PAT Final Assignment: Automatic Function Inversion in Haskell

Klaus Philipp Theyssen - lwg303

NOTE: this is an unfinished version of the report, I will hopefully be able to upload a finished version later today (hint, hint..)

Abstract

Function inversion is a important concept and can be used to safe time and avoid bugs. Applications are for example compression/decompression and encryption/decryption algorithms. Using a reversible language like Janus, the programmer gets inverse functions for "free". In this work I discuss a specific method for automatic function inversion in Haskell implemented via a GHC plugin. Haskell has a larger user base than Janus and is also used in industry. Therefore, exploring automatic function inversion in Haskell gives insight into how automatic inversion can be applied outside research. First I explain in broad terms the approach used by the GHC plugin to generate inverses. Finally, I present results from implementing various algorithms, discussing limitations and benefits of the plugin.

1 Introduction

1.1 Motivation

As we have seen in the course function inversion is powerful and helps during software development. Saving time by avoiding having two functions implementation and preventing bugs for example in compression software. In the lectures we have seen function inversion in the setting of Janus, which is a research project focused on reversible computation. Trying to bring some of the benefits of function inversion to a mature system like Haskell is an worthwhile endeavor. As Haskell has seen increasing adoption in industry and is used at large companies like Facebook (Marlow et al. [2013]).

1.2 Goals of Work

The goal of this work is two-fold, first I want to make the work from Teegen et al. [2021] more accessible, as it is mostly focused on the Haskell part and less on the automatic function inversion part. By focusing on explaining the underlying algorithms, idea for generating the inverse function.

Secondly I want to give a outlook of the use-ability and benefits one might get from using the plugin and also present it shortcomings.

1.3 Summary of contributions

The contribution of this work is giving a high-level summary of the approach for automatic function inversion presented in Teegen et al. [2021], skipping the technical details related to Haskell.

Further I implemented various modules in Haskell using the plugin, measuring the performance of the generated inverses and also giving a short account of my experience using the plugin.

The remainder of this report is organized as follows:

- **Section 2:** In this section I talk about inversion in the setting of functional languages in general. For this I review and discuss various approaches from the literature.
- **Section 3:** Here I present the fundamental idea of the inversion framework using a functional-logic extension of Haskell.
- **Section 4:** Evaluating the GHC Plugin, its usability and performance, by implementing various algorithms and small programs.
- **Section 5:** In this part I give a short conclusion and discuss various other ideas to explore, but also limitations of my own work.

2 Inversion in Functional Languages

In this section I want to review of the literature on inverting functional languages. In the PAT course we have seen the imperative reversible language Janus, now I want to discuss what happens if we consider a functional language like Haskell.

The Universal Resolving Algorithm based on perfect process tree, another general approach for automatic inversion, as seen in the lecture, algorithm for inverse interpreter in Abramov and Glück [2000], which we have also shortly seen in the lecture.

The URA has even been extended to lazy functional languages in Abramov et al. [2007], which is interesting since Haskell is a lazy functional language.

Another approach making only use of equality and constructors is presented in Glück and Kawabe [2003]

A different kind of approach is to design a reversible functional language, similar to how Janus is a reversible imperative language in Yokoyama et al. [2012].

If we limit our self to only dealing injective functions one can even use techniques from LR parsing to generate inverses as presented in Glück and Kawabe [2004].

In Teegen et al. [2021] the authors do not limit themselves to injective functions, which would be impractical to incorporate while using a general purpose language like Haskell. Instead they employ techniques found in logic programming Languages like Prolog, to provide a non-deterministic search for inverses. Therefore, the generated inverse function return a list of possible results.

This idea is closely connected to functional-logic programming languages like Curry (see Hanus [2013]).

3 Approach to Automatic Function Inversion in Haskell

The core concept of the plugin is utilizing free-variables and non-determinism in order to generate inverses. This is possible by extending Haskell computational model to a functional-logic computational model using Monads. All functions, constructs and expressions of a Haskell module are lifted into the Monad.

Thus, the underlying inverse computation is similar to that of logic programming languages like Prolog, where one can call predicates with free variables as arguments to find variable bindings such that the predicate is satisfied.

```
prolog> append(Xs, Ys, [17,42]).
Xs = [], Ys = [17, 42] ;
Xs = [17], Ys = [42] ;
Xs = [17, 42], Ys = [] ;
false.
```

3.1 Functional-logic extension of Haskell

The functional-logic extension is realized using free variables and non-determinism.

