Specialization of Distributed Actors by Partial Evaluation

Germán Vidal
MiST, DSIC, Universitat Politècnica de València
Valencia, Spain
gvidal@dsic.upv.es

***Abstract*—Partial evaluation is a well-established technique for program specialization that might achieve dramatic runtime speedups. While it has been widely studied for sequential languages, partial evaluation of concurrent and distributed programs has received little attention. In this paper, we consider an asynchronous message passing language that can be seen as a simple but significant version of the concurrent and distributed language Erlang. We introduce a hybrid partial evaluation scheme for this language, and illustrate it with an example. A prototype tool has been implemented and is publicly available. To the best of our knowledge, this is the first approach to the partial evaluation of an asynchronous message passing language like Erlang.**

***Index Terms*—partial evaluation, specialization, actor model, concurrency, distributed systems**

I. Introduction

The actor model [1] is a formalism for concurrent computation which is based on the notion of *actor*. In this model, actors can only communicate through messages (i.e., there is no shared memory). Every actor can send and receive messages (which are stored in a local mailbox until they are consumed), as well as spawn new actors. This formalism has been used for modeling distributed systems like, e.g., web services. Erlang [2] is a message passing concurrent and distributed functional programming language based on the actor model. It has been successfully used in the development of many applications where massive concurrency and fault-tolerance are essential. In this work, we consider a simplified but significant version of Erlang in order to focus on some interesting aspects of the language.

Partial evaluation is a well-known technique for program specialization [3]. Essentially, it takes a program and part of its input data, the so called *static* data, and produces a *residual* program which is specialized for these data. Following [3], let us consider that the *semantics* of a program P is denoted by $[\![P]\!]$, i.e., $[\![P]\!]$ is a function from P's inputs to P's outputs, and let mix be a partial evaluator. Given a program P with two inputs, in_1 and in_2, the partial evaluation of P w.r.t. in_1 (static data) is denoted by $[\![\mathsf{mix}]\!](P, in_1)$. Correctness is then stated as follows:

$$[\![[\![\mathsf{mix}]\!](P, in_1)]\!](in_2) = [\![P]\!](in_1, in_2)$$

This work has been partially supported by the EU (FEDER) and the *Spanish Ministerio de Ciencia, Innovación y Universidades*/AEI under grant TIN2016-76843-C4-1-R and by the *Generalitat Valenciana* under grant PROMETEO-II/2015/013 (SmartLogic).

i.e., running the partially evaluated program with the remaining data in_2, the so called *dynamic* data, must return the same result as the original program with all input data in_1, in_2.

Performance improvements are basically achieved by propagating known values and unfolding operations when enough information is available. In some cases, dramatic speedups can be achieved (while performance penalties are unusual).

There are basically two types of partial evaluators, depending on when termination issues are considered: offline and online. *Offline* partial evaluators consider two separate stages: binding-time analysis (BTA) and proper partial evaluation. The BTA typically produces a program instrumented with termination annotations that ensure the termination of partial evaluation. In contrast, *online* partial evaluators are monolithic procedures that perform partial evaluation and deal with termination issues at the same time.

We can find many approaches to partial evaluation of functional languages in the literature (e.g., [4]–[6]). Most of these approaches consider an offline scheme (a notable exception being [7], which is online). Another close techniques are (positive) supercompilation [8] and narrowing-driven partial evaluation [9], [10], which are typically online and achieve more powerful optimizations than standard partial evaluation.

While partial evaluation has been widely studied for sequential languages, partial evaluation of concurrent and distributed programs has received little attention. Dealing with concurrent programs is difficult since the propagation of static data should also take into account shared data structures and/or the values sent through messages. One of the first approaches was introduced by Hosoya et al [11]. They present an online approach to partial evaluation of a concurrent process calculus, which is still far from a realistic programming language like Erlang. Most of the previous approaches, though, consider an offline partial evaluation scheme which includes a BTA that infers information about the possible runtime communications (e.g., [12]–[14]). While [12] considers a CSP-like language, [13] and [14] deal with concurrent ML. In all these cases, in contrast to our considered language, communication is synchronous. Moreover, they aim at removing communications at partial evaluation time, which may change the possible runtime behaviour of the programs.

In this paper, we present a novel approach to partial evaluation in Erlang. We consider a hybrid scheme where a static analysis is used to infer a sound underapproximation of the

$$
\begin{array}{rcl}
\mathit{program} & ::= & \mathit{fun}_1\ \cdots\ \mathit{fun}_n \\
\mathit{fun} & ::= & f(X_1,\ldots,X_n) \doteq e \\
\mathit{lit} & ::= & \mathit{Atom} \mid \mathit{Integer} \mid \mathit{Float} \mid [\,] \\
e \ni \mathit{expr} & ::= & \mathit{Var} \mid \mathit{lit} \mid [e_1|e_2] \mid \{e_1,\ldots,e_n\} \\
& \mid & \alpha(p_1,\ldots,p_n) \mid f(p_1,\ldots,p_n) \\
& \mid & \mathsf{let}\ X = e_1\ \mathsf{in}\ e_2 \\
& \mid & \mathsf{case}\ e\ \mathsf{of}\ cl_1;\ldots;cl_m\ \mathsf{end} \\
& \mid & \mathsf{spawn}(f(p_1,\ldots,p_n)) \\
& \mid & \mathsf{send}(p_1,p_2) \mid \mathsf{self}() \\
& \mid & \mathsf{receive}\ cl_1;\ldots;cl_n\ \mathsf{end} \\
cl \ni \mathit{clause} & ::= & p\ \mathsf{when}\ e_1 \to e_2 \\
p \ni \mathit{pat} & ::= & \mathit{Var} \mid \mathit{lit} \mid [p_1|p_2] \mid \{p_1,\ldots,p_n\}
\end{array}
$$

Fig. 1. Language syntax rules

messages sent at runtime (i.e., using the static data available at partial evaluation time). An online partial evaluator then takes this information into account in order to also specialize the statements for receiving messages. In this way, we can obtain a degree of specialization that previous approaches based solely on the specialization of sequential Erlang code (like [15]) cannot achieve. A prototype implementation of the partial evaluator, called erlmix, is publicly available from `https://github.com/mistupv/erlmix`.

