Real World Idris

Ioan Luca
201638554
supervised by Dr. Robert Atkey
University of Strathclyde

March 20, 2019

List of Figures

List of Tables

Acknowledgements

Abstract

Salut c'est quoi?

Contents

1 Introduction **2**
- 1.1 Overview . . . 2
- 1.2 Background . . . 2
  - 1.2.1 Idris . . . 2
  - 1.2.2 OCaml . . . 3
  - 1.2.3 Mirage . . . 3
  - 1.2.4 The Idris-Malfunction compiler back-end . . . 4
- 1.3 Objectives . . . 4
  - 1.3.1 Basic . . . 4
  - 1.3.2 Intermediate . . . 5
  - 1.3.3 Advanced . . . 5
- 1.4 Preliminary survey of related work . . . 5
- 1.5 Methodology . . . 6
  - 1.5.1 Specification . . . 6
  - 1.5.2 Design . . . 6
  - 1.5.3 Implementation . . . 7
  - 1.5.4 Verification . . . 7
  - 1.5.5 Evaluation . . . 7

2 Related Work **8**

3 Problem Description and Specification **9**

4 System Design **10**

5 Detailed Design and Implementation **11**

6 Verification and Validation **12**

7 Results and Evaluation **13**

8 Summary and Conclusions **14**

A Appendix Title **15**

Chapter 1

Introduction

1.1 Overview

"Real World Idris" aims to enhance the Idris ecosystem by building a Foreign Function Interface (FFI) to OCaml, to be used as a mechanism for calling functions and services written in OCaml from Idris.

The idea of this upgrade is to enable Idris programs to build on top of "battle-tested" software which would ultimately make Idris more suitable for practical programming. For instance, one could write a secure, high-performance network application in Idris running as a unikernel constructed by MirageOS, an OCaml built library operating system. This functionality will be added on top of the Idris-Malfunction compiler back-end.

1.2 Background

1.2.1 Idris

Idris is a general purpose, dependently typed programming language that has an advanced type system which encodes complex properties about programs into types, which means that the compiler checks the programs for correctness before they run. This technique is described as "Type-Driven development" and Idris leverages types as the foundation of the code, allowing relationships and assumptions to be expressed using language constructs directly.

Idris is at the forefront of a new generation of programming languages that support "lightweight verification" — specifications are defined early during development, enabling writing maintainable code that is guaranteed to work. This has the potential to drastically reduce the costs of reliable software by integrating verification into development, instead of considering it a separate

concern.

Despite being novel and popular in the Programming Language research community Idris did not catch the attention of the industrial developer communities. This resulted in a lack of reliable, secure and optimized tools and libraries in the Idris ecosystem. Firstly, despite having an extensible compiler that targets more than one platform, the Idris main/native runtime currently lags wise behind its industry competitors in terms of performance. Secondly, Idris has a steep learning curve partly because it requires a somewhat strong background in functional programming on which it relies heavily. In addition, the "Type Driven" paradigm is based on the Dependent Types theory which is widely inaccessible outside the academic world.

1.2.2 OCaml

OCaml is a general purpose industrial-strength functional programming language with a high-performance runtime system that focuses on safety and expressiveness.

Since its development has started in 1996, an active community of academics and their industry counterparts have contributed to a rich set of libraries and tools for building advanced, secure and reliable software. Mirage is an example of such a system.

1.2.3 Mirage

Mirage is a library operating system written in OCaml, used to build network applications with an emphasis on safety and speed. Such an application is designed to run as a unikernel that can be deployed across a variety of platforms.

A library OS offers typical functionality, such as networking, in the form of a collection of software libraries that can be mixed with configuration code to encapsulate an application in a unikernel.

Unikernels are specialized machine images. They are said to be "single address space" because they provide only one address space that is shared globally between all processes. Unikernels can run directly on a so called hypervisor — a piece of hardware, firmware or software that creates and runs virtual machines.

In essence, Mirage enables programmers to select a minimal set of OS specific low-level functionality required for their applications which are then deployed as fixed-purpose programs that don't need an underlying OS to be executed.

1.2.4 The Idris-Malfunction compiler back-end

Recently work has been put into solving the Idris performance gap by creating a new code generation back-end that compiles Idris to Malfunction — a thin wrapper around the OCaml Lambda intermediate representation. Malfunction is then compiled to native code by the OCaml compiler using "flambda" optimizations for common high level abstractions found in functional languages — higher-order functions, parametric polymorphism, higher-kinded types, lambdas — anonymous functions, pattern matching and lazy evaluation.

1.3 Objectives

The work will involve mapping the OCaml types to Idris equivalents, implementing the Foreign Function Interface and there will also be opportunities to fix some of the ergonomic issues within the current Idris-Malfunction project.

The upgrade will be carried out using the Haskell programming language which is what the compiler back-end is currently written in. It will, however, involve coding in OCaml and Idris as well.

