

Statically reasoning about R vectors

Designing an abstract domain for a real language

Manuel Di Agostino¹, Florian Sihler², Vincenzo Arceri¹, Oliver Gerstl², Matthias Tichy²

28 October 2026

¹ University of Parma, ² Ulm University

Static Analysis and Software Verification, Seminar

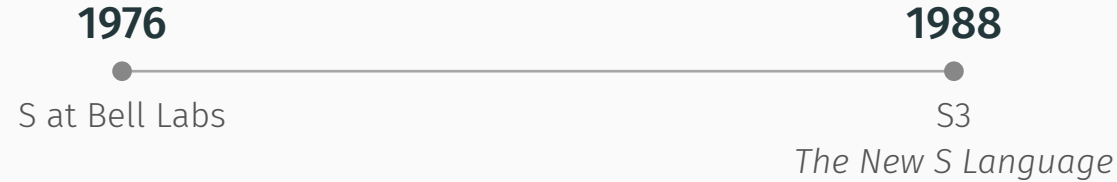
manuel.diagostino@unimore.it

M. Di Agostino, F. Sihler, V. Arceri, O. Gerstl, M. Tichy. *Towards Statically Reasoning about R Vectors*. NSAD '26, pp. 18–25.

doi:10.1145/3840563.3843708

$R \rightarrow \mu R \rightarrow \text{Domain} \rightarrow \text{Operators} \rightarrow \text{Validation}$

1. From S to R



S: an interactive environment for statisticians.¹

¹ Dates: R. Ihaka, *R: Past and Future History*, 1998; R FAQ §3.3.1.

Batch (Fortran)

```
PROGRAM ANALYSIS
READ (5, *) N, (X(I), I = 1, N)
CALL MEAN(X, N, AVG)
WRITE (6, *) AVG
STOP
END
C    unit 5 = card reader
C    unit 6 = line printer
```

edit → submit → queue → print, in **hours**

Batch (Fortran)

```
PROGRAM ANALYSIS
READ (5, *) N, (X(I), I = 1, N)
CALL MEAN(X, N, AVG)
WRITE (6, *) AVG
STOP
END
C    unit 5 = card reader
C    unit 6 = line printer
```

edit → submit → queue → print, in **hours**

Interactive (S)

```
x <- c(3.1, 4.2, 5.0)
mean(x)           # whole column
plot(x, y)
lm(y ~ x1 + x2)   # formula
```

type → see → adjust, in **seconds**

*“programming with data”*¹

¹ J. M. Chambers, *Programming with Data: A Guide to the S Language*, Springer, 1998.

Batch (Fortran)

```
PROGRAM ANALYSIS
READ (5, *) N, (X(I), I = 1, N)
CALL MEAN(X, N, AVG)
WRITE (6, *) AVG
STOP
END
C    unit 5 = card reader
C    unit 6 = line printer
```

edit → submit → queue → print, in **hours**

Interactive (S)

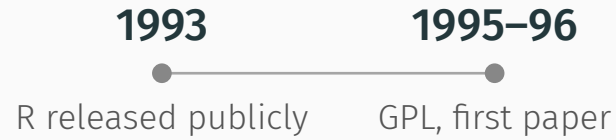
```
x <- c(3.1, 4.2, 5.0)
mean(x)           # whole column
plot(x, y)
lm(y ~ x1 + x2)   # formula
```

type → see → adjust, in **seconds**

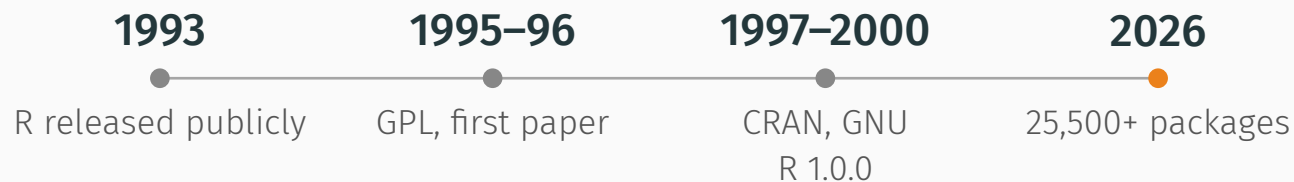
*“programming with data”*¹

Ergonomics first: designed for the analyst, not the compiler.