The free variables are implemented implemented with identifiers and a heap, which keeping track of already initiated free variables and computational branches.

In addition, non-determinism is modeled via multiple computation branches using a tree. The tree and its different computational branches are explored in a breadth-first manner, in order to avoid getting stuck in an infinite computation of a single computation branch.

Since general purpose functions one can write in Haskell are non-injective, the inverses return a list of possible results.

Further, this approach makes it possible to also implemented partial inverses, since we can simply fix the additional arguments instead of using free variables.

The whole framework for automatic inversion is described in figure 1.

3.2 Example: Inverse Generation of List Concatenation

To illustrate the approach of the plugin I want to give a high level overview how inverse of the list concatenation function (++) is generated.

```
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x : xs ++ ys
```

With the plugin we can define the reverse function, which computes all possible splits for a given list in the following way:

```
split :: [Int] -> [([Int], [Int])]
split = $(inv '(++))
```

1. Lift function (type, definition + expressions) into functional-logic computational model (Monad):
    - simplify pattern matching into unary lambda abstraction
    - non-exhaustive pattern augmented with pattern match failure
    - lifting expressions (technical details skipped over)
2. Run lifted function, with free variables as arguments, Heap keeps track of instantiated free variables, non-determinism modeled via tree
3. Start functional-logic computation: monad starter that traverses search tree of all possible computation branches in breadth-first manner
4. pattern matching should leads to instantiation of free variables (narrowing), narrow will return list of constructors of datatype and thereby spawning new computation branches
5. use ground normal form to ensure result does not contain free variables
6. match results with argument for inverse function, every matching computation branch contains heap with free variables which we will return, the other ones will fail fast (Haskell laziness)

Figure 1: Steps of Inversion Framework

In a **ghci** session we can do the following:

```
*Main> split [17,42]
[([],[17,42]),([17],[42]),([17,42],[])]
```

Now I want to describe the what happens internally in the framework:

1. call $(inv '(++)) [17, 42]
2. take the lifted version of (++) the arguments are free-variables: V0, V1
3. run it in tree (multiple computation branches), in each time we pattern match on a free variable, instantiate it using narrow, thereby creating more computation branches, if primitive types are involved add a constraint
4. call ground on it (instantiate all free variables, "normal form")
5. check for computation branches which resulted in [17,42], if a computation branch matches, return the bindings of the free variables V0, V1 from the heap of the computation branch

3.3 Primitive types

Dealing with primitive types like Int, is not easy in this setting. Since the implementation uses narrowing, by pattern matching on the constructor of types when initiating free variables we would have to try all possible values of an integer. To circumvent this the plugin instead introduces constraints each time a pattern match on a primitive type is made, and each time a constraint is added it checks if it conflicts with the previous ones, if yes the computation branch is terminated. This is implemented using an SMT solver like Z3 from de Moura and Bjørner [2008].

3.4 Partial inverses

The plugin easily allows computing partial inverse. It is done similarly to regular inverse, but now the fixed argument is also provided to the lifted function.

As an example consider the following insertAt function:

```
insertAt :: Int -> Int -> [Int] -> [Int]
insertAt 0 x xs = (x:xs)
insertAt i x l@(y:ys) =
  case i < 0 of
    True -> error "invalid index"
    False ->
      case i > (length l) of
        True -> error "invalid index"
        False -> y : (insertAt (i-1) x ys)
```

Which takes a index, an integer and a list of integers and inserts the integer into the list at the specified index.

If we now fix its first argument (the index) and compute its partial inverse, we get a function which given a index and a list returns the element at that index and the remaining list after removing the element.

```
removeAt1 :: Int -> [Int] -> [(Int, [Int])]
removeAt1 = $(partialInv 'insertAt [1])
```

4 Evaluating the GHC Plugin

For my evaluation I decided to implement various examples.

First one has to note that Haskell has a somewhat interesting relation with language extensions and plugins compared to other programming languages. Due to the high use in academics and innovating new Programming language ideas, techniques there is a plethora of concepts provided by various extension and plugins, which are basically adding additional steps in the GHC (Haskell compiler), where one can perform additional transformation the the core language (internal intermediate language of the Haskell compiler).

