Real World Idris

Project Progress Presentation

Ioan Luca
201638554
supervisor: Dr Robert Atkey
second marker: Dr Dmitri Roussinov

University of Strathclyde

January 11, 2019

A little bit about Idris

- pure functional language inspired by Haskell

A little bit about Idris

- pure functional language inspired by Haskell
- has dependent types, so can encode complex properties

A little bit about Idris

- pure functional language inspired by Haskell
- has dependent types, so can encode complex properties about programs into types enabling "Type-Driven Development"
- i.e. allows formal lightweight verification — proving correctness of programs inside the source code itself, much like mathematical theorem proving

A little bit about Idris

- pure functional language inspired by Haskell
- has dependent types, so can encode complex properties about programs into types enabling "Type-Driven Development"
- i.e. allows formal lightweight verification — proving correctness of programs inside the source code itself, much like mathematical theorem proving
- programming becomes a very interactive experience — devs think about interesting functionality of their programs and the compiler figures out where it is/could be useful

A little bit about Idris

- pure functional language inspired by Haskell
- has dependent types, so can encode complex properties about programs into types enabling "Type-Driven Development"
- i.e. allows formal lightweight verification — proving correctness of programs inside the source code itself, much like mathematical theorem proving
- programming becomes a very interactive experience — devs think about interesting functionality of their programs and the compiler figures out where it is/could be useful
- so it's quite new and innovative — mostly in the academic world at the moment

A little bit about Idris

- pure functional language inspired by Haskell
- has dependent types, so can encode complex properties about programs into types enabling "Type-Driven Development"
- i.e. allows formal lightweight verification — proving correctness of programs inside the source code itself, much like mathematical theorem proving
- programming becomes a very interactive experience — devs think about interesting functionality of their programs and the compiler figures out where it is/could be useful
- so it's quite new and innovative — mostly in the academic world at the moment
- Idris-Malfunction, recent work to solve the performance gap

Overview

- essentially an enhancement of the Idris ecosystem that aims to make it more suitable for practical programming (for the "Real World")

Overview

- essentially an enhancement of the Idris ecosystem that aims to make it more suitable for practical programming (for the "Real World")
- a (two-way) Foreign Function Interface (FFI) that allows calling OCaml from Idris and vice-versa — to be built on top of the compiler backend I wrote

Overview

- essentially an enhancement of the Idris ecosystem that aims to make it more suitable for practical programming (for the "Real World")
- a (two-way) Foreign Function Interface (FFI) that allows calling OCaml from Idris and vice-versa — to be built on top of the compiler backend I wrote
- Why? — OCaml is a very popular "battle-tested" functional language

Overview

- essentially an enhancement of the Idris ecosystem that aims to make it more suitable for practical programming (for the "Real World")
- a (two-way) Foreign Function Interface (FFI) that allows calling OCaml from Idris and vice-versa — to be built on top of the compiler backend I wrote
- Why? — OCaml is a very popular "battle-tested" functional language
- Lots of modular, secure and fast software has been written in it

Overview

- essentially an enhancement of the Idris ecosystem that aims to make it more suitable for practical programming (for the "Real World")
- a (two-way) Foreign Function Interface (FFI) that allows calling OCaml from Idris and vice-versa — to be built on top of the compiler backend I wrote
- Why? — OCaml is a very popular "battle-tested" functional language
- Lots of modular, secure and fast software has been written in it
- MirageOS is on such example and it works as a library, so we'd like to use Mirage in Idris for instance

Overview

- essentially an enhancement of the Idris ecosystem that aims to make it more suitable for practical programming (for the "Real World")
- a (two-way) Foreign Function Interface (FFI) that allows calling OCaml from Idris and vice-versa — to be built on top of the compiler backend I wrote
- Why? — OCaml is a very popular "battle-tested" functional language
- Lots of modular, secure and fast software has been written in it
- MirageOS is on such example and it works as a library, so we'd like to use Mirage in Idris for instance
- to build an HTTP server...