¹ J. M. Chambers, *Programming with Data: A Guide to the S Language*, Springer, 1998.



R: a free S, built by statisticians.



The ecosystem grew faster than the tooling.¹

¹ Dates: R. Ihaka, *R: Past and Future History*, 1998; R FAQ §3.3.1.
Package count: CRAN, *Contributed Packages*, accessed 25 Oct 2026: 25,529.

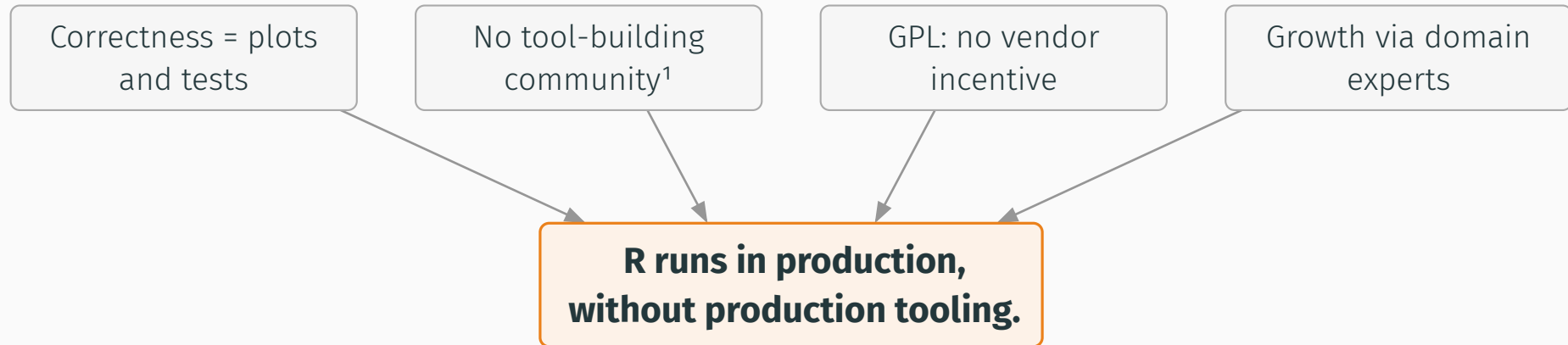
Correctness = plots
and tests

No tool-building
community¹

GPL: no vendor
incentive

Growth via domain
experts

¹ J. Fox, *Aspects of the Social Organization and Trajectory of the R Project*, The R Journal 1(2), 2009.



¹ J. Fox, *Aspects of the Social Organization and Trajectory of the R Project*, The R Journal 1(2), 2009.

2. Why R Resists Static Analysis

Everything is a vector:

- **TRUE**, *logical* of length 1
- **3** or **3.14**, *double* of length 1
- **c(1L, 2L, 3L)**, *integer* of length 3
- **"hello"**, *character* of length 1

There is **no scalar type**.

Everything is a vector:

- **TRUE**, *logical* of length 1
- **3** or **3.14**, *double* of length 1
- **c(1L, 2L, 3L)**, *integer* of length 3
- **"hello"**, *character* of length 1

There is **no scalar type**.

NA follows the same rule. It is the **sentinel for a missing value**: it occupies a position and it propagates.

Everything is a vector:

- **TRUE**, logical of length 1
- **3** or **3.14**, double of length 1
- **c(1L, 2L, 3L)**, integer of length 3
- **"hello"**, character of length 1

There is **no scalar type**.

NA follows the same rule. It is the **sentinel for a missing value**: it occupies a position and it propagates.

```
length(NA)           # 1           one element, and it is missing
c(1, 2, NA)          # [1, 2, NA]   NA occupies position 3
mean(c(1, 2, NA))    # NA          and it propagates
```

The two “nothings” of R are different objects.

NA: the value is unknown

- it is an element: it sits at a position
- any operation on it yields **NA**

```
x <- c(1, NA, 3)
length(x)      # 3
sum(x)         # NA
```

The two “nothings” of R are different objects.

