A Concurrency System for Idris & Erlang

Archibald Samuel Elliott

10 April 2015

University of
St Andrews

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.

Declaration

I declare that the material submitted for assesment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated.

The main text of this project is 13,267 words long, including project specification and plan.

In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

Contents

Chapter 1

Introduction

A programmer had a problem, so they decided to use threads.

Now two have they problems.

1.1 Concurrent Confusion

Concurrent Programming is hard.

Whereas sequential programming only requires the programmer to reason about time — i.e. how their program progresses — concurrent programming requires the programmer to reason not only about time, but also about space — i.e. how data moves between concurrent agents in their program — and about causality — i.e. how the computational- and data-dependencies of their program manifest themselves.

While it may seem that concurrency is a straightforward concept, the implications of concurrency models are not generally well understood by developers, which leads to bugs in programs. For example, in a shared memory concurrency model (such as *POSIX Threads*) programmers frequently misunderstand the atomicity of memory operations, which leads to difficult-to-debug race conditions.

The C code in Listing 1.1 contains one such race condition. When running this code, it's almost equally likely that it will print a value of less than 100 as it will print a value of 100, and this value is computed entirely nondeterministically.

The problem lies in the fact that += (line 15) isn't an atomic operation, and instead is split into an access and a separate update. These accesses and updates in each thread can be interleaved in any way the computer wants. These interleavings include doing multiple accesses before multiple updates, which has the effect of losing updates. Race conditions can be a lot harder to debug than this.

One way of avoiding this kind of bug is to use a totally different concurrency model. The Actor model [10] is one such model. Instead of communic-

```c
#include <stdio.h>
#include <assert.h>

#include <unistd.h>
#include <pthread.h>

#define N_THREADS 100

static int x = 0;

void *increment(void* argument) {
    sleep(1);

    int* value = (int*) argument;
    *value += 1;

    return NULL;
}

int main(void) {
    pthread_t threads[N_THREADS];
    void* res[N_THREADS];

    for (int i = 0; i < N_THREADS; i++)
        pthread_create(&threads[i], NULL, increment, (void*) &x);

    for (int j = 0; j < N_THREADS; j++)
        pthread_join(threads[j], &res[j]);

    printf("X = %d\n", x);
    assert(x == N_THREADS); /* Nondeterministically Fails */
}
```

Listing 1.1: A Race Condition in C

ating via sharing state, actors are processes that share state by communicating. This makes it impossible to access or modify the state of another process without coordinating with it first.

1.1.1 Erlang

One of the most respected implementations of the Actor model is Erlang, a programming language designed in the CS Lab at Ericsson. Erlang's designers originally settled on the actor model for the increased fault tolerance and isolation it provided, something they required for their software that runs telephone call switches.

These same properties have in fact proved amazingly useful and versatile for other applications. Erlang is now used in a wide variety of software such as Basho's Riak Database; ejabberd, an XMPP chat server; and the LINC OpenFlow switch, a software defined networking system.

Of course, it is still possible to write programs with race conditions or deadlocks in Erlang. For instance, a common Erlang deadlock is code like Listing 1.2. This code deadlocks in `test/0`, because the calls to `req/2` in `handle_request/1` call the current process, and it can't proceed to receive this new request from its mailbox until it has finished processing the current

```
-module(deadlock).
-compile(export_all).

req(Pid, Request) ->
    UniqueRef = make_ref(),
    Pid ! {req, {self(), UniqueRef}, Request},
    receive {reply, UniqueRef, Reply} ->
            Reply
    end.

server_loop() ->
    receive {req, {Pid, Ref}, Request} ->
            Response = handle_request(Request),
            Pid ! {reply, Ref, Response}
    end,
    server_loop().

handle_request({add, X, Y}) ->
    X + Y;

handle_request({add, X, Y, Z}) ->
    XY = req(self(), {add, X, Y}), %% Deadlock Here
    req(self(), {add, XY, Z}).

test() ->
    Pid = spawn(fun server_loop/0),
    req(Pid, {add, 3, 2, 4}).
```

Listing 1.2: A Deadlock in Erlang

request.

1.2 Banishing Bugs

Unfortunately, regardless of the programming model they use, programmers are only human and will inevitably still make mistakes, and no programming model exists that can prevent every single bug.

Programmers, however, are clever enough to know what they're not good at, and have devised systems that allow them to use computers, which are far more systematic than humans, to verify their programs. These systems fall into two broad categories: testing or static analysis systems.

Testing systems rely on running parts of the system and checking their behaviour against a finite number of test cases. These tests can only show the presence of bugs; they can never guarantee that a program is totally bug-free. This doesn't mean they're useless, instead like all things, they have their limitations.

There are lots of forms of static analysis. By far the most popular static analysis systems are Type Systems. There are two main forms of type systems: dynamically-checked type systems, and statically-checked type systems.

Statically-checked type systems are conventionally implemented in a compiler phase, preventing the production of executable code that doesn't agree with the specified type information. Haskell and Java have a statically-checked type systems which primarily focus on checking the arguments to and results

of sequential procedure calls. This turns out to be very successful at finding sequential bugs, but concurrent systems need a novel approach.

Dynamically checked type systems, on the other hand, check the types of values during run time, as functions are called or values are returned. This means that, like testing, the program must be comprehensively run to find out if it has any type errors.

Erlang has an optional static analysis tool called *dialyzer*. It can do type checking, but it is not integrated into compilation, so a programmer does not have to run the tool. The type checking it does is also weaker than most type systems, employing a mechanism called *success typing*, which can cope with the dynamic nature of existing Erlang library code, but cannot provide particularly strong guarantees of correctness.

Instead, I chose dependent typing to express the concurrent properties I want to be able to reason about in our programs. Dependent typing, unlike most type systems, makes no distinction between types and values, allowing values to be part of the types and vice versa. This makes them able to provide even stronger guarantees than conventional static typing. While dynamically-checked dependent types may be possible, statically-checked dependent types seem more viable.

```
1  data List a = Nil | Cons a (List a)

   -- Correct Version
   append :: List a → List a → List a
5  append Nil         ys = ys
   append (Cons x xs) ys = Cons x (append xs ys)

   -- Incorrect version, but type checks
   append' :: List a → List a → List a
10 append' xs ys = xs
```

Listing 1.3: A Buggy Haskell Program

To demonstrate the power of dependent types, I'm going to illustrate what they can do with an example.

In Listing 1.3, we see a list type and two version of a function called append. The list type allows me to know that not only do I have a list, but also that lists with different types of elements are different types, so I can reason about their contents. In the case of the two `append` functions, they both have the same type, but these functions do different things, and there's no way Haskell can make sure that all the elements of both lists go into their result. The type checker has helped me, but hasn't prevented this bug.

The same code might look like Listing 1.4 in a dependently typed language, in this example Idris. In this case, I'm using a type called `Vect`, which is exactly like the list type above, only it also knows how many elements are in the list (the `Nat` argument). This number is just a regular Idris value, and the `+` function used is also just a normal Idris function. Now I can use this extra information in `append` to verify that the size of the returned list is the sum of the sizes of the two original lists. The only way to satisfy this for all values of `n` and `m` is for me to actually append the first list onto the second.

```
data Vect : Nat → Type → Type where
  Nil  : Vect 0 a
  Cons : a → Vect n a → Vect (1 + n) a

-- Correct Version
append : Vect n a → Vect m a → Vect (n + m) a
append Nil         ys = ys
append (Cons x xs) ys = Cons x (append xs ys)

-- Incorrect Version
append' : Vect n a → Vect n a → Vect (n + m) a
append' xs ys = xs
```

Listing 1.4: Verified Append in IDRIS.

In the case of `append'`, I have given an incorrect definition, and the type system will tell me this at compile time. This is exactly what I want our system to be able to do, especially with more complex programs.

```
-module(append).
-export([append/2, append_dyn/2]).

-spec append([any()], any()) → any().
append([], Ys) →
    Ys;
append([X|Xs], Ys) →
    [X|append(Xs,Ys)].

-spec append_dyn([any()], [any()]) → [any()].
append_dyn([], Ys) when is_list(Ys) →
    Ys;
append_dyn([X|Xs], Ys) when is_list(Ys) →
    [X|append_dyn(Xs, Ys)].
```

Listing 1.5: Type-annotated Append in ERLANG

By comparison, dialyzer, the ERLANG static analysis toolkit thinks that the type specification on `append/2` in Listing 1.5 is correct but this is almost entirely useless. It tells me this function will take a list of anything and another anything, and give us back something. This is only marginally more information to verify a program against than the signature of the most general 2-argument function `f(any(), any()) -> any()`. The way to make this code safer is to introduce run-time checks, such as `when is_list(Ys)` in `append_dyn/2`.

1.2.1 Idris

IDRIS is a dependently-typed pure functional programming language. It is under active development for research, lead by Edwin Brady. IDRIS has a variety of interesting features including the ability to write new code generation backends without affecting the rest of the compiler, and the ability to interact easily with existing code written in other languages via its foreign function system.

1.3 Objectives

My project, as described, fits into three strands: Compilation, Library Support, and Modelling.

Compiling Idris programs into Erlang

1. Formalise how Idris will compile into Erlang (**Basic**, Document)
2. Create an Idris to Erlang compiler backend (**Basic**, Executable)
3. Document how to create new compiler backends for Idris (**Basic**, Document)
4. Create a small set of example Idris programs to demonstrate the new Erlang compiler backend (**Basic**, Example Programs)
5. Devise a foreign call interface for Idris (**Intermediate**, Idris Extension & Document)

Providing ways to verify the behaviour of Erlang programs

6. Create a small set of concurrent Idris example programs that can compile into Erlang (**Intermediate**, Example Programs)
7. Give a typed API to Erlang's runtime system (**Intermediate**, Library)
8. Give a dependently-typed API to Erlang and its runtime system (**Advanced**, Library)
9. Create a small set of dependently-typed concurrent Idris example programs that can compile into Erlang (**Advanced**, Example Programs)
10. Devise a Hoare-like logic for Erlang and its runtime system (**Advanced**, Document)

Modelling concurrency calculi in Idris

11. Model a concurrency calculus in Idris (**Basic**, Example Programs)
12. Create a verified and executable concurrency library for Idris based on Erlang and the Actor model (**Intermediate**, Library)
13. Create a verified and executable concurrency library for Idris based on another concurrency calculus (**Advanced**, Library)

1.4 Contributions

The main contributions of my dissertation are:

- A new compiler backend so that Idris programs can be compiled to run on the Erlang runtime system.
- A dependently-typed system that allow us to reason about concurrent Idris programs.
- A library that formalises the behaviour of several Erlang/OTP behaviours.

1.5 Outline

This dissertation is split into five parts. I start by describing various related work and alternative approaches in Chapter 2. In Chapter 3 I describe the design decisions behind my compiler and related libraries, including how to create your own code generator for Idris, followed by a brief description of their implementation details in Chapter 4.

Chapter 5 evaluates my approach based upon its own merit and compares it to other approaches. Chapter 6 concludes my dissertation.

Chapter 2

Related Work

Knock Knock
Race Condition
Who's there?

Anon

There has been lots of research done around the topic of concurrency and program verification, some of which I outline below.

2.1 Modelling Concurrency

Fundamentally, programming languages are just a way of presenting the behaviour of a given system to a programmer in a coherent and understandable way. This means that not only does a language have structure and syntax, but we also understand its behaviour against an abstract model that explains the bigger picture.

Concurrency is all about decoupling and encapsulating computations (usually called *Threads* or *Processes*) such that they may be performed independently of each other, and such that the dependencies between computations are clarified and may be minimised.

I've already mentioned shared-memory concurrency, where each thread of concurrent computation has access to some (or maybe all) of the same memory as other threads. Any thread is free to alter any memory it has access to, but without mutexes or locks, the possibility of changing state that another thread is accessing or changing is present, which could lead to bugs. Proper use of mutexes prevent this kind of bug by preventing more than one thread from being in a critical section where it alters that memory at the same time. This is the model used by POSIX Threads, and the runtime systems of Java, C#, amongst many others.

Unfortunately, shared-memory concurrency requires a very low-level understanding of what's happening to memory during execution. Another problem with this approach is that without mutexes or locks, the result of the computation can depend on how the scheduler chooses to run the threads, which the programmer has no control over.

