# Concolic Execution in Functional Programming by Program Instrumentation[⋆]

Adrián Palacios[⋆⋆] and Germán Vidal

MiST, DSIC, Universitat Politècnica de València
Camino de Vera, s/n, 46022 Valencia, Spain
{apalacios,gvidal}@dsic.upv.es

**Abstract.** Concolic execution, a combination of concrete and symbolic execution, has become increasingly popular in recent approaches to model checking and test case generation. In general, an interpreter of the language is augmented in order to also deal with symbolic values. In this paper, in contrast, we present a lightweight approach that is based on a program instrumentation. Basically, the execution of the instrumented program in a standard environment produces a sequence of events that can be used to reconstruct the associated symbolic execution. We discuss the usefulness of our approach for test case generation.

## 1 Introduction

Software testing is one of the most widely used approaches for program validation. In this context, symbolic execution [9] was introduced as an alternative to random testing —which usually achieves a poor code coverage— or the complex and time-consuming design of test-cases by the programmer or software tester. In symbolic execution, one replaces the input data by symbolic values. Then, at each branching point of the execution, all feasible paths are explored and the associated contraints on symbolic values are stored. Symbolic states thus include a so called *path condition* with the constraints stored so far. Test cases are finally produced by solving the constraints in the leaves of the symbolic execution tree, which is typically incomplete since the number of states is infinite.

Unfortunately, both the huge search space and the complexity of the constraints make test case generation based on symbolic execution difficult to scale. For instance, most approaches only deal with linear constraints. Therefore, as soon as a non-linear constraint is collected, the execution of this branch is terminated in order to ensure soundness, giving rise to a poor coverage in many cases.

[⋆] This work has been partially supported by the EU (FEDER) and the Spanish *Ministerio de Economía y Competitividad* under grant TIN2013-44742-C4-1-R and by the *Generalitat Valenciana* under grant PROMETEOII2015/013.

[⋆⋆] Partially supported by the the EU (FEDER) and the Spanish *Ayudas para contratos predoctorales para la formación de doctores de la Secretaría de Estado de Investigación, Desarrollo e Innovación del Ministerio de Economía y Competitividad* under FPI grant BES-2014-069749.

Concolic execution [6, 11] is a recent proposal that combines *conc*rete and symb*olic* execution, and overcomes some of the drawbacks of previous approaches. Essentially, concolic execution takes a program and some (initially random) concrete input data, and performs both a concrete and a symbolic execution that mimics the steps of the concrete execution. In this context, symbolic execution is simpler since we know the execution path that must be followed (the same of the concrete execution). Moreover, if the current constraint becomes too complex (e.g., non-linear), we can still push some concrete data from the concrete execution, thus symplifying it and often allowing the symbolic execution to continue. In general, this technique —called concolic *testing*— is being used both for model checking and test-case generation (see, e.g., SAGE [7] and Java Pathfinder [10]). Test cases produced with this technique usually achieve a better code coverage than previous approaches based solely on symbolic execution. Moreover, it scales up better to complex or large programs.

Despite its popularity in the imperative and object-oriented programming paradigms, we can only find a few preliminary approaches to concolic testing in the context of functional and logic programming. On the one hand, [12] introduced a formalization of both concrete and symbolic execution for the functional and concurrent language Erlang [2], but the concolic testing procedure was barely sketched. More recently, [5] presented the design and implementation of a concolic testing tool for a functional subset of Erlang (i.e., the concurrency features are not considered in the paper). The tool, called `CutEr`, is publicly available from `https://github.com/aggelgian/cuter`. On the other hand, within logic programming, [13] presented a first approach to concolic testing in Prolog. However, this scheme was only aimed at *statement* coverage and, thus, it is simpler than other approaches aimed at full path coverage (like, e.g., [5]).

However, the essential component of all these approaches is an interpreter augmented to also deal with symbolic values. Besides involving a huge implementation effort, these approaches are difficult to maintain and usually do not scale up well to medium and large applications.

In contrast, in this paper, we present a lightweight approach that is mainly based on *instrumenting* the source program. Here, we deal with a simple functional eager language that can be seen as a purely functional subset of Erlang. First, we present an *instrumented semantics*, a conservative extension of the standard semantics that also produces a sequence of events that suffice to reconstruct the associated symbolic execution. Then, we present a program instrumentation such that the execution of the instrumented program with the standard semantics produces the same events as the original program with the instrumented semantics.

## 2 The Language

In this section, we introduce the language considered in this paper. Our language is inspired in the concurrent functional language Erlang [2], which has a number of distinguishing features, like dynamic typing, concurrency via asynchronous

$$
\begin{array}{rcl}
\mathit{pgm} & ::= & \mathrm{a}/n = \mathsf{fun}\ (X_1,\ldots,X_n) \to e. \ \mid\ \mathit{pgm}\ \mathit{pgm} \\
\mathsf{Exp} \ni e & ::= & \mathrm{a} \mid X \mid [\,] \mid [e_1|e_2] \mid \{e_1,\ldots,e_n\} \mid \mathsf{apply}\ e_0\ (e_1,\ldots,e_n) \\
 & \mid & \mathsf{case}\ e\ \mathsf{of}\ \mathit{clauses}\ \mathsf{end} \mid \mathsf{let}\ p = e_1\ \mathsf{in}\ e_2 \mid \mathsf{do}\ e_1\ e_2 \\
\mathit{clauses} & ::= & p_1 \to e_1; \ldots; p_n \to e_n \\
\mathsf{Pat} \ni p & ::= & [p_1|p_2] \mid [\,] \mid \{p_1,\ldots,p_n\} \mid \mathrm{a} \mid X \\
\mathsf{Value} \ni v & ::= & [v_1|v_2] \mid [\,] \mid \{v_1,\ldots,v_n\} \mid \mathrm{a}
\end{array}
$$

**Fig. 1.** Core Erlang Syntax

message passing or hot code loading, that make it especially appropriate for distributed, fault-tolerant, soft real-time applications. Erlang's popularity is growing today due to the demand for concurrent services. But this popularity will also demand the development of powerful testing and verification techniques, thus the opportunity of our research.

Despite the fact that we plan to deal with full Erlang in the future, in this paper we only consider a functional subset of *Core Erlang* [3], an intermediate language used internally by the compiler, similarly to [5].

