Verification of Erlang Programs: Factoring out the Side-effect-free Fragment*

Dilian Gurov   Gennady Chugunov

Swedish Institute of Computer Science,
Box 1263, SE-164 29 Kista, Sweden,
dilian|gena@sics.se

Abstract

Erlang is a functional programming language developed at Ericsson for writing economical and yet powerful and efficient telecommunication applications. Correctness is of major importance in such applications, and since they usually exhibit a high degree of concurrency, testing is often not sufficient. Verification, namely formally proving that a system is correct, is becoming a more and more widespread practice. Due to the complexity of Erlang, there is no general method for verification of arbitrary Erlang programs which is effective and at the same time leads to economic proofs. However, one can do much better in specialised cases which are well-understood. A main direction of research is the identification of fragments of Erlang for which efficient verification methods exist. One such fragment is the side-effect-free one, in which an Erlang expression is evaluated purely for its value, and is not affecting the environment in which it is evaluated in terms of sending messages, reading from the message queue, or process spawning. This is furthermore a very common situation, given the number of libraries of side-effect-free functions used extensively in practice. The present paper presents work in progress and outlines an idea for compositional reasoning about the behaviour of an Erlang system modulo replacement of side-effect-free Erlang expressions with the result of their evaluation.

Keywords: software verification, Erlang, side-effect-free evaluation, compositional reasoning.

*Work partially supported by the Computer Science Laboratory of Ericsson Utvecklings AB, Stockholm, the Swedish National Board for Technical and Industrial Development (NUTEK) through the ASTEC competence centre, and a Swedish Foundation for Strategic Research Junior Individual Grant.

1 Introduction

Software written for telecommunication applications has to meet high quality demands. *Correctness* is one major concern; the activity of proving formally that a system is correct is called *verification*. Telecommunications software is highly concurrent, and *testing* is often not capable of guaranteeing correctness to a satisfactory degree. The software we are faced with consists of many, relatively small modules, written in the functional language Erlang [1]. These modules define the behaviour of a number of processes operating in parallel and communicating through asynchronous message-passing. New processes can be generated during execution. Because of the complexity of such software, our approach to verification is to prove that the software satisfies a set of properties formalized in a suitable logic language. The specification language we use is based on Park's μ-calculus [12, 11], extended with Erlang-specific features. This is a very powerful logic, due to the presence of least and greatest fixed point recursion, allowing the formalization of a wide range of behavioural properties. Verification in this context is not decidable, but can be automated to a large extent, requiring human intervention in a few, but critical points.

For a few years now, the Formal Design Techniques group at the Swedish Institute of Computer Science has pursued a programme aimed at enabling formal verification of complex open distributed systems (ODSs) through program code verification. Previous work by the group has been directed towards establishing the mathematical machinery [5, 6], providing basic tool support [3], performing case studies [2], and motivating the chosen verification framework [8].

Due to the complexity of Erlang, there is obviously no general method for verification of arbitrary Erlang programs which is effective and at the same time leads to economic proofs. However, one can do much better in specialised cases which are well-understood. A main direction of research is the identification of fragments of Erlang for which efficient verification methods exist. One such fragment is the side-effect-free one, in which an Erlang expression is evaluated purely for its value, and is not affecting the environment in which it is evaluated in terms of sending messages, reading from the message queue, or process spawning. A very common situation are function calls to functions, the body of the definition of which are side-effect-free expressions. A large number of libraries of such functions are used extensively in practice. In all the case studies we have performed so far we had repeatedly to deal with library functions for manipulation of lists, numbers etc. Following the *compositional reasoning* paradigm used for reasoning about large component-based software, one would like in such cases is to be able to reason modulo replacement of side-effect-free function calls with the result of their evaluation. The present paper is dedicated to technically achieving this goal in a systematic fashion within our verification framework by factoring out the reasoning about the behaviour of side-effect-free Erlang expressions from the reasoning about general Erlang systems.

The paper is organised as follows. The next section summarises our verification framework. Section 3 describes the general problem of factoring out the reasoning about side-effect-free Erlang expressions from the reasoning about general Erlang systems, and presents a systematic way of performing this within our verification framework. The following section focuses on the subtask of verifying side-effect-free Erlang expressions, which is illustrated on a concrete example in section 5. The last section gives a summary and concluding remarks.