Because of all these problems, computer scientists have looked for other models to use to reason about concurrency.

2.1.1 Message-Passing Concurrency

The overall guideline chosen by message-passing concurrency turns the conventional wisdom of shared-memory concurrency on its head. In shared-memory concurrency, programs communicate by sharing state, i.e. one thread alters some memory such that other threads may use the value it has left behind.

In message-passing concurrency, however, programs share state by communicating, i.e. a process[1] sends messages to other processes that contains the information they require to continue. This means that each process can be isolated from all the rest, so that no process may directly alter the internal state of another.

The advantage of this way of organising a system is that when programming the behaviour of a single process, the programmer does not have to think about any of the other processes, except to remember the protocol by which they communicate.

There are several existing message-passing models, all of which differently codify the details of how communication work.

- Milner's *Calculus of Communicating Systems* [19] proposes a system of agents that communicate by offering or receiving labels from or to each agent's ports. This makes no mention of the underlying runtime system implementation, but assumes synchronous communication. An interesting part of this work is the composition system, whereby agents are linked together by their compatible ports in order to organise them into communicating with each other.

- Hoare's *Communicating Sequential Processes* [11] chose a system of individually synchronous channels, whereby each process can have references to multiple channels for sending or receiving messages, but sending a message blocks the sender until the receiver retrieves the message. This model has been adopted in programming languages like occam or Go.[2]

- In Hewitt and Baker's *Actor model* [10, 2], each process has a single message queue, which any actor can send messages to. Sending a message doesn't block the sender, and the receiver can choose when (and in which order) to process messages from its queue. The Actor model has been adopted by languages such as Erlang[3] and Scala.[4]

  Actors can not only send messages, but they can also spawn new actors. This makes the Actor model strictly more powerful than the sequential

[1] Conventionally, *Thread* refer to a shared-memory concurrent computation, and *Process* refers to a message-passing concurrent computation.
[2] http://golang.org/
[3] http://www.erlang.org/
[4] http://www.scala-lang.org/

computation model (where there is no concurrency at all), and CCS and CSP (where processes are organised statically).

2.2 Erlang

I chose Erlang in particular for a few reasons. Firstly, and probably most importantly, it's a language I'm familiar with after working with it on-and-off for over five years.

Erlang, while still relatively unpopular,[5] is seeing adoption in various specific applications around networking and the internet. I would ascribe this to the fact that it was expressly designed for controlling network switches that had to be highly resilient to failure.

Another good reason for choosing Erlang is that its implementation of the Actor model is both simple and powerful. There are effectively only three concurrency operations: `send`, `receive`, and `spawn`.

To promote resiliency in Erlang programs where actors are being created and dying dynamically, Erlang introduces one further concept called a *link*, which allows one actor to subscribe to be notified when a linked actor exits. This allows systems to be resilient to dynamic failure, to the point where individual process crashes are encouraged so that errors don't spread through an application. While links are interesting, I will not be mentioning them again in my dissertation except to discuss some related work.

A more complete description and history of Erlang is in [1], which includes discussion of various of the language's design decisions.

2.3 Dependent Types & Idris

In the same way that programmers need models to reason about concurrency, type systems are also a model for reasoning about computation. In particular, programmers use statically checked type systems to enforce particular behaviour in our programs, according to programmer-defined specifications (type annotations).

I have previously illustrated why dependently-typed languages are interesting and useful in Section 1.2, but why in particular did I choose Idris[6] instead of Agda[7] or Coq[8]?

I chose Idris because I was also more familiar with it than the others, but also because Brady, Idris' creator, was willing to supervise me and help me with any questions I had.

Another reason for choosing Idris above Agda or Coq is that it has been designed for compiling to executable general-purpose code, rather than just for defining and proving theorems. With Coq or Agda, you can synthesise definitions from your code into other languages, but this isn't the same as

[5] Erlang did not feature in the top fifty places of TIOBE's Popular Programming Languages Index in March 2015. http://www.tiobe.com/index.php/content/paperinfo/tpci/

[6] http://www.idris-lang.org/

[7] http://wiki.portal.chalmers.se/agda/pmwiki.php

[8] https://coq.inria.fr/

direct compilation, as these definitions need more code around them to work in a complete executable system.

Idris, on the other hand, has a foreign function system built-in, and allows the programmer to choose from various code generation backends. The compiler infrastructure has been designed expressly so third-party backends, like the one that I have produced, integrate seamlessly with the core of the Idris compiler.

2.4 Reasoning about Concurrency

There is some recent work by Brady on using Idris for concurrent programming, with a system he called *ConcIO*.[9] How this system works is that you define a protocol as a type, and then this allows you to define programs for each party in the protocol such that they behave in a way that satisfies the protocol, up to completion. ConcIO is a good solution to this problem, however it uses a system with channels, which does not correspond to how Erlang and the Actor model work. I will, however, be coming back to Brady's idea of a token that ensures programs make progress through the defined protocol.

Using Idris is not the only way to use types to reason about concurrency. *Session Types* [12, 13, 14] were devised to directly solve the problem about reasoning about concurrency and distribution using type systems. They take a similar approach to ConcIO, where the protocol is defined, and then programs are defined for each party according to the protocol. The main difference between session types and ConcIO is that ConcIO still has all the power of Idris' dependent types for reasoning about the sequential computation in the implementation of each party.

There are also other tools for creating specifications of concurrent behaviour and verifying programs against them, such as *Spin*[10] and *TLA+*,[11] but these are external tools that don't have the same language integration that statically checked type systems do.

2.5 Reasoning about Erlang Programs

There has already been some work done to verify the behaviour of concurrent Erlang programs.

As I have mentioned, Erlang does have a type system [16], but it's limited. Programmers can only work with existing types defined in the language, but which they can use these to define new structures. Functions can operate on whichever data types they want, including what might appear to be disparate types. There are functions for checking types at run time, but no static type checking system is built into the language.

Sagonas has done some work on tooling around Erlang's type annotations, devising a system called *dialyzer* [15]. This is an external suite of tools that you

[9] Edwin Brady. *ConcIO*. 22nd Feb. 2015. URL: https://github.com/edwinb/ConcIO.
[10] http://spinroot.com/spin/whatispin.html
[11] https://tla.msr-inria.inria.fr/tlaps/content/Home.html

can run on an ERLANG program that will check the provided type annotations. Not only that, but dialyzer also has an inbuilt system for detecting common race conditions in ERLANG programs [6].

In terms of using dependent-types to verify ERLANG programs, the most complete project is called *verlang*,[12] which synthesises ERLANG code from Coq programs. Unfortunately, this requires writing the Coq theorems in a certain way to avoid run-time exceptions and to have them callable from ERLANG. This approach also generates inefficient code which requires wrapping to be useful in existing ERLANG applications [17].

There has been some other work on generating distributed ERLANG from Coq code, but this used Haskell as an intermediate synthesis language, which is also inefficient and over-complex [18]. This research approaches the problem of verified concurrent programs.

There has also been work by the Session Types community on Erlang, with a paper about using Session Types to reason about a subset of Erlang, in particular using its unique reference system to emulate channels [20]. Fowler's work on using Session Types to reason about dynamic supervision of ERLANG programs[13] is also interesting, but doesn't use dependent types, and it has been developed concurrently to this work.

[12] Tim Carstens. *verlang*. 3rd Aug. 2013. URL: https://github.com/tcarstens/verlang.
[13] Simon Fowler. *Monitoried Session Erlang*. 13th Mar. 2015. URL: https://github.com/SimonJF/monitored-session-erlang.

Chapter 3

Design

> Formal mathematics is nature's way of letting you know how sloppy your mathematics is
>
> Leslie Lamport

There are two main parts to this project: an Idris to Erlang compiler; and a system for verifying the behaviour of concurrent programs written for the Erlang runtime system.

3.1 Creating a New Compiler Backend

The first major part of my project is a new backend for the Idris compiler, such that I can compile Idris into Erlang.

While it may seem like a hard task to add a new backend to the Idris compiler, recent changes have meant that it's much easier than before, and that no official acceptance by the Idris team is required. Writing an Idris backend does, however, require proficiency with Haskell, as this is how the backend will interface with the Idris compiler.

As I was only working on a code generator, the interface between my code and the Idris compiler is small. At no point during writing my compiler was I required to learn about dependent types, type checking, proof obligations, or any of the other wonderful features that Idris has. All Idris code is fully type-checked and erasure is performed before the intermediate representation (IR) gets to the code generator.

A Code Generator is made up of two parts, conventionally. There is usually an executable, in my case named `idris-erlang`. The other part is a Haskell module, in my case named `IRTS.CodegenErlang`, which contains all the functions that actually generate code. Brady has released an example repository with which to get started.[1]

[1] Edwin Brady. *Idris Empty Code Generator*. 3rd Mar. 2015. URL: https://github.com/idris-lang/idris-emptycg.

The executable (in `src/Main.hs` in the `idris-erlang`[2] repository) doesn't do very much. The only thing that I changed from the example is that: any references to `emptycg` were replaced with references to `erlang`; the import of `IRTS.CodegenEmpty` was changed to `IRTS.CodegenErlang`, likewise the call to `codegenEmpty` was replaced with a call to `codegenErlang`; and the default output file name was changed to `main.erl` to match what Erlang expects.

Now, onto the code generation module. This part of the Idris compiler has been designed to allow for a large amount of flexibility. I created a function of type `CodeGenerator`, which can be found in `src/IRTS/CodegenCommon.hs` in the Idris repository.

3.1.1 Code Generation Information

Most backends will only require a couple of fields from `CodegenInfo`, in my case I only used `defunDecls`, `outputFile` and `exportDecls`.

```
24 data CodegenInfo = CodegenInfo { outputFile :: String,
25                                  -- elided fields
                                    includes :: [FilePath],
                                    importDirs :: [FilePath],
                                    compileObjs :: [String],
                                    compileLibs :: [String],
30                                  compilerFlags :: [String],
                                    simpleDecls :: [(Name, SDecl)],
                                    defunDecls :: [(Name, DDecl)],
                                    liftDecls :: [(Name, LDecl)],
                                    interfaces :: Bool,
35                                  exportDecls :: [ExportIFace]
                                  }
```

Listing 3.1: Abridged `CodegenInfo` from `src/IRTS/CodegenCommon.hs` in the Idris repository

- `outputFile` contains the path to the file that will be created by the Erlang backend, if everything is successful.
- `includes` is a list of file names to include, as specified with `%include` directives in source files. These are filtered so you only get the include files specified for your backend. This is in particular used by the C backend for headers, but also the Java backend for imports.
- `importDirs` is the list of paths on the idris import path, as specified by `--idrispath` at the command line. This is used by `%link` directives as a search path when for specified files.
- `compileObjs` is a list of file names to link with, as specified with `%link` directives. These are filtered in the same way as `includes`, i.e. by matching backend. This is in particular used by the C backend.

[2] Archibald Elliott. *Idris-Erlang*. Apr. 2015. URL: https://github.com/lenary/idris-erlang.

- `compileLibs` is a list of libraries to link against, as specified with `%lib` directives. They are filtered in the same way as `includes`, i.e. by matching backend. This is in particular used by the C backend, but also by the Java backend for specifying dependencies.

- `compilerFlags` is a list of compiler flags to add, as specified with `%flag` directives. They are filtered in the same way as `includes`, i.e. by matching backend. This is in particular used by the C backend.

- `simpleDecls`, `defunDecls`, and `liftDecls` I will get to in just a moment, they're three different intermediate representations that code could be generated from, though a code generator will only use one of them.

- `interfaces` and `exportDecls` are parts of the new export infrastructure, respectively they are an option that tells you whether to only generate an interface (rather than an executable file), and a list of functions and types to export to the host system that's having code generated for it (i.e. a way to call IDRIS generated functions from C programs).

Looking back over the three kinds of declarations (`simpleDecls`, `defunDecls`, and `liftDecls`), we have to choose one (and only one) to generate code from. They all represent the same information, but they assume different features in the language we're generating code for, so using one that is too expressive for a low-level language would be a bad idea. Full descriptions of these forms are provided in Appendix C.

The highest level version of these is `liftDecls` which assume a language with lambdas and laziness. I think you'd only want to use these if you were generating code for a higher-order functional language like Haskell or OCaml.