The basic objects of the language are variables (denoted by $X, Y, \ldots \in \mathsf{Var}$), atoms (denoted by a, b, ...) and constructors (which are fixed in Erlang to lists, tuples and atoms); defined functions are named using atoms too (we will use, e.g., $\mathrm{f}/n, \mathrm{g}/m, \ldots$). The syntax for Core Erlang programs and expressions obeys the rules shown in Figure 1. Programs are sequences of function definitions. Each function $\mathrm{f}/n$ is defined by a rule $\mathsf{fun}\ (X_1,\ldots,X_n) \to e.$ where $X_1,\ldots,X_n$ are distinct variables and the body of the function, $e$, can be an atom, a process identifier, a variable, a list, a tuple, a function application, a case distinction, a let expression or a do construct (i.e., $\mathsf{do}\ e_1\ e_2$ evaluates sequentially $e_1$ and, then, $e_2$, so the value of $e_1$ is lost). Patterns are made of lists, tuples, atoms, and variables. Values are similar to patterns but cannot contain variables.

*Example 1.* Consider the Erlang function (left) and its translation to Core Erlang (right) shown in Figure 2, where some minor simplifications have been applied. Observe that Erlang's sequence operator "," is translated to a do operator when no value should be passed (using pattern matching) to the next element in the sequence, and to a let expression otherwise. Note also that, despite the fact that this is not required by the syntax, some function applications are *flattened* in order to avoid nested applications. For this purpose, some additional let expressions are introduced. Moreover, additional default alternatives are added to each case expression in order to catch pattern matching errors, so it is common to have overlapping patterns in the clauses of a case construct.

As we will see later, for our instrumentation to be correct, we require some additional constraints on the syntax of programs. Basically, we require the following:

– both the name and the arguments of a function application must be patterns,

```
f(X, Y) → g(X),
          case h(X) of
            a → A = h(Y),
                g(A);
            b → g(h([]))
          end.
```

```
f/2 = fun (X, Y) → do apply g/1 (X),
                      case apply h/1 (X) of
                        a → let Z = apply h/1 (Y)
                              in apply g/1 (Z);
                        b → let V = apply h/1 ([])
                              in apply g/1 (V);
                        W → fail
                      end.
```

**Fig. 2.** Erlang function and its translation to Core Erlang

$$
\begin{array}{rcl}
\mathit{pgm} & ::= & \mathrm{a}/n = \mathsf{fun}\ (X_1,\ldots,X_n) \to \mathsf{let}\ X = e\ \mathsf{in}\ X. \ \mid\ \mathit{pgm}\ \mathit{pgm} \\
\mathsf{Exp} \ni e & ::= & \mathrm{a} \ \mid\ X \ \mid\ [\,] \ \mid\ [p_1|p_2] \ \mid\ \{p_1,\ldots,p_n\} \ \mid\ \mathsf{let}\ p = e_1\ \mathsf{in}\ e_2 \ \mid\ \mathsf{do}\ e_1\ e_2 \\
& \mid & \mathsf{let}\ p = \mathsf{apply}\ p_0\ (p_1,\ldots,p_n)\ \mathsf{in}\ e \ \mid\ \mathsf{let}\ p_1 = \mathsf{case}\ p_2\ \mathsf{of}\ \mathit{clauses}\ \mathsf{end}\ \mathsf{in}\ e \\
\mathit{clauses} & ::= & p_1 \to e_1; \ldots ; p_n \to e_n \\
\mathsf{Pat} \ni p & ::= & [p_1|p_2] \ \mid\ [\,] \ \mid\ \{p_1,\ldots,p_n\} \ \mid\ \mathrm{a} \ \mid\ X \\
\mathsf{Value} \ni v & ::= & [v_1|v_2] \ \mid\ [\,] \ \mid\ \{v_1,\ldots,v_n\} \ \mid\ \mathrm{a}
\end{array}
$$

**Fig. 3.** Flat language syntax

- the return value of a function must always be a pattern,
- the argument of a case expression must be a pattern, and
- both function applications and case expressions can only occur in the right-hand side of a let expression.

The new constraints are needed in order to keep track of the intermediate values returned by expressions. These values are stored in a pattern, which can then be used by other expressions or returned as the result of a function application.

The restricted syntax is shown in Figure 3. In the following, we call the programs fulfilling this syntax *flat programs*. In practice, one can transform (purely functional) Core Erlang programs to our flat syntax using a simple pre-processing transformation.

Furthermore, in the flat language we also require the *bound* variables in the body of the functions to have unique, fresh names. This is not strictly necessary, but it simplifies the presentation by avoiding the use of context scopes associated to every let expression, etc. (as in [8], where the *last* binding of a variable in the environment should be considered to ensure that the right scope is used). In the following, we denote with $\overline{o_n}$ a sequence of objects $o_1, \ldots, o_n$. $\mathcal{V}ar(e)$ denotes the set of variables appearing in an expression $e$; moreover, we say that $e$ is *ground* if $\mathcal{V}ar(e) = \emptyset$.

Here, we use the function $\mathsf{bv}$ to gather the bound variables of an expression:

**Definition 1 (bound variables, $\mathsf{bv}$).** *Let $e$ be an expression. The function $\mathsf{bv}(e)$ returns the set of bound variables of $e$ as follows:*

$$\mathsf{bv}(e) = \begin{cases} \{\,\} & \textit{if } e \in \mathsf{Pat} \\ \mathcal{V}ar(p) \cup \mathsf{bv}(e') & \textit{if } e \equiv \mathsf{let}\ p\ = \mathsf{apply}\ p_0\ (p_1, \ldots, p_n)\ \mathsf{in}\ e' \\ \mathcal{V}ar(p_0) \cup \ldots \cup \mathcal{V}ar(p_n) & \textit{if } e \equiv \mathsf{let}\ p_0 = \mathsf{case}\ p\ \mathsf{of}\ \overline{p_n \to e_n}\ \mathsf{end}\ \mathsf{in}\ e' \\ \ \cup\ \mathsf{bv}(e_1) \cup \ldots \cup \mathsf{bv}(e') & \\ \mathcal{V}ar(p) \cup \mathsf{bv}(e_1) \cup \mathsf{bv}(e_2) & \textit{if } e \equiv \mathsf{let}\ p = e_1\ \mathsf{in}\ e_2 \\ \mathsf{bv}(e_1) \cup \mathsf{bv}(e_2) & \textit{if } e \equiv \mathsf{do}\ e_1\ e_2 \end{cases}$$

*where, in the fourth case, we assume that $e_1$ is neither an application nor a case expression (i.e., it is a pattern or another let expresssion).*