2 Verification of Erlang Programs

In this section we summarise our verification framework as presented in [3, 8].

The Erlang Programming Language. We consider a core fragment of the Erlang programming language with dynamic networks of processes operating on data types such as natural numbers, lists, tuples, or process identifiers (pid's), using asynchronous, first-order call-by-value communication via unbounded ordered message queues called mailboxes. Real Erlang has several additional features such as communication guards, exception handling, modules, distribution extensions, and a host of built-in functions.

Besides Erlang *expressions* e we operate with the syntactical categories of *matches* m, *patterns* p, and *values* v. The abstract syntax of Core Erlang expressions is summarised as follows:

$$
\begin{array}{rcl}
e & ::= & V \ \mid \texttt{self} \ \mid op(e_1,\ldots,e_n) \ \mid \\
 & & e_1\ e_2 \ \mid e_1, e_2 \ \mid \texttt{case}\ e\ \texttt{of}\ m \ \mid \texttt{spawn}(e_1,e_2) \ \mid \\
 & & \texttt{receive}\ m\ \texttt{end} \ \mid e_1!e_2 \\
m & ::= & p_1 \rightarrow e_1;\ \cdots\ ; p_n \rightarrow e_n \\
p & ::= & op(p_1,...,p_n) \mid V \\
v & ::= & op(v_1,...,v_n)
\end{array}
$$

Here op ranges over a set of primitive constants and operations including zero 0, successor $e+1$, tupling $\{e_1, e_2\}$, the empty list $[]$, list prefix $[e_1|e_2]$, pid constants ranged over by pid, and atom constants ranged over by a, f, and g. The constructs involving side effects are: `self`, evaluating to the pid of the process evaluating this expression; `spawn`, resulting in a new process being generated; `receive` for reading from the mailbox which is assotiated with the process evaluating the expression; and "!" for sending a value to a process identified by its pid. These constructs will not be discussed in further detail here, since we focus on the side-effect-free part of the language.

To reason in a formal fashion about the behaviour of an Erlang program, a suitable formal semantics of the Erlang language is needed. This can be done in different styles, depending on the intended style of reasoning. Our approach is tailored to *small-step operational semantics*, although other formal notions of behaviour are derivable in our framework, supporting reasoning in different flavours. Operational semantics are usually presented by transition rules involving labelled transitions between structured states [13]. A natural approach to handling the different conceptual layers of entities in the Erlang language, namely expressions, processes, and systems, is to organise the semantics hierarchically, in layers, using different sets of transition labels at each layer, and extending at each layer the structure of the state with new components as needed. A suitable formal semantics for Erlang has been recently developed, and can be found in a (yet unpublished) manuscript by Fredlund [7].

The Property Specification Language. Reasoning about complex systems requires *compositional reasoning*, i.e. the capability to reduce arguments about the behaviour of compound entities to arguments about the behaviours of its parts. To support compositional reasoning, a specification language should capture the labelled transitions at each layer of the transitional semantics. Polymodal logic is particularly suitable for the task, employing box and diamond *modalities* labelled by the transition labels: a structured state s satisfies formula $\langle\alpha\rangle\,\Phi$ if there is an α-derivative of s (i.e. a state s' such that $s \xrightarrow{\alpha} s'$ is a valid labelled transition) satisfying Φ, while s satisfies $[\alpha]\,\Phi$ if all α-derivatives of s (if any) satisfy Φ. Additionally, *state predicates* are needed to capture the "local", unobservable characteristics of structured states, such as e.g. the value of a local variable. The presence of recursion on different layers requires also the specification language to be recursive. Adding recursion in the form of least and greatest fixed-points to the modalities described above results in a powerful specification language, broadly known as the *μ-calculus* [12, 11]. Roughly speaking, least fixed-point formulas $\mu X.\phi$ express eventuality properties, while greatest fixed-point formulas $\nu X.\phi$ express invariant properties. Nesting of fixed points allows complicated reactivity and fairness properties to be expressed.