The next level is `defunDecls` which is like `liftDecls` only without explicit laziness or lambdas, aka *Defunctionalised* form. In this case, all functions are fully applied when it comes to evaluation. This is the level I chose for my ERLANG backend, given ERLANG has no concept of laziness, and functions in ERLANG have to be fully applied. One of the interesting things about this approach is that all functions are applied via two large case statements.

The simplest level is `simpleDecls`, which is like `defunDecls` except that functions are only applied to variables (rather than arbitrary expressions), aka *Applicative Normal* form. IDRIS has a bytecode format it can generate from these simplified declarations, which is bytecode for a stack-based VM. The C and JavaScript backends both use this bytecode for code generation.

As I mentioned, I chose the defunctionalised form as it most closely resembled ERLANG without having to add significant complexity to the compiler.

One of the slight peculiarities of IDRIS is that every single included function, across all used modules, is included for code generation into a single file. ERLANG does have a module system, but I essentially completely ignored it and put all generated functions into the same file. That said, the export system does include a way to compartmentalise functions, which I shall return to.

3.1.2 Declarations

As the IDRIS compiler is defined, there are essentially two main different kinds of declaration: Constructors and Functions.

```
37 data DDecl = DFun Name [Name] DExp
               | DConstructor Name Int Int
```

Listing 3.2: `DDecl` definition from `src/IRTS/Defunctionalise.hs` in the IDRIS repository

Constructors (`DConstructor`) are really simple. They have 3 parts: a unique name, a unique tag, and an arity. The name inside the constructor will always match the name in the pair you find the constructor declaration in. Secondly, any time the name is referred to, for instance in construction or projection, the unique tag will also be included in the reference, so you don't need to keep track of the mapping in the code generator. The tag is in particular used by the C backend to identify constructors, because switch statements in C can only operate on integers.

Functions, (`DFun`) of course are a lot more complex. They also have 3 parts: a name, a list of argument variables, and an expression. As with constructors, the name will always match the name in the pair you get in `defunDecls`. The arguments are a list of names of argument variables that will be used later in the body of the function. Finally, the expression is the body of the function.

3.1.3 Functions

Expressions are the contents of each IDRIS function. They are the main part of where code generation happens.

```
16 data DExp = DV LVar
             | DApp Bool Name [DExp]
             | DLet Name DExp DExp
             | DUpdate Name DExp
20           | DProj DExp Int
             | DC (Maybe LVar) Int Name [DExp]
             | DCase CaseType DExp [DAlt]
             | DChkCase DExp [DAlt]
             | DConst Const
25           | DForeign FDesc FDesc [(FDesc, DExp)]
             | DOp PrimFn [DExp]
             | DNothing
             | DError String
```

Listing 3.3: `DExp` from `src/IRTS/Defunctionalise.hs` in the IDRIS repository

Again, I'm going to go through in order and describe each of the kinds of expressions in Listing 3.3.

- `DV` is just a variable reference. An `LVar` can either be a *local* variable, or what the code generation calls a *global* variable which is in fact just a named variable.

Local variables are unnamed and instead are numbered back through recent scopes, with the most recently defined local variable at index 0 and the last function argument at the highest index. This is called *De Bruijn indexing*. A code generator will need to cope with variable scope, including the ability to push new scopes and pop old scopes. In the defunctionalised form, full De Bruijn indexing hasn't happened, so most variables will still use global names. Applicative Normal (aka simplified) form uses full De Bruijn indexing.

- `DApp` is ostensibly function application. The expression includes the function name to call, and a list of expressions which correspond to the arguments to that function. The boolean specifies whether or not the application is a tail-call, which may require special treatment in the language you're generating code for.

  I say *ostensibly* function application, because the name could refer to a constructor, in which case the arguments are each a field. If the backend is not generating a function for each constructor, it will need to have a way to check if the name corresponds to a known constructor, at which point it should instead construct a value. In this case it won't get the constructor unique tag. There is the expectation that each constructor will turn into a function of the same name and arity, so if the code generator does do so, it won't need to keep track of constructors.

- `DLet` corresponds to a let statement, i.e. `let x = f y in g x`. The first expression is the expression to bind to the named variable. The second expression will be in a new scope including the new variable, and may refer to this new variable.

- `DUpdate` doesn't require any special treatment, except to generate the expression it contains. There is no requirement to update the named variable with the result of this expression.

- `DProj` is a *projection*, i.e. accessing a field in a constructed value. The given expression will produce a constructor, and the integer is a zero-indexed access into the constructor's fields.

- `DC` is a constructor invocation. The expressing includes the constructor's name, the constructor's unique tag (mentioned above), and a list of expressions that correspond to the constructor's fields. This may also contain the name of a variable where the code generator can reassign the constructor into, if it is generating code with mutable variables, but this is not required.

- `DCase` is a case statement. The aim is to evaluate the result of the expression against alternative branches (`DAlts`).

  A case branch (`DAlt`) has one of three forms:

  - `DConCase` matches against the given constructor (identified with both name and unique tag), providing variable names for each

field in the constructor which will be free variables in the provided expression.

  - `DConstCase` matches against the given constant, evaluating the given expression. No extra variables are bound during matching.
  - `DDefaultCase` is a final case which will always come last, and should match anything. It won't bind any variables, and just evaluates the given expression.

- `DChkCase` is like `DCase`, only the compiler doesn't know what type it will return. This is in particular used in the case statements that all function application goes through. The variants of the case branches are `DAlt`, the same as for `DCase`.
- `DConst` is a literal constant. These I shall explain further later.
- `DForeign` is a foreign function call. These are used for a lot of functionality in the Prelude (especially around files and processes), so they're not something code generation can avoid. The system around foreign functions has recently changed, and I will explain it fully later. The arguments, in order, are: the return type Idris requires, the function identifier, and a list of pairs of argument type and argument expression.
- `DOp` is a call to one of the primitive functions, as defined in `PrimFn` (in `src/IRTS/Lang.hs`). They cover a wide variety of functionality from basic mathematics through casting to runtime behaviour like forking processes. A full listing of these functions and their expected types is in Section C.2.
- `DNothing` represents something that has been erased from that position, and won't be used or inspected. It stays as a placeholder so that the Idris compiler doesn't need to change the arity of functions during erasure. The arity of constructors will be changed during erasure to remove erased fields (such as implicit type arguments or proof objects).
- `DError` represents the program throwing a run time error, with the given string as a message. This should not cause the code generator to throw that error, but it should be thrown at run time.

3.1.4 Constants

There are various constant literals that a code generator will need to be able to generate, and because Idris operates at both the type and the value level, some are for values and some are for types, for instance the C code generator requires knowledge of types for declaring variables.

Constants are fairly self-explanatory beyond this. `I` and `BI` differ only because the latter represents an arbitrary sized integer, whereas the former is for fixed-width integers. There are various types of integers. Firstly, `AType AFloat` represents a double-precision floating-point number. Then `AType (ATInt x)` represents an Integer, with `x` being either `ITBig` for arbitrary-precision integers, `ITNative` for fixed-precision integers (the size of these

integers is up to the backend), `ITChar` for the representation of characters, and `ITn` represent unsigned `n`-bit integers.

There is no assumed encoding of Characters or Strings, instead the backend gets to make its own choice.

The constructors starting `BnV` (where `n` is a number) are for vectors of various sizes. These are only used by the C backend, and I think it is reasonable to throw a compile-time error if a user tries to use them.

3.1.5 Primitive Operators

Idris has around 60 primitive operators (many more if you count each numeric variant as a different operator). They can be categorised in various ways, with operators for arithmetic, operators for type casting, operators for strings, operators for vectors, and operators for run-time functionality. A complete listing of the Primitive Operators is in Section C.2.

The arithmetic operators are parameterised by the numerical type they operate on, for instance Integers or Doubles (see the description in Constants about numerical types). In the case of the comparison operators, the result of this operator will usually have case analysis performed on it, and the cases expect a `1` for true and a `0` for false (as in C).

The casting operators are much the same, though — in the case of the integral casts — they are also parameterised by the two types you are casting between.

The string operators assume the underlying string implementation works like a list of characters, but you can substitute in your own representation and corresponding implementation of the operations

The operators for buffers and vectors can be ignored, as I have ignored buffer and vector literals, again a compile-time error is reasonable.

The operators for runtime functionality are a bit of a mixed bag, and really require individual explanation, which I have done in Section C.2. These are more interesting because they're now arbitrarily extensible using external primitives.

3.1.6 Foreign Calls & Exports

I will fully address foreign calls in Section 3.2, but right now is a good point to talk about exactly what a foreign function system is there to do.

I need a way of calling native Erlang functions from my generated functions. Because they have been generated from Idris, the generated functions won't work like native functions, so I need to do some translation of the data before I do the call. For each argument to the call, it must be translated from my generated interpretation of that data type into the native data type. And for each return type, it must be converted from the native type back into the generated representation of that type.

Exported functions work the opposite way around, their arguments need to be dynamically type checked, then converted into the representation used by the Idris backend. Return types are then converted from the backend representation back into their native data types.

How do we know how to translate the types? The code generator author gets to choose how each type is represented, as long as the functions match their implementation.

3.2 The New IO & Foreign System

Given the Idris compiler supports multiple backends, it is not a big leap to think that different generated code may be running in different computational environments. As `IO` is the way of encapsulating these changes to the outside environment, via Foreign calls, it too must be able to cope with these different environments.

For instance, the interactive environment requirements of an Idris program being compiled to an executable (via C) are different to those required when compiling Idris into JavaScript for the browser. The former has an environment containing files, processes, and other POSIX constructs; whereas the latter has an environment containing web pages, HTML and the DOM.

I was proposing to augment the primitives functionality (which is aimed at pure functions, despite the presence of IO functions) with a negotiated system of possible side-effecting interactions based on [9]. This might have allowed the backend to find out all the different interactions (and their types) that each program used, and check to make sure they were all possible within that backend. So, for example, if a program used an `appendElement` interaction, then only backends which allowed this interaction could compile this program, for instance browser-based backends. In the same way, if your program relies upon file input and output, that could have been a different set of interactions.

Several things were going to make this hard, not least allowing any new backend to propose new interactions with new types that would augment, rather than replace, the existing interactions.

In the end, this work got superseded by Brady extending how foreign functions worked, which is a much more sensible way of addressing the problem, as this would have had to change anyway too.

Brady's redesigned `IO` and foreign system was borne out of wanting to make it easier for backend designers to expose differing environments to programs written in Idris, but with a minimum of changes.

3.2.1 How New IO Solves This

Until recently, Idris's Foreign call system assumed that all foreign calls are into C, and therefore the types of these calls would be C-like in nature, and would conform to C's type system, so would include things not only like Strings, Floats or Pointers, but also Voids and Managed Pointers.[3]

In Erlang, for instance, this isn't great, as Erlang doesn't have the concept of Void, and does not require any Managed Pointers. On the other hand, there are types like Lists, Tuples, Atoms[4] and Functions that we would like to be able to use from Idris programs.

[3]Managed Pointers are a way of registering some C memory such that its lifetime will be controlled by the Idris GC system.

[4]Erlang Atoms are akin to Lisp Symbols, they're string-like, but only used for identity and equality, not for string manipulation.

The other large difference is how functions in C and Erlang (and other environments) are identified. In C, there is a single global namespace for functions, whereas in Erlang, functions are spread between different modules. In an object-oriented language, like JavaScript, some functions may have to be called on a particular object instead of just called.

These two related pieces of information: a set of types that the outside environment understands, and what amounts to a calling convention for environment functions describe the FFI system.

`IO'` has been introduced, which is parameterised by a structure containing the FFI description. Then `IO` has just been defined as `IO' C_FFI`, or the environment for interacting with C programs.

This means I can provide a library with my Erlang backend which defines how to interact with the Erlang environment. I called this environment `EIO`.

Functions that make foreign calls are implemented using the `foreign` function. `foreign` takes the FFI description, a function identifier (the type of which is deduced from information in the FFI description), and a type, and attempts to devise a function of the specified type. How it devises this function is that it does a proof search to construct the structure that `DForeign` is expecting, using the types of each argument, and the types that the FFI description contains.

