Erlang/OTP Meets Dependent Types

Archibald Samuel Elliott
Adviser: Edwin Brady
School of Computer Science
University of St Andrews
St Andrews, Scotland
sam@lenary.co.uk

1. Introduction

Concurrent programming is hard.

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

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*, a static analysis tool that includes success-type checking, or *QuickCheck*, a model checking tool, 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 which 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. The usefulness of a dependently-typed language is that types can be predicated on values, which allows more flexibility than conventional type systems.

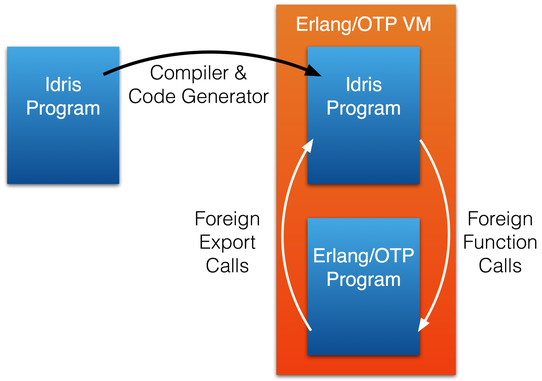
I have:

- 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. 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 A-normal form IR.

I chose the defunctionalised IR as it maps most closely to ERLANG syntax. IDRIS had a simple C foreign function system, which was redesigned so it could be extended for any foreign language, including ERLANG. Brady and I also designed a simple way of exporting IDRIS functions to foreign languages. Both of these mean that IDRIS can call any ERLANG function, and ERLANG can call any (exported) IDRIS function.



Compiler and Foreign Call System Schematic

3. 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. 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
```

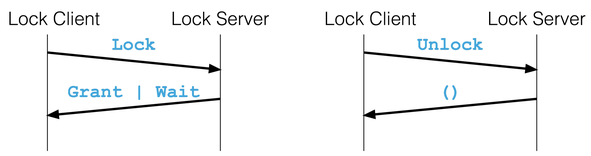
Send sends a message to another actor. The function uses the message type embedded in the actor identifier to make sure the receiving actor is expecting the type of message being sent.

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

3.1 Requests and Responses

This send and receive approach is very low-level. In lots of cases, I want a process to make a synchronous request to another concurrent process, and then receive a response.

For example, I may have a concurrent lock server. This server accepts requests of either `Lock` or `Unlock`. If we make a `Lock` request, the response is either `Grant` or `Wait`. For `Unlock` requests, the response is always `()`.



Concurrent Lock Server Example

```
data LockReq      = Lock | Unlock
data LockLockResp = Grant | Wait

total
LockResp : LockReq → Type
LockResp Lock = LockLockResp
LockResp Unlock = Unit

total
lock_srv : (r:LockReq) → LockResp r
lock_srv Lock = Wait
lock_srv Unlock = ()
```

In this example, the type of the response depends on the value of the request, which is why I need dependent types. I can define the request-response protocol in terms of the request type, and the function that computes the response type. This allows me to have a single `req` function that computes its type not only off the request-response process id, but also the request value.

```
data ReqResI : (r : Type)
             → (r → Type) → Type

data ReqResId : ReqResI r f → Type

req : {i : ReqResI r f}
    → ReqResId i
    → (m : r) → IO (f m)
```

This can be enforced on the server side by making sure the function that is spawned to become the server process has the same type as that specified in the interface.

```
spawn : {i : ReqResI r f}
      → ((m : r) → f m)
      → IO (ReqResId i)
```

To bring this all together, I can revisit my example, showing the types of various calls:

```
lock_i : ReqResI LockReq LockResp

spawn lock_srv : IO (ReqResId lock_i)

lock_id : ReqResId lock_i

req lock_id Lock   : LockLockResp
req lock_id Unlock : Unit
```

4. 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 both synchronously and asynchronously. We can model this as a combination of the models we have defined above. A similar approach seems to be applicable to concurrent FSMs and event handling systems.

Not only can I verify all communication 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.

5. Related Work

My approach differs from Session Types 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* in that ERLANG programs are produced after type checking and erasure, rather than by program synthesis.

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.

ICFP-SRC 2015, International Conference on Functional Programming, Student Research Competition (Undergraduate), September 2015, Vancouver, Canada