flowR – A Dataflow Analysis Framework and Program Slicer for R

Florian Sihler, Oliver Gerstl, Julian Schubert, Matthias Tichy | useR! 2026 • Lightning Talk

Software Engineering
Programming Languages

universität ulm

We do **static analysis** for R.

Analyzing the code without executing it.

We do **static analysis** for R.

Analyzing the code without executing it.

Non Executable — research artifacts might be broken

We do **static analysis** for R.

Analyzing the code without executing it.

Non Executable — research artifacts might be broken

Security — no potential side-effects

We do **static analysis** for R.

Analyzing the code without executing it.

Non Executable — research artifacts might be broken

Security — no potential side-effects

IDE & Development — linting, auto-completion & more

We do **static analysis** for R.

Analyzing the code without executing it.

Non Executable — research artifacts might be broken

Security — no potential side-effects

IDE & Development — linting, auto-completion & more

I use **LLMs**?

Program Slicing

focus on a single visualization


```
1 sum  ← 0
2 prod ← 1
3 n    ← 10
4 for (i in 1:(n-1)) {
5   sum  ← sum + i
6   prod ← prod * i
7 }
8 cat("Sum:", sum, "\n")
9 cat("Product:", prod, "\n")
```

Program Slicing

focus on a single visualization


```
1 sum  ← 0
2 prod ← 1
3 n    ← 10
4 for (i in 1:(n-1)) {
5   sum  ← sum + i
6   prod ← prod * i
7 }
8 cat("Sum:", sum, "\n")
9 cat("Product:", prod, "\n")
```

Program Slicing

focus on a single visualization


```
1 sum  ← 0
2 prod ← 1
3 n    ← 10
4 for (i in 1:(n-1)) {
5   sum  ← sum + i
6   prod ← prod * i
7 }
8 cat("Sum:", sum, "\n")
9 cat("Product:", prod, "\n")
```

Program Slicing

focus on a single visualization


```
1 sum  ← 0
2 prod ← 1
3 n    ← 10
4 for (i in 1:(n-1)) {
5   sum  ← sum + i
6   prod ← prod * i
7 }
8 cat("Sum:", sum, "\n")
9 cat("Product:", prod, "\n")
```

slice(8@sum)

```
1 sum  ← 0
3 n    ← 10
4 for (i in 1:(n-1)) {
5   sum  ← sum + i
7 }
8 cat("Sum:", sum, "\n")
```

Program Slicing

focus on a single visualization


```
1 sum  ← 0
2 prod ← 1
3 n    ← 10
4 for (i in 1:(n-1)) {
5   sum  ← sum + i
6   prod ← prod * i
7 }
8 cat("Sum:", sum, "\n")
9 cat("Product:", prod, "\n")
```

slice(8@sum)

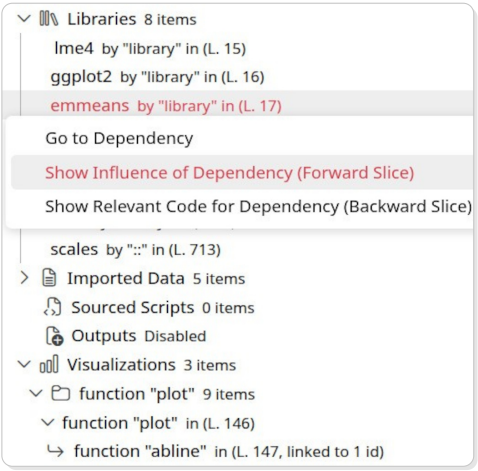
```
1 sum  ← 0
3 n    ← 10
4 for (i in 1:(n-1)) {
5   sum  ← sum + i
7 }
8 cat("Sum:", sum, "\n")
```

On real scripts, slices are 80 % to 90 % smaller.[1]

[1] Sihler et al., “Statically Analyzing the Dataflow of R Programs” (2025, OOPSLA)

Dependency View

Dependency View



Dependency view • wiki/Interface

Data-Frame Shapes

patients.csv

```
id,age,weight,height
1,34,70,1.75
2,51,82,1.80
3,29,60,1.65
4,42,95,1.90
```

```
1 df ← read.csv("patients.csv") |>
2   mutate(bmi = weight / height^2) |>
3   select(-c(weight, height))
```