Therefore the plugin is for example tied to a specific GHC version, since the interface is subject to regular change by the compiler developers, also since it makes heavy use of specific implementation details.

Therefore it seems a little bit unrealistic to use this plugin in a real project in the wild, but it is still a prototype, but this is also made clear by the authors in Teegen et al. [2021].

I will discuss the examples, and then some benchmarks.

4.1 Experiments

4.1.1 Free Fall

I started with the simple free fall simulation we have seen in the course, this involves some basic arithmetic.

4.1.2 Run Length encoding

- bad run time noticeable First I want to provide a compression, decompression algorithm where I want to compare writing both compression and decompression function by hand vs. generating the decompression function with the plugin, and I want to compare speed for theses two (maybe other metrics important here?)

4.1.3 TEA Tiny encryption algorithm

TEA tiny encryption algorithm

-> after doing some research on bit manipulations in Haskell, it seems very natural to use the packages Data.Bits and Data.Word to implement the tiny encryption algorithms. Unfortunately external libraries are not supported in the current Plugin version, even Prelude (standard library) of haskell is causing trouble

-> From this we see that, the plugin is not very well suited for implementing real world applications of inverse computation.

- still the plugin seems quiet powerful, and of course the task/problem of changing the entire computational model of a programming language and especially the wide range of complicated abstractions and features in haskell seem to make it very difficult to provide a stable and complete implementation of such and feature using a plugin.

- I think it is also remarkable that this is even possible, having the GHC provide interfaces for plugin developer to extend the computational model of the language in a meaningful way, what other languages provide this?

4.2 Benchmarks

- explain hardware setup (laptop hardware spec)

- run length decoder probably so slow because using Integer (primitive type) constraints? –> could try to use peano numbers (Datatype with constructor?)

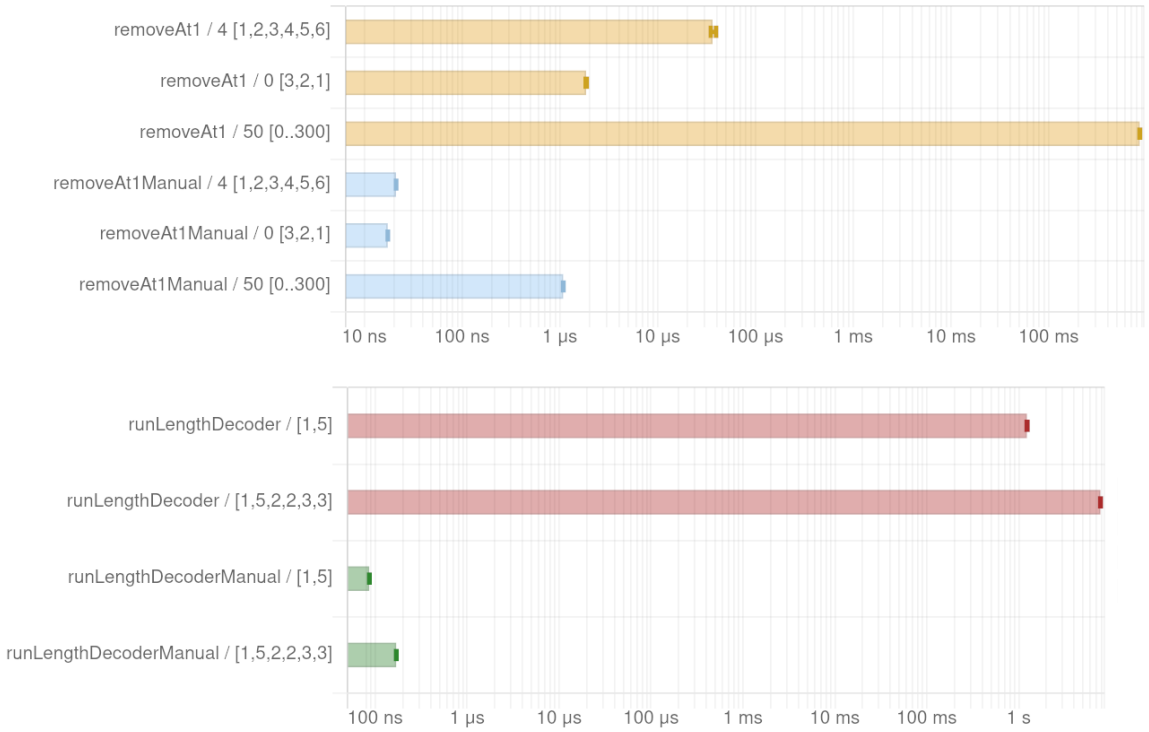

Figure 2: Top: Results from comparing automatic generated removeAt1 (partial inverse) function with manual written inverse. Bottom: Results from comparing automatic generated run length decoder vs. manual written one.

