Idris 2: Quantitative Type Theory in Practice (Artifact)

Edwin Brady
School of Computer Science, University of St Andrews, Scotland, UK

Abstract

Dependent types allow us to express precisely what a function is intended to do. Recent work on Quantitative Type Theory (QTT) extends dependent type systems with linearity, also allowing precision in expressing when a function can run. This is promising, because it suggests the ability to design and reason about resource usage protocols, such as we might find in distributed and concurrent programming, where the state of a communication channel changes throughout program execution.

Idris 2 is a new version of Idris, implemented in itself, and based on Quantitative Type Theory. The paper introduces Idris 2 and describes how QTT has influenced its design, as well as giving several examples of how to use QTT in practice. The artifact, correspondingly, provides an implementation of Idris 2, running on a virtual machine, along with runnable examples from the paper. This document explains how to install the artifact, how to run the examples, and suggests some small ways to experiment with and modify the examples.

2012 ACM Subject Classification Software and its engineering → Functional languages

Keywords and phrases Dependent types, linear types, concurrency

Digital Object Identifier 10.4230/DARTS.7.2.10

Acknowledgements This work was funded by EPSRC grant EP/T007265/1. Thanks to the Idris community for their many contributions to this Idris 2 project.

Related Article Edwin Brady, "Idris 2: Quantitative Type Theory in Practice", in 35th European Conference on Object-Oriented Programming (ECOOP 2021), LIPIcs, Vol. 194, pp. 9:1–9:26, 2021. `https://doi.org/10.4230/LIPIcs.ECOOP.2021.9`

Related Conference 35th European Conference on Object-Oriented Programming (ECOOP 2021), July 12–16, 2021, Aarhus, Denmark (Virtual Conference)

1 Scope

The artifact includes the main examples from the paper, installed on a vrtual machine running a minimal installation of Alpine Linux, with 4GB RAM. The image can be imported as an appliance to VirtualBox and used as follows:

1. Choose `File...Import Appliance` from the VirtualBox menu
2. Choose the file `VM/idris-v0.3.0-playground.ova`
3. Choose settings for the VM (the default settings should be fine)
4. Click `Import`
5. Start the newly imported Virtual Machine. Once booted, you can log in with:
   - Username: `idris-playground`
   - Password: `idris-playground`
6. Check that Idris2 is installed and executable by entering `idris2` at the shell prompt.
   - You can now try evaluating some Idris expressions (e.g. arithmetic expressions, or functions from the standard prelude) or use `:q` to quit.

The VM installation also includes basic utilities, including text editors `vim`, `mg` (a lightweight emacs-compatible editor). If required, you can discover further utilities using the command `apk search [name]` and install packages with `sudo apk add [packagename]`.

© Edwin Brady;
licensed under Creative Commons License CC-BY 4.0
Dagstuhl Artifacts Series, Vol. 7, Issue 2, Artifact No. 10, pp. 10:1–10:7
Dagstuhl Artifacts Series
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1.1 Manual Installation of Idris

Alternatively, you can install Idris 2 on your own machine, provided you have either of the following:

- Chez Scheme (`https://cisco.github.io/ChezScheme/`), version 9.5 or later, *or*
- Racket (`https://racket-lang.org/`), version 7 or later

You can find full instructions on how to install Idris here: `https://github.com/idris-lang/Idris2/blob/master/INSTALL.md`. Briefly, unpack the artifact, navigate to the `Idris2` directory, then, if you have Chez Scheme, enter:

- `make bootstrap SCHEME=scheme` (where `scheme` is the command used to start up Chez Scheme. On some systems this may be `chez`, `chezscheme`, or `chezscheme9.5`)

Or, if you have Racket, enter:

- `make bootstrap-racket`

Then, enter:

- `make install .`

The default installation prefix is `$HOME/.idris2`, so you will need to add `$HOME/.idris2/bin` to your `PATH`.

