A Concurrency System for Idris & Erlang

Archibald Samuel Elliott
School of Computer Science
University of St Andrews
St Andrews, Scotland
ashe@st-andrews.ac.uk

Abstract

Concurrent programming is notoriously difficult, due to needing to reason not only about the sequential progress of any algorithms, but also about how information moves between concurrent agents.

What if programmers were able to reason about their concurrent programs and statically verify both sequential and concurrent guarantees about those programs' behaviour? That would likely reduce the number of bugs and defects in concurrent systems.

I propose a system combining dependent types, in the form of the Idris programming language, and the Actor model, in the form of Erlang and its runtime system, to create a system which allows me to do exactly this. By expressing these concurrent programs and properties in Idris, and by being able to compile Idris programs to Erlang, I can produce statically verified concurrent programs.

Given that these programs are generated for the Erlang runtime system, I also produce verified, flexible Idris APIs for Erlang/OTP behaviours.

What's more, with this description of the Idris code generation interface, it is now much easier to write code generators for the Idris compiler, which will bring dependently-typed programs to even more platforms and situations. I, for one, welcome our new dependently-typed supervisors.

1. Idris to Erlang Compiler

My compiler uses the existing Idris code generation system. There are three possible Idris intermediate representations that I could generate code from: a high-level IR with lambdas and laziness, a defunctionalised IR with only fully-applied functions, and an applicative-normal form IR.

In my case, we used the defunctionalised IR to generate Erlang source code from, according to the following translation, $\mathcal{E}[\![\langle expression \rangle]\!]$.

$\mathcal{V}[\![\langle variable \rangle]\!]$ turns IR variables into valid Erlang variable names; $\mathcal{C}[\![\langle name \rangle \, ; \langle expression \rangle^*]\!]$ creates constructors; $\mathcal{N}[\![\langle name \rangle]\!]$ turns IR names into valid Erlang Atoms; $\mathcal{L}[\![\langle constant \rangle]\!]$ turns IR constants into Erlang expressions; and $\mathcal{O}[\![\langle operation \rangle \, ; \langle expression \rangle^*]\!]$ turns primitive operators into their Erlang equivalent.

$$
\begin{aligned}
\mathcal{E}[\![\langle variable \rangle]\!] &\Rightarrow \mathcal{V}[\![\langle variable \rangle]\!] \\
\mathcal{E}[\![\langle name \rangle(\langle expression \rangle^*)]\!] &\Rightarrow \mathcal{C}[\![\langle name \rangle \, ; \langle expression \rangle^*]\!] \quad \text{when } \langle name \rangle \text{ is a Constructor} \\
&\Rightarrow \mathcal{N}[\![\langle name \rangle]\!]\,(\mathcal{E}[\![\langle expression \rangle_0]\!], \cdots, \mathcal{E}[\![\langle expression \rangle_{n-1}]\!]) \quad \text{otherwise} \\
\mathcal{E}[\![\text{let } \langle name \rangle := \langle expression \rangle_0 \text{ in } \langle expression \rangle_1]\!] &\Rightarrow \mathcal{V}[\![\text{global} \langle name \rangle]\!] \texttt{ = begin } \mathcal{E}[\![\langle expression \rangle_0]\!] \texttt{ end, } \\
&\quad \mathcal{E}[\![\langle expression \rangle_1]\!] \\
\mathcal{E}[\![\text{update } \langle name \rangle := \langle expression \rangle]\!] &\Rightarrow \mathcal{E}[\![\langle expression \rangle]\!] \\
\mathcal{E}[\![\langle expression \rangle_n]\!] &\Rightarrow \texttt{element}\,(n{+}2,\ \mathcal{E}[\![\langle expression \rangle]\!]) \\
\mathcal{E}[\![\text{new } \langle name \rangle(\langle expression \rangle^*)]\!] &\Rightarrow \mathcal{C}[\![\langle name \rangle \, ; \langle expression \rangle^*]\!] \\
\mathcal{E}[\![\text{case } \langle expression \rangle \text{ of } \langle alternative \rangle^* \text{ end }]\!] &\Rightarrow \texttt{case } \mathcal{E}[\![\langle expression \rangle]\!] \texttt{ of} \\
&\quad \mathcal{A}[\![\langle alternative \rangle_0]\!]; \\
&\quad \cdots; \\
&\quad \mathcal{A}[\![\langle alternative \rangle_{n-1}]\!] \\
&\quad \texttt{end} \\
\mathcal{E}[\![\text{chkcase } \langle expression \rangle \text{ of } \langle alternative \rangle^* \text{ end }]\!] &\Rightarrow \mathcal{E}[\![\text{case } \langle expression \rangle \text{ of } \langle alternative \rangle^* \text{ end }]\!] \\
\mathcal{E}[\![\langle constant \rangle]\!] &\Rightarrow \mathcal{L}[\![\langle constant \rangle]\!] \\
\mathcal{E}[\![\text{foreign } \langle fdesc \rangle_0 \langle fdesc \rangle_1 (\langle farg \rangle^*)]\!] &\Rightarrow \langle fdesc \rangle_1\,(\mathcal{E}[\![\langle expression \rangle_2]\!], \cdots, \mathcal{E}[\![\langle expression \rangle_{n+1}]\!]) \\
\mathcal{E}[\![\text{operator } \langle operator \rangle(\langle expression \rangle^*)]\!] &\Rightarrow \mathcal{O}[\![\langle operator \rangle \, ; \langle expression \rangle^*]\!] \\
\mathcal{E}[\![\text{nothing }]\!] &\Rightarrow \texttt{`undefined'} \\
\mathcal{E}[\![\text{error } \langle string \rangle]\!] &\Rightarrow \texttt{erlang:error}(\langle string \rangle) \\
\mathcal{A}[\![\text{match } \langle name \rangle_0(\langle name \rangle^*) \rightarrow \langle expression \rangle]\!] &\Rightarrow \mathcal{C}[\![\langle name \rangle_0 \, ; \text{ global } \langle name \rangle_1 \cdots \text{ global } \langle name \rangle_n]\!] \texttt{ -> } \\
&\quad \mathcal{E}[\![\langle expression \rangle]\!] \\
\mathcal{A}[\![\langle constant \rangle \rightarrow \langle expression \rangle]\!] &\Rightarrow \mathcal{L}[\![\langle constant \rangle]\!] \texttt{ -> } \mathcal{E}[\![\langle expression \rangle]\!] \\
\mathcal{A}[\![\text{default } \rightarrow \langle expression \rangle]\!] &\Rightarrow \texttt{\_ ->} \mathcal{E}[\![\langle expression \rangle]\!]
\end{aligned}
$$