This powerful logic is capable of expressing a wide range of important system properties, ranging from type-like assertions to complex reactivity properties of the interaction behaviour of a telecommunication system. For instance, the type of natural numbers is the least set containing zero and closed under successor. The property of being a natural number can hence be defined recursively as a least fixed-point:

$$
\begin{aligned}
&N : nat \Leftarrow \\
&\qquad N = 0 \\
&\quad \vee\, \exists V.\, (V : nat \;\wedge\; N = V + 1)
\end{aligned}
$$

where $\Leftarrow$ is used for least-fixed-point definitions (i.e., for denoting the least solution of a recursive definition), while $\Rightarrow$ is used for greatest-fixed-point definitions (overloaded with implication).

The Proof System. Reasoning about open distributed systems written in Erlang requires reasoning about their interface behaviour relativised by assumptions about certain system parameters. Technically, this can be achieved by using Gentzen-style proof systems, allowing free parameters to occur within the *proof judgments* of the proof system. The judgments are of the form $\Gamma \vdash \Delta$, where Γ and Δ are sets of assertions. A judgment is deemed *valid* if, for any interpretation of the free variables, some assertion in Δ is valid whenever all assertions in Γ are valid. Parameters are simply variables ranging over specific types of entities, such as messages, functions, or processes. For example, the proof judgment $x : \Psi \vdash P(x) : \Phi$ states that object P has property Φ provided the parameter x of P satisfies property Ψ.

This idea of open correctness assertions gave rise to the development of a Gentzen-style proof system [6] that serves as the basis for the implementation of a verification tool. On top of a fairly standard proof system we added two rules: the first a "term-cut" rule for decomposing proofs about a compound system to proofs about the components, the second a discharge rule based on detecting loops in the proof. Roughly, the goal is to identify situations where a latter proof node is an instance of an earlier one on the same proof branch, and where appropriate fixed points have been safely unfolded. The discharge rule thus takes into account the history of assertions in the proof tree. In terms of the implementation this requires the preservation of the proof tree during proof construction. Combined, the term-cut rule and the discharge rule allow general and powerful induction and co-induction principles to be applied, ranging from induction on the dynamically evolving architecture of a system, to induction on finitary and co-induction on infinitary datatypes.

The Erlang Verification Tool. From a user's point of view, proving a property of an Erlang program using the verification tool involves "backward" (i.e., goal-directed) construction of a proof tree (tableau). The user is provided with commands for defining the initial node of the proof tree, for expanding a proof tree node ('the current proof node can be considered proved if the following nodes are proved instead'), for navigating through the proof tree, for checking whether the discharge rule is applicable, and for visualizing the current state of the proof tree using the daVinci graph visualization tool. Since the whole proof tree is maintained, proof reuse and sharing is greatly facilitated. Automation is achieved through a set of proof tactics and tacticals.

At the present point in time a prototype tool has been completed with the functionality described above. The largest case study performed so far is the verification of a distributed database lookup manager written in Erlang [2].

A high degree of mechanization of the low-level reasoning steps is crucial for making our verification method industrially applicable. This is the primary motivation for the ideas presented below, since the side-effect-free fragment of Erlang is well understood and classic methods exist to allow the treatment of this fragment to be mechanized to a satisfactory degree.

3 Factoring out the Side-effect-free Fragment

Due to the complexity of Erlang, there is obviously no general method for verification of arbitrary Erlang programs, which is effective and at the same time leads to economic proofs. However, one can do much better in specialised cases which are well-understood. One such fragment is the side-effect-free one, in which a (possibly recursively defined) Erlang expression is evaluated purely for its value, and does not affect the environment in which it is evaluated in terms of sending messages, reading from the message queue, or process spawning.