## 3 Instrumented Semantics

In this section, we present an instrumented semantics for flat programs that produces a sequence of events that will suffice to reconstruct the associated symbolic execution. Essentially, we need to keep track of function calls, returns, let bindings and case selections. Formally,

- The first event, $\mathsf{call}(params, vars, p, [p_1, \ldots, p_n])$, is associated to a function application $\mathsf{let}\ p = \mathsf{apply}\ p_0\ (p_1, \ldots, p_n)\ \mathsf{in}\ e$. Here, *params* and *vars* refer to the function where the application occurs; *params* is the list with the current function parameters, and *vars* is the list with the bound variables. Then, $[p_1, \ldots, p_n]$ are the arguments of the function call, and $p$ will be used to store the return value of the function call.
- The second event is $\mathsf{exit}(params, vars, p)$, where $p$ is the pattern used to store the return value of the function body.
- The next event is $\mathsf{bind}(params, vars, p, p')$, which binds the pattern $p$ from a generic let expression (i.e., a let expression whose argument is neither an application nor a case expression) to the return value $p'$ of that expression (see function $\mathsf{ret}$ below).
- Finally, for each expression of the form

  $$\mathsf{let}\ p = \mathsf{case}\ p_0\ \mathsf{of}\ p_1 \to e_1; \ldots; p_n \to e_n\ \mathsf{end}\ \mathsf{in}\ e$$

  we have two associated events. The first one is

  $$\mathsf{case}(params, vars, i, p_0, p_i, [(p_0, 1, p_1), \ldots, (p_0, n, p_n)])$$

  Here, we store the position of the selected branch, $i$, the case argument $p_0$, the selected pattern $p_i$, as well a list with all case branches, which will become useful for producing alternative input data in concolic testing. The second event is $\mathsf{exitcase}(params, vars, p, p')$, where $p'$ is the return value of the selected branch (see below).

Before presenting the instrumented semantics, we need the following auxiliary function that identifies the *return value* of an expression:

**Definition 2 (return value, ret).** *Let $e$ be an expression. We let $\mathsf{ret}(e)$ denote the return value of $e$ as follows:*

$$\mathsf{ret}(e) = \begin{cases} e & \textit{if } e \in \mathsf{Pat} \\ \mathsf{ret}(e') & \textit{if } e \equiv \mathsf{let}\ p\ = \mathsf{apply}\ p_0\ (p_1,\ldots,p_n)\ \mathsf{in}\ e' \\ \mathsf{ret}(e') & \textit{if } e \equiv \mathsf{let}\ p_0 = \mathsf{case}\ p\ \mathsf{of}\ \overline{p_n \to e_n}\ \mathsf{end}\ \mathsf{in}\ e' \\ \mathsf{ret}(e_2) & \textit{if } e \equiv \mathsf{let}\ p = e_1\ \mathsf{in}\ e_2 \\ \mathsf{ret}(e_2) & \textit{if } e \equiv \mathsf{do}\ e_1\ e_2 \end{cases}$$

*where, in the fourth case, we assume that $e_1$ is neither an application nor a case expression (i.e., it is a pattern or another let expresssion).*

The instrumented semantics for flat programs is formalized in Figure 4 following the style of a natural —big-step— semantics [8]. Observe that we do not need *closures* (as it is common in the natural semantics [8]) since we do not allow fun expressions in the body of a function in this paper. Here, we use an *environment* $\theta$ —i.e., a mapping from variables to patterns— because we need to know the static values of the variables for the instrumentation (e.g., we use the case argument that appears statically in the program, rather than the instantiated run time value). The main novelty is that, for the instrumentation, we also need to keep track of the function where an expression occurs. For this purpose, we also introduce a simple context $\pi$ that stores this information, i.e., for a given function $\mathsf{fun}\ (X_1,\ldots,X_n) \to e$ we store a tuple $\langle [X_1,\ldots,X_n], [\mathsf{bv}(e)]\rangle$. The environment is only updated in function applications, where $[\mathsf{bv}(e)]$ denotes a list with the variables returned by $\mathsf{bv}(e)$.

Let us briefly explain the rules of the semantics (events are depicted in blue). Statements have the form $\pi, \theta \vdash e \Downarrow_\tau p$, where $\pi$ is the aforementioned context, $\theta$ is a substitution (the environment), $e$ is an expression, $\tau$ is a sequence of events, and $p$ is a pattern —the value of $e$.

The first rule deals with patterns (including variables, atoms, tuples and lists). Here, the evaluation just proceeds by applying the current environment $\theta$ to the pattern $p$ to bind its variables (if any), which is denoted by $p\theta$. The associated sequence of events is $\epsilon$ denoting an empty sequence.

The next rule deals with function applications. In this case, the context is necessary for setting the first and second parameters of call and exit events. Note that since we only consider flat programs, both the function name and the arguments are patterns; thus, their evaluation amounts to binding their variables using the current environment, which explains why the associated sequences of events are $\epsilon$. Note also that, when recursively evaluating the body of the function, we update the context with the information of the function called. The bound variables are collected using the function bv; and, as mentioned before, in the flat language we assume that they all have different, fresh names. Observe that the subcomputation for evaluating the body of the function called is preceeded by the call event and followed by an exit event. Here, we use the auxiliary function

$$\frac{}{\pi, \theta \vdash p \Downarrow_{\epsilon} p\theta}$$

$$\frac{\begin{array}{c}\langle vs, ps\rangle, \theta \vdash p_0 \Downarrow_{\epsilon} \mathrm{f}/m \quad \ldots \quad \langle vs, ps\rangle, \theta \vdash p_m \Downarrow_{\epsilon} p'_m \\ \langle [\overline{Y_m}], [\mathsf{bv}(e_2)]\rangle, \theta \cup \sigma \vdash e_2 \Downarrow_{\tau_1} p' \qquad \langle vs, ps\rangle, \theta \cup \sigma' \vdash e \Downarrow_{\tau_2} p''\end{array}}{\langle vs, ps\rangle, \theta \vdash \mathsf{let}\ p = \mathsf{apply}\ p_0\ (\overline{p_m})\ \mathsf{in}\ e \Downarrow_{\mathsf{call}(vs,ps,p,[\overline{p_m}])+\tau_1+\mathsf{exit}([\overline{Y_m}],[\mathsf{bv}(e_2)],p''_2)+\tau_2} p''}$$

