ERLANG/OTP Meets Dependent Types

Archibald Samuel Elliott

University of St Andrews

sam@lenary.co.uk

ACM Member Number: [REDACTED]

Category: Undergraduate

Adviser: Edwin Brady, University of St Andrews

Postal Address: [REDACTED]

1. Introduction

Concurrent programming is hard.

Combining the Actor model with a resilient runtime system and well-understood generic concurrent patterns, ERLANG/OTP [1] has provided an environment that helps programs achieve reliablility other systems can only aspire to[1].

It has done this in the distinct absence of any particularly advanced static verification system, despite how useful this could be for producing correct programs. ERLANG programmers usually turn to *dialyzer* [6, 12, 13], a static analysis tool that includes success-type checking, or *QuickCheck* [2] when looking to verify their programs.

What happens if I can use a type system to codify and verify how actors in a program communicate with each other? Surely this would make it easier to construct correct concurrent programs.

To do this, I require a way of expressing and verifying types that is compatible with existing ERLANG codebases. The way I chose to do this is to compile from a statically-typed programming language into ERLANG.

IDRIS is a general-purpose dependently-typed pure functional programming language [4]. The usefulness of a dependently-typed language is that types can be predicated on values, which allows more flexibility than conventional type systems. The reason for choosing IDRIS above *Agda* or *Coq* is that the IDRIS compiler provides a mechanism for writing new code generators without having to modify the rest of the IDRIS compiler.

[1] `https://pragprog.com/articles/erlang`

I have:

- produced a compiler that can turn dependently-typed IDRIS programs into ERLANG programs;
- devised a system for reasoning about concurrent IDRIS programs; and
- constructed a system for reasoning about the generic concurrent patterns that ERLANG/OTP provides.

2. Related Work

My approach differs from Session Types [7, 9–11, 15] in that it does not specify anything at the protocol level, and it supports dependently-typed proofs about the compiled programs.

The approach also differs from existing systems such as *verlang* [5, 14] in that ERLANG programs are produced after type checking and erasure, rather than by program synthesis.

3. Compiler

The IDRIS compiler can provide three different intermediate representations for code generation. All no longer include any dependent types. They are, from highest- to lowest-level: an IR that includes lambdas and laziness; a defunctionalised IR; and an applicative normal form IR.

I chose the defunctionalised IR as it maps most closely to ERLANG syntax. IDRIS foreign function call system required a small redesign to support more languages, and the design of a foreign export system which I assisted Brady with. IDRIS programs can now call and be called by ERLANG programs.

4. Verified Actor Systems

An actor is an isolated process with a mailbox of incoming messages, and the ability to send messages to other actors, or to spawn other actors [3, 8]. The fundamental concurrent interface to specify is to type the messages the actor will receive.

An actor-based computation, denoted `Actor l a`, is parameterised over two types: the type of messages it expects `l`, and the type the computation will perform `a`. There are three operations on these actors: receive, spawn and send.

```
data Actor : Type → Type → Type
```

Receive returns a message from the mailbox, so the value has the same type as that which it expects to receive.

```
receive : Actor l l
```

Spawn starts a new actor, which may have a different message type, and returns a reference to the new actor to the spawning actor. An actor identifier embeds the expected message type of its actor.

```
spawn : (Actor l' a) → Actor l (ActorID l')
data ActorID : Type → Type
```

Send sends a message to another actor. The function uses the message type embedded in the actor identifier to make sure the receiving actor is expecting the type of message being sent.

```
send : ActorID l' → l' → Actor l ()
```

5. Verified ERLANG/OTP Behaviours

The ERLANG/OTP libraries contain higher-level generic concurrent patterns, which build upon these basic building blocks to provide more useful abstractions such as concurrent servers (*gen_server*), concurrent FSMs (*gen_fsm*), and concurrent event handling systems (*gen_event*).

In particular, a *gen_server* is a concurrent actor which can be communicated with synchronously (calls) and asynchronously (casts). Casts can be modelled just like I have done for actors above.

Calls, on the other hand, have both a request and a response component, where the value of the request can choose the type of the response using dependent types. This allows more flexibility than a conventional type system could provide.

A similar approach seems to be applicable to concurrent FSMs and event handling systems.

Not only can I verify all calls and casts against the specified concurrent interfaces, but I can use totality checking to make sure the servers handle all possible messages. In conventional Erlang programs, concurrent interfaces are usually a lot harder to discover and understand.

6. Conclusion

I have shown that IDRIS can be used successfully for concurrent programming. With my new ERLANG code generator and associated IDRIS libraries, I can now write and run safe, flexible actor-based programs which conform to statically verified guarantees.

References

[1] J. Armstrong. A history of Erlang. In *the third ACM SIGPLAN conference*, pages 6–1–6–26, New York, NY, USA, June 2007. ACM Press.

[2] T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing telecoms software with quviq QuickCheck. In *ERLANG '06: Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, pages 2–10, New York, NY, USA, Sept. 2006. ACM Request Permissions.

[3] H. Baker and C. Hewitt. Laws for communicating parallel processes. 1977.

[4] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of functional programming*, 23:552–593, Sept. 2013.

[5] T. Carstens. verlang. URL `https://github.com/tcarstens/verlang`.

[6] M. Christakis and K. Sagonas. Static detection of race conditions in Erlang. pages 119–133, 2010.

[7] S. Fowler. Monitoried session erlang. URL `https://github.com/SimonJF/monitored-session-erlang`.

[8] C. Hewitt and H. Baker. Actors and continuous functionals. 1977.

[9] K. Honda. Types for dyadic interaction. pages 509–523, 1993.

[10] K. Honda, V. T. Vasconcelos, and M. Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *ESOP '98: Proceedings of the 7th European Symposium on Programming: Programming Languages and Systems*. Springer-Verlag, Mar. 1998.

[11] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Request Permissions, Jan. 2008.

[12] T. Lindahl and K. Sagonas. Practical type inference based on success typings. In *PPDP '06: Proceedings of the 8th ACM SIGPLAN symposium on Principles and practice of declarative programming*. ACM Request Permissions, July 2006.

[13] S. Marlow and P. Wadler. A practical subtyping system for Erlang. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*. ACM Request Permissions, Aug. 1997.

[14] C. Meiklejohn. Vector Clocks in Coq: An Experience Report. *arXiv.org*, June 2014.

[15] D. Mostrous and V. T. Vasconcelos. Session typing for a featherweight Erlang. In *COORDINATION'11: Proceedings of the 13th international conference on Coordination models and languages*. Springer-Verlag, June 2011.