In each case, this will build an initial Idris 2 executable from either Chez Scheme or Racket source (generated by an existing Idris 2 compiler), then use that newly built executable to build Idris 2 and its libraries from scratch. You can check whether this succeeded by entering:

- `make test .`

1.2 Using the Artifact

The subdirectory `ECOOP21-IdrisQTT` includes the examples from the paper, divided into subdirectories for each section. To load an example into Idris2, navigate into one of the subdirectories, and load the file with the command:

- `idris2 File.idr -p contrib`

Where `File.idr` is the file name of the example, and `-p contrib` is a flag to tell Idris to look for imports in the `contrib` package, which is an additional package beyond the standard library consisting of user contributions. Some (but not all) of the examples require modules from `contrib`.

Note that Idris 2 does not currently use `readline` or support command history. You can, however, run it via `rlwrap`, which is installed on the virtual machine:

- `rlwrap idris2 File.idr -p contrib`

Once loaded, you can try evaluating the example code by entering expressions at the REPL, or start up an editor to edit the loaded file using the command `:e`. The default editor is `vim`, but you can change this either by setting the `EDITOR` environment variable, or entering `:set editor [editorcommand]` at the REPL prompt.

In the rest of this section, I briefly describe the examples and suggest some things to try. For a full tutorial on Idris 2, see `https://idris2.readthedocs.io/en/latest/tutorial/index.html`.

These examples illustrate the contributions of the paper:

- There is a full, self-hosted, implementation of Idris 2
- The examples illustrate quantities for erasure (run-length encoding) and linearity (resource usage for an ATM protocol)
- There is an executable implementation of bidirectional session types

1.2.1 Idris 2 source

The Idris 2 source is available in the `idris-playground` home directory, under `Idris2-0.3.0` (already built) and `Idris2-head` (directly pulled from github on 16th April 2021, and ready to build). You can try building Idris 2 from this source following the instructions below. To install using Chez Scheme, then test the result, type:

- `make bootstrap SCHEME=chez`
- `make test`

Note, however, that the tests run in parallel and you may need to increase the memory of the VM!

1.2.2 Overview of Idris (`Section2`)

This section introduces Idris, with some basic examples, in the file `Section2/Examples2.idr`. Navigate to the `Section2` subdirectory, then load `Examples2.idr`:

```
$ cd ECOOP21-IdrisQTT
$ idris2 Examples2.idr -p contrib


     ____    __     _         ___
    /  _/___/ /____(_)____   |__ \
    / // __  / ___/ / ___/   __/ /     Version 0.3.0-7714bdf3f
  _/ // /_/ / /  / (__  )   / __/      https://www.idris-lang.org
 /___/\__,_/_/  /_/____/   /____/      Type :? for help

Welcome to Idris 2.  Enjoy yourself!
1/1: Building Examples2 (Examples2.idr)
Main>
```

Note that you *must* load this from the correct subdirectory, due to the way Idris handles module paths.

You can inspect the types of the defined names, or any subexpression:

```
Main> :t append
Main.append : List a -> List a -> List a

Main> :t printf (Num (Lit " " (Str End)))
printf (Num (Lit " " (Str End))) : Int -> String -> String
```

You can evaluate expressions:

```
Main> append ['a','b'] ['c']
['a', 'b', 'c']
```

```
Main> printf (Num (Lit " " (Str End))) 99 "Red Balloons"
"99 Red Balloons"
```

You can also inspect the types of *holes*. To try this, edit the file and replace the definition of `printfFmt` starting on line 65 with the (commented out) definition above, quit the editor and return to the REPL. Then try:

```
Main> :t printfFmt_rhs_1
   x : Format
   acc : String
------------------------------
printfFmt_rhs_1 : Int -> PrintfType x
```

1.2.3 Quantities in Types (`Section3`)

This section describes the new features in Idris 2 arising from the core language, QTT (Quantitative Type Theory). Each source file corresponds to examples in the paper. These are:

- `Syntax.idr`, a small example showing the syntax of quantities.
- `RLE.idr`, the run-length encoding example showing how quantities support explicit erasure at the type level
- `Linearity.idr`, a small example showing a linear quantity

The example in Section 3.3.2 of the paper, showing how Idris implements I/O via linearity, can be seen as part of the Idris 2 libraries. This is in the Idris 2 source tree, under `libs/prelude/PrimIO.idr`.