$$\begin{array}{l}\text{if } \mathrm{f}/m = \mathsf{fun}\ (\overline{Y_m}) \to e_2 \in \mathsf{pgm},\ \mathsf{ret}(e_2) = p''_2, \\ \mathsf{match}(\overline{Y_m}, \overline{p'_m}) = \sigma,\ \mathsf{match}(p, p') = \sigma'\end{array}$$

$$\frac{\langle vs, ps\rangle, \theta \vdash p \Downarrow_{\epsilon} p' \qquad \langle vs, ps\rangle, \theta \cup \sigma \vdash e_i \Downarrow_{\tau_1} p \qquad \langle vs, ps\rangle, \theta \cup \sigma' \vdash e \Downarrow_{\tau_2} p''}{\langle vs, ps\rangle, \theta \vdash \mathsf{let}\ p_0 = \mathsf{case}\ p\ \mathsf{of}\ \mathit{clauses}\ \mathsf{end}\ \mathsf{in}\ e \Downarrow_{\mathsf{case}(vs,ps,i,p,p_i,\mathit{alts})+\tau_1+\mathsf{exitcase}(vs,ps,p_0,p'_i)+\tau_2} p''}$$

$$\begin{array}{l}\text{if } \mathit{clauses} = p_1 \to e_1; \ldots; p_m \to e_m,\ \mathsf{cmatch}(p', \mathit{clauses}) = (i, p_i, \sigma), \\ \mathit{alts} = [(1, p, p_1), \ldots, (m, p, p_m)],\ \mathsf{ret}(e_i) = p'_i,\ \mathsf{match}(p_0, p) = \sigma'\end{array}$$

$$\frac{\pi, \theta \vdash e_1 \Downarrow_{\tau_1} p'_1 \qquad \pi, \theta \cup \sigma \vdash e_2 \Downarrow_{\tau_2} p}{\pi, \theta \vdash \mathsf{let}\ p_1 = e_1\ \mathsf{in}\ e_2 \Downarrow_{\tau_1+\mathsf{bind}(vs,ps,p_1,\mathsf{ret}(e_1))+\tau_2} p} \quad \text{if } \mathsf{match}(p_1, p'_1) = \sigma$$

$$\frac{\pi, \theta \vdash e_1 \Downarrow_{\tau_1} p_1 \qquad \pi, \theta \vdash e_2 \Downarrow_{\tau_2} p_2}{\pi, \theta \vdash \mathsf{do}\ e_1\ e_2 \Downarrow_{\tau_1+\tau_2} p_2}$$

**Fig. 4.** Flat language instrumented semantics

match to compute the matching substitution (if any) between two patterns, i.e., $\mathsf{match}(p_1, p_2) = \sigma$ if $\mathcal{D}om(\sigma) \subseteq \mathcal{V}ar(p_1)$ and $p_1\sigma = p_2$, and fail otherwise. In this rule, $\mathsf{match}(\overline{Y_m}, \overline{p'_m})$ just returns the substitution $\{Y_1 \mapsto p'_1, \ldots, Y_m \mapsto p'_m\}$. The *update* of an environment $\theta$ using $\sigma$ is denoted by $\theta \cup \sigma$. Formally, $\theta \cup \sigma = \delta$ such that $X\delta = \sigma(X)$ if $X \in \mathcal{D}om(\sigma)$ and $X\delta = X\theta$ otherwise (i.e., $\sigma$ has higher priority than $\theta$). Observe that we use the evaluated patterns $p'_1, \ldots, p'_m$ to update the environment, but the original, static patterns $p_1, \ldots, p_m$ in the call event.

The next rule is used to evaluate case expressions. Here, we produce case and exitcase events that also include the parameter variables of the function and the bound variables. For selecting the matching branch of the case expression, we use the auxiliary function cmatch that is defined as follows: $\mathsf{cmatch}(p, p_1 \to e_1; \ldots; p_n \to e_n) = (i, p_i, \sigma)$ if $\mathsf{match}(p, p_i) = \sigma$ for some $i \in \{1, \ldots, n\}$ and $\mathsf{match}(p, p_j) = \mathsf{fail}$ for all $j < i$. Informally speaking, cmatch selects the first matching branch of the case expression, which follows the usual semantics of Erlang. As in the previous rule, note that we use $p'$ in cmatch but the original, static pattern $p$ in the case event.

The following rule is is used to evaluate let expressions. It produces a single bind event which includes, as usual, the parameter variables of the function and the bound variables. Finally, the last rule deals with do expressions. Here, we proceed as expected and return the concatenation of the sequences of events produced when evaluating the subexpressions.

```
main/1 = fun $(X) \to$ let $W =$ apply app/2 $(X, X)$ in $W$

app/2 = fun $(X, Y) \to$ let $W_1 =$ case $X$ of
                                      $[\,] \to Y$
                                      $[H|T] \to$ let $W_2 =$ apply app/2 $(T, Y)$ in $[H|W_2]$
                                    end
                         in $W_1$
```

**Fig. 5.** Example flat program

In the following, without loss of generality, we assume that the entry point to the program is always the distinguished function main$/n$.

**Definition 3 (instrumented execution).** *Given a flat program pgm and an initial expression,* apply main$/n$ $(p_1, \ldots, p_n)$*, with* main$/n =$ fun $(X_1, \ldots, X_n) \to e \in$ pgm*, its evaluation is denoted by*

$$\langle[\overline{X_n}], [\mathsf{bv}(e)]\rangle, \theta \vdash e \Downarrow_\tau v$$

*where* $\theta = \{X_1 \mapsto p_1, \ldots, X_n \mapsto p_n\}$ *is a substitution,* $v$ *is the computed value and* $\tau + \mathsf{exit}([\overline{X_n}], [\mathsf{bv}(e)], \mathsf{ret}(e))$ *is the associated sequence of events.*

*Example 2.* Let us consider the flat program shown in Figure 5. An example computation with the instrumented semantics is shown in Figure 6. Therefore, the associated sequence of events[1] is the following:

$\mathsf{call}([X], [W], W, [X, X])$
$\mathsf{case}([X, Y], [W_1, W_2], 2, X, [H|T], [(1, X, [\,]), (2, X, [H|T])])$
$\mathsf{call}([X, Y], [W_1, W_2], W_2, [T, Y])$
$\mathsf{case}([X, Y], [W_1, W_2], 1, X, [\,], [(1, X, [\,]), (2, X, [H|T])])$
$\mathsf{exitcase}([X, Y], [W_1, W_2], W_1, Y)$
$\mathsf{exit}([X, Y], [W_1, W_2], W_1)$
$\mathsf{exitcase}([X, Y], [W_1, W_2], W_1, [H|W_2])$
$\mathsf{exit}([X, Y], [W_1, W_2], W_1)$
$\mathsf{exit}([X], [W], W)$