Compositional Verification. The essence of compositional verification is the reduction of an argument about the behaviour of a compound system to arguments about the behaviour of its components. A system P containing component Q can be represented through term substitution as $P[Q/X]$, where X is a variable ranging over entities of the type of Q. We can relativize an assertion $P[Q/X] : \Phi$ about the compound object $P[Q/X]$ to a certain property Ψ of its component Q by considering Q as a parameter for which property Ψ is assumed, provided we can show that Q indeed satisfies the assumed property Ψ. Technically, we achieve this through a *term-cut* proof rule of the shape:

$$\textsf{(TermCut)}\ \frac{\Gamma \vdash Q : \Psi, \Delta \qquad \Gamma, X : \Psi \vdash P : \Phi, \Delta}{\Gamma \vdash P[Q/X] : \Phi, \Delta}$$

We consider the case when component Q is a function call to a function the evaluation of which involves no side-effects. In this case we can offer a more powerful (de)composition principle than the one explained above.

Side-effect-free Function Calls. Let us consider a common situation when verifying Erlang programs. Let proc<e, pid, q> be an Erlang process with process identifier pid and message queue q evaluating expression e. Assume now that the redex (i.e., the current control point) of e is a function call of the shape f(Y), where Y is a value variable. Then e is equal to e'[f$(Y)/X$] for some Erlang expression e'(X) having a single occurrence of expression variable X (at redex point). A proof goal involving this (open) process would generally have the form:

$$\Gamma \vdash \texttt{proc<e, pid, q>} : \phi, \Delta \qquad (1)$$

where ϕ is a desired property of the process, and where Γ might put some constraints on the free variables in the process, such as Y. If function f(Y) is side-effect-free, it is evaluated only for its value. The specification of the process, and hence ϕ, should not depend on the actual number of internal steps which the evaluation of the function call requires. Therefore, we should be able to replace the above proof goal with:

$$\Gamma, V : \theta(Y) \vdash \texttt{proc<e'(}V\texttt{), pid, q>} : \phi, \Delta \tag{2}$$

where V is a value variable and θ states the relation of V to the input parameter(s) Y of function `f`. This reduction of goal (1) to goal (2) is what we call *factoring out the side-effect-free fragment*, since it enables us to reason modulo replacement of (open) side-effect-free function calls with the result of their evaluation.

The Reduction Steps. There is a systematic way of performing this reduction within our proof system. It is based on the following assumptions:

- the redex of expression `e` is a function call of the shape $\texttt{f}(Y_1, ..., Y_n)$;
- the body of the definition of `f` is side-effect-free;
- property ϕ is insensitive to the number of side-effect-free actions (usually denoted by τ), such as "eventually the process sends out a reply ...", e.g. of the shape $\mu Z.(\phi' \vee \langle\tau\rangle\, Z)$.

We explain the method on the example given above, i.e. starting with proof goal (1). First, we relativize the goal w.r.t. the specification of $\texttt{f}(Y)$, which is given as a formula of the shape $\texttt{prepost}(\psi, \theta)$ relating input values to output values of `f`. Intuitively, $\texttt{prepost}(\psi, \theta)$ states that given pre-condition ψ holds of the input value Y, evaluation of $\texttt{f}(Y)$ terminates with a value satisfying post-condition θ. Formalising `prepost` in our logic and verifying side-effect-free expressions is the topic of the next section. This relativization can be achieved through applying the term-cut rule, resulting in two new proof goals replacing goal (1):

$$\Gamma \vdash \texttt{f}(Y) : \texttt{prepost}(\psi, \theta), \Delta \tag{3}$$

$$\Gamma, X : \texttt{prepost}(\psi, \theta) \vdash \texttt{proc<e'(}X\texttt{), pid, q>} : \phi, \Delta \tag{4}$$

where `e'` is as explained above. Ideally, $\texttt{f}(Y)$ is a library function which has been already specified and verified; in this case goal (3) is (an instance of) a lemma and can be eliminated. If not, we use the method described in the next section to achieve this. It is also possible, that we have no access to the implementation of the function; in this case we have to leave the goal open and the final proof will be relative to the correctness of the specification. We focus our attention on goal (4). Since we intend to show that the function call results in a value, we obviously hope to be able to prove that the pre-condition ψ is a consequence of the assumptions Γ. So, dealing with `prepost` should result in the two proof goals replacing goal (4):

$$\Gamma \vdash \psi, \Delta \tag{5}$$

$$\Gamma, X : \texttt{eval}(\theta) \vdash \texttt{proc<e'}(X)\texttt{, pid, q>} : \phi, \Delta \tag{6}$$