NA: the value is unknown

- it is an element: it sits at a position
- any operation on it yields **NA**

```
x <- c(1, NA, 3)
length(x)      # 3
sum(x)         # NA
```

NULL: there is no value

- it is not an element: it has no position
- it disappears from a vector, so the length shrinks

```
y <- NULL
length(y)      # 0
c(1, NULL, 3)  # [1, 3]
```

The two “nothings” of R are different objects.

NA: the value is unknown

- it is an element: it sits at a position
- any operation on it yields **NA**

```
x <- c(1, NA, 3)
length(x)      # 3
sum(x)         # NA
```

NULL: there is no value

- it is not an element: it has no position
- it disappears from a vector, so the length shrinks

```
y <- NULL
length(y)      # 0
c(1, NULL, 3)  # [1, 3]
```

NULL is the empty vector, and `c()` is its literal. **NA** is a value that says **missing**.

→ The analysis must know: **how long is this value, and what sits at each position?**

Remark

Type Hierarchy

Vectors elements must be homogeneous in their type:

`logical < integer < double < character`

Implicit **coercion** occurs: when types are mixed, the elements are promoted to their least upper bound.

Remark

Type Hierarchy

Vectors elements must be homogeneous in their type:

logical < integer < double < character

Implicit **coercion** occurs: when types are mixed, the elements are promoted to their least upper bound.

```
c(TRUE, "hello")  # ["TRUE", "hello"]
```

Silent: no error, no warning, and nothing in the source says character.

`NA` exists once per type. It prints without quotes, so it is not the string `"NA"`:

```
c(NA, "hello")      # [NA, "hello"]
```

`NA` exists once per type. It prints without quotes, so it is not the string `"NA"`:

```
c(NA, "hello")      # [NA, "hello"]
```

The result takes the least upper bound of the element types:

```
c(TRUE, 3L, 2.5)    # [1.0, 3.0, 2.5]    double  
1L + TRUE           # 2L              integer
```

`NA` exists once per type. It prints without quotes, so it is not the string `"NA"`:

```
c(NA, "hello")      # [NA, "hello"]
```

The result takes the least upper bound of the element types:

```
c(TRUE, 3L, 2.5)    # [1.0, 3.0, 2.5]    double  
1L + TRUE           # 2L                integer
```

→ The analysis must know: **which type can each element have?**

R operations are **element-wise** across vectors.

```
c(1, 2, 3) + c(3, 2, 1)           # ?  
c(0, 0, 0, 0) + c(1, 2)           # ?  
c(1, 2, 3, 4, 5, 6) + c(10, 20, 30) # ?  
c(1, 5, 2, 8) > 3                  # ?  
c(1, 5, 2, 8) > c(4, 6)            # ?
```

R operations are **element-wise** across vectors.

```
c(1, 2, 3) + c(3, 2, 1)           # [4, 4, 4]
c(0, 0, 0, 0) + c(1, 2)           # ?
c(1, 2, 3, 4, 5, 6) + c(10, 20, 30) # ?
c(1, 5, 2, 8) > 3                  # ?
c(1, 5, 2, 8) > c(4, 6)            # ?
```

R operations are **element-wise** across vectors.

```
c(1, 2, 3) + c(3, 2, 1)           # [4, 4, 4]
c(0, 0, 0, 0) + c(1, 2)           # [1, 2, 1, 2]
c(1, 2, 3, 4, 5, 6) + c(10, 20, 30) # ?
c(1, 5, 2, 8) > 3                 # ?
c(1, 5, 2, 8) > c(4, 6)           # ?
```

R operations are **element-wise** across vectors.

```
c(1, 2, 3) + c(3, 2, 1)           # [4, 4, 4]
c(0, 0, 0, 0) + c(1, 2)           # [1, 2, 1, 2]
c(1, 2, 3, 4, 5, 6) + c(10, 20, 30) # [11, 22, 33, 14, 25, 36]
c(1, 5, 2, 8) > 3                  # ?
c(1, 5, 2, 8) > c(4, 6)            # ?
```

R operations are **element-wise** across vectors.

```
c(1, 2, 3) + c(3, 2, 1)           # [4, 4, 4]
c(0, 0, 0, 0) + c(1, 2)           # [1, 2, 1, 2]
c(1, 2, 3, 4, 5, 6) + c(10, 20, 30) # [11, 22, 33, 14, 25, 36]
c(1, 5, 2, 8) > 3                  # [FALSE, TRUE, FALSE, TRUE]
c(1, 5, 2, 8) > c(4, 6)            # ?
```