Note that the semantics is a conservative extension of the standard semantics in the sense that the generation of events does not affect the evaluation, i.e., if we remove the context information and the events labeling the arrows, we are back to the standard semantics of an eager functional language essentially equivalent to that in [8]. In the following, we denote computations with the standard, non-instrumented semantics, with $e \Downarrow v$. In this case, we do not use contexts nor

[1] Note that the flat program is not syntactically correct according to Fig. 3 since the right-hand side of the functions do not have the form let $X = e$ in $X$ with $e$ a pattern, a let binding or a do expression. Here, we keep this simpler formulation for clarity, and it also simplifies the sequence of events by avoiding some redundant bind events.

$$\dfrac{\dfrac{\dfrac{\dfrac{\pi_2, \theta_4 \vdash Y \Downarrow_{\epsilon} [\mathrm{a}] \qquad \pi_2, \theta_5 \vdash W_1 \Downarrow_{\epsilon} [\mathrm{a}]}{\pi_2, \theta_4 \vdash \mathsf{let}\ W_1 = \mathsf{case}\ldots \Downarrow_{\tau_1} [\mathrm{a}]} \qquad \pi_2, \theta_6 \vdash [H|W_2] \Downarrow_{\epsilon} [\mathrm{a}, \mathrm{a}]}{\pi_2, \theta_3 \vdash \mathsf{let}\ W_2 = \mathsf{apply}\ldots \Downarrow_{\tau_2} [\mathrm{a}, \mathrm{a}]} \qquad \pi_2, \theta_7 \vdash W_1 \Downarrow_{\epsilon} [\mathrm{a}, \mathrm{a}]}{\pi_2, \theta_2 \vdash \mathsf{let}\ W_1 = \mathsf{case}\ldots \Downarrow_{\tau_3} [\mathrm{a}, \mathrm{a}]} \qquad \pi_1, \theta_8 \vdash W \Downarrow_{\epsilon} [\mathrm{a}, \mathrm{a}]}{\pi_1, \theta_1 \vdash \mathsf{let}\ W = \mathsf{apply}\ \mathsf{app}/2\ (X, X)\ \mathsf{in}\ W \Downarrow_{\tau_4} [\mathrm{a}, \mathrm{a}]}$$

with

$$\pi_1 = \langle [X], [W] \rangle \quad \text{and} \quad \pi_2 = \langle [X, Y], [W_1, W_2 H, T] \rangle$$

$$\begin{array}{ll}
\theta_1 = \{X \mapsto [\mathrm{a}]\} & \theta_2 = \{X \mapsto [\mathrm{a}], Y \mapsto [\mathrm{a}]\} \\
\theta_3 = \{X \mapsto [\mathrm{a}], Y \mapsto [\mathrm{a}], H \mapsto \mathrm{a}, T \mapsto [\,]\} & \theta_4 = \{X \mapsto [\,], Y \mapsto [\mathrm{a}]\} \\
\theta_5 = \{X \mapsto [\,], Y \mapsto [\mathrm{a}], W_1 \mapsto [\mathrm{a}]\} & \theta_6 = \{X \mapsto [\mathrm{a}], Y \mapsto [\mathrm{a}], H \mapsto \mathrm{a}, T \mapsto [\,], W_2 \mapsto [\mathrm{a}]\} \\
\theta_7 = \{X \mapsto [\mathrm{a}], Y \mapsto [\mathrm{a}], W_1 \mapsto [\mathrm{a}, \mathrm{a}]\} & \theta_8 = \{X \mapsto [\mathrm{a}], W \mapsto [\mathrm{a}, \mathrm{a}]\}
\end{array}$$

$$\begin{aligned}
\tau_1 &= \mathsf{case}([X, Y], [W_1, W_2], 1, X, [\,], [(1, X, [\,]), (2, X, [H|T])]) \\
&\quad + \mathsf{exitcase}([X, Y], [W_1, W_2], W_1, Y) \\
\tau_2 &= \mathsf{call}([X, Y], [W_1, W_2], W_2, [T, Y]) + \tau_1 + \mathsf{exit}([X, Y], [W_1, W_2], W_1) \\
\tau_3 &= \mathsf{case}([X, Y], [W_1, W_2], 2, X, [H|T], [(1, X, [\,]), (2, X, [H|T])]) + \tau_2 \\
&\quad + \mathsf{exitcase}([X, Y], [W_1, W_2], W_1, [H|W_2]) \\
\tau_4 &= \mathsf{call}([X], [W], W, [X, X]) + \tau_3 + \mathsf{exit}([X, Y], [W_1, W_2], W_1)
\end{aligned}$$

**Fig. 6.** Example computation with the instrumented semantics

environments, and assume that the bindings are always applied to the current expression.

The relevance of the computed sequences of events is that one can easily reconstruct a symbolic execution that mimics the steps of the concrete execution that produced the sequence of events. For instance, one can use the simple Prolog-like program shown in Fig. 7.[2] For example, given the sequence of events of Example 2 and the initial call $sym(Res, Vars)$, the above program returns:

$$Res = [X, X], \quad Vars = [X]$$

which obviously produces less instantiated values than the concrete execution (where we had $Res = [\mathrm{a}, \mathrm{a}]$, $Vars = [a]$). An extended version of this procedure can be used to generate new test cases (see Section 5).

## 4 Program Instrumentation

In this section, we present a program transformation that instruments a program so that its standard execution will return the same sequence of events produced with the original program and the instrumented semantics of Figure 4.

[2] Here, we assume that the elements of $\tau$ are renamed apart. In the implementation, this is achieved when consulting them from a file. Indeed, this is why a context is needed in the events, to be able to recover the right bindings.