II. The Language

In this section, we introduce the syntax and (informal) semantics of the considered language. Its syntax can be found in Fig. 1. Here, a function is defined by a single rule, whose left-hand side contains different variables as arguments and whose right-hand side is an expression possibly containing atoms (i.e., user-defined constants), variables, numbers, tuples, lists,[1] calls to built-in operators (e.g., arithmetic or relational operators), function calls, let bindings, and case expressions. Atoms are typically denoted with roman letters, while variables start with an uppercase letter; built-in operators are denoted with the greek symbol α, and user-defined functions with the letters $f, g, h, \ldots$ Without loss of generality, we require the arguments of function and built-in calls to be patterns. This simplifies our developments, and functions with arbitrary arguments can still be transformed into the required form by using let bindings.

In addition to the previous constructs, we also have some basic non-functional operations with side-effects to spawn new processes, send messages, receive messages, and get the *process identifier* (pid) of a given process.

We consider the usual domains in this language: expressions, patterns (built from variables, literals, lists and tuples) and *values* (i.e., ground—without variables—patterns). Values are denoted by $v, v', v_1, v_2, \ldots$ A *substitution* θ is a mapping from variables to expressions, and $\mathcal{D}om(\theta) = \{X \in \mathit{Var} \mid X \neq \theta(X)\}$ is its domain. Substitutions are usually denoted by sets of bindings, e.g., $\{X_1 \mapsto e_1, \ldots, X_n \mapsto e_n\}$. Substitutions are extended to morphisms from expressions to expressions in the natural way. We consider a postfix notation for substitution application, i.e., given an expression e and a substitution σ, the application $\sigma(e)$ is denoted by $e\sigma$. The identity substitution is denoted by id. Composition of substitutions is denoted by juxtaposition, i.e., $\theta\theta'$ denotes a substitution θ'' such that $\theta''(X) = \theta'(\theta(X))$ for all $X \in \mathit{Var}$.

The intuitive semantics of the functional constructs is the usual one. Built-in's are evaluated using an external function eval. For user-defined function calls, we consider a call-by-value semantics where function parameters are evaluated before unfolding a call. In a let binding of the form let $X = e_1$ in e_2, we first evaluate e_1 to a value, say v, and then proceed with the evaluation of $e_2\{X \mapsto v\}$. In a case expression "case e of p_1 when $e_1 \to e'_1$; $\ldots$; p_n when $e_n \to e'_n$ end" we first evaluate e to a value, say v; then, we find (if it exists) the first clause p_i when $e_i \to e'_i$ such that v matches p_i, i.e., there exists a substitution σ for the variables of p_i such that $v = p_i\sigma$, and $e_i\sigma$—the *guard*—evaluates to true; we use the external function eval to evaluate guards, so the above condition is denoted by $\mathsf{eval}(e_i\sigma) = \mathit{true}$. If these conditions hold, the case expression above reduces to $e'_i\sigma$. Guards can only contain calls to built-ins (typically, arithmetic and relational operators).

The execution of a program gives rise to a number of processes (actors) that interact only through message passing (i.e., there is no shared memory). Messages are stored in a FIFO queue, which is local to each process, until they are consumed by a receive expression. Message sending is asynchronous, while receive instructions block the execution of a process until a matching message reaches its local queue.

Let us now consider the functions with side-effects (most of which are built-in's in the Erlang programming language): self, spawn, send, and receive. The expression self() returns the pid of a process, while send(π, v) sends a message v to the process with pid π, which will be eventually stored in π's local queue. New processes are spawned with a call of the form spawn$(f(v_1,\ldots,v_n))$, so that the new process begins with the evaluation of $f(v_1,\ldots,v_n)$. Finally, an expression

$$\mathsf{receive}\ p_1\ \mathsf{when}\ e_1 \to e'_1;\ldots;p_n\ \mathsf{when}\ e_n \to e'_n\ \mathsf{end}$$

traverses the messages in the process' queue until one of them matches a branch in the receive statement; i.e., it should find the *first* message v in the process' queue (if any) such that case v of p_1 when $e_1 \to e'_1;\ldots;p_n$ when $e_n \to e'_n$ end can be reduced; then, the receive expression evaluates to the same expression to which the above case expression would be evaluated, with the additional side effect of deleting the message v from the process' queue. If there is no matching message in the queue, the process *suspends* its execution until a matching message arrives.

