.812	4,4
Core Erlang	Wrong case guard semantics	4.56	1
Core Erlang	Environment is not updated in let	5.705	1.4
Core Erlang	Div instead of quot	800.445	204
Core Erlang	Wrong app	214.048	54.64
Core Erlang	Division by zero	10.98	2.65
Erlang	List normalisation without anywhere	15.886	4
Erlang	Missing rem	15.556	4
Erlang	Wrong addition	21.756	4.5
Erlang	Wrong rewrite rule for app	14.052	3.25
Erlang	List comprehension wrong guard	317.24	81.56
Erlang	Generator of list comprehension assumed to be a list	22.473	5.68
Erlang	Wrong tuple comparison	205.497	53

Table 2. Mean time to failure using few hundred LOC programs

While we reached the maximal rule coverage on the Erlang semantics (at least together with the test suite), we cannot say the same about the Core Erlang semantics, where only 80% of the rules were covered by the test suite.

Although every kind of expression (formalised in the Core Erlang semantics) is translated from Erlang (based on our observations), this does not imply maximal rule coverage of the Core Erlang semantics. Actually, there is a simple explanation: during the translation process of the Erlang code, among other transformations, the evaluation order of parameters is explicitly determined by nested `let` expressions in Core Erlang (see Section 3) and for `case` expressions, the compiler generates a "catch-all" clause where an exception is thrown. That is, we cannot produce Core Erlang code translated from Erlang such that an exception occurs during the evaluation of parameters, or because there was no matching clause in the case expression; for these corner cases we needed to write tests by hand in Core Erlang which can be included in this testing process only for the semantics of Core Erlang.

The general message of this observation is that when there are two languages, say L_1 and L_2, and L_1 can be translated to L_2, it should be considered whether all kinds of expressions or statements can be translated from L_1 to L_2. Moreover, even if this is the case, it does not necessarily mean that full rule coverage on L_1 semantics guarantees full rule coverage on L_2 semantics.

5.3 Execution Speed

The other important aspect of effectiveness is execution speed. We re-injected some of the bugs mentioned above and we measured the average time and the average number of test cases until the error was found (mean time to failure). We provide two different measurements in the next paragraphs (the terminology in Table 3 will be used to denote program complexity). Thereafter, we also discuss the shrinking speed.

Size	LOC	AST node count	AST height
small	19.65	407.7	123.27
medium	83.63	2233.29	680.06
large	161.02	4033.29	1227.94

Table 3. Size terminology

Generating only large programs. The statistics about the re-injected bugs are described in Table 2. These measurements represent our initial approach only using large random programs, and do not include the time of shrinking.

The mean times as well as the number of tests needed to trigger particular bugs shows significant differences. This is because some of the bugs can be hidden (e.g. the difference between `div` and `quot` during evaluation only appears when one of their parameters is negative) while others are easy to encounter (e.g. missing `rem` operation appears every time such a construct is evaluated).

To speed up execution, the Core Erlang semantics was extracted to Haskell; moreover, the generator for random programs was extended with options to configure the shape of generated programs (e.g. size, number of binary operators and compound expressions, etc.):

- Recursion depth limit determines how many times a recursive rule can be used within an expression.
- Expression size determines the complexity of base values and the size of collections.
- Recursive rule probability determines the probability of using recursive rules over base cases.

Testing with programs of different sizes. Using the extracted Haskell semantics and adjusting the options mentioned above and the recursion depth limit of the generator (which allows us to set the size of generated programs), we managed to speed up the process to find specific faults. We present our preliminary results in Table 4 targeting two specific bugs.

Semantics	Error type	Program size	Time (s)	Number of tests
Core Erlang	Missing rem	small	26.96	6.44
Core Erlang	Missing rem	medium	12.89	3.63
Core Erlang	Missing rem	large	12.66	3.33
Core Erlang	Div instead of quot	small	560.89	167.33
Core Erlang	Div instead of quot	medium	250.11	62.5
Core Erlang	Div instead of quot	large	218.47	50.83

Table 4. Mean time to failure using programs of different sizes

As the preliminary data suggests, we managed to improve our initial corresponding results, even in case of generating large programs. Moreover, while program size grows, the number of tests and time to find the failure decreases. This is because the execution path contains more and more language constructs to evaluate. On the other hand, the decrease in time is less and less significant as the size grows, so it does not worth testing with very large programs.