Mirage

- written in OCaml

Mirage

- written in OCaml
- library OS — amazing for building safe and blazing fast network apps

Mirage

- written in OCaml
- library OS — amazing for building safe and blazing fast network apps
- basically provides only the minimal set of low-level functionality that your app needs and strips down all the heavy bits of a fully-fledged OS

Mirage

- written in OCaml
- library OS — amazing for building safe and blazing fast network apps
- basically provides only the minimal set of low-level functionality that your app needs and strips down all the heavy bits of a fully-fledged OS
- constructs unikernels that can be deployed on a variety of platforms — hardware ($5 ESP32 chip) or hypervisor (AWS)

Mirage

- written in OCaml
- library OS — amazing for building safe and blazing fast network apps
- basically provides only the minimal set of low-level functionality that your app needs and strips down all the heavy bits of a fully-fledged OS
- constructs unikernels that can be deployed on a variety of platforms — hardware ($5 ESP32 chip) or hypervisor (AWS)
- secure and incredibly fast

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 code generation project.

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 code generation project.

- Basic (Currently working on)

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 code generation project.

- Basic (Currently working on)
  - basic Idris to OCaml FFI

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 code generation project.

- Basic (Currently working on)
  - basic Idris to OCaml FFI
  - described using Idris support for describing foreign function calls

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 code generation project.

- Basic (Currently working on)
  - basic Idris to OCaml FFI
  - described using Idris support for describing foreign function calls
  - i.e. fill in fields of the FFI record: *ffi_types*, *ffi_fn*, *ffi_data*

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 code generation project.

- Basic (Currently working on)
  - basic Idris to OCaml FFI
  - described using Idris support for describing foreign function calls
  - i.e. fill in fields of the FFI record: *ffi_types*, *ffi_fn*, *ffi_data*
  - successfully call simple functions like $val\ floor : float \rightarrow float$

Objectives (Intermediate)

- export Idris monomorphic (types are concrete) functions to OCaml

Objectives (Intermediate)

- export Idris monomorphic (types are concrete) functions to OCaml
- essentially the other way around

Objectives (Intermediate)

- export Idris monomorphic (types are concrete) functions to OCaml
- essentially the other way around
- at this point support 2-way conversion between the Idris and OCaml type systems

Objectives (Intermediate)

- export Idris monomorphic (types are concrete) functions to OCaml
- essentially the other way around
- at this point support 2-way conversion between the Idris and OCaml type systems
- safely exchanging complex data between the 2 ecosystems

Objectives (Intermediate)

- export Idris monomorphic (types are concrete) functions to OCaml
- essentially the other way around
- at this point support 2-way conversion between the Idris and OCaml type systems
- safely exchanging complex data between the 2 ecosystems
- challenging because some of the Idris types don't make sense in OCaml and hence, don't have equivalents

Objectives (Intermediate)

- export Idris monomorphic (types are concrete) functions to OCaml
- essentially the other way around
- at this point support 2-way conversion between the Idris and OCaml type systems
- safely exchanging complex data between the 2 ecosystems
- challenging because some of the Idris types don't make sense in OCaml and hence, don't have equivalents
- anyway, for now it's only interesting to call OCaml from Idris, but for the sake of completeness

Objectives (Intermediate)

- export Idris monomorphic (types are concrete) functions to OCaml
- essentially the other way around
- at this point support 2-way conversion between the Idris and OCaml type systems
- safely exchanging complex data between the 2 ecosystems
- challenging because some of the Idris types don't make sense in OCaml and hence, don't have equivalents
- anyway, for now it's only interesting to call OCaml from Idris, but for the sake of completeness
- so we need to implement the *FFI_Export* type in Idris

Objectives (Intermediate)