The proof search in particular has to know how to translate native Idris types into native backend types, so we in fact describe the Erlang types in terms of a translation from Idris types, as shown in Listing 3.4.

The important part of this code is `Erl_Types`, which relies on the other definitions. `Erl_Types` shows the proof search what types it is possible to represent in Erlang, by presenting constructors for the representable types. This means that the proof search will be able to work out that functions involving strings can be translated, but those involving Managed Pointers have to use `ErlRaw`.

In the last 4 constructors, we introduce some extra conditions on the translatable types. So, for instance, we can only translate a type of `List a` (a list only containing terms of type `a`) if we can translate the type `a`. The same applies to pairs — we can only translate them if we can translate both elements of the pair. `Erl_FunT` essentially says that we can only translate functions (as arguments) if they operate on and return a translatable type. `Erl_NumT` just says the only kinds of numbers we can have are Characters, Integers or Doubles. Lastly, we use the `Erl_Any` to allow us to pass arbitrary Idris terms into Erlang which it won't understand, but also won't modify.

Because we very carefully designed how Idris compiles into Erlang, including special casing constructors for lists and pairs, no translation is required when passing these kinds of arguments from Idris to Erlang.

If we needed to have any guards, the `FDesc` structure would inform the backend of which parts of `Erl_Types` had been used to construct the type using the `FCon` and `FApp` constructors, which would allow us to correctly translate both the arguments and the return types.

```
  MkERaw : (x:t) → ErlRaw t

abstract
data Atom : Type where
  MkAtom : (x : String) → Atom

data Erl_NumTypes: Type → Type where
  Erl_IntChar    : Erl_NumTypes Char
  Erl_IntNative  : Erl_NumTypes Int
  Erl_Double     : Erl_NumTypes Double

mutual
  data Erl_FunTypes : Type → Type where
    Erl_Fun     : Erl_Types s
                → Erl_FunTypes t
                → Erl_FunTypes (s → t)
    Erl_FunIO   : Erl_Types t → Erl_FunTypes (IO' l t)
    Erl_FunBase : Erl_Types t → Erl_FunTypes t

  data Erl_Types : Type → Type where
    Erl_Str  : Erl_Types String
    Erl_Atom : Erl_Types Atom
    Erl_Ptr  : Erl_Types Ptr
    Erl_Unit : Erl_Types ()
    Erl_Any  : Erl_Types (ErlRaw a)
    Erl_List : Erl_Types a → Erl_Types (List a)
    Erl_Tupl : Erl_Types a → Erl_Types b → Erl_Types (a,b)
    Erl_FunT : Erl_FunTypes a → Erl_Types (ErlFn a)
    Erl_NumT : Erl_NumTypes t → Erl_Types t
```

Listing 3.4: `Erl_Types` from `libs/erlang/ErlPrelude.idr`

3.2.2 Foreign's Dual: Exports

Of course, the other side of foreign functions (which call native code from generated code) is to allow exported functions (which allow native code to call generated code).

This gets more complex, because not only do we have to be able to export functions, we also need to export data types, because the functions won't be very useful if they can only operate on foreign types.

The most major problem is that Data Types can be parameterised over other types, for example `List a`, but languages like C won't cope with this. The current exports system in Idris disallows parameterised types until we can find a better design to allow more powerful type information to be generated. As we can see from Listing 3.5, this Idris program exports both data types (`Data` entries in `testList`) and functions (`Fun` entries in `testList`).

For exported data types, they are just given an identifier. In this case, it's a string, but because different backends work differently, the type of this identifier is also left up to the backend designer, and is included in the data inside the FFI description. It is required that all data types which are required by the functions you export are exported too.

In the case of exported functions, they're given an identifier (of the same type as the one you used for foreign calls), and the export system will its type (in a proof search) to provide the backend with information as to its type.

```
import Data.List

addLists : List Int → List Int → List Int
addLists xs ys = xs ++ ys

nil : List Int
nil = []

cons : Int → List Int → List Int
cons x xs = x :: xs

show' : List Int → IO String
show' xs = do putStrLn "Ready to show..."
              return (show xs)

testList : FFI_Export FFI_C "testHdr.h" []
testList = Data (List Int) "ListInt" $
           Data (List Nat) "ListNat" $
           Fun addLists "addLists" $
           Fun nil "nil" $
           Fun cons "cons" $
           Data Nat "Nat" $
           Fun Strings.length "lengthS" $
           Fun show' "showList" $
           End
```

Listing 3.5: An Example of IDRIS Exports (from `tests/ffi006/ffi006.idr`)

With multiple `FFI_Export` definitions in a single IDRIS program, it is possible to generate functions in multiple files or modules.

3.3 IDRIS LIBRARIES

The second part of my project is about using IDRIS to reason about concurrent behaviour in programs, especially assuming an Actor-based system.

3.3.1 The Problem of General Communication

In ERLANG, as I have mentioned before, there are two completely general communication operators, `send` and `receive`.

Send is an asynchronous operation, returning immediately, rather than blocking until the other process accepts the message. Messages are stored in the process' message queue, and are retrieved using a receive statement. A receive statement pattern-matches in the same way that a case statement would, but will scan the whole mailbox, not just the first message. This means receive statements in ERLANG do not have to be exhaustive, so can opt not to deal with certain kinds of messages, either immediately, or at all.

But if we have an IDRIS program, given our aim for function totality, surely this is exactly what we don't want. What we want is that every message is handled, in the same way that we want every possible function input to produce an output.

It turns out that starting with example ERLANG programs, with their arbitrary communication schemes is not sensible, so instead I think it's best to start with much more predictable communication schemes.

3.3.2 Erlang's Patterns

During the development of ERLANG, the designers released it internally to other groups at Ericsson, and then later surveyed the various ways in which these groups were organising their systems. This work lead to the development of the Open Telecom Platform (aka OTP), which is a set of libraries and systems on top of the ERLANG language that codify most ways in which ERLANG systems are designed.

Firstly, OTP brought *behaviours*, which are the module equivalent of interfaces. Programmers implement their functionality in modules that conform to these behaviours so they can use the functionality that these patterns and libraries bring.

The three main behaviours that were devised are `gen_server`, a single process that operates as a server to other client processes; `gen_fsm`, a more complex version of `gen_server` which operates with an internal finite-state machine; `gen_event`, a system for broadcasting or subscribing to event streams; and `supervisor`, a system for creating supervision trees to dynamically monitor other processes using the links I mentioned earlier.

These systems are much easier to codify because they have much more regular communication protocols. Of course, I will be creating a dependently-typed interface for them, which restricts them to communicating only in a well-typed and sound way.

`gen_server` is the most widely used of these behaviours, so I'll start with it. There are two ways of communicating (in a safe manner) with a server: a synchronous *call*, to which the server will reply; or an asynchronous *cast*, to which the server won't reply, but it also won't block the client. Given we want to put a type on these communications, these would seem like a good starting point.

Firstly, I'm going to call the types any process (including these `gen_*` behaviours) uses to communicate its *language*. We include this language within the type of a reference to a process, so that we can use this information when checking messages we send to that process.

In the case of `gen_server`, the language consists of three parts, the type of calls, the type of replies, and they type of casts. However, in the case of the reply type, it would be very restrictive to make all calls have the same reply type. In this case, we can use the power of dependent types to afford us some flexibility that we can still reason about. Instead of only having a single reply type, we can specify a function that uses the call value to compute the reply type.

For instance, a server managing users for an application may have two different calls: one that requests the number of users, to which the reply type is a number; and one that adds a user, to which the reply type is whether or not the operation succeeded or not. Without the flexibility of using a function to compute the response type, this style of communication would not be possible.

I felt that keeping details about the type of initialization data, and the internal state data out of the process language was fundamental for the language to be useful enough to be re-used. For instance, the user management service

above might get rewritten to be more efficient, changing the internal state type, but given it still uses the same communication language, processes that interact with this service won't need to be significantly rewritten. Codifying your internal protocols like this is fundamental to modularity.

Despite introducing this dependency between the types, it is still possible to know which type will be in the reply to a call, as the caller knows what the call is and the language the server communicates with, and the server will also know what type to reply with, as it knows the value of the call it just got.

In the case of `gen_fsm`, we essentially have the same communication scheme, only with an extra type that specifies the particular states this FSM can be in. In this case we can also use the current state, along with the call value, to compute the reply type.

In the case of `gen_event`, calls are a lot more complex because there are multiple event handlers, so I ignored them and only allowed the implementation to handle events, which are like casts, in that they're asynchronous.

3.3.3 Returning to General Communication

I now have an approach to well defined protocols that looks like it will work, I can retrace my steps and apply this system to general processes.

We take the idea of a language, above, and for a process, we reduce it down to one thing: the type of messages which that process expects to receive. This language is again included in the type of any reference to a process, which lets us enforce that we only send messages to a process of a type that it expects.

It is also common that we may want to do remote procedure calling between two processes, much as calls work for `gen_server`. In this case, we have a lot less information about the interaction to check against. The caller knows the type it is expecting to receive, and the type the remote process is expecting to receive, but the remote process only knows what it should receive, and not what type the caller is expecting.

However, as Brady has done in ConcIO, we can use Idris' `UniqueType`, a kind of types that programs are only allowed a single reference to a value of, to enforce progress through the RPC flow. In the caller we generate a tag that the calling process must pass to receive a reply, and in the remote process we generate a tag that we pass along with the reply message in order to send a reply. This `UniqueType` system also prevents the remote process from sending two replies to an RPC message, or the caller trying to receive two replies to the same RPC.

3.4 Concurrency Calculi

The `Process` system that I created above to reason about (well typed) general actor behaviour is a well-typed subset of the possible behaviour that the Actor system, in general, supports.

While it is not expressly a Concurrency Calculi, my typed systems for the OTP behaviours are based on Hancock and Setzer's work on interactive dependently-typed programs [9]. This work allows us to define *worlds*, which define how we interact with an environment, in a pure, co-inductive way.

Dependently-typed programs can then be defined in terms of these worlds of interactions.

Because these programs are defined in terms of a pure and co-inductive abstraction over the interactions, much more of the program is total, which allows us to reason about and check more behaviour for correctness. While the function that runs these programs may be partial, it will usually be a lot smaller than the main program itself.

This work requires co-induction because we want these programs to be able to loop or recurse infinitely, something that regular induction prevents. Thankfully, the programmer does not necessarily have to understand co-induction to be able to use this system for interactive programs, as Idris can deal with all the required laziness.

Chapter 4

Implementation

> How do you solve the dining philosopher's problem?
>
> Just add more `fork()`
>
> Anon

4.1 Erlang Code Generation

So how exactly do we generate code for Erlang? `src/IRTS/CodegenErlang.hs` contains the entirety of the code to do so.

In particular, we want to look at the `generateErl` function, which is where all the specific functionality we need lies. This operates within the `ErlCG` monad, which sounds scary, but is only used to let us store various bits of information about the progress of code generation and to let us throw exceptions to abandon code generation if it finds a structure it doesn't understand.

The following formalisation should be read with respect to the Idris IR that I describe in Appendix C, in particular with the defunctionalised version of ⟨*expression*⟩, as given in Figure C.3.

I won't be giving the definitions of $\mathcal{N}[\![\langle name \rangle]\!]$, which changes Idris names into Erlang function names (`erlAtom`); $\mathcal{V}[\![\langle variable \rangle]\!]$, which changes Idris variables into valid Erlang variable names (`erlVar`); $\mathcal{L}[\![\langle constant \rangle]\!]$, which changes Idris constants into valid Erlang terms (`generateConst`); or $\mathcal{O}[\![\langle operation \rangle \ ; \ \langle expression \rangle *]\!]$, which produces valid calls for all the primitive operations (`generatePrim`). Not only is their source code available in my repository, but they're all long-winded, boring functions.