Foreign Calls

In Idris, the type of `IO a` is in fact parameterised by the types that the foreign function interface uses. So for instance, `IO a` is the type of computations that use C-like types in their foreign functions.

Erlang doesn't have a type system like C's, so instead we defined our new computation type, `EIO a` which is parameterised by the types that Erlang uses, so we can make foreign calls to Erlang code.

2. Concurrency in Idris

Of course, our Idris to Erlang compiler is uninteresting unless we can use it to write concurrent programs.

The system I settled on implements the Actor model — Erlang's underlying concurrency model — as a type `Process l a`, where `l` is the type of messages that process can receive (something I like to refer to as the language of a process), and `m` is the type of what the computation creates.

The type of `m` is in fact a `Type*` so that we can use Idris' Uniqueness Types to provide guarantees about protocol progress.

We can then embed important details about how spawning, sending and receiving work into the types of the functions that do so, for instance:

```
-- The Process Type, from above
data Process : Type → Type* → Type

-- A reference to a process, as used for messaging.
-- This embeds the process' language in its type.
data ProcRef : Type → Type

-- Spawn gives us a reference to the spawned process,
-- indexed by the correct language.
spawn : Process l' a → Process l (ProcRef l')

-- Receive returns us a value with a type that matches
-- the process' language.
receive : Process l l

-- Send can only send messages of the correct type
send : ProcRef l' → l' → Process l ()
```

For Instance, this program will only run concurrently (it uses some basic constructs on top of `Process` to provide RPC functionality):

```
module Main

import ErlPrelude
import Erlang.Process
import Erlang.RPC

-- This is the handler in the RPC process.
-- `i` is the current count,
-- `Nothing` will read the current count,
-- `Just x` will add `x` to the count and return the new count,
-- The returned pair contains the reply and the new count
counter_rpc : Int → Maybe Int → Process (Maybe Int) (Int,Int)
counter_rpc i Nothing  = return (i,i)
counter_rpc i (Just x) = return (i+x,i+x)

test_proc : Process Int ()
test_proc = do counter ← spawn_rpc counter_rpc 0
               x ← rpc counter (Just 3)
               lift ∘ putStrLn $ "x (3) = " ++ (cast x)
               y ← rpc counter (Nothing)
               lift ∘ putStrLn $ "y (3) = " ++ (cast x)
               return ()

main : EIO ()
main = run test_proc
```

3. Erlang Concurrency Patterns

Erlang/OTP contains several other higher-order actor patterns which most programmers structure their applications around. For instance, there are behaviours for generalised servers (`gen_server`), generalised finite state machines (`gen_fsm`), and generalised event handlers (`gen_event`).

We can put types on these patterns, especially adding languages for these processes. In this way, we use dependent typing to allow for flexible but strong types about various parts of the languages.

For instance, with a generalised server, there are two ways of communicating with it. Firstly, there is a synchronous call, which will receive a reply. Then there is also an asynchronous cast, which will not receive a reply.

We can encode this inside a more advanced language type, like so:

```
-- A Gen Server language
-- cl is the call type
-- (cl → Type) computes the reply type from the call value
-- the third type is the cast type
data GSL : (cl:Type) → (cl → Type) → Type → Type

-- Type of computations that can interact with a Gen Server
GSP : Type → Type → Type

-- A Gen Server reference includes the language
data GSRef : (GSL _ _ _) → Type

-- Making a synchronous call to a Gen Server
call : {l:GSL cl cr ct} → GSRef l → (m:cl) → GSP (cr m)

-- Making an asynchronous cast to a Gen Server
cast : {l:GSL _ _ ct} → GSRef l → ct → GSP Unit
```

Conclusion

I have shown that Idris is an entirely adequate programming language for concurrent programming. With the new Erlang code generation system that I have built, we can now write and run safe, flexible actor-based programs which conform to statically proven guarantees.

I have built an Idris to Erlang compiler, which supports concurrent Idris code. Along the way I have documented exactly how to produce a compiler and how my compiler works.

I have also devised a way to model the Actor model in Idris' dependent types in a lightweight way such that we can run the resultant concurrent programs on Erlang, including doing inter-process RPC.

I looked into a heavily dependently-typed model of the three main Erlang/OTP behaviours, which I finished, but are not executable.

I wanted to look into modelling other concurrency calculi in Idris, but was unable to do so, and likewise I was unable to devise a Hoare-like logic for Erlang and its runtime system.

I hope that the work in my dissertation paves the way for future research into concurrent programming with Idris and other dependently-typed programming languages.