Data-Frame Shapes

Data-Frame Shapes
hover tooltips in the editor

patients.csv

```
id,age,weight,height
1,34,70,1.75
2,51,82,1.80
3,29,60,1.65
4,42,95,1.90
```

```
1 df ← read.csv("patients.csv") |>
2   mutate(bmi = weight / height^2) |>
3   select(-c(weight, height))
```

```
data frame shape:
Rows:    4
Cols:    3
Columns: id, age, bmi
```

Linting

Linter
reproducibility & security

```
1  applyTwice ← function(f, x) f(f(x))
2  square     ← function(x) x * x
3  cube       ← function(x) x * x * x
4
5  p ← readline()
6  pdf(p)
7  print(applyTwice(square, 3))
```

Linting

Linter
reproducibility & security

unused-definitions
cube is never called

```
1  applyTwice ← function(f, x) f(f(x))
2  square     ← function(x) x * x
3  cube       ← function(x) x * x * x
4
5  p ← readline()
6  pdf(p)
7  print(applyTwice(square, 3))
```

wiki/Linter

Linting

Linter
reproducibility & security

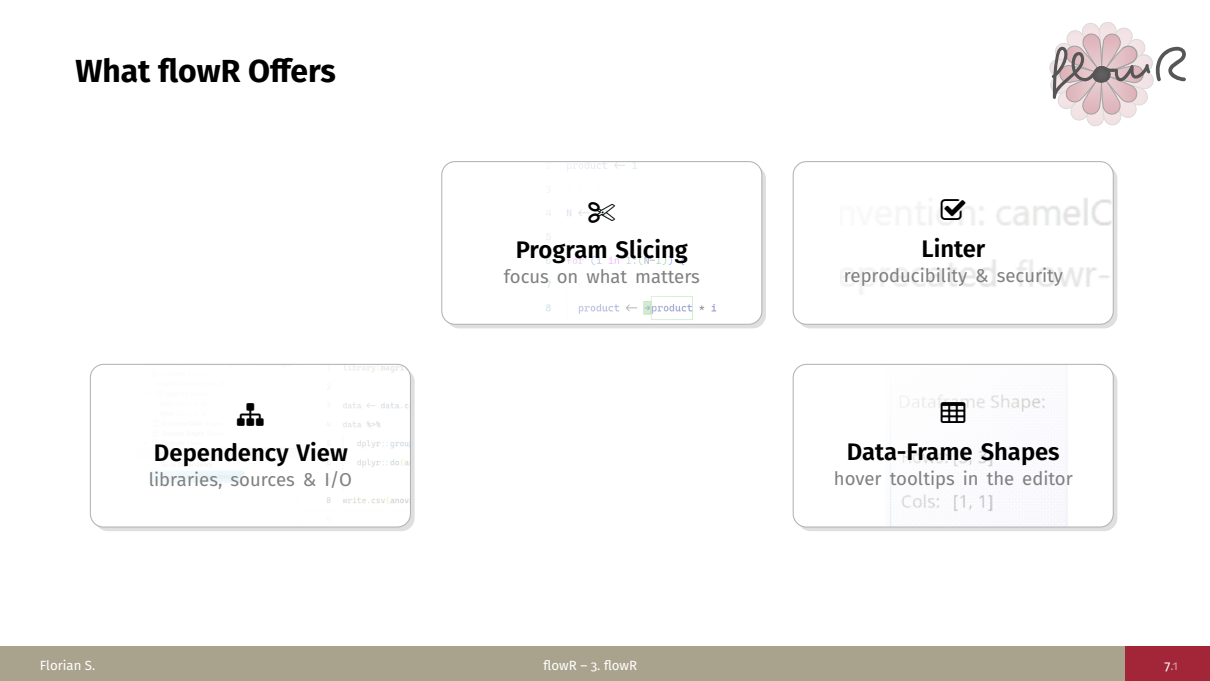
unused-definitions
cube is never called

```
1  applyTwice ← function(f, x) f(f(x))
2  square     ← function(x) x * x
3  cube       ← function(x) x * x * x
4
5  p ← readline()
6  pdf(p)
7  print(applyTwice(square, 3))
```