Example 1: Figure 2 shows a program implementing a simple client-server scheme with one server and a generic number N of clients. Execution starts with a call to $\mathit{main}(N, L)$, where N is a natural number and L is a list. It calls to function $\mathit{init}(N, L, S)$ in order to create the N clients, where S is the pid of the current process (i.e., the server), and then calls function $\mathit{server}()$ in order to start the server. Every client

[1] Following Erlang, [] denotes an empty list while $[H|T]$ denotes a list with head element H and tail T (a list).

$$
\begin{array}{rl}
main(N,L) \doteq & \mathsf{let}\ S = \mathsf{self}() \\
& \mathsf{in\ let}\ W = init(N,L,S)\ \mathsf{in}\ server() \\
server() \doteq & \mathsf{receive} \\
& \quad \{C,L,N\} \to \mathsf{let}\ V = \{\mathrm{ack}, rep(L,N)\} \\
& \qquad\qquad \mathsf{in\ let}\ W = \mathsf{send}(C,V) \\
& \qquad\qquad \mathsf{in}\ server(); \\
& \quad X \to \mathrm{error} \\
& \mathsf{end} \\
init(N,L,S) \doteq & \mathsf{case}\ N\ \mathsf{of} \\
& \quad 0 \to \mathrm{done}; \\
& \quad M\ \mathsf{when}\ M > 0 \\
& \qquad \to \mathsf{let}\ C = spawn(client(M,L,S)) \\
& \qquad\quad \mathsf{in}\ init(M-1,L,S) \\
& \mathsf{end} \\
client(N,L,S) \doteq & \mathsf{let}\ C = \mathsf{self}()\ \mathsf{in\ let}\ W = \mathsf{send}(S,\{C,L,N\}) \\
& \mathsf{in\ receive} \\
& \quad \{\mathrm{ack}, Res\} \to \mathrm{ok} \\
& \mathsf{end} \\
rep(L,N) \doteq & \mathsf{case}\ L\ \mathsf{of} \\
& \quad [\,] \to [\,]; \\
& \quad [H|R] \to [aux(H,N)|rep(R,N)] \\
& \mathsf{end} \\
aux(X,N) \doteq & \mathsf{case}\ N\ \mathsf{of} \\
& \quad 0 \to [\,]; \\
& \quad M\ \mathsf{when}\ M > 0 \to [X|aux(X,M-1)] \\
& \mathsf{end}
\end{array}
$$

Fig. 2. A simple message-passing program

performs a simple request of the form $\{C, L, N\}$ where C is the pid of the client. For each request, the server performs a simple operation: it replaces every element X of L by a list with N repetitions of element X, returns this new list to the client, and calls to function $server$ again in an endless loop. If the message does not have the right structure, the catch-all clause "$X \to \mathrm{error}$" returns error and stops.

Despite its simplicity, the interesting point in this example is that, even if we know some of the input parameters, say the value of N in the initial call to $main$, existing specialization techniques for (sequential) functional programs (and also for Erlang [15]) are not able to achieve any specialization since the value N is passed to the server through messages. □

By lack of space, we will not introduce the formal semantics of the language in this paper but refer the interested reader to [16] where a similar language is introduced.

III. The Specialization Scheme

In this section, we present a specialization scheme for our asynchronous message passing language.

In the following, we consider a fixed program Pgm and assume that partial evaluation begins with a call of the form $main(p_1, \ldots, p_n)$, where p_i is either a value v_i (a static parameter) or a variable X_i (a dynamic parameter).

A. Message Analysis

In general, a message analysis computes an approximation of the messages sent at runtime. In [17], for instance, the authors present a message analysis for Erlang that *overapproximates* the values of runtime messages. In contrast to [17], we need an analysis that takes into account the static data; moreover, an underapproximation is more appropriate in our setting (here, the degree of precision would only affect to the quality of the specialization but not to its correctness).

$$
\begin{array}{l}
\mathsf{mix}(calls, seen, \Delta) = \\
\quad \mathsf{if}\ calls = [\,]\ \mathsf{then}\ \emptyset \\
\quad \mathsf{else\ let}\ [\{\pi, f(p_1,\ldots,p_n)\}|calls'] = calls\ \mathsf{in} \\
\qquad \mathsf{let}\ e = [\![f(p_1,\ldots,p_n)]\!]^{[\,]}_{\Delta(\pi)}\ \mathsf{in} \\
\qquad \mathsf{let}\ calls_e = \mathsf{succ}(\pi, e)\ \mathsf{in} \\
\qquad \mathsf{let}\ nseen = seen \mathbin{+\!\!+} [\{\pi, f(p_1,\ldots,p_n)\}]\ \mathsf{in} \\
\qquad \mathsf{let}\ ncalls = (calls' \mathbin{+\!\!+} calls_e) \setminus nseen\ \mathsf{in} \\
\qquad [f(p_1,\ldots,p_n) \doteq e] \mathbin{+\!\!+} \mathsf{mix}(ncalls, nseen, \Delta)
\end{array}
$$

Fig. 3. Partial evaluation procedure

Essentially, we consider a symbolic execution version of the language semantics in order to simulate all possible executions with the static data. Termination can be ensured, e.g., using a depth bound or a timeout. In this way, we compute a sound *underapproximation* of the (partially known) messages sent at runtime. The output of this stage is a set, Δ, with pairs of the form $\{\pi, p\}$, where π is a pid and p is pattern, i.e., a partial value denoting a message sent to process π.

Example 2: Let us consider, for instance, the program of Example 1 and the initial call $main(2, L)$. The execution of (any instance of) this call will create three processes, with pids, e.g., s (for server), c1 (for client #1) and c2 (for client #2). For simplicity, we consider that the assignment of new pids is deterministic, so they will be the same in all the executions of the program. In practice, one can use the Erlang built-in "register" to assign a fixed name to every process (as we do in erlmix). Here, the message analysis returns the following sets of messages:

$$
\Delta = \left\{ \begin{array}{l} (\mathsf{s}, \{\mathsf{c1}, L, 1\}), (\mathsf{s}, \{\mathsf{c2}, L, 2\}), \\ (\mathsf{c1}, \{\mathrm{ack}, Res\}), (\mathsf{c2}, \{\mathrm{ack}, Res'\}) \end{array} \right\}
$$

B. Online Partial Evaluation

The next stage is the proper partial evaluation. It takes the program Pgm, the initial call $main(p_1, \ldots, p_n)$, and the set of (possibly partial) messages, Δ, and produces a residual, specialized program. In contrast to other approaches, we do not change the runtime communications of the program (as in, e.g., [12]–[14]). In other words, in our setting, the same messages will be sent in the original and specialized programs at runtime, although these messages will (hopefully) be processed in a more efficient way thanks to the specialization.