where `eval` is like `prepost` but is not relativised on a precondition. `eval` states, that either (i) X is a value variable V satisfying θ, or otherwise (ii) X reduces via a silent (i.e. side-effect-free) step to another expression X' satisfying $\texttt{eval}(\theta)$. In other words, dealing with `eval` should result in two proof goals replacing goal (6):

$$\Gamma, V : \theta \vdash \texttt{proc<e'}(V)\texttt{, pid, q>} : \phi, \Delta \tag{7}$$

$$\Gamma, X \xrightarrow{\tau} X', X' : \texttt{eval}(\theta) \vdash \texttt{proc<e'}(X)\texttt{, pid, q>} : \phi, \Delta \tag{8}$$

where goal (7) is the desired goal (2). It remains to eliminate goal (8). We assumed that variable X stands for a side-effect-free Erlang expression occurring at redex point in `e'`. As a consequence, the only actions of `e'`(X) are the silent actions of X. We also assumed that property ϕ is insensitive to the number of internal actions, e.g. of the shape $\mu Z.(\phi' \vee \langle\tau\rangle\, Z)$. These considerations imply, that dealing with ϕ should result in a goal replacing goal (8):

$$\Gamma, X' : \texttt{eval}(\theta) \vdash \texttt{proc<e'}(X')\texttt{, pid, q>} : \phi, \Delta \tag{9}$$

This goal is an instance of goal (6). It was obtained through unfolding a least-fixed-point formula (namely `eval`) on the left-hand side of the turnstyle symbol, and can hence be eliminated by using the discharge mechanism mentioned in the previous section.

For given shapes of formula ϕ the reduction outlined above can even be performed algorithmically. An important such case is when ϕ is itself a `prepost` formula; in this case the transition from goal (8) to goal (9) becomes straightforward and easy to mechanize.

4 Specification and Verification of Side-effect-free Expressions

In general, specification of the interaction behaviour of Erlang programs is a difficult task for which no systematic method has been developed so far. In the previous section we showed how to factor out the reasoning about side-effect-free Erlang expressions from the general reasoning about Erlang systems. Once we are in the realm of side-effect-free Erlang expressions we can apply well-known verification techniques.

Natural Semantics. Our verification method is based on a small-step operational semantics for Erlang. The proof rules contain, among other types of essertions, labelled-transition assertions of the shape $P \xrightarrow{\alpha} Q$. When reasoning about side-efect-free expressions, however, one usually prefers to work directly with the reflexive and transitive closure $e \longrightarrow^* v$ relating the expression e with the value v resulting from the (terminating) evaluation (i.e. sequence of side-effect-free computations) of e. This kind of assertion is usually denoted $e \Downarrow v$, and the semantics based on a set of rules for reasoning in this style is usually called *natural (operational) semantics* [9]. It is the style of reasoning that we would like to employ, and this is easy to achieve, since the $\Downarrow$ predicate is definable as a least fixed-point formula through the transition relation. However, there is one significant complication: such a semantics is sufficient only if we are dealing with closed expressions e. But our specification and verification method is parametric in its nature, and in this case we have to somehow relate in a single construct the resulting values to the (free) parameters in e. This idea brings us to another well-known concept, namely the one of (weakest) pre-conditions and (strongest) post-conditions.

Pre-conditions and Post-conditions. A classical method for verification of sequential programs is the axiomatic method of Hoare [10]. It is based on assertions of the shape $\{\psi\}e\{\theta\}$, the intuitive semantics of which, in our context, is: given the parameters of e satisfy the pre-condition ψ, then execution of e, provided it terminates, results in a value satisfying the post-condition θ. We follow the same idea, but require termination; a correctness notion known as *total correctness.*