The `generateErl` is implemented like the $\mathcal{D}[\![\langle declaration \rangle]\!]$ function defined in Figure 4.1. To interpret this mathematical notation, it just says I generate function forms[1] for each function declaration, and don't generate anything for constructor declarations. I don't completely ignore constructor declarations because they tell me what constructors I should expect (and with which

[1] Erlang calls the elements of its own IR *forms*.

names), but I don't need to add any specific ERLANG forms to the file for each one.

$$\mathcal{D}[\![\ \text{function}\ \langle name\rangle\langle name\rangle^*\langle expression\rangle]\!] \Rightarrow \mathcal{N}[\![\langle name\rangle_0]\!]\texttt{(}\mathcal{V}[\![\texttt{global}\ \langle name\rangle_1]\!]\texttt{,}\cdots\texttt{,}\mathcal{V}[\![\texttt{global}\ \langle name\rangle_n]\!]\texttt{)->}\ \mathcal{E}[\![\langle expression\rangle]\!]\texttt{.}$$

$$\mathcal{D}[\![\ \text{constructor}_n\ \langle name\rangle]\!] \Rightarrow \epsilon$$

Figure 4.1: $\mathcal{D}[\![\langle declaration\rangle]\!]$ — Defunctionalised Declaration Translation

The $\mathcal{E}[\![\langle expression\rangle]\!]$ defined in Figure 4.2 corresponds to `generateExp`. For each kind of IDRIS expression, we generate the corresponding ERLANG expression. As you can see, the difference between IDRIS' IR and ERLANG code is not that large, and we are not doing anything complex. There are various complexities to these transformation rules, which I shall now explain.

As I mentioned in Subsection 3.1.3, function application — the second clause — may in fact contain a constructor, so we use the information about which constructors exist that we gathered above, and if the $\langle name\rangle$ refers to a constructor, we just construct the value rather than calling a function. If $\langle name\rangle$ isn't that of a constructor, we just compile it into a function application, which uses parentheses around the arguments in ERLANG's syntax.

In the case of update expressions, I in fact just generate the internal expression, and ignore other information. This is especially useful as ERLANG doesn't have mutable variables.

Projections use ERLANG's inbuilt `element/2` function, which pulls the requested field out of a tuple. The field number depends on how we generate the constructors, but as the function is one-indexed, and we are using tagged-tuples where the constructor name is in the first element, we have to add 2 to the zero-indexed index argument. We know the value of the index argument at compile-time, which is useful, and means we can do the addition at compile-time too.

Case expressions in ERLANG operate almost exactly like they do in IDRIS. Because ERLANG is almost entirely homoiconic — i.e. data types are destructed with the same syntax they're constructed with — and variables don't require special declarations when they appear on the left-hand side of a binding (assignment), generating the alternatives to match against is simple, and can reuse the exact same special-cased constructor translation that we use during construction. The full translation of $\mathcal{A}[\![\langle alternative\rangle]\!]$ is in Figure 4.3. Like in IDRIS, the _ variable is a "don't care" variable that matches with anything but doesn't bind a value. Both kinds of case expression in IDRIS use the same syntax because ERLANG doesn't mind that chkcase doesn't know the resultant type, after all it's dynamically typed.

I play fast and loose with the foreign calls, but this approach is simple and seems to work. I ignore all the foreign annotations, as I assume the stated type for the function is correct. I just call the named function name and pass in all the $\langle expressions\rangle$ from the $\langle farg\rangle$ pairs.

It doesn't matter what the result of $\mathcal{E}[\![\ \text{nothing}\ ]\!]$ is, as long as it's an expression. I chose the atom `undefined`, because it's used like NULL would be (ERLANG lacks a null type or value). The second reason for choosing this is

that if there's a bug in the compiler, this value should help the programmer find it, though I admit that an atom like `'the_idris_compiler_has_a_bug'` would perhaps be more obvious. The resultant value doesn't actually matter in the general case, as this expression replaces erased values, and erased values shouldn't be inspected at runtime.

Lastly, we use the function `erlang:error/1` to raise a runtime error for $\mathcal{E}[\![$ error $\langle string \rangle]\!]$. This should not raise an error at compile-time.

$$
\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]\!]\texttt{(}\mathcal{E}[\![\langle expression\rangle_0]\!]\texttt{, } \cdots\texttt{, } \mathcal{E}[\![\langle expression\rangle_{n-1}]\!]\texttt{)} \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, } \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\texttt{+2, } \mathcal{E}[\![\langle expression\rangle]\!]\texttt{)} \\
\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]\!]\texttt{;} \\
&\quad \cdots\texttt{;} \\
&\quad \mathcal{A}[\![\langle alternative\rangle_{n-1}]\!] \\
&\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 \text{ 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\texttt{(}\mathcal{E}[\![\langle expression\rangle_2]\!]\texttt{, } \cdots\texttt{, } \mathcal{E}[\![\langle expression\rangle_{n+1}]\!]\texttt{)} \\
\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\texttt{)}
\end{aligned}
$$

Figure 4.2: $\mathcal{E}[\![\langle expression\rangle]\!]$ — Defunctionalised Expression Translation

$$
\begin{aligned}
\mathcal{A}[\![\text{ match } \langle name\rangle_0(\langle name\rangle*) \rightarrow \langle expression\rangle]\!] &\Rightarrow \mathcal{C}[\![\langle name\rangle_0\ ;\ \text{global } \langle name\rangle_1 \cdots \text{ global } \langle name\rangle_n]\!]\texttt{ -> } \mathcal{E}[\![\langle expression\rangle]\!] \\
\mathcal{A}[\![\langle constant\rangle \rightarrow \langle expression\rangle]\!] &\Rightarrow \mathcal{L}[\![\langle constant\rangle]\!]\texttt{ -> } \mathcal{E}[\![\langle expression\rangle]\!] \\
\mathcal{A}[\![\text{ default } \rightarrow \langle expression\rangle]\!] &\Rightarrow \texttt{\_ -> } \mathcal{E}[\![\langle expression\rangle]\!]
\end{aligned}
$$

Figure 4.3: $\mathcal{A}[\![\langle alternative\rangle]\!]$ — Case Alternative Translation

Constructor Translation — $\mathcal{C}[\![\langle name\rangle\ ;\ \langle$<expression>$\rangle*]\!]$, in Figure 4.4 — is a little bit more complex, but not hugely. In order to make foreign function calls really simple, I compile the Idris list constructors into the equivalent Erlang list constructors (the first two transformations). I also compile the Unit value into a 0-tuple, which seems equivalent enough, though the 0-tuple isn't used in Erlang. Idris pairs are translated into 2-tuples, which seems fitting, and means we can use pairs in foreign-function calls.

I compile Idris' True and False into their erlang equivalents, but this is not the same as run-time booleans, which are the integers 0 and 1. The sole reason for compiling the booleans to their equivalents is that the code is far easier

to inspect for something going wrong. This serves no purpose for foreign function calls either.

Lastly, there are two final options for how I compile a constructor. If it has no arguments, it can safely be translated to an Erlang Atom (which will match and compare for equality). If, on the other hand, it has arguments, it gets compiled to a tagged-tuple.

$$
\begin{aligned}
\mathcal{C}\,[\![\texttt{Prelude.List.Nil}\,\S\,]\!] &\Rightarrow \texttt{[]} \\
\mathcal{C}\,[\![\texttt{Prelude.List.(::)}\,\S\,\langle expression\rangle_0\langle expression\rangle_1]\!] &\Rightarrow \texttt{[}\,\mathcal{E}\,[\![\langle expression\rangle_0]\!]\ \texttt{|}\ \mathcal{E}\,[\![\langle expression\rangle_1]\!]\texttt{]} \\
\mathcal{C}\,[\![\texttt{MkUnit}\,\S\,]\!] &\Rightarrow \texttt{\{\}} \\
\mathcal{C}\,[\![\texttt{Builtins.MkPair}\,\S\,\langle expression\rangle_0\langle expression\rangle_1]\!] &\Rightarrow \texttt{\{}\,\mathcal{E}\,[\![\langle expression\rangle_0]\!]\texttt{,}\ \mathcal{E}\,[\![\langle expression\rangle_1]\!]\texttt{\}} \\
\mathcal{C}\,[\![\texttt{Prelude.Bool.True}\,\S\,]\!] &\Rightarrow \texttt{'true'} \\
\mathcal{C}\,[\![\texttt{Prelude.Bool.False}\,\S\,]\!] &\Rightarrow \texttt{'false'} \\
\mathcal{C}\,[\![\langle name\rangle\,\S\,]\!] &\Rightarrow \mathcal{N}\,[\![\langle name\rangle]\!] \\
\mathcal{C}\,[\![\langle name\rangle\,\S\,\langle expression\rangle\texttt{*}]\!] &\Rightarrow \texttt{\{}\mathcal{N}\,[\![\langle name\rangle]\!]\texttt{,}\mathcal{E}\,[\![\langle expression\rangle_0]\!]\texttt{,}\cdots\texttt{,}\mathcal{E}\,[\![\langle expression\rangle_{n-1}]\!]\texttt{\}}
\end{aligned}
$$

Figure 4.4: $\mathcal{C}\,[\![\langle name\rangle\,\S\,\langle$<expression>$\rangle\texttt{*}]\!]$ — Constructor Translation

Using these defined translations, and the information about the primitives in Section C.2 it should be possible to recreate my compiler without seeing the source.

As Erlang only has arbitrary precision floating-point numbers, this solved a lot of issues with generating numbers and primitives, as all the arithmetic primitives all worked and there was only one kind of constant to generate. The only exception to this is that Erlang has a different operator for integral division (`div`) vs floating-point division (`/`).

In the case of Erlang characters, they don't actually exist, and are just represented as integer Latin1 codepoints. This means converting between characters and integers is trivial.

Erlang also doesn't have a string type, choosing instead to represent strings as lists of characters. This means that most of the string primitives are actually using Erlang's list handling functions.

4.2 Running Concurrent Idris Programs

My `Process` definition (from `Erlang/Process.idr`) is just defined in terms of `EIO`, so that any processes can have whichever side effects they want, like spawning another process, or sending and receiving messages, which are all foreign function calls.

Processes can handle `Type*`s for reasons we'll see in Section 4.3.

The Erlang source for these send and receive foreign functions are defined within `irts/idris_erlang_conc.erl`, the support library for concurrent Idris in Erlang.

We use a wrapper around messages for a few reasons, the primary one of which is type safety. If we were to just send any Idris term without wrapping it, then when it came time for a process to receive its messages, it would retrieve any message, even those from the Erlang runtime, which are almost

certainly not the type it is expecting. To get around this, we wrap Idris messages in a structure, `?IDRIS_MSG/2` that identifies them and keeps them from getting confused with other system messages. Within this message we also store the sender of the message, so the receiver can choose to only receive messages from a particular sender (casually emulating asynchronous channels), or from any sender either including or discarding the sender information.

That said, we only keep the process id of the sender, and I'm not convinced we can rebuild the information about the type that the sender communicates with.

One of the interesting features of the Actor model is the operation *become*, which changes the functionality of a given process without affecting the process id or existing message queue. This allows a process to start dealing with messages in a different way, should they wish. I find it quite interesting, mostly because it's similar to, but simpler than, *redirect* from [9].

```
become : Process l a → (a → Process l' b) → Process l' b
```

Listing 4.1: Perhaps the Type of *become*

If I had to give *become* a type, I think it would be something like that in Listing 4.1. This looks similar to the type of `(>>=)`, only it also gives the process a chance to change the type of messages it receives. It's this last part which is why we can't implement this functionality — if a process *a* has a ProcRef to a process *b*, and process *b* becomes a process that receives a different language, how does process *a* know to change the type of its ProcRef to match the new language the process receives? If it uses the ProcRef of the old type, it has the capacity to ruin type-safety, something we have worked hard to achieve.

There is another form of *become*, which is the form that forwards all messages to another process: `ProcRef l' -> (l -> l') -> Process l ()`. In this case we provide a mapping function so that all messages delivered to our process can be translated before forwarding them to the requested process. This at least is more type-safe than the other version, but also is not nearly as interesting for modelling protocols.

4.3 Enforcing Progress with Uniqeness Types

There are two halves to my implementation of RPC (synchronous calls between processes): the Erlang side, and the Idris side. The Erlang side is implemented in `irts/idris_erlang_conc.erl`, and the Idris side is implemented in `Erlang/RPC.idr`.

On the Idris side, we create a unique token when a process sends an RPC to another. This essentially identifies the single RPC transaction. The advantage of this token is that we can use it to compare for equality, and we know if anyone has another token that is equal, it can only have come from the same place originally. This means we can match up replies to the right

calls, even if the replies are coming in quicker than the process is retrieving them.