The kernel of our specialization scheme is the partial evaluation procedure mix shown in Fig. 3. It takes a list of calls to be partially evaluated with their associated pids (initially $[\{\mathsf{s}, main(p_1, \ldots, p_n)\}]$, a list of calls already partially evaluated (initially $[\,]$), and a set of messages Δ. It then proceeds as follows:

- The first function call in $calls$ is specialized using the partial evaluation semantics shown in Fig. 4 (see below), thus producing a residual expression e. The partial evaluation semantics has two additional parameters: the list of unfolded calls (initially empty), which is used to

ensure the so called *local* termination, and the list of messages for the current process, which is needed to produce specialized clauses in receive statements.
- Then, we update the list $seen$ by adding the pair $\{\pi, f(p_1, \ldots, p_n)\}$ just partially evaluated, resulting in the new list $nseen$.
- Function succ is used to extract the function calls in e. The set $ncalls$ removes from $calls'$ ++ $\mathsf{succ}(\pi, e)$ those calls already partially evaluated (i.e., $nseen$). Here, we consider that ++ and $\setminus$ denote list concatenation and list subtraction modulo *variants* (i.e., considering all variables as a single value).
- Finally, we produce a residual, specialized function, $f(p_1, \ldots, p_n) \doteq e$, and call recursively to function mix with the updated parameters until there are no more calls to be specialized.

The partial evaluation semantics is shown in Fig. 4.

In the following, we let $[\![e]\!]_q^{\mathcal{A}} \downarrow$ denote the normal (or irreducible) form of e according to the partial evaluation semantics, i.e., an expression e' such that $[\![e]\!]_q^{\mathcal{A}} \Rightarrow \ldots \Rightarrow e' \not\Rightarrow$. By definition, e' cannot contain occurrences of $[\![\ ]\!]$. Moreover, we consider a well-founded order [18], denoted by $\triangleright$, which guarantees that we cannot have an infinite sequence of the form $e_1 \triangleright e_2 \triangleright \ldots$ Given a list of calls $\mathcal{A}$ and another call e, we let $\mathcal{A} \triangleright e$ denote that e is smaller than the last call with the same function symbol in $\mathcal{A}$ (this is safe since the number of different function symbols in a program is finite). Le us now briefly explain the rules of the partial evaluation semantics:

- The first rules are self-explanatory: patterns are not partially evaluated (they are considered *values* at partial evaluation time), while tuples and lists are partially evaluated by recursively partially evaluating their elements.
- In rule Built-in, we first partially evaluate the parameters of the operation. If all of them can be reduced to a value, then we execute the built-in using the auxiliary call eval; otherwise, we residualize it.
- The partial evaluation of a function call starts by partially evaluating its arguments. Then, we distinguish two cases: if the new call is "smaller" than the last call with the same function symbol (if any) in $\mathcal{A}$, we unfold the call and continue partially evaluating it; we also add this call to the current set of unfolded calls. Otherwise, we residualize the call and partially evaluate its arguments.
- In a let expression of the form let $X = e_1$ in e_2, we first partially evaluate e_1 up to its normal form. If we get a pattern, p_1, then we continue with the partial evaluation of $e_2\{X \mapsto p_1\}$. Otherwise, we partially evaluate both e_1 and e_2 but residualize the let statement.
- Case expressions are partially evaluated in a similar way. First, the case argument is partially evaluated to its normal form. If we get a pattern, we execute the case statement and continue with the partial evaluation of the selected branch. Otherwise, we residualize the case statement and partially evaluate its clauses.
- The concurrent actions spawn, send and self are always residualized (and their arguments partially evaluated).
- Finally, given a receive statement, we keep adding new, specialized clauses as long as there are messages in q that match (in a non-trivial way) some pattern and the corresponding guard, if any, evaluates to $true$. Once no more specialized clauses are added, the second rule residualizes the receive statement[2] and continue partially evaluating its clauses.

Termination of the partial evaluation semantics is guaranteed by the fact that no concurrent action is executed and the use of a well-founded order to avoid unfolding infinite calls.

C. Post-Processing Renaming

Once the partial evaluation procedure of Fig. 3 terminates computing a residual program Pgm' for the set of calls $\mathcal{C}$ (i.e., the left-hand sides of the program rules or, equivalently, the value of $seen$ in the last iteration of the partial evaluation procedure), we apply a standard post-processing of renaming. Renaming is necessary for Pgm' to respect the syntax of Fig. 1. It also helps to further optimize the code, removing unnecessary data and avoiding overlapping functions.

Basically, we map every call $f(p_1, \ldots, p_n)$ in $\mathcal{C}$ to an expression of the form $f'(X_1, \ldots, X_m)$, where $X_1, \ldots, X_m$ are the variables of $f(p_1, \ldots, p_n)$. We then use this mapping to rename the calls to these functions in the bodies of the program functions. By construction, we know that these functions must be instances of some call in $\mathcal{C}$. Therefore, given a call $f(p'_1, \ldots, p'_n)$ such that $f(p'_1, \ldots, p'_n) = f(p_1, \ldots, p_n)\sigma$, we replace it by $f'(X_1, \ldots, X_n)\sigma$. See, e.g., [19], for more details on the post-processing of renaming.