The typical sequent we have to consider is of the following shape:

$$\Gamma \vdash \mathtt{f}(Y_1, ..., Y_n) : \mathtt{prepost}(\psi, \theta), \Delta$$

where $\mathtt{f}(Y_1, ..., Y_n)$ is a function call, the body of the definition of which is side-effect free, and where `prepost` relates the values resulting from evaluating the function call to the values of the input parameters $Y_1, ..., Y_n$ of `f`. Intuitively, $\mathtt{prepost}(\psi,\theta)$ states that given ψ holds, evaluation of $\mathtt{f}(Y_1, ..., Y_n)$ terminates with a value satisfying θ. In our property specification language `prepost` can be defined as follows:

$$\mathtt{prepost}(\psi, \theta) \;=\; (\psi \Rightarrow \mathtt{eval}\,\theta)$$

$$\begin{array}{l} \mathtt{eval}\,\theta \;\Leftarrow \\ \quad \lambda E : \mathtt{ErlangExpression.} \\ \qquad\quad \exists V : \mathtt{ErlangValue.}\ (V = E \,\wedge\, \theta\, V) \\ \qquad \vee\, E : \langle\tau\rangle\,\mathtt{true} \wedge [\tau]\,\mathtt{eval}\,\theta \end{array}$$

where τ refers to side-effect-free (also called *silent*) computation steps.

The crucial question is how to handle function calls within the body of $\mathtt{f}(Y_1, ..., Y_n)$. A natural approach is to use the same technique as outlined in the previous section. When ϕ is a `prepost` formula, the transition from goal (8) to goal (9) becomes straightforward. We obtain the following admissable proof rule, given that e_2 occurs at redex point of e_1 (and given ϕ is a `prepost` formula):

$$(\mathsf{ValCut})\ \frac{\Gamma \vdash e_2 : \texttt{prepost}(\psi,\theta), \Delta \qquad \Gamma, V : \theta \vdash e_1(V) : \phi, \Delta \qquad \Gamma \vdash \psi, \Delta}{\Gamma \vdash e_1(e_2) : \phi, \Delta}$$

which plays an important rôle in our proofs.

5 Example: The Quicksort Algorithm

In this section we illustrate our approach on the well-known quicksort algorithm. Figure 1 gives an implementation of the algorithm as an Erlang module.

Specification. The algorithm is to be specified as a satisfaction pair of the shape $\texttt{sort}(L) : \phi_{sort} L$ where $\phi_{sort} L$ is a formula of type $\texttt{prepost}(\psi_{sort} L, \theta_{sort} L)$. The pre-condition $\psi_{sort}\ L$ can be given as a satisfaction pair $L : \texttt{list}$ where `list` is a list specification playing the rôle of a list type. For the purposes of the present paper it is sufficient to consider a list as either being the emptylist or being decomposable into a head element and a tail list:

$$
\begin{array}{l}
\texttt{list} \Leftarrow \\
\quad \lambda L : \texttt{ErlangValue}. \\
\qquad L = \texttt{[]} \\
\quad \lor\ \exists P, R : \texttt{ErlangValue}. \\
\qquad\quad L = \texttt{[}P|R\texttt{]} \\
\qquad \land\ R : \texttt{list}
\end{array}
$$

The post-condition $\theta_{sort}\ L$ should relate the resulting list L' to the argument list L. The usual way to relate the two lists is to require L' to be a sorted permutation of L:

$$
\begin{array}{l}
\theta_{sort}\ L \stackrel{\Delta}{\equiv} \\
\quad \lambda L' : \texttt{ErlangValue}. \\
\qquad \texttt{isSorted}\ L' \\
\quad \land\ \texttt{isPermutation}\ L\ L'
\end{array}
$$

where the predicates `isSorted` and `isPermutation` are as expected (definitions omitted).

We have also to specify the `split` function, which is called within the body of `sort` (we omit the specification of `append`). As with `sort`, the specification $\phi_{split}\ (P, R)$ of `split` is given as a formula $\texttt{prepost}(\psi_{split}\ (P, R), \theta_{split}\ (P, R))$. The pre-condition $\psi_{split}\ (P, R)$ can be taken to be $R : \texttt{list}$. The post-condition $\theta_{split}\ (P, R)$ specifies the resulting value as a pair $\{S, B\}$ of Erlang values (lists), such that the concatenation $S \cdot B$ is a permutation of R, P is smaller than any

```
-module(sort).
-export([sort/1]).

sort ([]) -> [];
sort ([Pivot|Rest]) ->
  case split(Pivot, Rest) of
    {Smaller, Bigger} ->
      append(sort(Smaller), [Pivot|sort(Bigger)])
  end.

split(Pivot, L) ->
  split(Pivot, L, [], [])

split(Pivot, [], Smaller, Bigger) ->
  {Smaller, Bigger};
split(Pivot, [H|T], Smaller, Bigger) when H < Pivot ->
  split(Pivot, T, [H|Smaller], Bigger);
split(Pivot, [H|T], Smaller, Bigger) when H >= Pivot ->
  split(Pivot, T, Smaller, [H|Bigger]).
```