On the IDRIS side, the call to send an RPC returns an `RPCSenderTag`. This is a `UniqueType` that wraps a copy of the unique token from the ERLANG side. When the sender wants to retrieve the reply to their RPC, they have to pass this `RPCSenderTag` to the receive function, which will consume the tag block until the reply comes back (or return immediately if the reply was in the process' message queue). Because the tag has been consumed, a programmer can not attempt to retrieve a reply to the same RPC a second time, which would block forever, as the reply will only be sent once. So, in the case of the sender's tag, it prevents the sender trying to replay the receive.

I also use the sender tag to encode the expected message response type, which we calculated at send-time. This means that had I modelled the Actor model's *become* command, these tokens would still mean we could compute the right type of the reply.

We do almost exactly the same for the RPC Handler side in IDRIS. When an RPC message comes in, we get both the message and a `RPCRecvTag`. When we want to respond to that RPC message (most likely immediately, but there's really no requirement for that if we want to interleave RPCs for some reason), we pass the reply and tag into the reply function. The reply function consumes the tag, meaning that we can't reply again. If we were to attempt to repeat the `rpc_send_rep tag rep` from line 44, this would be a type error and the program wouldn't compile, preventing us trying to reply to a message twice and fill up a process' message queue with responses they will never read.

In fact, even more helpfully, it's perfectly possible to send the `RPCRecvTag` in a message to another function, and it could reply on behalf of the original RPC handler. This could be useful if the request required heavy computation that can be spawned into a separate process. The only problem with doing this is that I haven't managed to work out how to associate the type that the reply is required to be with the reply tag, in order to make sure that replies are of the correct type.

Both these Unique Types allow me to make sure that only one reply is sent to a process, and only one reply is ever fetched from the mailbox. An incredibly useful property of safe RPC systems, even if I don't have all the type information I want. Another useful property I haven't worked out how to prove is that an RPC process will eventually respond to a message. I had a few ideas a while ago about it, but none of them were satisfactory.

I imagine the way to encode RPC properly is to have an RPC language that specifies both the request and the response type (the latter could even be dependent on the request type). Then when spawning an RPC process, I can put this language in the type of the process reference so I know what the reply will come back as. That said, if I get up to this stage, it's essentially no great leap up to using a `gen_server`.

Chapter 5

Evaluation

> Computers let you make more mistakes than any other invention in history.
>
> With the possible exception of handguns and tequila.
>
> Anon

I am proud of the systems that I have produced while exploring a topic I'm interested in, and I think I have done this research well.

In this project, first and foremost, I have learned about how Idris works, at both ends of the compiler — I'm now a lot more confident writing Idris code than I was when I started, and I'm also happy to build code generators for other languages. I hope to improve some of the Idris code generation system with changes that were out of scope of this project.

I've also spent a lot of time reading and thinking about concurrency models and how they work, which I found enlightening. I'm particularly interested in formal verification of Distributed Systems, but I felt I first needed to work on a smaller set of concurrency problems as a precursor to that research. This work has served well to get me starting to think about these sorts of problems and how I can go about solving them.

Lastly, I've learned how to work my way through a large research project, finding and solving interesting problems along the way. I'm sure if I did this project again, I'd certainly do it a different way, but I feel that's a lot of the point.

5.1 The Value of My Work

This work will become valuable to Erlang programmers who are looking for ways of developing verified concurrent systems. Whilst dialyzer is a great tool, concurrent Erlang programs really need a better set of verification tools than its static analysis and QuickCheck.

This work could stand as the basis of a better type system for Erlang programs, but it would need a lot more work. I think the system deals with type-safe actor systems well, and the typed assertions it gives are useful.

However, it doesn't deal with some of the really hard problems that Erlang programmers need to have solved, like tools to reason about distributed programming or hot code swapping. On the other hand, dialyzer doesn't come close to helping with these either, and to verify this with QuickCheck requires very complex tests.

I realise that a lot of the theory and details from my work are inscrutable to many programmers, but I hope that these same checks will be built into an Erlang static analysis tool or DSL.

In terms of immediate impact, one of the most useful parts of this dissertation is the documentation of how Idris' code generation system works, which will be useful to other people who are implementing (or thinking of implementing) code generators. This work is the first to coherently codify and explain how all the parts of that system work together, though there are existing examples of other Idris code generators.

5.2 Example Programs

Despite their brevity, my three example programs actually exercise a surprising amount of the code generator's functionality, and therefore I'd claim they were quite useful.

The `test_primitives/MainErl.idr` example I put together early on in the project to test out coverage of a new set of interactive primitives I had proposed. In the end, the changes to the FFI system meant I could reuse this example to check and debug foreign calls, which is how most of `EIO`'s effectful interactions are made.

This example also contains an equivalent C version in `MainC.idr` which does exactly the same thing, proving I have got compilation correct. The C code generator is the canonical backend, as it is used to create normal Idris binaries if no other backend is specified.

The `test_special_ctors/MainErl.idr` example was also put together very quickly to debug and check the special-cased constructors, but it now serves to prove that these have been implemented correctly.

Lastly for the standalone examples, `rpc_example/Main.idr` contains a simple concurrent example, which shows a process spawning a counter process, then communicating with it over the RPC system I built. This shows that both the `Process 1` and the RPC systems I build work, which is great to know, but also that it is possible to reason about this kind of system.

Then in the `Gen*.idr` files, each has an `Example` namespace with a single example inside it. Respectively: the `gen_server` has an echo server (for any type); the `gen_fsm` has an FSM that either replies with a number or a string based on the state its in, which is changed with an asynchronous event; and the `gen_event` just contains an event handler that counts how many events it receives. I felt these examples were enough to be able to check if these approaches were viable and would type check without too much bother, and it seems this is the case.

Though in each of these cases I feel it's clearest if the communication language is specified separately to the handler implementations, which just shows how useful defining the protocol up front is.

5.3 Were My Objectives Met?

I think I have managed to achieve most of my objectives, but I'll go through each one.

Compiling Idris programs into Erlang

1. *Formalise how* Idris *will compile into* Erlang. I have done this, the formalisation is in Section 4.1.

2. *Create an* Idris *to* Erlang *compiler backend.* My Idris to Erlang backend[1] can produce working Erlang programs from Idris code.

   It is definitely research-quality software — it does everything it needs to, but is not a high quality program. If I were going to be building production Erlang systems with Idris, I would rewrite this compiler completely, of which the largest change would be to generate *Core Erlang*, a desugared intermediate representation of Erlang that most of the other languages for the Erlang VM compile into.

3. *Document how to create new compiler backends for* Idris. Section 3.1 is the first viable piece of documentation for creating a new backend beyond reading the code of existing backends. Together with the information in Appendix C about the Idris IRs, this document should be invaluable to backend developers.

4. *Create a small set of example* Idris *programs to demonstrate the new* Erlang *compiler backend.* There are two sequential example programs that illustrate the Erlang backend in `examples` in the `idris-erlang` repository, both of which compile and run with my backend. They may seem small, but they test the full range of type checking, foreign calls, and constructor usage. I would have liked to create more, larger examples, but I ran out of time to come back to these.

5. *Devise a foreign call Interface for* Idris. There is now a foreign call interface for Idris, implemented by Brady, but based in part on discussions we had about how it should work. This needs to be revisited in order to support exporting more complex types for some backends, but it is adequate for C and Erlang. Unfortunately I didn't manage to integrate it into my compiler yet, which I would have liked to do.

Providing ways to verify the behaviour of Erlang programs

6. *Create a small set of concurrent* Idris *example programs that can compile into* Erlang. There is an example of my RPC example working in the `examples` directory, which compiles and runs as a concurrent Erlang program, based upon the `libs/erlang/Erlang/Process.idr` and `libs/erlang/Erlang/RPC.idr`. I would have liked to produce more examples, but ran out of time to come back to this.

[1] Archibald Elliott. *Idris-Erlang*. Apr. 2015. URL: https://github.com/lenary/idris-erlang.

7. *Give a typed API to* Erlang *and it's runtime system.* A dependently-typed API is, trivially, a typed API. See the next point.

8. *Give a dependently-typed API to* Erlang *and its runtime system.* I designed a simple typed API (in `libs/erlang/Erlang/Process.idr`) for concurrent message-passing in Idris programs, which we know runs because of the example. It uses dependent types to make sure we send the right type of message to the right process.

   In `OTP/Gen*.idr` there are dependently-typed versions of the three major OTP behaviours. These use dependent-types to allow more flexibility in how they communicate than they would be allowed with less powerful statically-checked type system.

9. *Create a small set of dependently-typed concurrent* Idris *example programs that can compile into* Erlang. The RPC example in `examples` uses the dependently-typed system from `Erlang/Process.idr`, so this work is an example that can compile into Erlang.

   Each file in `libs/erlang/OTP/Gen*.idr` also contains an example of that behaviour which type checks, but unfortunately none of them will run on Erlang yet.

   For the `GenFsm.idr` system, I decided that the language would be parameterised by the current state of the `gen_fsm` when I communicated. While I think this was an interesting idea, it seemed the dependent types became increasingly overambitious, because there was no way the calling process could know what state that FSM process was in at the moment the FSM process started processing the call, meaning the caller could never know for sure what thpe they would get back. A simpler system, where each synchronous event only has one reply type, like a `gen_server` seems sensible.

10. *Devise a Hoare-like logic for* Erlang *and its runtime system.* In the end I didn't even begin to work on this objective, as it turned out to require a lot more prior knowledge than I was expecting, and I felt that it wouldn't be as useful as putting my time into other parts of the project. I hope to revisit this sometime in the future to devise a firmer basis for reasoning about Erlang programs.

Modelling concurrency calculi in Idris

11. *Model a concurrency calculus in* Idris. There is a typed interface to the Actor model in `libs/erlang/Erlang/Process.idr` that I can not only use during type checking, but can also execute, I think I've gone further than just modelling a concurrency calculus.

12. *Create a verified and executable concurrency library for* Idris *based on* Erlang *and the Actor model.* I have implemented this in `Process.idr`, which is both typed and executable. While it doesn't offer particularly strong guarantees, I feel it does satisfy this objective.

13. *Create a verified and executable concurrency library for* Idris *based on another concurrency calculus.* I did not manage to do any work towards this objective.

5.4 What Didn't I Manage?

There are two advanced objectives that I was unable to complete in the time available.

Firstly, I'd have loved to devise a Hoare-like logic for reasoning about the Erlang language and system in particular. I think this would not only have been interesting to devise, but also really useful for other researchers working on static or dynamic analysis of Erlang programs.

Secondly, I would have loved to have more time to study other concurrency calculi, with a view to implementing one of them, in a well-typed way, on top of Erlang. In the end, it was a lot of work just to devise how to model the actor model well, which is why I didn't manage to start on a second concurrency calculus.

I would also have liked to produce a lot more example programs to show exactly what was possible with the system I have designed and built. The existing examples are not enough to really show off the project.

Chapter 6

Conclusion

> In theory, there is no difference between theory and practice, but in practice I'm the wrong person to ask.
>
> @AcademicsSay

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.

6.1 What's Next?

This project leads into a few further projects, which I would like to think about.

First and foremost, I'm interested in Distributed Systems. Concurrency is required for distribution, but there are a lot of ways in which distributed systems are much harder to verify, all to do with how they fail and how information propagates around distributed systems. I see this work with Idris

as a stepping stone to do research around using type systems (or other static analysis) to verify distributed systems.

However, this is not the only direction this work leads in. There is lots of work that could still be done about writing IDRIS programs to run on ERLANG.

The first thing should be to complete the behaviours in `OTP/Gen*.idr` such that they can run on ERLANG. The compiler also needs polishing such that ERLANG programs can call into functions generated from IDRIS using the foreign exports I designed.

I also wondered about integrating this work on processes into IDRIS' Effects library, essentially having two different kinds of Effects: the ability to send messages to a particular single Process; and the ability to spawn new processes that use a given language. This would then allow the programmer to much more tightly control the side-effects of their functions.

This should be able to be extended into the OTP behaviours in the same way, with separate effects for communicating with a particular `gen_server` (for example), and for spawning new instances of a `gen_server`. Perhaps it would be possible to restrict the effects even further to only being able to make a singular particular call or cast to a particular `gen_server`. The same applies for the other OTP behaviours.