Upon completion of a working FFI, an HTTP server will be written in Idris as proof of concept. The server will be deployed to either a chip or to AWS.

1.3.1 Basic

Use the provided high level data types for describing foreign function function calls in order to implement a simple Foreign Function Interface (FFI). Specifically, it should be possible to call simple functions — that return and accept primitive types, from OCaml. For example, calling functions like $val\ floor : float \rightarrow float$ from Idris.

This should be achieved by using the foreign function call construct in Idris and defining parts of the fields defined in the FFI record, namely:

- the predicate ffi_types that describes which types can be passed to and from foreign functions
- the type of function descriptors ffi_fn
- the type of exported data descriptors ffi_data

1.3.2 Intermediate

Export Idris monomorphic functions like $not : Bool \to Bool$ to "header" files, so that they can be called from OCaml, via the Idris-Malfunction backend. Describing the foreign exports can be achieved by implementing the FFI_Export type in Idris.

Support conversion between Idris high level types and OCaml types, being able to safely exchange complex data. More specifically, come up with ways of mapping data types like records and functions to OCaml equivalents. The challenging parts will involve finding a general strategy for converting dependent types to regular types that exist in OCaml. However, not all the types are required — indeed, some may be inappropriate or impossible in this case.

1.3.3 Advanced

Prove the underlying concept of the project — create and deploy a Mirage unikernel that runs a networking application (HTTP) written in Idris and deploy it to a chip (ESP32) or cloud (Amazon Web Services):

- create Idris bindings for the required Mirage modules
- write a configuration file for Mirage
- deploy to the target platform

1.4 Preliminary survey of related work

The "Cross-platform Compilers for Functional Languages" draft research paper by Edwin Brady, the creator of Idris, provided useful insights about the generic FFI in Idris.

The C and JavaScript code generation back-ends that are distributed with the official Idris platform show the steps they take to support interfacing with the target languages.

A study of basic programs written in OCaml and Idris was performed to gain an initial understanding of their type systems.

An initial reading on unikernels and how they work in the context of Mirage was done.

A meeting with Lucas Pluvinage took place at ICFP'18 where a discussion about his recent successful work in compiling Mirage to an ESP32 chip turned out to be an inspiring opportunity for this project. His talk where

he described his work in more detail was attended as part of the OCaml workshop.

Ultimately, opportunities to fix some of the ergonomic issues with the current Idris-Malfunction back-end have been identified. Some preliminary work will include:

- dummy parameter optimization
- support for Unicode
- improve compilation time
- code generation refactoring
- implement any non-implemented Idris primitives
- find a reasonably good trade off regarding block tags being limited to 200
- test BigIntegers
- deal with floating point numbers
- bump to latest Idris, Stack, Opam and OCaml

1.5 Methodology

The current plan is to develop this project using an Agile methodology and document progress as often as possible. The idea is to start writing the report and the source code early and keep them synchronized throughout the stages of the project.

1.5.1 Specification

The specification will start as a more detailed description of the deliverables and it will be added on to as the project evolves and more features can be identified. Finally, it needs to accurately resemble the accompanying artefact at all times.

1.5.2 Design

The design will heavily be influenced by other Idris code generation back-ends and the FFI upgrade has to fit nicely in the current Idris-Malfunction back-end. As it is usually the case with Agile development, the design will suffer many iterations.

1.5.3 Implementation

The report will be written using *Tex technologies while the software part will involve coding in Haskell, Idris and OCaml on a Unix system. The VS-Code text editor with specific plug-ins installed will serve as development environment, while Trello is going to aid project management. Version control will be carried out using Git. Stack will be leveraged as a build tool. More software that automates the Software Development Life Cycle will need to be used — Travis for continuous integration for instance.

1.5.4 Verification

To test the Haskell code base, some of the following will be used:

- QuickCheck or SmallCheck — generative testing
- HUnit for — testing
- Criterion — benchmarking
- HSpec or Tasty — testing framework wrapper

1.5.5 Evaluation

Evaluation will mainly consist of writing Idris code that uses OCaml bindings and testing that it has the correct behaviour. Moreover, Idris and OCaml semantically equivalent programs should behave the same. This will also be an incremental process. Even from the very beginning stages of development, small Idris programs should should be built to test small bits of functionality. This will provide confidence when attempting to extend pieces of functionality and then more complex Idris programs can be built on top of the small ones. Nevertheless, some things will have to be tested in isolation as well.

Upon completion of the advanced deliverable, the running Idris networking application will be validated for correctness.

Chapter 2

Related Work

Chapter 3

Problem Description and Specification

Chapter 4

System Design

Chapter 5

Detailed Design and Implementation

Chapter 6

Verification and Validation

Chapter 7

Results and Evaluation

Chapter 8

Summary and Conclusions

Appendix A

Appendix Title

appendices