Figure 1: The Quicksort Algorithm as an Erlang Module.

element of B, and P is bigger than any element of S:

$$
\begin{array}{l}
\theta_{split}\ (P,R) \stackrel{\Delta}{\equiv} \\
\quad \lambda V : \texttt{ErlangValue}. \\
\quad\quad \exists S, B : \texttt{ErlangValue}. \\
\quad\quad\quad V = \{S, B\} \\
\quad\quad \wedge\ \texttt{isPermutation}\ (S \cdot B)\ R \\
\quad\quad \wedge\ \texttt{isSmaller}\ P\ B \\
\quad\quad \wedge\ \texttt{isBigger}\ P\ S
\end{array}
$$

Verification. We now proceed with sketching a correctness proof. The initial proof goal is as follows:

$$\vdash\ \texttt{sort}(L) : \phi_{sort}\ L \tag{10}$$

Unfolding ϕ_{sort} leads to the new goal:

$$L : \texttt{list}\ \vdash\ \texttt{sort}(L) : \texttt{eval}(\phi_{sort}\ L) \tag{11}$$

Unfolding the definition of `list` yields the two new goals:

$$\vdash\ \texttt{sort([])} : \texttt{eval}(\phi_{sort}\ \texttt{[]}) \tag{12}$$

$$R : \texttt{list}\ \vdash\ \texttt{sort}(\texttt{[}P|R\texttt{]}) : \texttt{eval}(\phi_{sort}\ \texttt{[}P|R\texttt{]}) \tag{13}$$

Goal (12) is easily eliminated by chosing the empty list for the existentially quantified Erlang value in the body of the `eval` property, since the empty list is (trivially) both sorted and a permutation of itself. Proceeding with goal (13), we unfold the definition of `sort`, obtaining the following goal:

$$R : \texttt{list}\ \vdash\ \texttt{case split}(P,R)\ \texttt{of}\ \ \texttt{...} : \texttt{eval}(\phi_{sort}\ \texttt{[}P|R\texttt{]}) \tag{14}$$

We have reached a point where the redex of the expression is a function call. This is where we can apply rule (ValCut) introduced in the previous section, yielding three new goals:

$$R : \texttt{list}\ \vdash\ R : \texttt{list} \tag{15}$$

$$R : \texttt{list}\ \vdash\ \texttt{split}(P,R) : \phi_{split}\ (P,R) \tag{16}$$

$$R : \texttt{list}, V : \theta_{split}\ (P,R)\ \vdash\ \texttt{case}\ \ V\ \ \texttt{of}\ \ \texttt{...} : \texttt{eval}(\phi_{sort}\ \texttt{[}P|R\texttt{]}) \tag{17}$$

Goal (15) is trivial. Goal (16) is just another instance of the typical sequent adressed in the previous section. By unfolding θ_{split} and the `case` statement, goal (17) is reduced to:

$$R : \texttt{list}, \Gamma\ \vdash\ \texttt{append}(\texttt{sort}(S), \texttt{[}P|\texttt{sort}(B)\texttt{]}) : \texttt{eval}(\phi_{sort}\ \texttt{[}P|R\texttt{]}) \tag{18}$$

where Γ is the formula list:

$$\texttt{isPermutation}\ (S \cdot B)\, R, \texttt{isSmaller}\ P\, B, \texttt{isBigger}\ P\, S$$