- the results show that a real use case is not viable, its just to slow, although this maybe actually from my implementation, especially the runlengthencoder, but the removeAt1 function on the other hand seems very straightforward to me, and there is no real reason why it is slow

- I think the plugin is not feasible for any real project, but also the authors already only said that a use case like automatic test case generation seems viable, so its the plugin is less about bringing inversion for free to haskell programmers, but rather demonstrating the strength and flexibility in implementing and extending Haskell via a GHC plugin, this has been done before in various forms. Like liquid haskell, dependent types in haskell.

- run handwritten inverses vs. automatic inverses

4.3 Limitations, Problems encountered

- inverses for non-injective functions -> may not always terminate in search for inputs, because there may be infinitely many, not really limitation but inherent property of the functional-logic based approach

- no I/O operations, and problems with external libraries / prelude

- type synonym does not work with plugin, when implementing the free fall function would come in handy to use type synonym for integers

- example while writing first version runLengthencoding, I learned that the plugin cannot generate a inverse for dropWhile

Most of the definitions from the Prelude are not supported at the moment. For now, the best idea is to not use any imported definitions from the Prelude or other modules. We will lift this restriction in the future!!! -> pretty big limit, as I do not have access to head, drop, dropWhile, splitAt etc.

- run time, for run length encoding it seems takes way too long, this of course can be due to my inefficient implementation, but on the other hand since I cannot use prelude functions making it efficient is also kind of a large endeavor.

- some obscure GHC error messages, not very stable. To be expected from such a research project

$ is just a syntactic construct in haskell for setting parentheses, see section 5.1.

While trying to use implement transpose function on lists I got a GHC error see section 5.1.

5 Conclusion

- property based testing with automatic inverse? -> have run length encoding and quickcheck

- Several observations are interesting: - logic programming and its ability to easily generate inverses - power of GHC plugins, how easily one can extend Haskell in various ways - ease of use of the plugin, some drawbacks (no prelude etc.) - research prototype, but seems possible to bring benefit of automatic inversion to real broader industry developers -> industrial use of Haskell is big and growing (even meta/facebook) etc.

5.1 Further Work

- in this work it was nice to use the plugin and have some working experiments and read about it, but this work definitely lacks some theoretical depth, first in explaining exactly all the subtleties of the plugin (Haskell related, type theoretic, monads etc.) but also the review of other approaches in the literature for inversion in a functional language is rather shallow, it would be nice to take more time and go into more detail, but due to time constraints this work remained on the surface

- my experiments, examples all involve integers (primitive) types, which works for the framework, but might be a big bottleneck, I would have liked to do some more interesting examples with custom datatype, where the instatiation of free variables should be cheaper. -> maybe run a lot quicker?

- the plugin has additional features not discussed here: functional patterns, higher order function inverses?

-> use quickcheck, generator to generate lists of integers and check that x === runLength-EncoderInv (runLengthEncoder x)

- discussing the implementation, plugin internals in greater detail

- doing further benchmarks more experiments, comparing with handwritten inverses

- limitations of my own work:

- theory is to thin, better presentation, since I did such high level presentation, I definitely have got somethings inaccurate

- benchmark could be more comprehensive, more realistic, different functions

References

S. Abramov and R. Glück. The universal resolving algorithm: Inverse computation in a functional language. *Mathematics of Program Construction*, page 187–212, 2000. ISSN 1611-3349. doi: 10.1007/10722010_13.

S. Abramov, R. Glück, and Y. Klimov. An universal resolving algorithm for inverse computation of lazy languages. *Lecture Notes in Computer Science*, page 27–40, 2007. doi: 10.1007/978-3-540-70881-0_6.