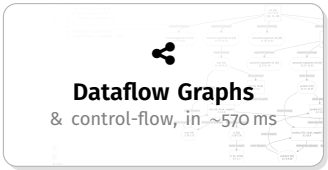
tainted-input
untrusted p reaches pdf()

wiki/Linter

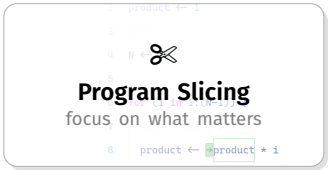
What flowR Offers



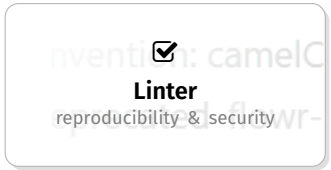
What flowR Offers



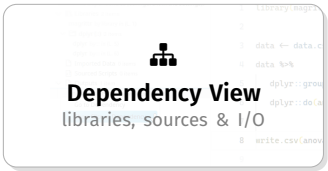
Dataflow Graphs
& control-flow, in ~570 ms



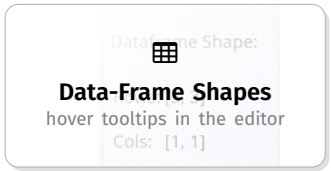
Program Slicing
focus on what matters



Linter
reproducibility & security



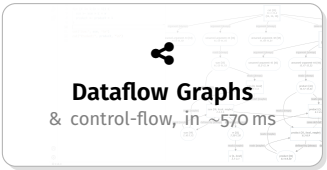
Dependency View
libraries, sources & I/O



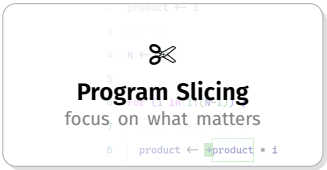
Data-Frame Shapes
hover tooltips in the editor

github.com/flowr-analysis/flowr

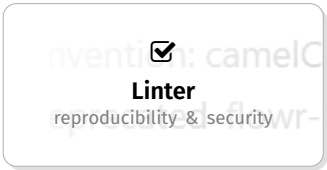
What flowR Offers



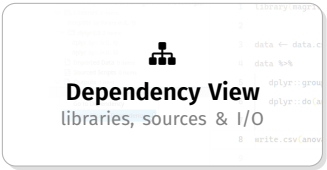
Dataflow Graphs
& control-flow, in ~570 ms



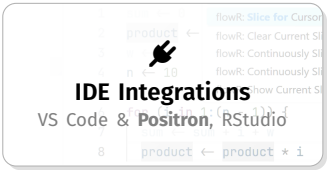
Program Slicing
focus on what matters



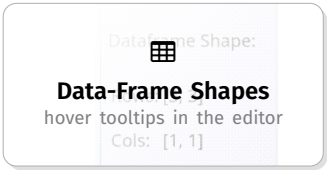
Linter
reproducibility & security



Dependency View
libraries, sources & I/O



IDE Integrations
VS Code & **Positron**, RStudio

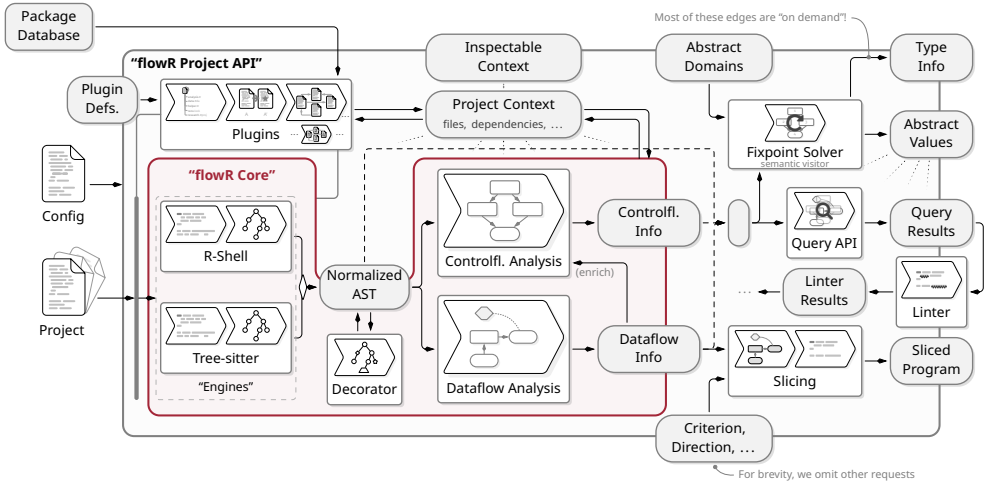


Data-Frame Shapes
hover tooltips in the editor

github.com/flowr-analysis/flowr

flowR's Architecture

flowR's Architecture



Try flowR and tell us what you want!

github.com/flowr-analysis/flowr

VS Code & **Positron** extension

Florian Sihler • florian.sihler@uni-ulm.de