Program size	Shrinking time (s)
small	126.83
medium	177.55
large	978.47

Table 5. Mean shrinking time for programs of different sizes

Shrinking speed. We have measured the shrinking speed of the counterexamples depending on their size (Table 5). Not surprisingly, these showed exponential growth as the size increased. From the point of view of execution times, it is worth generating medium-sized programs, and shrinking them to discover well-hidden errors in the semantics; however, for simpler bugs, it is more efficient to use small random programs which can be understood without shrinking.

We should also note that the use of the inductive big-step semantics would have increased the execution time by orders of magnitude, given that the computer would not have run out of memory first. Just to mention concrete numbers, for a smaller Erlang program (around one hundred lines of code) which contained only arithmetic operations, the Core Erlang inductive semantics execution run for 30 minutes on average.

6 Conclusion and Future Work

In conclusion, we have described an approach to validating formal semantics by testing them against each other and a reference implementation in a property-based setting, combining a number of semantics validation methods. The grammar-based random generator for test programs uses well-defined rules for simplifying counterexamples. We applied our approach to testing Erlang and Core Erlang semantics against the reference implementations of these languages, also covering technical issues such as the efficient execution of formal semantics. We evaluated our approach, measuring its performance and coverage.

We found that our testing approach is effective and useful. Property-based testing excels in finding "sneaky" problems (e.g. using `div` or `quot` in the Core Erlang semantics to formalise Erlang's `div`), and the automatic simplification of counterexamples is invaluable in these cases. In addition, cross-testing allows us to solve similar problems in both semantics the same way, and one semantics definition can aid the rectification of the other.

We have also found some disadvantages of the approach. Full rule coverage for the higher level language semantics does not ensure full rule coverage in the translation language semantics; in such cases, additional tests must be written for the latter (Core Erlang in our case). Furthermore, when applying moderate shrinking rules, simplification of large programs takes an unreasonable amount of time; this needs tweaking the generator to promote smaller syntax trees.

We are working on speeding up the testing process: we are porting the Erlang semantics to the latest version of the $\mathbb{K}$ framework, and fine-tuning the generator to get more exhaustive coverage with smaller but more complex programs (see preliminary data in Table 4). We will also investigate QuickCheck's parallel testing capabilities for scalability.

In the future, we plan to increase the language coverage of the semantics and the generator, including concurrency features of Erlang and Core Erlang. Our long-term goal is to use the validated formalisations of these languages for reasoning about behaviour-preservation of program transformations.

Acknowledgments

This work has been supported by the European Union, co-financed by the European Social Fund projects EFOP-3.6.2-16-2017-00013, "Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications (3IN)" and "Integrált kutatói utánpótlás-képzési program az informatika és számítástudomány diszciplináris területein (Integrated program for training new generation of researchers in the disciplinary fields of computer science)", No. EFOP-3.6.3-VEKOP-16-2017-00002.

It has also received support from the National Research, Development and Innovation Fund of Hungary, financed by the Thematic Excellence Programme TKP2020-NKA-06 (National Challenges Subprogramme) scheme, project "Application Domain Specific Highly Reliable IT Solutions".

Supported by the ÚNKP-20-4 New National Excellence Program of the Ministry for Innovation and Technology from the source of the National Research, Development and Innovation Fund.