L. de Moura and N. Bjørner. Z3: An efficient smt solver. *Lecture Notes in Computer Science*, page 337–340, 2008. ISSN 1611-3349. doi: 10.1007/978-3-540-78800-3_24.

R. Glück and M. Kawabe. A program inverter for a functional language with equality and constructors. *Lecture Notes in Computer Science*, page 246–264, 2003. ISSN 1611-3349. doi: 10.1007/978-3-540-40018-9_17.

R. Glück and M. Kawabe. Derivation of deterministic inverse programs based on lr parsing. *Lecture Notes in Computer Science*, page 291–306, 2004. ISSN 1611-3349. doi: 10.1007/978-3-540-24754-8_21.

M. Hanus. Functional logic programming: From theory to curry. *Lecture Notes in Computer Science*, page 123–168, 2013. ISSN 1611-3349. doi: 10.1007/978-3-642-37651-1_6.

S. Marlow, J. Coens, L. Brandy, and J. Purdy. The haxl project at facebook. In *Proceedings of the Code Mesh London*, 2013.

F. Teegen, K.-O. Prott, and N. Bunkenburg. Haskell-1: automatic function inversion in haskell. *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell*, Aug 2021. doi: 10.1145/3471874.3472982.

T. Yokoyama, H. B. Axelsen, and R. Glück. Towards a reversible functional language. *Lecture Notes in Computer Science*, page 14–29, 2012. ISSN 1611-3349. doi: 10.1007/978-3-642-29517-1_2.

Appendix

Compression.hs

```haskell
{-# OPTIONS_GHC -fplugin Plugin.InversionPlugin #-}
{-# OPTIONS_GHC -Wno-incomplete-patterns #-}
{-# OPTIONS_GHC -Wno-overlapping-patterns #-}
module Compression where

import Prelude hiding ((++), map, takeWhile, dropWhile, length, head)

--------------------------------------------------------------------------------
-- Version without prelude functions:
--------------------------------------------------------------------------------
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs

head :: [a] -> a
head (x:_) = x
head [] = error "bad head"

length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + (length xs)

takeWhile               :: (a -> Bool) -> [a] -> [a]
takeWhile _ []          =  []
takeWhile p (x:xs) = case p x of
  True -> x : takeWhile p xs
  False -> []

dropWhile               :: (a -> Bool) -> [a] -> [a]
dropWhile _ []          =  []
dropWhile p xs@(x:xs') = case p x of
  True -> dropWhile p xs'
  False -> xs

splitList :: [Int] -> [[Int]]
splitList [] = []
splitList [x] = [[x]]
splitList l@(x:_) = (takeWhile eqX l) : (splitList (dropWhile eqX l))
```

```haskell
  where
    eqX = (\y -> x == y)

mergeLists :: [Int] -> [Int] -> [Int]
mergeLists [] [] = []
mergeLists (x:xs) (y:ys) = [x,y] ++ (mergeLists xs ys)
mergeLists [] y = y
mergeLists x [] = x

runLengthEncoder :: [Int] -> [Int]
runLengthEncoder xs = mergeLists digits times
  where
    subLists = splitList xs
    digits = map head subLists
    times = map length subLists
```

Encryption.hs

```haskell
module Encryption where

import Prelude hiding ((++), map, takeWhile, dropWhile, length, head)

import Data.Bits
import Data.Word

--------------------------------------------------------------------------------
-- Implementing the tiny encryption algorithms by David Wheeler, Roger Needham
--------------------------------------------------------------------------------

-- 128 bit key
data TEAKey = TEAKey {-# UNPACK #-} !Word32 {-# UNPACK #-} !Word32 {-# UNPACK #-} !Word32 {-# UNPACK #-} !Word32
  deriving (Show)


secretKey :: TEAKey
secretKey = (TEAKey  0xdeadbeef 0xdeadbeef 0xdeadbeef 0xdeadbeef)

delta :: Word32
delta = 0x9e3779b9

myData :: Word64
myData = 0xbadf00000000000d
```

```haskell

rounds :: Int
rounds = 32

-- Plugin cannot handle Data.Bits, Data.Word :(
teaEncrypt :: TEAKey -> Word64 -> Word64
teaEncrypt (TEAKey k0 k1 k2 k3) v = doCycle rounds 0 v0 v1 where
  v0, v1 :: Word32
  v0 = fromIntegral v
  v1 = fromIntegral $ v `shiftR` 32
  doCycle :: Int -> Word32 -> Word32 -> Word32 -> Word64
  doCycle 0 _ v0 v1 = (fromIntegral v1 `shiftL` 32)
                      .|. (fromIntegral v0 .&. 0xffffffff)
  doCycle n sum v0 v1 = doCycle (n - 1) sum' v0' v1'
    where
      sum' = sum + delta
      v0' = v0 + (((v1 `shiftL` 4) + k0) `xor` (v1 + sum') `xor` ((v1
      ↪  `shiftR` 5) + k1))
      v1' = v1 + (((v0 `shiftL` 4) + k2) `xor` (v0 + sum') `xor` ((v0
      ↪  `shiftR` 5) + k3))
```