R operations are **element-wise** across vectors.

```
c(1, 2, 3) + c(3, 2, 1)           # [4, 4, 4]
c(0, 0, 0, 0) + c(1, 2)           # [1, 2, 1, 2]
c(1, 2, 3, 4, 5, 6) + c(10, 20, 30) # [11, 22, 33, 14, 25, 36]
c(1, 5, 2, 8) > 3                  # [FALSE, TRUE, FALSE, TRUE]
c(1, 5, 2, 8) > c(4, 6)            # [FALSE, FALSE, FALSE, TRUE]
```

R operations are **element-wise** across vectors.

```
c(1, 2, 3) + c(3, 2, 1)           # [4, 4, 4]
c(0, 0, 0, 0) + c(1, 2)           # [1, 2, 1, 2]
c(1, 2, 3, 4, 5, 6) + c(10, 20, 30) # [11, 22, 33, 14, 25, 36]
c(1, 5, 2, 8) > 3                  # [FALSE, TRUE, FALSE, TRUE]
c(1, 5, 2, 8) > c(4, 6)            # [FALSE, FALSE, FALSE, TRUE]
```

When two vectors have different lengths, the shorter is **recycled** to match the longer: that is why `x > 3` works without writing `c(3, 3, 3, 3)`.

When the longer length is not a multiple of the shorter, R **warns** and computes the result anyway.

Example

Not a multiple

```
r <- c(1, 1, 1)
if (runif() >= 0.5) {
  r <- r + c(41, 41, 41, 41)  # [WARN]
                              # [1, 1, 1] recycled to [1, 1, 1, 1]
} else {
  r <- r - c(1)              # [1] recycled to [1, 1, 1]
}
```

When the longer length is not a multiple of the shorter, R **warns** and computes the result anyway.

Example

Not a multiple

```
r <- c(1, 1, 1)
if (runif() >= 0.5) {
  r <- r + c(41, 41, 41, 41)  # [WARN]
                             # [1, 1, 1] recycled to [1, 1, 1, 1]
} else {
  r <- r - c(1)              # [1] recycled to [1, 1, 1]
}
```

After the `if-else`, `r` is either `[42, 42, 42, 42]` or `[0, 0, 0]`.

When the longer length is not a multiple of the shorter, R **warns** and computes the result anyway.

Example

Not a multiple

```
r <- c(1, 1, 1)
if (runif() >= 0.5) {
  r <- r + c(41, 41, 41, 41)  # [WARN]
                              # [1, 1, 1] recycled to [1, 1, 1, 1]
} else {
  r <- r - c(1)              # [1] recycled to [1, 1, 1]
}
```

After the `if-else`, `r` is either `[42, 42, 42, 42]` or `[0, 0, 0]`.

→ The analysis must know: **the branch taken decides the length**, not the syntax.

Any R object can carry *attributes*: `dim`, `class` and `names` change what it is.

Example

From vector to matrix

```
x <- 1:10
dim(x) <- c(2, 5)
#      [,1] [,2] [,3] [,4] [,5]
# [1,]    1    3    5    7    9
# [2,]    2    4    6    8   10
```

Any R object can carry *attributes*: `dim`, `class` and `names` change what it is.

Example

From vector to matrix

```
x <- 1:10
dim(x) <- c(2, 5)
#      [,1] [,2] [,3] [,4] [,5]
# [1,]    1    3    5    7    9
# [2,]    2    4    6    8   10
```

The ten integers are unchanged, the **shape** is not: without `dim` a plain integer vector, with it a matrix, `x[1, 2]` a legal index, and `%*%`, `t`, `solve` apply.