One final place I might have taken this project was to see whether we could design a system for protocols from the ground up. I.e. define processes with typed message-passing-interfaces, and then composing these definitions together such that subsets of the sends and receives match together, somewhat like the approach in CCS, but I'm not totally sure of the viability of creating correct protocols like this. Protocols almost certainly do need to be defined starting from the interactions and proceeding down to the parties, rather than starting from the parties and building up to the interactions.

As you can see, this project is the stepping stone to a lot of interesting research.

Appendices

Appendix A

Listings

The code that I developed or used as part of this project is hosted on GitHub, in the following repositories:

- The Idris to Erlang compiler and libraries that I developed is in a repository at https://github.com/lenary/idris-erlang. This repository includes some examples as well.
- The Idris repository is at https://github.com/idris-lang/Idris-dev.

Appendix **B**

Bibliography

[1] Joe Armstrong. 'A history of Erlang'. In: *the third ACM SIGPLAN conference*. New York, NY, USA: ACM Press, June 2007, pp. 6–1–6–26 (cit. on p. 11).

[2] Henry Baker and Carl Hewitt. 'Laws for communicating parallel processes'. In: (1977) (cit. on p. 10).

[3] Edwin Brady. *ConcIO*. 22nd Feb. 2015. URL: https://github.com/edwinb/ConcIO (cit. on p. 12).

[4] Edwin Brady. *Idris Empty Code Generator*. 3rd Mar. 2015. URL: https://github.com/idris-lang/idris-emptycg (cit. on p. 15).

[5] Tim Carstens. *verlang*. 3rd Aug. 2013. URL: https://github.com/tcarstens/verlang (cit. on p. 13).

[6] Maria Christakis and Konstantinos Sagonas. 'Static detection of race conditions in Erlang'. In: (2010), pp. 119–133 (cit. on p. 13).

[7] Archibald Elliott. *Idris-Erlang*. Apr. 2015. URL: https://github.com/lenary/idris-erlang (cit. on pp. 16, 37).

[8] Simon Fowler. *Monitoried Session Erlang*. 13th Mar. 2015. URL: https://github.com/SimonJF/monitored-session-erlang (cit. on p. 13).

[9] Peter Hancock and Anton Setzer. 'Interactive Programs in Dependent Type Theory'. In: *Proceedings of the 14th Annual Conference of the EACSL on Computer Science Logic*. Springer-Verlag, Aug. 2000 (cit. on pp. 22, 27, 33).

[10] Carl Hewitt and Henry Baker. 'Actors and continuous functionals'. In: (1977) (cit. on pp. 1, 10).

[11] Charles Anthony Richard Hoare. 'Communicating Sequential Processes'. In: *Communications of the ACM* 21.8 (1978), pp. 666–677 (cit. on p. 10).

[12] Kohei Honda. 'Types for dyadic interaction'. In: (1993), pp. 509–523 (cit. on p. 12).

[13] Kohei Honda, Vasco Thudichum Vasconcelos and Makoto 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 (cit. on p. 12).

[14] Kohei Honda, Nobuko Yoshida and Marco 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 (cit. on p. 12).

[15] Tobias Lindahl and Konstantinos 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 (cit. on p. 12).

[16] Simon Marlow and Philip 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 (cit. on p. 12).

[17] Christopher Meiklejohn. 'Vector Clocks in Coq: An Experience Report'. In: *arXiv.org* (June 2014). arXiv: 1406.4291v1 [cs.DC] (cit. on p. 13).

[18] Marino Miculan and Marco Paviotti. 'Synthesis of distributed mobile programs using monadic types in Coq'. In: (May 2012), pp. 183–200 (cit. on p. 13).

[19] Robin Milner. *A Calculus of Communicating Systems*. Vol. 92. Lecture Notes in Computer Science. Berlin New York: Springer-Verlag, 1980 (cit. on p. 10).

[20] Dimitris Mostrous and Vasco 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 (cit. on p. 13).

Appendix C

The Idris Language

C.1 Intermediate Representation BNFs

The Idris intermediate representation (IR) is defined in `src/Idris/Lang.hs`, and below we include a representation of this language and its variants in Bakus-Naur Form. The form has been very slightly simplified for clarity.

Declarations are essentially all the same, whether they are an `LDecl`, a `DDecl`, or an `SDecl`, what changes is the structure of the ⟨*expression*⟩.

Figure C.2 is the structure of `LExp`, the highest-order of the expression representations.

⟨*declaration*⟩	::=	function ⟨*name*⟩ [⟨*name*⟩*] ⟨*expression*⟩
	|	constructor$_n$ ⟨*name*⟩

Figure C.1: `LDecl`, `DDecl`, and `SDecl`

⟨*expression*⟩	::=	⟨*variable*⟩	– LV
	|	⟨*expression*⟩ (⟨*expression*⟩*)	– LApp
	|	lazy ⟨*name*⟩ (⟨*expression*⟩*)	– LLazyApp
	|	lazy ⟨*expression*⟩	– LLazy
	|	force ⟨*expression*⟩	– LForce
	|	let ⟨*name*⟩ := ⟨*expression*⟩ in ⟨*expression*⟩	– LLet
	|	lambda ⟨*name*⟩* → ⟨*expression*⟩	– LLam
	|	[⟨*expression*⟩]$_n$	– LProj
	|	new ⟨*name*⟩ (⟨*expression*⟩*)	– LCon
	|	case ⟨*expression*⟩ of ⟨*alternative*⟩* end	– LCase
	|	⟨*constant*⟩	– LConst
	|	foreign ⟨*fdesc*⟩ ⟨*fdesc*⟩ (⟨*farg*⟩*)	– LForeign
	|	operator ⟨*operator*⟩ (⟨*expression*⟩*)	– LOp
	|	nothing	– LNothing
	|	error ⟨*string*⟩	– LError

Figure C.2: `LExp` from `src/IRTS/Lang.hs`

⟨*expression*⟩	::=	⟨*variable*⟩	– DV
	|	⟨*name*⟩ (⟨*expression*⟩*)	– DApp
	|	let ⟨*name*⟩ := ⟨*expression*⟩ in ⟨*expression*⟩	– DLet
	|	update ⟨*name*⟩ := ⟨*expression*⟩	– DUpdate
	|	[⟨*expression*⟩]$_n$	– DProj
	|	new ⟨*name*⟩ (⟨*expression*⟩*)	– DC
	|	case ⟨*expression*⟩ of ⟨*alternative*⟩* end	– DCase
	|	chkcase ⟨*expression*⟩ of ⟨*alternative*⟩* end	– DChkCase
	|	⟨*constant*⟩	– DConst
	|	foreign ⟨*fdesc*⟩ ⟨*fdesc*⟩ (⟨*farg*⟩*)	– DForeign
	|	operator ⟨*operator*⟩ (⟨*expression*⟩*)	– DOp
	|	nothing	– DNothing
	|	error ⟨*string*⟩	– DError

Figure C.3: `DExp` from `src/IRTS/Defunctionalise.hs`

⟨*expression*⟩	::=	⟨*variable*⟩	– SV
	|	⟨*expression*⟩ (⟨*variable*⟩*)	– SApp
	|	let ⟨*variable*⟩ := ⟨*expression*⟩ in ⟨*expression*⟩	– SLet
	|	update ⟨*variable*⟩ := ⟨*expression*⟩	– SUpdate
	|	[⟨*variable*⟩]$_n$	– SProj
	|	new ⟨*name*⟩ (⟨*variable*⟩*)	– SCon
	|	case ⟨*variable*⟩ of ⟨*alternative*⟩* end	– SCase
	|	chkcase ⟨*variable*⟩ of ⟨*alternative*⟩* end	– SChkCase
	|	⟨*constant*⟩	– SConst
	|	foreign ⟨*fdesc*⟩ ⟨*fdesc*⟩ (⟨*fvar*⟩*)	– SForeign
	|	operator ⟨*operator*⟩ (⟨*variable*⟩*)	– SOp
	|	nothing	– SNothing
	|	error ⟨*string*⟩	– SError

Figure C.4: `SExp` from `src/IRTS/Simplified.hs`

The structure of `DExp` in Figure C.3 is much the same, only without the explicit laziness or lambdas. It introduces an '`update`' structure, to replace them. It also introduces a '`chkcase`' structure which works like a '`case`' structure, only the type it returns is unknown at compile time.

The structure of `SExp` (from `src/IRTS/Simplified.hs`) is further modified to be in applicative-normal form. In this case, all functions are applied to variables (rather than expressions).

Other parts of the grammar are in Figure C.5, and are common to all ⟨*expression*⟩ forms.

⟨*alternative*⟩ ::= match ⟨*name*⟩ (⟨*name*⟩*) → ⟨*expression*⟩
| ⟨*constant*⟩ → ⟨*expression*⟩
| default → ⟨*expression*⟩

⟨*variable*⟩ ::= local ⟨*integer*⟩
| global ⟨*name*⟩

⟨*name*⟩ ::= user ⟨*string*⟩
| namespace ⟨*name*⟩ (⟨*string*⟩*)
| machine ⟨*integer*⟩ ⟨*name*⟩

⟨*constant*⟩ ::= int ⟨*integer*⟩
| bigint ⟨*integer*⟩
| float ⟨*float*⟩
| char ⟨*char*⟩
| string ⟨*string*⟩
| arithtype ⟨*arithty*⟩
| stringtype
| worldtype
| world
| voidtype
| forgot

⟨*arithty*⟩ ::= int ⟨*intty*⟩ | float

⟨*intty*⟩ ::= fixed ⟨*nativety*⟩ | native | big | char

⟨*nativety*⟩ ::= it8 | it16 | it32 | it64

⟨*fdesc*⟩ ::= constructor ⟨*name*⟩
| string ⟨*string*⟩
| unknown
| io ⟨*fdesc*⟩
| apply ⟨*name*⟩ (⟨*fdesc*⟩*)

⟨*farg*⟩ ::= arg ⟨*fdesc*⟩ ⟨*expression*⟩

⟨*fvar*⟩ ::= var ⟨*fdesc*⟩ ⟨*variable*⟩

⟨*interface*⟩ ::= export ⟨*name*⟩ ⟨*string*⟩ (⟨*export*⟩*)

⟨*export*⟩ ::= data ⟨*string*⟩
| fun ⟨*name*⟩ ⟨*fdesc*⟩ ⟨*fdesc*⟩ (⟨*fdesc*⟩*)

Figure C.5: The Rest of the Idris Intermediate Representation. ⟨*operator*⟩ is defined in Figure C.6.

⟨*operator*⟩ ::= + ⟨*arithty*⟩ | − ⟨*arithty*⟩ | × ⟨*arithty*⟩ | udiv ⟨*arithty*⟩
| sdiv ⟨*arithty*⟩ | urem ⟨*arithty*⟩ | srem ⟨*arithty*⟩
| and ⟨*intty*⟩ | or ⟨*intty*⟩ | xor ⟨*intty*⟩ | complement ⟨*intty*⟩
| shiftl ⟨*intty*⟩ | lshiftr ⟨*intty*⟩ | ashiftr ⟨*intty*⟩
| eq ⟨*arithty*⟩ | lt ⟨*intty*⟩ | lte ⟨*intty*⟩ | gt ⟨*intty*⟩
| gte ⟨*intty*⟩
| slt ⟨*arithty*⟩ | slte ⟨*arithty*⟩ | sgt ⟨*arithty*⟩ | sgte ⟨*arithty*⟩
| signextend ⟨*intty*⟩ ⟨*intty*⟩ | zeroextend ⟨*intty*⟩ ⟨*intty*⟩
| truncate ⟨*intty*⟩ ⟨*intty*⟩
| int-float ⟨*intty*⟩ | float-int ⟨*intty*⟩
| int-str ⟨*intty*⟩ | str-int ⟨*intty*⟩
| float-str | str-float | ch-int ⟨*intty*⟩ | int-ch ⟨*intty*⟩
| bitcast ⟨*arithty*⟩ ⟨*arithty*⟩
| exp | log | sin | cos | tan | asin | acos | atan | sqrt
| floor | ceil | negate
| strconcat | strlt | streq | strlen | strhead | strtail
| strcons | strindex | strrev | readstr | writestr
| systeminfo | fork | par | noop
| external ⟨*name*⟩

⟨*rts-bool*⟩ ::= 1 — True
| 0 — False

Figure C.6: Idris Primitive Operators, from `src/IRTS/Lang.hs`

C.2 Primitives

Until this document, there has been no central documentation of all of the idris primitives, including their expected functionality and types. I won't claim this listing is canonical, but I hope it at least helps someone out.