Simulation.hs

```haskell
{-# OPTIONS_GHC -fplugin Plugin.InversionPlugin #-}
{-# OPTIONS_GHC -Wno-incomplete-patterns #-}
{-# OPTIONS_GHC -Wno-overlapping-patterns #-}
module Simulation where

import Prelude hiding ((++), lookup, Maybe(..), length)

-- ----------------------------------------------------------------------------
-- Implementing a simple simulation of free falling objects
-- ----------------------------------------------------------------------------

data Height = Height Int
  deriving (Show)
data Velocity = Velocity Int
  deriving (Show)
data Time = Time Int
  deriving (Show)
data TimeEnd = TimeEnd Int
  deriving (Show)

freeFall :: (Height, Velocity, Time, TimeEnd) -> (Height, Velocity, Time,
 ↪  TimeEnd)
```

```haskell
freeFall current@((Height h), (Velocity v), (Time t), (TimeEnd tEnd)) =
  case t == tEnd of
    True -> current
    False -> freeFall ((Height h'), (Velocity v'), (Time t'), (TimeEnd
    ↪  tEnd))
      where
        v' = v + 10
        h' = h - v' + 5
        t' = t + 1
```

PartialInverses.hs

```haskell
{-# OPTIONS_GHC -fplugin Plugin.InversionPlugin #-}
{-# OPTIONS_GHC -Wno-incomplete-patterns #-}
{-# OPTIONS_GHC -Wno-overlapping-patterns #-}
module PartialInverses where

import Prelude hiding (length)

--------------------------------------------------------------------------------
-- Implement insertAt from which we will generate a partial inverse
-- removeAt. By fixing the index argument and giving a list, removeAt will
-- return the element at that index of the list and the remaining list
--------------------------------------------------------------------------------

length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + (length xs)

insertAt :: Int -> Int -> [Int] -> [Int]
insertAt 0 x xs = (x:xs)
insertAt i x l@(y:ys) =
  case i < 0 of
    True -> error "invalid index"
    False ->
      case i > (length l) of
        True -> error "invalid index"
        False -> y : (insertAt (i-1) x ys)
```