$sym(\tau, Res, Vars) \leftarrow eval(\tau, [(Res, Vars)], [\_BVars]).$

$eval([\,], [\,], [\,]).$

$eval([call(Vars, BVars, NRes, NVars)|Tau], [(Res, Vars)|Env], [BVars|BEnv]) \leftarrow$
$\quad eval(Tau, [(NRes, NVars), (Res, Vars)|Env], [\_NBVars, BVars|BEnv]).$

$eval([case(Vars, BVars, \_N, Arg, Pat, \_Alts)|Tau], [(Res, Vars)|Env], [BVars|BEnv]) \leftarrow$
$\quad Arg = Pat,\ eval(Tau, [(Res, Vars)|Env], [BVars|BEnv]).$

$eval([exitcase(Vars, BVars, Arg, Pat)|Tau], [(Res, Vars)|Env], [BVars|BEnv]) \leftarrow$
$\quad Arg = Pat,\ eval(Tau, [(Res, Vars)|Env], [BVars|BEnv]).$

$eval([bind(Vars, BVars, Pat1, Pat2)|R], [(Res, Vars)|Env], [BVars|BEnv]) \leftarrow$
$\quad Pat1 = Pat2, eval(R, [(Res, Vars)|Env], [BVars|BEnv]).$

$eval([exit(Vars, BVars, Pat)|Tau], [(Res, Vars)|Env], [BVars|BEnv]) \leftarrow$
$\quad Res = Pat,\ eval(Tau, Env, BEnv).$

**Fig. 7.** Prolog-like procedure for symbolic execution

Let us first introduce the predefined function out. It outputs its first argument (e.g., to a given file or to the standard output) and returns its second argument. This function is implemented as a function *call* (i.e., not as a function application) so that there is not any conflict when performing the instrumentation.

**Definition 4 (program instrumentation).** *Let pgm be a flat program. We instrument pgm by replacing each function definition:*

$$\mathrm{f}/k = \mathsf{fun}\ (X_1, \ldots, X_k) \to \mathsf{let}\ X = e\ \mathsf{in}\ X$$

*with a new function definition of the form*

$$\mathrm{f}/k = \mathsf{fun}\ (X_1, \ldots, X_k) \to [\![\mathsf{let}\ X = e\ \mathsf{in}\ \mathsf{out}(\text{"}\mathsf{bind}(vs, bs, X, \mathsf{ret}(e))\text{"}, \mathsf{out}(\text{"}\mathsf{exit}(vs, bs, X)\text{"}, X))]\!]^{vs,bs}_{\mathsf{F}}$$

*where* $vs = [\overline{X_k}]$, $bs = [\mathsf{bv}(e)]$, $\mathsf{F}$ *is a flag to determine if an* exitcase *event should be produced (see below), and the auxiliary function* $[\![\ ]\!]$ *is shown in Figure 8.*

Let us briefly explain the rules of the instrumentation. First, we add bind and exit events at the end of each function. Then, we also add call and case events in each occurrence of a function application and a case expression, respectively. Finding the value returned by a case expression is a bit more subtle. For this purpose, we introduce a flag that is propagated through the different cases so that only when the expression is the last expression in a case branch (a pattern) we produce an exitcase event. For let expressions, we produce a bind event and continue evaluating both the expression in the right-hand side of the binding and the result. Finally, the *default* case —the last equation in Figure 8— is only used to ignore the call to the predefined function out/2.

*Example 3.* Consider again the flat program of Example 2. The instrumented program is shown in Figure 9, where the new code is shown in blue.

$$
\begin{array}{rcl}
[\![e]\!]^{vs,bs}_{\mathsf{F}} & = & e \quad \text{if } e \in \mathsf{Pat}\\
[\![e]\!]^{vs,bs}_{\mathsf{T}(p)} & = & \mathsf{out}(\text{“}\mathsf{exitcase}(vs,bs,p,e)\text{”}, e) \quad \text{if } e \in \mathsf{Pat}\\
[\![\mathsf{let}\ W = \mathsf{apply}\ p_0\ (\overline{p_n})\ \mathsf{in}\ e]\!]^{vs,bs}_{b} & = & \mathsf{let}\ W = \mathsf{out}(\text{“}\mathsf{call}(vs,bs,W,[p_1,\ldots,p_n])\text{”},\\
 & & \qquad\qquad\quad \mathsf{apply}\ p/0\ (p_1,\ldots,p_n)\ )\\
 & & \mathsf{in}\ [\![e]\!]^{vs,bs}_{b}\\
[\![\mathsf{let}\ W = \mathsf{case}\ p\ \mathsf{of} & = & \mathsf{let}\ W = \mathsf{case}\ p\ \mathsf{of}\\
p_1 \to e_1; & & \qquad\quad p_1 \to \mathsf{out}(\text{“}\mathsf{case}(vs,bs,1,p,p_1,alts)\text{”},\\
 & & \qquad\qquad\qquad\qquad [\![e_1]\!]^{vs,bs}_{\mathsf{T}(W)}\ )\\
\ldots & & \qquad\quad \ldots\\
p_n \to e_n & & \qquad\quad p_n \to \mathsf{out}(\text{“}\mathsf{case}(vs,bs,n,p,p_n,alts)\text{”},\\
 & & \qquad\qquad\qquad\qquad [\![e_n]\!]^{vs,bs}_{\mathsf{T}(W)}\ )\\
\mathsf{end} & & \qquad \mathsf{end}\\
\mathsf{in}\ e]\!]^{vs,bs}_{b} & & \mathsf{in}\ [\![e]\!]^{vs,bs}_{b}\\
[\![\mathsf{let}\ p = e_1\ \mathsf{in}\ e_2]\!]^{vs,bs}_{b} & = & \mathsf{let}\ p = [\![e_1]\!]^{vs,bs}_{\mathsf{F}}\ \mathsf{in}\ \mathsf{out}(\text{“}\mathsf{bind}(vs,bs,p,\mathsf{ret}(e_1))\text{”},\\
 & & \qquad\qquad\qquad\qquad [\![e_2]\!]^{vs,bs}_{b}\ )\\
[\![\mathsf{do}\ e_1\ e_2]\!]^{vs,bs}_{b} & = & \mathsf{do}\ [\![e_1]\!]^{vs,bs}_{\mathsf{F}}\ [\![e_2]\!]^{vs,bs}_{b}\\
[\![e]\!]^{vs,bs}_{b} & = & e \quad \text{otherwise}
\end{array}
$$