References

[1] Péter Bereczky, Dániel Horpácsi, and Simon Thompson. 2020. A Proof Assistant Based Formalisation of a Subset of Sequential Core Erlang. In *Trends in Functional Programming*, Aleksander Byrski and John Hughes (Eds.). Springer International Publishing, Cham, 139–158. https://doi.org/10.1007/978-3-030-57761-2_7

[2] Péter Bereczky, Dániel Horpácsi, and Simon J. Thompson. 2020. Machine-Checked Natural Semantics for Core Erlang: Exceptions and Side Effects. In *Proceedings of Erlang 2020*. ACM, 1–13. https://doi.org/10.1145/3406085.3409008

[3] Péter Bereczky, Dániel Horpácsi, and Simon Thompson. 2020. A Comparison of Big-step Semantics Definition Styles. arXiv:2011.10373 [cs.PL]

[4] Sandrine Blazy. 2007. Experiments in validating formal semantics for C. In *C/C++ Verification Workshop*. Oxford, United Kingdom, 95–102. https://hal.inria.fr/inria-00292043

[5] Sandrine Blazy and Xavier Leroy. 2009. Mechanized Semantics for the Clight Subset of the C Language. *Journal of Automated Reasoning* 43, 3 (Jul 2009), 263–288. https://doi.org/10.1007/s10817-009-9148-3

[6] Martin Bodin et al. 2014. A Trusted Mechanised JavaScript Specification. *SIGPLAN Not.* 49, 1 (Jan. 2014), 87–100. https://doi.org/10.1145/2578855.2535876

[7] Martin Bodin, Tomás Diaz, and Éric Tanter. 2018. A Trustworthy Mechanized Formalization of R. *SIGPLAN Not.* 53, 8 (Oct. 2018), 13–24. https://doi.org/10.1145/3393673.3276946

[8] Richard Carlsson et al. 2004. *Core Erlang 1.0.3 language specification*. Technical Report. https://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf

[9] Francesco Cesarini and Simon Thompson. 2009. *ERLANG Programming* (1st ed.). O'Reilly Media, Inc.

[10] Arthur Charguéraud. 2013. Pretty-Big-Step Semantics. In *Programming Languages and Systems*, Matthias Felleisen and Philippa Gardner (Eds.). Springer. https://doi.org/10.1007/978-3-642-37036-6_3

[11] Coq documentation 2021. *Ltac documentation*. Retrieved April 21st, 2021 from https://coq.inria.fr/refman/proof-engine/ltac.html

[12] Coq introduction 2021. *A short introduction to Coq*. Retrieved April 21st, 2021 from https://coq.inria.fr/a-short-introduction-to-coq

[13] Cross-testing semantics 2021. *Erlang semantics testing*. Retrieved April 21st, 2021 from https://github.com/harp-project/erlang-semantics-testing

[14] Dániel Drienyovszky, Dániel Horpácsi, and Simon Thompson. 2010. Quickchecking Refactoring Tools. In *Proceedings of Erlang '10*. ACM, 75–80. https://doi.org/10.1145/1863509.1863521

[15] Erlang/OTP Syntax Tools 2021. *Erlang/OTP Syntax Tools*. Retrieved April 21st, 2021 from http://erlang.org/documentation/doc-11.1.4/lib/syntax_tools-2.4/doc/pdf/syntax_tools-2.4.pdf

[16] ErLLVM-bench 2021. *ErLLVM-bench*. Retrieved April 21st, 2021 from https://github.com/cstavr/erllvm-bench

[17] Daniele Filaretti and Sergio Maffeis. 2014. An Executable Formal Semantics of PHP. In *ECOOP 2014*, Richard Jones (Ed.). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-662-44202-9_23

[18] Paolo Guagliardo and Leonid Libkin. 2017. A Formal Semantics of SQL Queries, Its Validation, and Applications. *Proc. VLDB Endow.* 11, 1 (Sept. 2017), 27–39. https://doi.org/10.14778/3151113.3151116

[19] F. Hebert. 2019. *Property-Based Testing with PropEr, Erlang, and Elixir: Find Bugs Before Your Users Do*. Pragmatic Bookshelf. https://books.google.hu/books?id=SGI0uQEACAAJ

[20] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2018. Grammarinator: A Grammar-Based Open Source Fuzzer. In *Proceedings of A-TEST 2018*. ACM, 45–48. https://doi.org/10.1145/3278186.3278193

[21] Dániel Horpácsi. 2018. *Verification and Application of Program Transformations*. Ph.D. Dissertation. Eötvös Loránd University.