Here again we can apply rule (ValCut), as well as some properties of `list`, to obtain:

$$S : \texttt{list}, \Gamma \vdash \texttt{sort}(S) : \texttt{eval}(\theta_{sort}\, S) \tag{19}$$

$$B : \texttt{list}, \Gamma \vdash \texttt{sort}(B) : \texttt{eval}(\theta_{sort}\, B) \tag{20}$$

$$\Gamma \vdash \texttt{append}(L_S, \texttt{[}P|L_B\texttt{]}) : \phi_{append}\, (L_S, \texttt{[}P|L_B\texttt{]}) \tag{21}$$

$$\Gamma, L_S : \theta_{sort}\, S, L_B : \theta_{sort}\, B, L : \theta_{append}\, (L_S, \texttt{[}P|L_B\texttt{]}) \vdash L : \theta_{sort}\, \texttt{[}P|R\texttt{]} \tag{22}$$

Goals (19) and (20) are instances of goal (11), and can therefore be eliminated (note that the least fixed-point formula `list` was unfolded along the way, allowing application of the rule of discharge). Goal (21) is once again an instance of the typical sequent adressed in the previous section. What is left is goal (22), in which no Erlang expressions occur anymore, just variables. The treatment of such sequents is entirely standard and outside the scope of this paper.

The details of the verification of the quicksort algorithm can be found in a separate paper [4]. The approach to verification of side-effect free code is highly compositional which greatly facilitated the reuse of proof results. Around 30 lemmata were proved and used to produce a well-structured and economic proof.

6 Conclusion

We outlined an idea for an approach for factoring out the reasoning about the behaviour of side-effect-free Erlang expressions from the reasoning about general Erlang systems, and the subtask of verifying side-effect-free Erlang expressions, which was illustrated on the concrete example of the well-known quicksort algorithm. Future effort is needed to support the proposed approach by theoretical results, such as a proof of admissability of the ValCut rule.

Because of its simplicity, it is appealing to attempt to adapt the same reasoning scheme to Erlang expressions which do exhibit side-effects. This will in general be connected with significant complications, especially when the side-effects include process spawning, and remains a topic of future research.

Acknowledgement. The authors would like to thank Mads Dam and Lars-åke Fredlund at the Swedish Institute of Computer Science for carefully reading and commenting on the manuscript. Special thanks are also due to the anonymous referree who rejected the paper but provided a series of excellent suggestions for improving the scientific value of the paper.

References

[1] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang (Second Edition).* Prentice-Hall International (UK) Ltd., 1996.

[2] T. Arts and M. Dam. Verifying a distributed database lookup manager written in Erlang. In *Proc. Formal Methods Europe'99,* Lecture Notes in Computer Science, 1708:682–700, 1999.

[3] T. Arts, M. Dam, L.-å. Fredlund, and D. Gurov. System description: Verification of distributed Erlang programs. In *Proc. CADE'98,* Lecture Notes in Artificial Intelligence, 1421:38–41, 1998.

[4] G. Chugunov and L.-å. Fredlund. Verifying sequential Erlang programs. Technical Report T2000:02, Swedish Institute of Computer Science, 2000.

[5] M. Dam. Proving properties of dynamic process networks. *Information and Computation*, 140:95–114, 1998.

[6] M. Dam, L.-å. Fredlund, and D. Gurov. Toward parametric verification of open distributed systems. In *Compositionality: the Significant Difference,* H. Langmaack, A. Pnueli and W.-P. de Roever (eds.), Springer, 1536:150–185, 1998.

[7] L.-å. Fredlund. Towards a semantics for Erlang. Unpublished manuscript, *Swedish Institute of Computer Science*, 1999.

[8] L.-å. Fredlund and D. Gurov. A framework for formal reasoning about open distributed systems. In *Proc. ASIAN'99,* Lecture Notes in Computer Science, 1742:87–100, 1999.

[9] C. A. Gunter. *Semantics of Programming Languages.* The MIT Press, 1992. (See chapter 4.1).

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[11] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, **27**:333–354, 1983.

[12] D. Park. Finiteness is mu-Ineffable. *Theoretical Computer Science*, **3**:173–181, 1976.

[13] G. D. Plotkin. A structural approach to operational semantics. Aarhus University report DAIMI FN-19, 1981.