where $alts = [(p,1,p_1),\ldots,(p,n,p_n)]$

**Fig. 8.** Program instrumentation

$$
\begin{array}{l}
\mathrm{main}/2 \ = \ \mathsf{fun}\ (X) \to \mathsf{let}\ W = \ \mathsf{out}(\text{“}\mathsf{call}([X],[W],W,[X,X])\text{”},\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{apply}\ \mathrm{app}/2\ (X,X))\\
\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ \mathsf{out}(\text{“}\mathsf{exit}([X],[W],W)\text{”},W)\\[1ex]
\mathrm{app}/2 \ = \ \mathsf{fun}\ (X,Y) \to\\
\qquad \mathsf{let}\ W_1 = \mathsf{case}\ X\ \mathsf{of}\\
\qquad\qquad\qquad [\,] \to \ \mathsf{out}(\text{“}\mathsf{case}([X,Y],[W_1,W_2,H,T],1,X,[\,],alts)\text{”},\\
\qquad\qquad\qquad\qquad\quad \mathsf{out}(\text{“}\mathsf{exitcase}([X,Y],[W_1,W_2,H,T],W_1,Y)\text{”},Y))\\
\qquad\qquad\qquad [H|T] \to \ \mathsf{out}(\text{“}\mathsf{case}([X,Y],[W_1,W_2,H,T],2,X,[H|T],alts)\text{”},\\
\qquad\qquad\qquad\qquad\quad \mathsf{let}\ W_2 = \ \mathsf{out}(\text{“}\mathsf{call}([X,Y],[W_1,W_2,H,T],W_2,[T,Y])\text{”},\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{apply}\ \mathrm{app}/2\ (T,Y)))\\
\qquad\qquad\qquad\qquad\quad \mathsf{in}\ \mathsf{out}(\text{“}\mathsf{exitcase}([X,Y],[W_1,W_2,H,T],W_1,[H|W_2])\text{”},\\
\qquad\qquad\qquad\qquad\qquad\quad [H|W_2])\\
\qquad \mathsf{in}\ \mathsf{out}(\text{“}\mathsf{exit}([X,Y],[W_1,W_2,H,T],W_1)\text{”},W_1)
\end{array}
$$

where $alts = [(1,X,[]),(2,X,[H|T])]$.

**Fig. 9.** Instrumented program

It can easily be shown that the instrumented program produces the same sequence of events of Example 2 (e.g., by executing the program in the standard environment of Erlang, together with an appropriate definition of out/2).

The correctness of the program instrumentation is stated in the next result:

**Theorem 1.** *Let pgm be a flat program and $pgm^I$ its instrumented version according to Definition 4. Given an initial expression,* apply $\mathrm{main}/n$ $(p_1, \ldots, p_n)$*, its execution using pgm and the instrumented semantics (according to Definition 3) produces the same sequence of events as its execution using $pgm^I$ and the standard semantics.*

The proof is not difficult and can be performed by induction on the structure of the proof trees.

## 5 Concolic Testing

In this section, we discuss the usefulness of our approach to concolic execution in the context of test case generation. Basically, the process is similar to previous algorithms (e.g., [13]). First, we introduce the following notion of *trace*:

**Definition 5 (trace).** *Let $\tau$ be the sequence of events produced by an evaluation with the instrumented semantics. Then,* $\mathsf{trace}(\tau)$ *denotes the associated trace, where the auxiliary function* trace *is defined as follows:*

$$\mathsf{trace}(\tau) = \begin{cases} \epsilon & \mathit{if}\ \ \tau = \epsilon \\ \mathsf{trace}(\tau') & \mathit{if}\ \ \tau = \mathsf{call}(vs, bs, p, ps) + \tau' \\ \mathsf{trace}(\tau') & \mathit{if}\ \tau = \mathsf{exit}(vs, bs, p) + \tau' \\ \mathsf{trace}(\tau') & \mathit{if}\ \ \tau = \mathsf{bind}(vs, bs, p, p') + \tau' \\ i, \mathsf{trace}(\tau') & \mathit{if}\ \ \tau = \mathsf{case}(vs, bs, i, p, p_i, alts) + \tau' \\ \mathsf{trace}(\tau') & \mathit{if}\ \ \tau = \mathsf{exitcase}(vs, bs, p, p') + \tau' \end{cases}$$

For instance, the trace associated to the sequence of events $\tau$ of Example 2 is $\mathsf{trace}(\tau) = 2, 1$. Roughly speaking, the trace just records the fact that two case expressions have been evaluated, first selecting the second branch and then the first branch.

Here, one is interested in computing all possible alternative input arguments from a given sequence of events, rather than only the one that mimics the concrete execution. This can be obtained, e.g., using the Prolog-like procedure shown in Figure 10 (a slight extension of the one in Section 3; the new case is in blue). Now, $\mathit{input\_alt}(\tau, \mathit{Vars})$ —where the events in $\tau$ are renamed apart, as before— considers all possible alternatives by matching a different branch in every evaluated case expression. For instance, for the sequence of events of Example 2, we get three (nondeterministic) answers:

$$\mathit{Vars} = [\,]\ ; \quad \mathit{Vars} = [X]\ ; \quad \mathit{Vars} = [X,\, Y|R]$$

In the following, we assume a function $\mathsf{alt}(\mathit{Traces}, \tau)$ that is based on $\mathit{input\_alt}$ but also fulfills the following conditions:

$input\_alt(\tau, Vars) \leftarrow eval(\tau, [(\_Res, Vars)], [\_BVars]).$

$eval([\,], [\,], [\,]).$

$eval([call(Vars, BVars, NRes, NVars)|Tau], [(Res, Vars)|Env], [BVars|BEnv])$
$\quad \leftarrow eval(Tau, [(NRes, NVars), (Res, Vars)|Env], [\_NBVars, BVars|BEnv]).$

$eval([case(Vars, BVars, \_N, Arg, Pat, \_Alts)|Tau], [(Res, Vars)|Env], [BVars|BEnv])$
$\quad \leftarrow Arg = Pat,\ eval(Tau, [(Res, Vars)|Env], [BVars|BEnv]).$

$eval([case(Vars, BVars, N, \_, \_, Alts)|Tau], [(Res, Vars)|Env], [BVars|BEnv])$
$\quad \leftarrow member((M, Arg, Pat), Alts), N \neq M,$
$\qquad Arg = Pat,\ eval(Tau, [(Res, Vars)|Env], [BVars|BEnv]).$

$eval([bind(Vars, BVars, Pat1, Pat2)|R], [(Res, Vars)|Env], [BVars|BEnv]) \leftarrow$
$\quad Pat1 = Pat2, eval(R, [(Res, Vars)|Env], [BVars|BEnv]).$

$eval([exitcase(Vars, BVars, Arg, Pat)|Tau], [(Res, Vars)|Env], [BVars|BEnv])$
$\quad \leftarrow Arg = Pat,\ eval(Tau, [(Res, Vars)|Env], [BVars|BEnv]).$

$eval([exit(Vars, BVars, Pat)|Tau], [(Res, Vars)|Env], [BVars|BEnv])$
$\quad \leftarrow Res = Pat,\ eval(Tau, Env, BEnv).$

**Fig. 10.** Prolog-like procedure for concolic testing

- When selecting a case branch, the bindings should ensure not only that the pattern of the branch is matched, but also that no previous patterns are matched.
- The set of visited traces is used to avoid producing input data whose execution will produce a trace that is a prefix of some of the already visited traces.
- Finally, when the returned arguments are not ground, we further apply a grounding substitution. Although any arbitrary values would be correct, one can get a more precise algorithm by using type information to restrict the possible values an argument may take.