[22] Xuan Huang. 2019. A Mechanized Formalization of the WebAssembly Specification in Coq. https://www.cs.rit.edu/~mtf/student-resources/20191_huang_mscourse.pdf

[23] K projects 2021. *K framework project catalogue*. Retrieved April 21st, 2021 from https://github.com/kframework

[24] Judit Kőszegi. 2018. KErl: Executable semantics for Erlang. *CEUR Workshop Proceedings* 2046 (2018), 144–160. http://ceur-ws.org/Vol-2046/koszegi.pdf

[25] P. M. Maurer. 1990. Generating test data with enhanced context-free grammars. *IEEE Software* 7, 4 (1990), 50–55. https://doi.org/10.1109/52.56422

[26] Module proper_erlang_abstract_code 2021. *PropEr generator of abstract code*. Retrieved April 21st, 2021 from https://github.com/proper-testing/proper/blob/master/src/proper_erlang_abstract_code.erl

[27] Natural Semantics for Core Erlang 2021. *Core Erlang Formalization*. Retrieved April 21st, 2021 from https://github.com/harp-project/Core-Erlang-Formalization

[28] Martin Neuhäußer and Thomas Noll. 2007. Abstraction and model checking of Core Erlang programs in Maude, In Proceedings of the 6th International Workshop on Rewriting Logic and its Applications. *ENTCS* 176, 4, 147–163. https://doi.org/10.1016/j.entcs.2007.06.013

[29] Official Core Erlang Parser 2018. *Core Erlang YECC Parser Grammar*. Retrieved April 21st, 2021 from https://github.com/erlang/otp/blob/master/lib/compiler/src/core_parse.yrl

[30] Scott Owens et al. 2016. Functional Big-Step Semantics. In *Programming Languages and Systems*, Peter Thiemann (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 589–615. https://doi.org/10.1007/978-3-662-49498-1_23

[31] Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an Optimising Compiler by Generating Random Lambda Terms. In *Proceedings of AST'11*. ACM, New York, NY, USA, 91–97. https://doi.org/10.1145/1982595.1982615

[32] Árpád Perényi and Jan Midtgaard. 2020. Stack-Driven Program Generation of WebAssembly. In *Programming Languages and Systems*, Bruno C. d. S. Oliveira (Ed.). Springer International Publishing, Cham, 209–230. https://doi.org/10.1007/978-3-030-64437-6_11

[33] Joe Gibbs Politz et al. 2012. A Tested Semantics for Getters, Setters, and Eval in JavaScript. *SIGPLAN Not.* 48, 2 (oct 2012), 1–16. https://doi.org/10.1145/2480360.2384579

[34] QuickCheck Documentation 2017. *QuviQ QuickCheck*. Retrieved January 7th, 2021 from http://quviq.com/documentation/eqc

[35] Ian Roessle, Freek Verbeek, and Binoy Ravindran. 2019. Formally Verified Big Step Semantics out of x86-64 Binaries. In *Proceedings of CPP 2019*. ACM, New York, NY, USA, 181–195. https://doi.org/10.1145/3293880.3294102

[36] Konstantinos Sagonas, Chris Stavrakakis, and Yiannis Tsiouris. 2012. ErLLVM: An LLVM Backend for Erlang. In *Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang Workshop*. ACM, 21–32. https://doi.org/10.1145/2364489.2364494

[37] The BEAM Book 2020. *The Erlang Runtime System*. Retrieved April 21st, 2021 from https://github.com/happi/theBeamBook/releases/download/0.0.14.fix/beam-book.pdf

[38] Zheng Yang and Hang Lei. 2018. Lolisa: Formal Syntax and Semantics for a Subset of the Solidity Programming Language. arXiv:1803.09885 [cs.PL]