How to Take the Inverse of a Type (Artifact)

Daniel Marshall
School of Computing, University of Kent, Canterbury, UK

Dominic Orchard
School of Computing, University of Kent, Canterbury, UK
Department of Computer Science and Technology, University of Cambridge, UK

Abstract

In functional programming, regular types are a subset of algebraic data types formed from products and sums with their respective units. One can view regular types as forming a commutative semiring but where the usual axioms are isomorphisms rather than equalities. In the pearl, we show that regular types in a linear setting permit a useful notion of multiplicative inverse, allowing us to 'divide' one type by another. We begin with an exploration of the properties and applications of this construction, and this artifact includes examples in Haskell demonstrating its relevance to various topics from the literature including program calculation, Laurent polynomials, and derivatives of data types. Some examples from the paper require a richer setting than Haskell can offer; the artifact replays the first set of examples in Granule, while also presenting additional examples demonstrating further algebraic structure through linear functions that incorporate local side effects.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases linear types, regular types, algebra of programming, derivatives

Digital Object Identifier 10.4230/DARTS.8.2.1

Funding This work is supported by an EPSRC Doctoral Training Award (Marshall) and EPSRC grant EP/T013516/1 (Verifying Resource-like Data Use in Programs via Types).

Related Article Daniel Marshall and Dominic Orchard, "How to Take the Inverse of a Type", in 36th European Conference on Object-Oriented Programming (ECOOP 2022), LIPIcs, Vol. 222, pp. 5:1–5:27, 2022.
https://doi.org/10.4230/LIPIcs.ECOOP.2022.5

Related Conference 36th European Conference on Object-Oriented Programming (ECOOP 2022), June 6–10, 2022, Berlin, Germany

Evaluation Policy The artifact has been evaluated as described in the ECOOP 2022 Call for Artifacts and the ACM Artifact Review and Badging Policy.

1 Scope

The artifact provides relevant samples of code for each section in the paper, bundled with a Docker image. Much of the code in the paper is given using Haskell's linear types extension, and we provide this but also the equivalent code in Granule where possible to demonstrate the translation. Some code (in particular, one direction of the involution from the "Communicating with Inverses" chapter) is only possible to represent in Granule, so the contents of the two files do not match exactly. There are some extra pieces of code not present in the paper, provided for interest and further experimentation.

1.1 Getting started

Load the file provided via:

```
docker load < ecoop22-paper13-artifact.tar.gz
```

This may take some time as Docker will unpack the zip file before loading the image. The Docker image can now be started as with any other Docker image; for example, the below command will start up an interactive terminal session.

```
docker run -it ecoop22-paper13-artifact bin/bash
```

This image comes with GHC 9.2.1 and Granule 0.8.2.0 pre-installed, so no additional software needs to be downloaded or installed. The source code files for the paper are in the `/code` directory, named `pearl.hs` and `pearl.gr` for the Haskell code and Granule code respectively.

These files can be compiled via `ghc pearl.hs` and `gr pearl.gr` to verify that they typecheck and behave correctly. The reviewer may feel free to modify these files as they wish to experiment with the various ideas from the paper as they follow along; the text editor `nano` is installed, or if another option would be preferred it can be installed using `apt` or otherwise. In order to experiment with the Haskell code more interactively, `ghci` will load GHC's interactive environment, and `:l pearl.hs` will load in the file in the usual way. The reviewer can then, for example, inspect the types of functions via `:t`. For experimentation with the Granule one can use `grepl`, which is an interactive Granule environment that behaves in much the same way (`:l`, `:r` and `:t` have the same meanings as in `ghci`).

Example usages of `ghci` and `grepl` are given below, for reference.

```
# ghci
GHCi, version 9.2.1: https://www.haskell.org/ghc/  :? for help
:ghci> :l pearl.hs
[1 of 1] Compiling Main             ( pearl.hs, interpreted )
Ok, one module loaded.
ghci> :t divide
divide :: a %1 -> Inverse a %1 -> ()
ghci> divide True boolDrop
()
ghci> :q
Leaving GHCi.
```

```
# grepl
Welcome to Granule interactive mode (grepl). Version 0.8.2.0
Granule> :l pearl.gr
/code/pearl.gr, checked.
Granule> :t divide
  divide : forall {a : Type} . a -> (a -> ()) -> ()
Granule> divide True boolDrop
()
Granule> :q
Leaving Granule interactive.
```

2 Content

The artifact package comprises a Docker image built upon a minimal installation of Debian, with GHC 9.2.1 and Granule 0.8.2.0 preinstalled. As this paper is a pearl, the primary purpose of the artifact is to allow for being able to play around with the code samples in the paper in a practical setting, to aid understanding. There are no experimental results to replicate; the focus is on providing an additional demonstration of the ideas.

There are code samples in most sections of the paper, and as stated above we provide the relevant code from each section. The code files themselves (found in the `/code` folder, and named `pearl.hs` and `pearl.gr`) are divided into sections by comments with the chapter titles from the pearl, for ease of reference.

The code can easily be modified to experiment further with the various types, functions and ideas from the paper. We suggest various possible avenues for further exploration in the later sections of the pearl, and some examples that did not make it into the body of the text are provided along with the artifact. We invite additional contributions from interested readers!

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 on Zenodo at: `https://doi.org/10.5281/zenodo.6275280`.

4 Tested platforms

There are no special hardware or software requirements for the artifact beyond the standard system requirements for Docker and GHC. Around 5GB of free space will be needed to store the uncompressed Docker image.

The artifact itself has been tested on a MacBook running macOS 12.1, with 16GB of RAM, but since the artifact runs through Docker it should work fine on a much wider variety of systems.

If the user wishes to experiment with the code on their own system, rather than through the Docker image, then this should also be possible. GHC can be installed through any of the standard means (though note that the version will need to be at least 9.0.1, as the linear types extension is required for all of our examples). Granule binaries or instructions for building the Granule compiler from source can be found at `https://github.com/granule-project/granule`.

5 License

The artifact is available under the BSD 3-Clause license (`https://opensource.org/licenses/BSD-3-Clause`), since this is the license for the Granule compiler which needs to be distributed with the artifact image.

6 MD5 sum of the artifact

b187f849e170e11fee955643a722add1

7 Size of the artifact

1.28 GiB