A data frame is not a primitive: it is a **named list** plus attributes.

```
l <- list(1:3, c("a", "b", "c"))
names(l)                                # NULL
names(l) <- c("id", "label")             # the names become the columns
df <- as.data.frame(l)                   # adds class and row.names
#   id label
# 1  1     a
# 2  2     b
# 3  3     c
```

A data frame is not a primitive: it is a **named list** plus attributes.

```
l <- list(1:3, c("a", "b", "c"))
names(l)                                # NULL
names(l) <- c("id", "label")            # the names become the columns
df <- as.data.frame(l)                  # adds class and row.names
#   id label
# 1  1     a
# 2  2     b
# 3  3     c
```

→ The analysis must know: **which attributes are present, not only the elements.**

The brackets are the same; the **index** decides the operation.

```
v <- c(10, 20, 30, 40)
v[c(2, 0, 5)]      # ?
v[c(-1, -3)]       # ?
v[c(TRUE, FALSE)]  # ?
```

The brackets are the same; the **index** decides the operation.

```
v <- c(10, 20, 30, 40)
v[c(2, 0, 5)]      # [20, NA]    0 is skipped, 5 is out of bounds
v[c(-1, -3)]       # ?
v[c(TRUE, FALSE)]  # ?
```

The brackets are the same; the **index** decides the operation.

```
v <- c(10, 20, 30, 40)
v[c(2, 0, 5)]      # [20, NA]    0 is skipped, 5 is out of bounds
v[c(-1, -3)]       # [20, 40]   positions 1 and 3 are removed
v[c(TRUE, FALSE)]  # ?
```

The brackets are the same; the **index** decides the operation.

```
v <- c(10, 20, 30, 40)
v[c(2, 0, 5)]      # [20, NA]    0 is skipped, 5 is out of bounds
v[c(-1, -3)]       # [20, 40]    positions 1 and 3 are removed
v[c(TRUE, FALSE)]  # [10, 30]    mask recycled to length 4
```

The brackets are the same; the **index** decides the operation.

```
v <- c(10, 20, 30, 40)
v[c(2, 0, 5)]      # [20, NA]    0 is skipped, 5 is out of bounds
v[c(-1, -3)]       # [20, 40]    positions 1 and 3 are removed
v[c(TRUE, FALSE)]  # [10, 30]    mask recycled to length 4
```

- **positive**: select by position; 0 is skipped, out-of-bounds gives **NA**

The brackets are the same; the **index** decides the operation.

```
v <- c(10, 20, 30, 40)
v[c(2, 0, 5)]      # [20, NA]    0 is skipped, 5 is out of bounds
v[c(-1, -3)]       # [20, 40]    positions 1 and 3 are removed
v[c(TRUE, FALSE)]  # [10, 30]    mask recycled to length 4
```

- **positive**: select by position; **0** is skipped, out-of-bounds gives **NA**
- **negative**: exclude by position; everything not listed is kept

The brackets are the same; the **index** decides the operation.

```
v <- c(10, 20, 30, 40)
v[c(2, 0, 5)]      # [20, NA]    0 is skipped, 5 is out of bounds
v[c(-1, -3)]       # [20, 40]    positions 1 and 3 are removed
v[c(TRUE, FALSE)]  # [10, 30]    mask recycled to length 4
```

- **positive**: select by position; 0 is skipped, out-of-bounds gives **NA**
- **negative**: exclude by position; everything not listed is kept
- **logical**: filter by mask; the mask is recycled to the vector's length

The brackets are the same; the **index** decides the operation.

```
v <- c(10, 20, 30, 40)
v[c(2, 0, 5)]      # [20, NA]    0 is skipped, 5 is out of bounds
v[c(-1, -3)]       # [20, 40]   positions 1 and 3 are removed
v[c(TRUE, FALSE)]  # [10, 30]   mask recycled to length 4
```

- **positive**: select by position; 0 is skipped, out-of-bounds gives **NA**
- **negative**: exclude by position; everything not listed is kept
- **logical**: filter by mask; the mask is recycled to the vector's length

R refuses to mix positive and negative indices; 0 is allowed with either.