In the following list, each operator is being given with a Haskell-like type signature, for clarity. The expected types of any external primitives will be declared in the code. The existing external primitives are declared in `libs/prelude/Builtins.idr`.

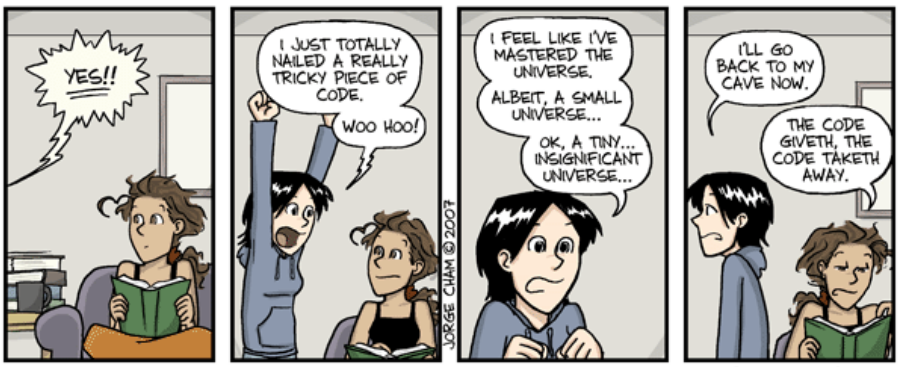
- $+\langle arithty\rangle :: \langle arithty\rangle \to \langle arithty\rangle \to \langle arithty\rangle$ — Arithmetic addition.
- $-\langle arithty\rangle :: \langle arithty\rangle \to \langle arithty\rangle \to \langle arithty\rangle$ — Arithmetic subtraction.
- $\times\langle arithty\rangle :: \langle arithty\rangle \to \langle arithty\rangle \to \langle arithty\rangle$ — Arithmetic multiplication.
- udiv$\langle arithty\rangle :: \langle arithty\rangle \to \langle arithty\rangle \to \langle arithty\rangle$ — Unsigned arithmetic division.
- sdiv$\langle arithty\rangle :: \langle arithty\rangle \to \langle arithty\rangle \to \langle arithty\rangle$ — Signed arithmetic division.
- urem$\langle arithty\rangle :: \langle arithty\rangle \to \langle arithty\rangle \to \langle arithty\rangle$ — Unsigned arithmetic remainder.

- srem⟨*arithty*⟩ :: ⟨*arithty*⟩ → ⟨*arithty*⟩ → ⟨*arithty*⟩ — Signed arithmetic remainder.
- and⟨*intty*⟩ :: ⟨*intty*⟩ → ⟨*intty*⟩ → ⟨*intty*⟩ — Bitwise logical AND.
- or⟨*intty*⟩ :: ⟨*intty*⟩ → ⟨*intty*⟩ → ⟨*intty*⟩ — Bitwise logical OR.
- xor⟨*intty*⟩ :: ⟨*intty*⟩ → ⟨*intty*⟩ → ⟨*intty*⟩ — Bitwise logical XOR (aka exclusive or).
- complement :: ⟨*intty*⟩ → ⟨*intty*⟩ — Bitwise logical negation.
- shiftl⟨*intty*⟩ :: ⟨*intty*⟩ → ⟨*intty*⟩ → ⟨*intty*⟩ — Left bitshift.
- lshiftr⟨*intty*⟩ :: ⟨*intty*⟩ → ⟨*intty*⟩ → ⟨*intty*⟩ — Right logical bitshift.
- ashiftr⟨*intty*⟩ :: ⟨*intty*⟩ → ⟨*intty*⟩ → ⟨*intty*⟩ — Right arithmetic bitshift.
- eq⟨*intty*⟩ :: ⟨*arithty*⟩ → ⟨*arithty*⟩ → ⟨*rts-bool*⟩ — Arithmetic equality.
- lt⟨*intty*⟩ :: ⟨*intty*⟩ → ⟨*intty*⟩ → ⟨*rts-bool*⟩ — Arithmetic less-than.
- lte⟨*intty*⟩ :: ⟨*intty*⟩ → ⟨*intty*⟩ → ⟨*rts-bool*⟩ — Arithmetic less-than-or-equal-to.
- gt⟨*arithty*⟩ :: ⟨*intty*⟩ → ⟨*intty*⟩ → ⟨*rts-bool*⟩ — Arithmetic greater-than.
- gte⟨*arithty*⟩ :: ⟨*intty*⟩ → ⟨*intty*⟩ → ⟨*rts-bool*⟩ — Arithmetic greater-than-or-equal-to.
- slt⟨*arithty*⟩ :: ⟨*arithty*⟩ → ⟨*arithty*⟩ → ⟨*rts-bool*⟩ — Arithmetic signed less-than.
- slte⟨*arithty*⟩ :: ⟨*arithty*⟩ → ⟨*arithty*⟩ → ⟨*rts-bool*⟩ — Arithmetic signed less-than-or-equal-to.
- sgt⟨*arithty*⟩ :: ⟨*arithty*⟩ → ⟨*arithty*⟩ → ⟨*rts-bool*⟩ — Arithmetic signed greater-than.
- sgte⟨*arithty*⟩ :: ⟨*arithty*⟩ → ⟨*arithty*⟩ → ⟨*rts-bool*⟩ — Arithmetic signed greater-than-or-equal-to.
- signextend⟨*intty*⟩⟨*intty*⟩ :: ⟨*intty*⟩ → ⟨*intty*⟩ — Sign extend from the first integral type to the second one. The first integral type should be smaller than the second one.
- zeroextend⟨*intty*⟩⟨*intty*⟩ :: ⟨*intty*⟩ → ⟨*intty*⟩ — Zero extend from the first integral type to the second one. The first integral type should be smaller than the second one.
- truncate⟨*intty*⟩⟨*intty*⟩ :: ⟨*intty*⟩ → ⟨*intty*⟩ — Truncate from the first integral type to the second one. The first integral type should be larger than the second one.

- int – float⟨*intty*⟩ :: ⟨*intty*⟩ → ⟨*float*⟩ — Cast from a particular integral type into a floating-point number.
- float – int⟨*intty*⟩ :: ⟨*intty*⟩ → ⟨*float*⟩ — Cast from a floating-point number into a particular integral type.
- int – str⟨*intty*⟩ :: ⟨*intty*⟩ → ⟨*string*⟩ — Cast from a particular integral type into a string.
- str – int⟨*intty*⟩ :: ⟨*string*⟩ → ⟨*intty*⟩ — Cast from a string into a particular integral type.
- float – str :: ⟨*float*⟩ → ⟨*string*⟩ — Cast from a floating-point number into a string.
- str – float :: ⟨*string*⟩ → ⟨*float*⟩ — Cast from a string into a floating-point number.
- ch – int⟨*intty*⟩ :: ⟨*character*⟩ → ⟨*intty*⟩ — Cast from a character into a particular integral type.
- int – ch⟨*intty*⟩ :: ⟨*intty*⟩ → ⟨*character*⟩ — Cast from a particular integral type into a character.
- bitcast⟨*intty*⟩⟨*intty*⟩ :: ⟨*intty*⟩ → ⟨*intty*⟩ — Cast from one integral type to another. Right now Idris doesn't actually ever generate this primitive, which means information about lacking, but that may change in the future.
- exp :: ⟨*float*⟩ → ⟨*float*⟩ — The Natural Exponential function on floating-point numbers
- log :: ⟨*float*⟩ → ⟨*float*⟩ — The Natural Logarithm function on floating-point numbers.
- sin :: ⟨*float*⟩ → ⟨*float*⟩ — The Sine function on floating-point numbers.
- cos :: ⟨*float*⟩ → ⟨*float*⟩ — The Cosine function on floating-point numbers.
- tan :: ⟨*float*⟩ → ⟨*float*⟩ — The Tangent function on floating-point numbers.
- asin :: ⟨*float*⟩ → ⟨*float*⟩ — The Arcsine function on floating-point numbers.
- acos :: ⟨*float*⟩ → ⟨*float*⟩ — The Arccosine function on floating-point numbers.
- atan :: ⟨*float*⟩ → ⟨*float*⟩ — The Arctangent function on floating-point numbers.
- sqrt :: ⟨*float*⟩ → ⟨*float*⟩ — The Square-root function on floating-point numbers.

- floor :: ⟨*float*⟩ → ⟨*float*⟩ — The Floor function on floating-point numbers.
- ceil :: ⟨*float*⟩ → ⟨*float*⟩ — The Ceiling function on floating-point numbers.
- negate :: ⟨*float*⟩ → ⟨*float*⟩ — The Negate function on floating-point numbers.
- strconcat :: ⟨*string*⟩ → ⟨*string*⟩ → ⟨*string*⟩ — String concatenation.
- strlt :: ⟨*string*⟩ → ⟨*string*⟩ → ⟨*rts-bool*⟩ — String comparison (less-than).
- streq :: ⟨*string*⟩ → ⟨*string*⟩ → ⟨*rts-bool*⟩ — String equality.
- strlen :: ⟨*string*⟩ → ⟨*intty*⟩ — String length. The returned length should be of the native integral type.
- strhead :: ⟨*string*⟩ → ⟨*character*⟩ — Return the first character of a string.
- strtail :: ⟨*string*⟩ → ⟨*string*⟩ — Return a copy of the string without the first character.
- strcons :: ⟨*character*⟩ → ⟨*string*⟩ → ⟨*string*⟩ — String Construction. Create a new string by prepending the given character to the start of the given string. This can be more inefficient than *strconcat*, as the Idris compiler will try to use that operator in as many places as possible.
- strindex :: ⟨*string*⟩ → ⟨*intty*⟩ → ⟨*character*⟩ — Return the character at the given index in the string. The index will be in the native integer type.
- strrev :: ⟨*string*⟩ → ⟨*string*⟩ — Return a copy of the string where the original string is reversed.
- readstr :: ⟨*world*⟩ → ⟨*string*⟩ — Read the first line of *stdin* and return it as a string. The ⟨*world*⟩ argument is so that Idris doesn't compile out this primitive because it thinks it is pure, and can be ignored.
- writestr :: ⟨*world*⟩ → ⟨*string*⟩ → ⟨*rts-bool*⟩ — Write the given string to *stdout* (without a newline), and return whether the operation was successful or not. Again, the ⟨*world*⟩ can be ignored.
- systeminfo :: ⟨*intty*⟩ → ⟨*string*⟩ — Generate a function that returns information about the system as a string. This will either be called with a native integer of either 0, 1 or 2. If the argument is 0, return a string representing the name of the backend. If the argument is 1, return a string representing the OS the program was compiled on. If the argument is 2, return a string representing the architecture target triple that the program was compiled on.
- fork :: IO*a* → ⟨*pointer*⟩ — Fork a new concurrent process to evaluate the given expression, returning a pointer to that process via which the process can be messaged with. The expression may have side effects, as it is an `IO` function.

- par :: $a \to a$ — Evaluate the argument in a parallel thread, returning the result. This doesn't even need to actually be done in parallel if the backend is implicitly single-threaded, as long as it returns the result of the computation.
- noop :: $a \to a$ — Returns its argument. Not currently generated by the compiler.
- external$\langle$*name*$\rangle$ — The type of these primitives is declared within IDRIS source code itself. Some examples are in `libs/prelude/Builtins.idr`.


YES!!
I JUST TOTALLY NAILED A REALLY TRICKY PIECE OF CODE.
WOO HOO!
JORGE CHAM © 2007
I FEEL LIKE I'VE MASTERED THE UNIVERSE.
ALBEIT, A SMALL UNIVERSE...
OK, A TINY... INSIGNIFICANT UNIVERSE...
I'LL GO BACK TO MY CAVE NOW.
THE CODE GIVETH, THE CODE TAKETH AWAY.
WWW.PHDCOMICS.COM