

FLOWR

Layout Preserving Reconstruction

Pascal Deutsch, Florian Sihler

FlowR

flowR is a static dataflow analyzer and program slicer for the R programming language.

- a statistics module
- a benchmark module
- a core module
- and a slicer module.

The statistics module is used to analyze R files and identify common patterns. The benchmarker module serves as a way to benchmark flowR's performance. The core module contains flowR's main definitions and is the home of flowR's read-eval-print loop (REPL) and flowR's server. The slicer module contains flowR's backward program slicer for R. This module consists of four submodules with a custom R bridge, a way to normalize the AST representation of the data flow graph, a data flow analyzer for said data flow graph, and the reconstruct module. My project is focused on the reconstruction module. There, the work was mostly centered around the implementation of a layout-preserving reconstruction.

**Reconstruction** The reconstruction works in simple steps. The R parser parses the R code into an AST, which then gets normalized. This normalized AST is then traversed through a fold, where the lexemes get put together with location information and additional tokens to create code parts. The code parts will, in a final step, get sorted and put together to form the final output. Currently, the reconstruction can generate a layout-preserving reconstruction using the location information and additional tokens, however, not all additional tokens are currently in use. Mostly parentheses are being used to preserve the layout of loops. This, however, will be expanded in future updates to flowR, to iron out some quirks of the layout preserving reconstruction and possibly include the option for amorphous slicing.

Layout Preserving Reconstruction

```
sum <- 0
product <- 1
w <- 7
N <- 10

for (i in 1:(N-1)) {
  sum <- sum + i + w
  product <- product * i
}

cat("Sum:", sum, "\n")
cat("Product:", product, "\n")
```

(a) Original R source code

```
sum <- 0
product <- 1
w <- 7
N <- 10

for (i in 1:(N-1)) {
  sum <- sum + i + w
  product <- product * i
}

cat("Sum:", sum, "\n")
cat("Product:", product, "\n")
```

(b) Original R source code with sliced tokens highlighted

```
product <- 1
N <- 10
for (i in 1:(N-1)) {
  product <- product * i
}
cat("Product:", product, "\n")
```

(c) Reconstructed slice for product in line 12

The pictures on the left show an example of the reconstruction. The specific challenge, in this case, was the reconstruction of the for-loop. The parser can differentiate between the single elements, however the implementation of the reconstruction had to be reworked significantly to use them. The first step was to take the single elements the parser gave us and store them with their location data. Those elements (parts) would then be stored in a monad imitating structure. We had to include the additional tokens the parser returns as well, since the parentheses, brackets, or even the "in" would not be available otherwise. This means that the reconstruction of the for loop has to construct every part of it separately and feed it into our output. To accomplish this we needed to add a way to put together multiple parts in their correct order. This gets accomplished over a single function (merge), which first sorts all parts by line, and afterward puts them together with sorted lines. Printing the output gave us another challenge, as we had to work on each line individually. We needed to keep track of each line length to avoid unwanted spaces but also make sure that all parts would be traversed in order.

My Contributions

During my project, I refactored the structure of the preexisting reconstruction implementation. This was done to increase the readability of the module, as well as improve the quality for future improvements. The main part of the project

so far has, however, been spent on improving the reconstruction by implementing layout-preserving reconstruction. To achieve this, we introduced two new parts. First, we utilized the location information of the R parser to give the lexemes a

position within the final output. Using this, we then build up the code representation of the slice using the location information by first sorting all parts into their correct order and then appending all parts together. Secondly, we introduced and uti-

lized additional tokens, which include parentheses and other special characters. These additional tokens then get added into the part with their locations, to get sorted and added to the output.