→ The analysis must know: **the mode of each index and the length of the result.**

Assignment is not local: `v[i] <- x` can **grow** the vector.

```
v <- c(10, 20, 30)
v[5] <- 99          # ?
v[-2] <- 5          # ?
v[c(TRUE, FALSE)] <- 0 # ?
v[0] <- 5           # ?
```

Assignment is not local: `v[i] <- x` can **grow** the vector.

```
v <- c(10, 20, 30)
v[5] <- 99           # [10, 20, 30, NA, 99]
v[-2] <- 5           # ?
v[c(TRUE, FALSE)] <- 0 # ?
v[0] <- 5            # ?
```

Assignment is not local: `v[i] <- x` can **grow** the vector.

```
v <- c(10, 20, 30)
v[5] <- 99           # [10, 20, 30, NA, 99]
v[-2] <- 5           # [5, 20, 5, 5, 5]
v[c(TRUE, FALSE)] <- 0 # ?
v[0] <- 5            # ?
```

Assignment is not local: `v[i] <- x` can **grow** the vector.

```
v <- c(10, 20, 30)
v[5] <- 99          # [10, 20, 30, NA, 99]
v[-2] <- 5          # [5, 20, 5, 5, 5]
v[c(TRUE, FALSE)] <- 0 # [0, 20, 0, 5, 0]
v[0] <- 5           # ?
```

Assignment is not local: `v[i] <- x` can **grow** the vector.

```
v <- c(10, 20, 30)
v[5] <- 99           # [10, 20, 30, NA, 99]
v[-2] <- 5           # [5, 20, 5, 5, 5]
v[c(TRUE, FALSE)] <- 0 # [0, 20, 0, 5, 0]
v[0] <- 5            # unchanged, and no error
```

Assignment is not local: `v[i] <- x` can **grow** the vector.

```
v <- c(10, 20, 30)
v[5] <- 99          # [10, 20, 30, NA, 99]
v[-2] <- 5          # [5, 20, 5, 5, 5]
v[c(TRUE, FALSE)] <- 0 # [0, 20, 0, 5, 0]
v[0] <- 5           # unchanged, and no error
```

- past the end: the vector is extended and the gap is filled with **NA**

Assignment is not local: `v[i] <- x` can **grow** the vector.

```
v <- c(10, 20, 30)
v[5] <- 99          # [10, 20, 30, NA, 99]
v[-2] <- 5          # [5, 20, 5, 5, 5]
v[c(TRUE, FALSE)] <- 0 # [0, 20, 0, 5, 0]
v[0] <- 5           # unchanged, and no error
```

- past the end: the vector is extended and the gap is filled with **NA**
- a negative index or a recycled mask writes **one value to many positions**

Assignment is not local: `v[i] <- x` can **grow** the vector.

```
v <- c(10, 20, 30)
v[5] <- 99           # [10, 20, 30, NA, 99]
v[-2] <- 5           # [5, 20, 5, 5, 5]
v[c(TRUE, FALSE)] <- 0 # [0, 20, 0, 5, 0]
v[0] <- 5            # unchanged, and no error
```

- past the end: the vector is extended and the gap is filled with **NA**
- a negative index or a recycled mask writes **one value to many positions**
- index **0** writes to no position, and R does not complain

Assignment is not local: `v[i] <- x` can **grow** the vector.

```
v <- c(10, 20, 30)
v[5] <- 99           # [10, 20, 30, NA, 99]
v[-2] <- 5           # [5, 20, 5, 5, 5]
v[c(TRUE, FALSE)] <- 0 # [0, 20, 0, 5, 0]
v[0] <- 5            # unchanged, and no error
```

- past the end: the vector is extended and the gap is filled with **NA**
 - a negative index or a recycled mask writes **one value to many positions**
 - index **0** writes to no position, and R does not complain
- The analysis must know: **whether the length is still known after the update.**

3. The Analysis Problem and the Soundness Contract

At every program point $\ell \in L^1$, we want the **set of states** the program can be in.

Definition

Collecting semantics

A state σ maps each variable to the vector it holds, $\sigma : \text{Var} \rightarrow \mathcal{V}$, and

$$\llbracket P \rrbracket(\ell) \stackrel{\text{def}}{=} \{\sigma \mid \text{some run of } P \text{ reaches } \ell \text{ in state } \sigma\}$$

¹ L is the set of possible labels of the program.

- For example, given:

```
v <- c(10, 20, 30)
if (runif() >= 0.5) v <- v * 2
```

two states reach the end, one per branch: $\{v \mapsto [10, 20, 30]\}$ or $\{v \mapsto [20, 40, 60]\}$.