D. The Partial Evaluation Scheme in Practice

Let us now illustrate the specialization process with an example. Consider the program from Example 1 (Fig. 2). Given the initial call $main(2, L)$, the message analysis computes the set Δ shown in Example 2. Thus, the partial evaluation procedure starts with the call $\mathsf{mix}([\{\mathsf{s}, main(2, L)\}], [\,], \Delta)$.

Therefore, our first call to the partial evaluation semantics is $[\![main(2, L)]\!]_q^{[\,]}$, with $q = \Delta(\mathsf{s}) = [\{\mathsf{c1}, L, 1\}, \{\mathsf{c2}, L', 2\}]$.

In the following, for clarity, we do not unfold all function calls even if they are smaller than previous calls; in this way, the partially evaluated program is kept as close to the original one as possible.[3] The associated derivation is as follows:

$$
\begin{array}{l}
[\![main(2, L)]\!]_q^{[\,]} \\
\Rightarrow [\![\mathsf{let}\ S = \mathsf{self}()\ \mathsf{in} \ldots]\!]_q^{\mathcal{A}_1} \\
\Rightarrow \mathsf{let}\ S = \mathsf{self}()\ \mathsf{in}\ [\![\mathsf{let}\ W = init(2, L, S)\ \mathsf{in} \ldots]\!]_q^{\mathcal{A}_1} \\
\Rightarrow \mathsf{let}\ S = \mathsf{self}()\ \mathsf{in}\ \mathsf{let}\ W = init(2, L, S)\ \mathsf{in}\ [\![server()]\!]_q^{\mathcal{A}_2} \\
\Rightarrow \mathsf{let}\ S = \mathsf{self}()\ \mathsf{in}\ \mathsf{let}\ W = init(2, L, S)\ \mathsf{in}\ server()
\end{array}
$$

[2]Note that receive statements are never removed, in contrast to case expressions, for instance.

[3]The implemented tool allows the user to add annotations on some functions to achieve this effect.

$$
\begin{array}{lrl}
\textsf{Pattern} & [\![p]\!]_q^{\mathcal{A}} & \Rightarrow p \\
\textsf{List} & [\![[e_1|e_2]]\!]_q^{\mathcal{A}} & \Rightarrow [[\![e_1]\!]_q^{\mathcal{A}}|[\![e_2]\!]_q^{\mathcal{A}}] \\
\textsf{Tuple} & [\![\{e_1,\ldots,e_n\}]\!]_q^{\mathcal{A}} & \Rightarrow \{[\![e_1]\!]_q^{\mathcal{A}},\ldots,[\![e_n]\!]_q^{\mathcal{A}}\} \\
\textsf{Built-in} & [\![\alpha(p_1,\ldots,p_n)]\!]_q^{\mathcal{A}} & \Rightarrow v \;\; \text{if } [\![p_i]\!]_q^{\mathcal{A}}{\downarrow} = v_i,\; i=1,\ldots,n \text{ and } \textsf{eval}(\alpha(v_1,\ldots,v_n)) = v \\
 & [\![\alpha(p_1,\ldots,p_n)]\!]_q^{\mathcal{A}} & \Rightarrow \alpha([\![p_1]\!]_q^{\mathcal{A}},\ldots,[\![p_n]\!]_q^{\mathcal{A}}) \;\; \text{otherwise} \\
\textsf{Fun call} & [\![f(p_1,\ldots,p_n)]\!]_q^{\mathcal{A}} & \Rightarrow [\![e\sigma]\!]_q^{\mathcal{A}'} \;\; \text{if } [\![p_i]\!]_q{\downarrow} = p_i',\; i=1,\ldots,n, \mathcal{A} \triangleright f(p_1',\ldots,p_n'),\; \mathcal{A}' = [f(p_1',\ldots,p_n') \mid \mathcal{A}], \\
 & & \qquad f(X_1,\ldots,X_n) \to e \ll Pgm, \text{ and } \sigma = \{X_1 \mapsto p_1',\ldots,X_n \mapsto p_n'\} \\
 & [\![f(p_1,\ldots,p_n)]\!]_q^{\mathcal{A}} & \Rightarrow f([\![p_1]\!]_q^{\mathcal{A}},\ldots,[\![p_n]\!]_q^{\mathcal{A}}) \\
\textsf{Let} & [\![\textsf{let } X = e_1 \textsf{ in } e_2]\!]_q^{\mathcal{A}} & \Rightarrow [\![e_2\sigma]\!]_q^{\mathcal{A}} \;\; \text{if } [\![e_1]\!]_q^{\mathcal{A}}{\downarrow} = p_1 \text{ and } \sigma = \{X \mapsto p_1\} \\
 & [\![\textsf{let } X = e_1 \textsf{ in } e_2]\!]_q^{\mathcal{A}} & \Rightarrow \textsf{let } X = [\![e_1]\!]_q^{\mathcal{A}} \textsf{ in } [\![e_2]\!]_q^{\mathcal{A}} \;\; \text{otherwise} \\
\textsf{Case} & [\![\textsf{case } e \textsf{ of } cl_1;\ldots;cl_n \textsf{ end}]\!]_q^{\mathcal{A}} & \Rightarrow [\![e_i'\sigma]\!]_q^{\mathcal{A}} \;\; \text{if } [\![e]\!]_q^{\mathcal{A}}{\downarrow} = p,\; cl_i = (p_i \textsf{ when } e_i \to e_i') \text{ is the first clause such that} \\
 & & \qquad p_i\sigma = p \text{ and } \textsf{eval}(e_i\sigma) = true \\
 & [\![\textsf{case } e \textsf{ of } cl_1;\ldots;cl_n \textsf{ end}]\!]_q^{\mathcal{A}} & \Rightarrow \textsf{case } [\![e]\!]_q^{\mathcal{A}} \textsf{ of } [\![cl_1]\!]_q^{\mathcal{A}};\ldots;[\![cl_n]\!]_q^{\mathcal{A}} \textsf{ end} \quad \text{otherwise} \\
\textsf{Spawn} & [\![\textsf{spawn}(f(p_1,\ldots,p_n))]\!]_q^{\mathcal{A}} & \Rightarrow \textsf{spawn}(f([\![p_1]\!]_q^{\mathcal{A}},\ldots,[\![p_n]\!]_q^{\mathcal{A}})) \\
\textsf{Send} & [\![\textsf{send}(p_1,p_2)]\!]_q^{\mathcal{A}} & \Rightarrow \textsf{send}([\![p_1]\!]_q^{\mathcal{A}},[\![p_2]\!]_q^{\mathcal{A}}) \\
\textsf{Receive} & [\![\textsf{receive } cl_1;\ldots;cl_n \textsf{ end}]\!]_q^{\mathcal{A}} & \Rightarrow [\![\textsf{receive } cl_1;\ldots;cl_{i-1};p \textsf{ when } e_i\sigma \to e_i'\sigma;cl_i;\ldots;cl_n \textsf{ end}]\!]_q^{\mathcal{A}} \\
 & & \qquad \text{if } p \in q,\; cl_i = (p_i \textsf{ when } e_i \to e_i') \text{ is the first clause such that } p_i\sigma = p, \\
 & & \qquad \textsf{eval}(e_i\sigma) = true \text{ and } \sigma \text{ is not a renaming} \\
 & [\![\textsf{receive } cl_1;\ldots;cl_n \textsf{ end}]\!]_q^{\mathcal{A}} & \Rightarrow \textsf{receive } [\![cl_1]\!]_q^{\mathcal{A}};\ldots;[\![cl_n]\!]_q^{\mathcal{A}} \textsf{ end} \quad \text{otherwise} \\
\textsf{Self} & [\![\textsf{self}()]\!]_q^{\mathcal{A}} & \Rightarrow \textsf{self}() \\
\textsf{Clause} & [\![p \textsf{ when } e \to e']\!]_q^{\mathcal{A}} & \Rightarrow p \textsf{ when } [\![e]\!]_q^{\mathcal{A}} \to [\![e']\!]_q^{\mathcal{A}}
\end{array}
$$