Now, let us present the kernel of a concolic testing procedure.

**Definition 6 (concolic testing).**

**Input:** *an instrumented program pgm and an expression* apply $\mathrm{main}/n\ (\overline{p_n})$.
**Output:** *a set TC of test cases.*

```
1. Let $Pending := \{\mathsf{apply}\ \mathrm{main}/n\ (\overline{p_n})\}$, $TC := \{\}$, $Traces := \{\}$.
2. While $|Pending| \neq 0$ do
   (a) Take $e \in Pending$, $Pending := Pending \setminus \{e\}$, $TC := TC \cup \{e\}$.
   (b) Let $\tau$ be the sequence of events from the execution of $e$ using the instrumented program, with $\mathsf{trace}(\tau) = \overline{i_n}$. Let $Traces := Traces \cup \{\overline{i_n}\}$.
   (c) $Pending := Pending \cup \mathsf{alt}(Traces, \tau)$
3. Return the set $TC$ of test cases
```

In general, the concolic testing algorithm will run forever since the possible execution paths are infinite. In practice, we can make it finite by either using a timeout or limiting the *term depth* of the input arguments. In either case, the process will be incomplete, as usual. In the context of concolic testing, soundness is the most relevant property in order to avoid false positives.

For instance, given the instrumented program of Example 3, the initial expression $\mathsf{apply}\ \mathrm{main}/2\ ([a])$, and by limiting the term depth to 3, the concolic testing algorithm produces the following test cases:

$\mathsf{apply}\ \mathrm{main}/2\ ([a])$
$\mathsf{apply}\ \mathrm{main}/2\ ([\,])$
$\mathsf{apply}\ \mathrm{main}/2\ ([a, b])$
$\mathsf{apply}\ \mathrm{main}/2\ ([a, b, c])$

Observe that concolic testing is not only useful for test case generation, but could also be used for a sort of model checking by analyzing the concrete executions that are run during the concolic testing procedure (i.e., if we get a run time error in these executions, this is surely a real bug since there is no abstraction involved in the process).

An implementation of the concolic testing tool has been undertaken. The first stage, flattening and instrumenting the source program, is being implemented in Erlang itself. In contrast, the concolic testing algorithm is being implemented in Prolog since the facilities of this language —unification and nondeterminism— make it very appropriate for dealing with symbolic executions.

## 6 Discussion

As mentioned in the introduction, there are only a couple of previous approaches to concolic execution in functional and logic programming. Nevertheless, let us mention some previous work on test case generation in functional and logic programming.

In the context of logic programming, SWI-Prolog [14] includes a unit testing tool which allows the interactive generation of test cases. The closest approach, though, is that of Albert et al. [1] for test case generation based on symbolic execution. However, their technique is based solely on traditional symbolic execution. As mentioned before, concolic testing may scale better since one can deal with more complex constraints by using data from the concrete execution. Another close approach is [13], where a concolic execution semantics for logic programs is presented. But this approach only considers a simpler *statement* coverage (rather than path coverage). Moreover, both [1] and [13] follow the interpreter-based model, in contrast to our approach based on a program transformation. In the context of functional programming, one of the most well known approaches for software testing is that of QuickCheck [4], a property-based framework that includes a random generation of input values (though they can also be driven by the user). In general, though, the quality of the achieved coverage depends on the user, since a pure random data generation would often achieve a poor coverage. As for concolic execution, [5] introduces CutEr, a concolic testing tool for the functional component of Erlang. This approach, as mentioned in the introduction, is also based on implementing an interpreter augmented to also deal with symbolic values.

To the best of our knowledge, our paper proposes the first approach to concolic execution by program instrumentation in the context of functional or logic programming. In contrast to using an interpreter-based design, our lightweight approach is easier to maintain and may scale up better to medium and large programs since one can still use the standard environment for execution (although the instrumentation will of course introduce some overhead).

As a future work, we plan to design a concolic testing algorithm, together with efficient heuristics —including type information— in order to maximize the coverage achieved by the generated test cases. We will also continue working on the implementation in order to make it fully automatic and cover most of the features of Erlang, concurrency being the main challenge.

## References

1. E. Albert, P. Arenas, M. Gómez-Zamalloa, and J.M. Rojas. Test Case Generation by Symbolic Execution: Basic Concepts, a CLP-Based Instance, and Actor-Based Concurrency. In *SFM 2014*, Springer LNCS 8483, pages 263–309. Springer, 2014.
2. Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent programming in ERLANG.* Prentice Hall, 1993.
3. R. Carlsson. An Introduction to Core Erlang. In *Proceedings of the PLI'01 Erlang Workshop*, 2001. Available from `http://www.erlang.se/workshop/carlsson.ps`.
4. K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proc. of ICFP '00)*, pages 268–279. ACM, 2000.
5. A. Giantsios, N. Papaspyrou, and K. Sagonas. Concolic testing for functional languages. To appear in *Proc. of PPDP'15*, ACM, 2015.
6. P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proc. of PLDI'05*, pages 213–223. ACM, 2005.
7. P. Godefroid, M.Y. Levin, and D.A. Molnar. Sage: whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, 2012.
8. G. Kahn. Natural Semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *Proc. of STACS'87*, pages 22–39, 1987.
9. James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
10. C.S. Pasareanu and N. Rungta. Symbolic PathFinder: symbolic execution of Java bytecode. In *ASE'10*, pages 179–180. ACM, 2010.
11. K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proc. of ESEC/SIGSOFT FSE 2005*, pages 263–272. ACM, 2005.
12. G. Vidal. Towards Symbolic Execution in Erlang (short paper). In *Proc. of the 9th International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics (PSI'14)*, pages 351–360. Springer LNCS 8974, 2014.
13. G. Vidal. Concolic Execution and Test Case Generation in Prolog. In M. Proietti and H. Seki, editors, *Proc. of LOPSTR'14*, pp. 167-181. Springer LNCS 8981, 2015.
14. Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.