- export Idris monomorphic (types are concrete) functions to OCaml
- essentially the other way around
- at this point support 2-way conversion between the Idris and OCaml type systems
- safely exchanging complex data between the 2 ecosystems
- challenging because some of the Idris types don't make sense in OCaml and hence, don't have equivalents
- anyway, for now it's only interesting to call OCaml from Idris, but for the sake of completeness
- so we need to implement the *FFI_Export* type in Idris
- successfully export functions like $not \ : \ Bool \rightarrow Bool$ to OCaml header (.mli) files that can then be called

Objectives (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):

Objectives (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

Objectives (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

Objectives (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

Progress to date

- working on the 1st deliverable

Progress to date

- working on the 1st deliverable
- looking at how the C and JavaScript backends implement their FFIs
  https://github.com/idris-lang/Idris-dev/tree/master/codegen

Progress to date

- working on the 1st deliverable
- looking at how the C and JavaScript backends implement their FFIs
  https://github.com/idris-lang/Idris-dev/tree/master/codegen
- set up the Development Environment

Progress to date

- working on the 1st deliverable
- looking at how the C and JavaScript backends implement their FFIs
  https://github.com/idris-lang/Idris-dev/tree/master/codegen
- set up the Development Environment
- prototyping with implementing the FFI records from the "Cross-platform Compilers for Functional Languages" paper by Edwin Brady

Progress to date

- working on the 1st deliverable
- looking at how the C and JavaScript backends implement their FFIs
  https://github.com/idris-lang/Idris-dev/tree/master/codegen
- set up the Development Environment
- prototyping with implementing the FFI records from the "Cross-platform Compilers for Functional Languages" paper by Edwin Brady
- Malfunction has recently been updated (now supports lazy evaluation natively and has floats) so I need to update the code generation back-end

Progress to date

- working on the 1st deliverable
- looking at how the C and JavaScript backends implement their FFIs
  https://github.com/idris-lang/Idris-dev/tree/master/codegen
- set up the Development Environment
- prototyping with implementing the FFI records from the "Cross-platform Compilers for Functional Languages" paper by Edwin Brady
- Malfunction has recently been updated (now supports lazy evaluation natively and has floats) so I need to update the code generation back-end
- working on fixing some of the ergonomic problems with Idris-Malfunction (dummy parameters optimization, support for Unicode, primitives, bump to latest Idris/Opam/OCaml compilation time)

Progress to date

- working on the 1st deliverable
- looking at how the C and JavaScript backends implement their FFIs
  https://github.com/idris-lang/Idris-dev/tree/master/codegen
- set up the Development Environment
- prototyping with implementing the FFI records from the "Cross-platform Compilers for Functional Languages" paper by Edwin Brady
- Malfunction has recently been updated (now supports lazy evaluation natively and has floats) so I need to update the code generation back-end
- working on fixing some of the ergonomic problems with Idris-Malfunction (dummy parameters optimization, support for Unicode, primitives, bump to latest Idris/Opam/OCaml compilation time)
- Getting familiar with Mirage

Evaluation

- write Idris code that calls OCaml routines and Services and test for correct behaviour (vice-versa too)
- compile programs with the 2 compilers and if they are semantically equivalent then the behaviour should be the same
- test early during Development
- upon completion of the advanced deliverable, the Idris networking application will be validated for correctness

Overall project plan

- week 12 — 14 Jan - 20 Jan, week 13 — 21 Jan - 27 Jan
  Implement basic deliverable and start writing the report
- week 14 — 28 Jan - 03 Feb, week 15 — 04 Feb - 10 Feb
  Implement the intermediate deliverable and report Testing
- week 16 — 11 Feb - 17 Feb, week 17 — 18 Feb - 24 Feb, week 18 — 25 Feb - 03 Mar
  Attempting the advanced deliverable, Report Testing, **Mirage**
- week 19 — 04 Mar - 10 Mar, week 20 — 11 Mar - 17 Mar
  week 21 — 18 Mar - 24 Mar — Need to be ready by now
  Validation, Verification, Proofread
  Refactor, Finish Writing up, Prepare Demo
- **week 22 — 25 Mar - 27 Mar — Submission Monday 25, 12:00 Demo is on Wednesday 27**
  Hooray!