Fig. 4. Partial evaluation semantics for actors

with $\mathcal{A}_1 = [main(2,L)], \mathcal{A}_2 = [init(2,L,S), main(2,L)]$. Therefore, the associated residual rule is

$$
\begin{array}{l}
main(2,L) \doteq \textsf{let } S = \textsf{self}() \textsf{ in} \\
\qquad\qquad\qquad \textsf{let } W = init(2,L,S) \textsf{ in } server()
\end{array}
$$

Then, function succ returns $[\{\textsf{s}, init(2,L,S)\}, \{\textsf{s}, server()\}]$, which are added to the set of calls to be partially evaluated. Let us first consider $init(2,L,S)$:

$$
\begin{array}{l}
[\![init(2,L,S)]\!]_q^{[\,]} \\
\Rightarrow [\![\textsf{case } 2 \textsf{ of} \ldots]\!]_q^{\mathcal{A}_1} \\
\Rightarrow [\![\textsf{let } C = \textsf{spawn}(client(2,L,S)) \textsf{ in} \ldots]\!]_q^{\mathcal{A}_1} \\
\Rightarrow \textsf{let } C = \textsf{spawn}(client(2,L,S)) \textsf{ in } [\![init(2-1,L,S)]\!]_q^{\mathcal{A}_1} \\
\Rightarrow \textsf{let } C = \textsf{spawn}(client(2,L,S)) \textsf{ in } [\![\textsf{case } 1 \textsf{ of} \ldots]\!]_q^{\mathcal{A}_2} \\
\Rightarrow \ldots \\
\Rightarrow \textsf{let } C = \textsf{spawn}(client(2,L,S)) \textsf{ in} \\
\quad \textsf{let } C' = \textsf{spawn}(client(1,L,S)) \textsf{ in done}
\end{array}
$$

with $\mathcal{A}_1 = [init(2,L,S), \mathcal{A}_2 = [init(1,L,S), init(2,L,S)]$. Here, $\mathcal{A}_1 \triangleright init(1,L,S)$ and, thus, the function call is unfolded. The associated residual rule is as follows:

$$
\begin{array}{l}
init(2,L,S) \doteq \textsf{let } C = \textsf{spawn}(client(2,L,S)) \textsf{ in} \\
\qquad\qquad\qquad \textsf{let } C' = \textsf{spawn}(client(1,L,S)) \textsf{ in done}
\end{array}
$$

Then, function succ extracts the following new calls to be partially evaluated: $[\{\textsf{c1}, client(1,L,S)\}, \{\textsf{c2}, client(2,L,S\}]$. The partial evaluation of these calls is very simple and we omit it.[4] The next call to be partially evaluated—the most interesting one—is the following:

$$
\begin{array}{l}
[\![server()]\!]_q^{[\,]} \\
\Rightarrow [\![\textsf{receive } \{C,L,N\} \to \ldots; X \to \ldots \textsf{ end}]\!]_q^{\mathcal{A}_1} \\
\Rightarrow [\![\textsf{receive } \{C,L,1\} \to \ldots; \{C,L,N\} \to \ldots; \\
\qquad\qquad X \to \ldots \textsf{ end}]\!]_q^{\mathcal{A}_1} \\
\Rightarrow [\![\textsf{receive } \{C,L,1\} \to \ldots; \{C,L,2\} \to \ldots; \\
\qquad\qquad \{C,L,N\} \to \ldots; X \to \ldots \textsf{ end}]\!]_q^{\mathcal{A}_1} \\
\Rightarrow \textsf{receive } \{C,L,1\} \to [\![\ldots]\!]_q^{\mathcal{A}_1}; \{C,L,2\} \to [\![\ldots]\!]_q^{\mathcal{A}_1}; \\
\qquad\qquad \{C,L,N\} \to [\![\ldots]\!]_q^{\mathcal{A}_1}; X \to [\![\ldots]\!]_q^{\mathcal{A}_1} \textsf{ end} \\
\Rightarrow \ldots \\
\Rightarrow \textsf{receive} \\
\quad \{C,L,1\} \to \textsf{let } V = \{\text{ack}, rep(L,1)\} \textsf{ in} \\
\qquad\qquad\qquad \textsf{let } W = \textsf{send}(C,V) \textsf{ in } server() \\
\quad \{C,L,2\} \to \textsf{let } V = \{\text{ack}, rep(L,2)\} \textsf{ in} \\
\qquad\qquad\qquad \textsf{let } W = \textsf{send}(C,V) \textsf{ in } server() \\
\quad \ldots \\
\textsf{end}
\end{array}
$$

with $\mathcal{A}_1 = [server()]$. The associated residual rule is shown in Fig. 5. Here, function succ returns the following function calls: $[\{\textsf{s}, rep(L,1)\}, \{\textsf{s}, rep(L,2)\}, \{\textsf{s}, rep(L,N)\}, \{\textsf{s}, server()\}]$. Observe that the last one is removed from the set in the

[4]Note that the receive statements are not modified since the messages gathered by the analysis are renamings of the receive's pattern.

$$
\begin{array}{rl}
main2(L) \doteq & \mathsf{let}\ S = \mathsf{self}() \\
& \mathsf{in\ let}\ W = init2(L,S)\ \mathsf{in}\ server() \\
server() \doteq & \mathsf{receive} \\
& \quad \{C,L,1\} \to \mathsf{let}\ V = \{\mathrm{ack}, rep1(L)\} \\
& \qquad\qquad \mathsf{in\ let}\ W = \mathsf{send}(C,V) \\
& \qquad\qquad \mathsf{in}\ server(); \\
& \quad \{C,L,2\} \to \mathsf{let}\ V = \{\mathrm{ack}, rep2(L)\} \\
& \qquad\qquad \mathsf{in\ let}\ W = \mathsf{send}(C,V) \\
& \qquad\qquad \mathsf{in}\ server(); \\
& \quad \{C,L,N\} \to \mathsf{let}\ V = \{\mathrm{ack}, rep(L,N)\} \\
& \qquad\qquad \mathsf{in\ let}\ W = \mathsf{send}(C,V) \\
& \qquad\qquad \mathsf{in}\ server(); \\
& \quad X \to \mathrm{error} \\
& \mathsf{end} \\
init2(L,S) \doteq & \mathsf{let}\ C = \mathsf{spawn}(client2(L,S))\ \mathsf{in} \\
& \mathsf{let}\ C' = \mathsf{spawn}(client1(L,S))\ \mathsf{in\ done} \\
client1(L,S) \doteq & \mathsf{let}\ C = \mathsf{self}()\ \mathsf{let}\ W = \mathsf{send}(S,\{C,L,1\})\ \mathsf{in} \\
& \mathsf{receive} \\
& \quad \{\mathrm{ack}, Res\} \to ok \\
& \mathsf{end} \\
client2(L,S) \doteq & \mathsf{let}\ C = \mathsf{self}()\ \mathsf{let}\ W = \mathsf{send}(S,\{C,L,2\})\ \mathsf{in} \\
& \mathsf{receive} \\
& \quad \{\mathrm{ack}, Res\} \to ok \\
& \mathsf{end} \\
rep1(L) \doteq & \mathsf{case}\ L\ \mathsf{of}\ [\,] \to [\,]; [H|R] \to [[X]|rep1(R)]\ \mathsf{end} \\
rep2(L) \doteq & \mathsf{case}\ L\ \mathsf{of}\ [\,] \to [\,]; [H|R] \to [[X,X]|rep2(R)]\ \mathsf{end}
\end{array}
$$

Fig. 5. Renamed partially evaluated program

algorithm of Fig. 3 since is has been already partially evaluated. Let us proceed with the first call:

$$
\begin{array}{l}
[\![rep(L,1)]\!]_q^{[\,]} \\
\Rightarrow [\![\mathsf{case}\ L\ \mathsf{of}\ \ldots]\!]_q^{\mathcal{A}_1} \\
\Rightarrow \mathsf{case}\ L\ \mathsf{of}\ [\,] \to [\![[\,]]\!]_q^{\mathcal{A}_1}; [H|R] \to [\![[aux(H,1)|rep(R,1)]]\!]_q^{\mathcal{A}_1} \\
\Rightarrow \mathsf{case}\ L\ \mathsf{of}\ [\,] \to [\,]; [H|R] \to [[\![aux(H,1)]\!]_q^{\mathcal{A}_1} | [\![rep(R,1)]\!]_q^{\mathcal{A}_1}] \\
\Rightarrow \mathsf{case}\ L\ \mathsf{of}\ [\,] \to [\,]; \\
\qquad\qquad [H|R] \to [[\![\mathsf{case}\ 1\ \mathsf{of}\ \ldots]\!]_q^{\mathcal{A}_2} | [\![rep(R,1)]\!]_q^{\mathcal{A}_1}] \\
\Rightarrow \ldots \\
\Rightarrow \mathsf{case}\ L\ \mathsf{of}\ [\,] \to [\,]; [H|R] \to [[X]|rep(R,1)]
\end{array}
$$