Main.hs

```haskell
{-# LANGUAGE TemplateHaskell, FlexibleContexts, ViewPatterns #-}
{-# OPTIONS_GHC -Wno-orphans #-}
module Main where

import Plugin.InversionPlugin

import Simulation
import Compression
import PartialInverses
import Criterion.Main
import Criterion.Types

import Prelude hiding (map, lookup, (++), last)

main :: IO ()
main = do
  putStrLn "Benchmarking automatically generated Inverses"
  defaultMainWith
    (defaultConfig {reportFile = Just "benchmarks.html",
                    csvFile = Just "benchmark-inverses.csv"}) $
    [bgroup "removeAt1" [ bench "4 [1,2,3,4,5,6]"
                          $ nf (\x -> take 1 (removeAt1 4 x))
                          ↪  [1,2,3,4,5,6],
                          bench "0 [3,2,1]"
                          $ nf (\x -> take 1 (removeAt1 0 x)) [3,2,1],
                          bench "50 [0..300]"
                          $ nf (\x -> take 1 (removeAt1 50 x)) [0..300]
                        ],
     bgroup "removeAt1Manual" [ bench "4 [1,2,3,4,5,6]"
                                $ nf (\x -> removeAt1Manual 4 x)
                                ↪  [1,2,3,4,5,6],
                                bench "0 [3,2,1]"
                                $ nf (\x -> removeAt1Manual 0 x) [3,2,1],
                                bench "50 [0..300]"
                                $ nf (\x -> removeAt1Manual 50 x) [0..300]
                        ],
     bgroup "runLengthDecoder" [ bench "[1,5]" $
                                 nf (\x -> take 1 (runLengthDecoder x))
                                 ↪  [1,5],
                                 bench "[1,5,2,2,3,3]" $
                                 nf (\x -> take 1 (runLengthDecoder x))
                                 ↪  [1,5,2,2,3,3]
                               ],
     bgroup "runLengthDecoderManual" [ bench "[1,5]" $
```

```
                                        nf (\x -> runLengthDecoderManual x)
                                        ↪  [1,5],
                                        bench "[1,5,2,2,3,3]" $
                                        nf (\x -> runLengthDecoderManual x)
                                        [1,5,2,2,3,3]
                                    ]
    ]


split :: [Int] -> [([Int], [Int])]
split = $(inv '(++))

-------------------------------------------------------------------------------
-- Simulations
-------------------------------------------------------------------------------

freeFallInv :: (Height, Velocity, Time, TimeEnd)
             -> [(Height, Velocity, Time, TimeEnd)]
freeFallInv = $(inv 'freeFall)

fallStart :: (Height, Velocity, Time, TimeEnd)
fallStart = ((Height 176), (Velocity 0), (Time 0), (TimeEnd 3))

fallDown :: (Height, Velocity, Time, TimeEnd)
fallDown = freeFall fallStart

fallUp :: [(Height, Velocity, Time, TimeEnd)]
fallUp = take 5 $ freeFallInv fallDown

-------------------------------------------------------------------------------
-- Compressions
-------------------------------------------------------------------------------

runLengthDecoder :: [Int] -> [[Int]]
runLengthDecoder = $(inv 'runLengthEncoder)

-- implemented for benchmarking manual vs automatic generated inverse
runLengthDecoderManual :: [Int] -> [Int]
runLengthDecoderManual [] = []
runLengthDecoderManual [x] = error "Invalid encoding"
runLengthDecoderManual (x:(y:ys)) = (take y $ repeat x) ++
↪  runLengthDecoderManual ys

dataToCompress :: [Int]
dataToCompress = [1,1,1,1,1,1,1,2,2,3,3,3,5]

```

```haskell
encoded :: [Int]
encoded =  runLengthEncoder dataToCompress

decoded :: [[Int]]
decoded = take 1 $ runLengthDecoder encoded

-------------------------------------------------------------------------------
-- Partial Inverses
-------------------------------------------------------------------------------

-- partial inverse of insertAt fixing the first argument (index)
removeAt1 :: Int -> [Int] -> [(Int, [Int])]
removeAt1 = $(partialInv 'insertAt [1])

-- implemented for benchmarking manual vs automatic generated inverse
removeAt1Manual :: Int -> [Int] -> (Int, [Int])
removeAt1Manual _ [] = error "empty list"
removeAt1Manual 0 (x:xs) = (x, xs)
removeAt1Manual i (x:xs) = removeAt1Manual (i-1) xs

removeAt1Example :: [(Int, [Int])]
removeAt1Example = (take 1 (removeAt1 3 [1,2,3,4,5]))
```

Error Messages

```
/home/pt/repos/inversion-plugin/firststeps/src/Compression.hs:1:1: error:
    No inverse available for: $
  |
1 | {-# OPTIONS_GHC -fplugin Plugin.InversionPlugin #-}
  | ^
Failed, no modules loaded.
```

```
 GHCi, version 8.10.4: https://www.haskell.org/ghc/  :? for help
[1 of 2] Compiling Compression      (
↪  /home/pt/repos/inversion-plugin/firststeps/src/Compression.hs,
↪  interpreted )
ghc: panic! (the 'impossible' happened)
  (GHC version 8.10.4:
        splitFunTy
  ListFL (ListFL a_ab0v)
  --> ((ListFL a_ab0v --> ListFL Any) --> ListFL Any)
  Call stack:
      CallStack (from HasCallStack):
        callStackDoc, called at compiler/utils/Outputable.hs:1179:37 in
        ↪  ghc:Outputable
        pprPanic, called at compiler/types/Type.hs:1018:32 in ghc:Type

Please report this as a GHC bug:  https://www.haskell.org/ghc/reportabug
```