You can try the type-directed program synthesis for `uncompress`, mentioned on page 11 of the paper, by deleting the definition on lines 21–24 (but keeping the type of `uncompress`), reloading `RLE.idr` at the REPL, then typing at the REPL:

```
RLE> :gd 20 uncompress

uncompress Empty = Val []
uncompress (Run n x y) = let Val ys = uncompress y in Val (x :: (rep n x ++ ys))
```

You can also try generating a definition for the initial type `uncompress : RunLength ty xs -> List ty` (given on page 10 of the paper). To do so, change the type, then try the above command at the REPL again:

```
RLE> :gd 20 uncompress

uncompress Empty = []
uncompress (Run n x y) = x :: uncompress y
```

Program synthesis recognises that it can't use the list `xs` directly, but the type isn't precise enough here for it to know that it needs to produce `n` copies of `x`.

Note that program synthesis, and interactive program editing in general, is designed to be used via a text editor, but as this is not within the scope of the paper, I have not included it as part of the VM in order to reduce the size of the image.

1.2.4 Linear Resource Usage Protocols (Section4)

This section describes a resource usage protocol and shows how to implement it using quantities to ensure that the protocol is followed accurately. There are two files:

- `ATM.idr` which includes just the types as described in the paper
- `ATMsim.idr`, a slightly extended version which can be compiled and executed, simulating the ATM's behaviour via console output.

To compile and execute this example, load `ATMsim.idr` then use the `:exec` command at the Idris REPL:

```
Main> :exec main

PIN OK
Dispensing cash
Card ejected
ATM shut down
```

It can be instructive to replace sub-programs with holes, and inspect them at the REPL. For example, create a new definition `runATMpart`:

```
runATMpart : L IO ()
runATMpart = do m <- initATM
                m <- insertCard m
                ?whatNow
```

Then, inspecting `whatNow` tells us what state the ATM is in at this point in execution:

```
Main> :t whatNow
 1 m : ATM CardInserted
------------------------------
whatNow : L IO ()
```

1.2.5 Session Types via QTT (Section5)

This section describes a larger example, implementing concurrent protocols with session types. There are four files:

- `Channel.idr`, the implementation of channels
- `TestProto.idr`, a small test protocol (not described in the paper)
- `UtilServer.idr`, the utility server described in the paper
- `DepSession.idr`, an example of a dependent session type from the paper

For each of `TestProto` and `UtilServer`, you can load the file into Idris and execute it with `:exec main`. In each case, it starts up the server, and a client sends a request to the server and receives and prints a response.

As with the previous example, you can insert holes to see how the types change during the protocol run. For example, in `utilServer`:

```
utilServer : (1 chan : Server Utils) -> L IO ()
utilServer chan
    = do cmd # chan <- recv chan
         case cmd of
              Add => do (x, y) # chan <- recv chan
                        ?whatNextAdd
              Reverse => do str # chan <- recv chan
                            ?whatNextRev
```

Inspecting `whatNextAdd` and `whatNextRev` shows what is needed on `chan` to complete the protocol run:

```
Main> :t whatNextAdd
   cmd : Command
   x : Int
   y : Int
 1 chan : Channel (Send Int (\res => Close))
------------------------------
whatNextAdd : L IO ()
```

At this point, the type of `chan` tells us that we need to send an `Int` on the channel, then close it.

2 Content

The artifact can be downloaded from `https://www.type-driven.org.uk/edwinb/idris-playground.tgz`. This archive consists of:

- The source code for Idris 2, in a subdirectory `Idris2`
- The examples from the paper, `idris-qtt.code.tgz`
- A virtual machine, `VM/idris-v0.3.0-playground.ova` with both of the above installed in the home directory of the user `idris-playground`

3 Getting the artifact

The artifact endorsed by the Artifact Evaluation Committee is available free of charge on the Dagstuhl Research Online Publication Server (DROPS). In addition, the artifact is also available at: `https://www.type-driven.org.uk/edwinb/idris-playground.tgz`.

4 Tested platforms

The Virtual Machine has been tested using VirtualBox 6.1, running Ubuntu Linux and MacOS. Idris 2 works on the following platforms, with 4GB RAM:

- Linux
- Mac
- Windows
- RaspberryPi

5 License

The artifact is available under the 3-Clause BSD licence (`https://opensource.org/licenses/BSD-3-Clause`)

6 MD5 sum of the artifact

d42094219eb7956180225142cf9b728b

7 Size of the artifact

363419183 bytes (0.338 GiB)