with $\mathcal{A}_1 = [rep(L,1)], \mathcal{A}_2 = [aux(H,1), rep(L,1)]$. Note that $\mathcal{A}_2 \triangleright aux(H,0)$ but $\mathcal{A}_1 \not\triangleright rep(R,1)$. The partial evaluation of $rep(L,2)$ proceeds analogously, thus producing the residual rules

$$
\begin{array}{l}
rep(L,1) \doteq \mathsf{case}\ L\ \mathsf{of}\ [\,] \to [\,]; [H|R] \to [[X]|rep(R,1)] \\
rep(L,2) \doteq \mathsf{case}\ L\ \mathsf{of}\ [\,] \to [\,]; [H|R] \to [[X,X]|rep(R,2)]
\end{array}
$$

and function succ does not return any new call from this code. The last call, $rep(L,N)$, returns the original functions rep and aux, so we omit it.

Finally, using a straightforward renaming for the residual rules, we get the specialized program shown in Fig. 5, together with the original definitions of functions rep and aux. Observe that there are several performance improvements: function $init$ is not recursive anymore and, more importantly, the specialized functions $rep1$ and $rep2$ have completely unrolled the calls to the original function aux.

IV. Discussion

We have presented a novel approach to the specialization of distributed actors using a hybrid partial evaluation scheme. The first stage is a message analysis that computes a sound underapproximation of the messages sent starting from some initial call with partially known arguments. The second stage is an (online) partial evaluation procedure that uses not only the static data but also the output of the message analysis to further specialize the code. Local termination of the partial evaluation semantics is ensured using a well-founded order.

As future work, we plan to focus on formalizing and proving the correctness the message analysis. In principle, one can use a symbolic execution extension of the standard semantics with some mechanism to ensure termination.

Acknowledgment

We would like to thank Adrián Palacios for his support with the implementation of erlmix.

References

[1] C. Hewitt, P. B. Bishop, and R. Steiger, "A universal modular ACTOR formalism for artificial intelligence," in *Proc. of IJCAI'73*, N. J. Nilsson, Ed. William Kaufmann, 1973, pp. 235–245.

[2] F. Cesarini and S. Thompson, *Erlang Programming - A Concurrent Approach to Software Development*. O'Reilly, 2009.

[3] N. Jones, C. Gomard, and P. Sestoft, *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.

[4] N. D. Jones, P. Sestoft, and H. Søndergaard, "Mix: A self-applicable partial evaluator for experiments in compiler generation," *Lisp and Symbolic Computation*, vol. 2, no. 1, pp. 9–50, 1989.

[5] A. Bondorf and O. Danvy, "Automatic autoprojection of recursive equations with global variables and abstract data types," *Science of Computer Programming*, vol. 16, no. 2, pp. 151–195, 1991.

[6] J. Launchbury, "A strongly-typed self-applicable partial evaluator," in *Proc. of FPCA'91*, vol. 523. Springer LNCS, 1991, pp. 145–164.

[7] D. Weise, R. Conybeare, E. Ruf, and S. Seligman, "Automatic online partial evaluation," in *Proc. of FPCA'91*, vol. 523. Springer LNCS, 1991, pp. 165–191.

[8] M. Sørensen, R. Glück, and N. Jones, "A Positive Supercompiler," *Journal of Functional Programming*, vol. 6, no. 6, pp. 811–838, 1996.

[9] E. Albert and G. Vidal, "The narrowing-driven approach to functional logic program specialization," *New Generation Computing*, vol. 20, no. 1, pp. 3–26, 2002.

[10] G. Vidal, "A partial evaluation tool for multi-paradigm declarative languages," in *Proc. of SMC 2002*, vol. 1, 2002, pp. 194–199.

[11] H. Hosoya, N. Kobayashi, and A. Yonezawa, "Partial evaluation scheme for concurrent languages and its correctness," in *Proc. of Euro-Par'96*, vol. 1123. Springer LNCS, 1996, pp. 625–632.

[12] M. Marinescu and B. Goldberg, "Partial-evaluation techniques for concurrent programs," in *Proc. of PEPM'97*, ACM, 1997, pp. 47–62.

[13] M. Martel and M. Gengler, "Partial evaluation of concurrent programs," in *Proc. of Euro-Par'01*, vol. 2150. Springer LNCS, 2001, pp. 504–513.

[14] J. H. Reppy and Y. Xiao, "Specialization of CML message-passing primitives," in *Proc. of POPL 2007*, ACM, 2007, pp. 315–326.

[15] G. Weinholt, "Supercompiling Erlang," Master's thesis, Chalmers University of Technology, 2013.

[16] I. Lanese, N. Nishida, A. Palacios, and G. Vidal, "A theory of reversibility for Erlang," Journal of Logical and Algebraic Methods in Programming, 2018. *To appear*.

[17] R. Carlsson, K. Sagonas, and J. Wilhelmsson, "Message analysis for concurrent programs using message passing," *ACM Transactions on Programming Languages and Systems*, vol. 28, no. 4, pp. 715–746, 2006.

[18] B. Martens, D. D. Schreye, and T. Horváth, "Sound and complete partial deduction with unfolding based on well-founded measures," *Theoretical Computer Science*, vol. 122, no. 1, pp. 97–117, 1994.

[19] J. G. Ramos, J. Silva, and G. Vidal, "Fast Narrowing-Driven Partial Evaluation for Inductively Sequential Systems," in *Proc. of ICFP'05*, ACM Press, 2005, pp. 228–239.