- For example, given:

```
v <- c(10, 20, 30)
if (runif() >= 0.5) v <- v * 2
```

two states reach the end, one per branch: $\{v \mapsto [10, 20, 30]\}$ or $\{v \mapsto [20, 40, 60]\}$.

- *Not computable* in general: deciding $\sigma \in \llbracket P \rrbracket(\ell)$ is deciding reachability, which is *undecidable*.

- For example, given:

```
v <- c(10, 20, 30)
if (runif() >= 0.5) v <- v * 2
```

two states reach the end, one per branch: $\{v \mapsto [10, 20, 30]\}$ or $\{v \mapsto [20, 40, 60]\}$.

- *Not computable* in general: deciding $\sigma \in \llbracket P \rrbracket(\ell)$ is deciding reachability, which is *undecidable*.
- So we compute an **approximation**: one abstract state per point (**flow-sensitive**), one set of vectors per variable (**non-relational**).

An analysis is **sound** when it never loses a run: at every point, the concrete states are among those its abstract state stands for.

Definition

Soundness

$$\forall \ell . \llbracket P \rrbracket(\ell) \subseteq \gamma(\llbracket P \rrbracket^\#(\ell))$$

An analysis is **sound** when it never loses a run: at every point, the concrete states are among those its abstract state stands for.

Definition

Soundness

$$\forall \ell . \llbracket P \rrbracket(\ell) \subseteq \gamma(\llbracket P \rrbracket^\#(\ell))$$

- γ may be **too large**, and that costs a false alarm.
- γ may never be **too small**, because a run outside it is a behaviour the analysis has lost.

An analysis is **sound** when it never loses a run: at every point, the concrete states are among those its abstract state stands for.

Definition

Soundness

$$\forall \ell . \llbracket P \rrbracket(\ell) \subseteq \gamma(\llbracket P \rrbracket^\#(\ell))$$

- γ may be **too large**, and that costs a false alarm.
- γ may never be **too small**, because a run outside it is a behaviour the analysis has lost.

The same obligation one level down, on operators:

$$c \in \gamma(v) \Rightarrow \text{op}(c) \in \gamma(\text{op}^\#(v))$$

Each neighbouring domain assumes something R does not.

- *FunArray*¹ — uniform segments, no value per position. Recycling forces a choice: unroll every position into its own segment, and the segmentation dissolves, or summarise and lose exactly the facts recycling asks for.
- *Giet et al.*² — facts on the whole sequence; recycling needs per-position values and does not preserve sortedness.
- *JS*³ and *Python*⁴ — the hard part is the set of keys; in R, homogeneous and position-indexed, it is the length.

¹ P. Cousot, R. Cousot, F. Logozzo, *A Parametric Segmentation Functor for Fully Automatic and Scalable Array Content Analysis*, POPL 2011.

² J. Giet, F. Ridoux, X. Rival, *A Product of Shape and Sequence Abstractions*, SAS 2023.

³ S. H. Jensen, A. Møller, P. Thiemann, *Type Analysis for JavaScript*, SAS 2009.

⁴ R. Monat, A. Ouadjaout, A. Miné, *Static Type Analysis by Abstract Interpretation of Python Programs*, ECOOP 2020.

Both stop one level above the vector.

- *Data frames*¹ — model the transformations of a frame; the vector semantics underneath stays unmodelled.
- *flowR*² — dataflow graphs and slicing, the platform Part 7 is built on; no abstract domain for vector contents.

¹ O. Gerstl, *Tracking the Shape of Data Frames in R Programs Using Abstract Interpretation*, master's thesis, Universität Ulm, 2025; L. Negrini, G. Shabadi, C. Urban, *Static Analysis of Data Transformations in Jupyter Notebooks*, SOAP 2023.

² F. Sihler, M. Tichy, *Statically Analyzing the Dataflow of R Programs*, PACMPL 9(OOPSLA2), art. 309, 2025.

Both stop one level above the vector.

- *Data frames*¹ — model the transformations of a frame; the vector semantics underneath stays unmodelled.
- *flowR*² — dataflow graphs and slicing, the platform Part 7 is built on; no abstract domain for vector contents.

Neither has a rule for *recycling*, and every element-wise R operation applies it first.

¹ O. Gerstl, *Tracking the Shape of Data Frames in R Programs Using Abstract Interpretation*, master's thesis, Universität Ulm, 2025; L. Negrini, G. Shabadi, C. Urban, *Static Analysis of Data Transformations in Jupyter Notebooks*, SOAP 2023.

² F. Sihler, M. Tichy, *Statically Analyzing the Dataflow of R Programs*, PACMPL 9(OOPSLA2), art. 309, 2025.

Part 2 presents what the analysis must know. Together those answers are the requirements on the domain.

- **Length**: which lengths are possible, from **NULL** (0) to unbounded.
- **Content**: which value, and which type, each element can have.
- **Operations**: which positions survive a selection, and whether an update grows the vector.
- **Attributes**: present for certain, or only possibly.
- **Termination**: the length can grow without bound, so the iteration must stop.

4. The μR Language

Real R is much wider than vectors:

- **environments** are first-class values

```
e <- new.env()  
e$x <- 1
```

Real R is much wider than vectors:

- **environments** are first-class values
- arguments are **lazy promises**

```
f <- function(a, b) a * 10
f(10, stop("boom")) # 100: b unused
f(stop("boom"), 1)  # Error: a used
```

Real R is much wider than vectors:

- **environments** are first-class values
- arguments are **lazy promises**
- code can be rewritten at run time by **metaprogramming**,

```
eval(quote(`+` <- function(a, b) 42))  
1 + 2    # 42
```

Real R is much wider than vectors:

- **environments** are first-class values
- arguments are **lazy promises**
- code can be rewritten at run time by **metaprogramming**,
- every call: R's **S3/S4** OOP dispatch.

```
length.foo <- function(x) 42  
length(structure(1, class = "foo")) # 42
```

Real R is much wider than vectors:

- **environments** are first-class values
- arguments are **lazy promises**
- code can be rewritten at run time by **metaprogramming**,
- every call: R's **S3/S4** OOP dispatch.

μR keeps the constructs that affect vectors and drops the rest:

- **NULL** is not a special object: it is the empty vector.
- An environment is not a value: it is a flat map from variables to vectors.

We omit usual `if-else` and `while` constructs for brevity.

$$\begin{aligned}
 E_{idx} &::= \varepsilon \\
 &| E \\
 Atm &::= Var \mid Float \mid Int \mid Bool \mid \text{NULL} \\
 Bool &::= \text{TRUE} \mid \text{FALSE} \mid \text{NA} \\
 \bowtie_{arith} &::= + \mid - \mid * \mid / \\
 \bowtie_{rel} &::= > \mid < \mid = \mid \neq \mid \leq \mid \geq \\
 \bowtie_{log} &::= \& \mid | \\
 \triangleright &::= ! \mid -
 \end{aligned}$$

$$\begin{aligned}
 E &::= Atm \mid \triangleright E \\
 &| e_1 \bowtie_{arith} e_2 \\
 &| e_1 \bowtie_{rel} e_2 \\
 &| e_1 \bowtie_{log} e_2 \\
 &| c() \\
 &| c(E_1, \dots, E_n) \\
 &| Var \leftarrow E \\
 &| Var[E_{idx}] \\
 &| Var[[E_{idx}]] \\
 &| Var[E_{idx}] \leftarrow E \\
 &| Var[[E_{idx}]] \leftarrow E \\
 &| \text{setAttr}(Var, \{s_1, \dots, s_n\}) \\
 &| \text{getAttr}(Var)
 \end{aligned}$$

$c(\dots) \longrightarrow \oplus$ $\bowtie \longrightarrow \text{Recycle}$ $v[i] \longrightarrow \text{Select}$ $v[i] \leftarrow e \longrightarrow \text{Update}$

We omit usual `if-else` and `while` constructs for brevity.

E_{idx}	$::=$	ε	E
Atm	$::=$	Var	$Float$ Int $Bool$ $NULL$
$Bool$	$::=$	$TRUE$ $FALSE$ NA	
$\bowtie_{arith}$	$::=$	$+$ $-$ $*$ $/$	
$\bowtie_{rel}$	$::=$	$>$ $<$ $=$ $\neq$ $\leq$ $\geq$	
$\bowtie_{log}$	$::=$	$\&$ $ $	
$\triangleright$	$::=$	$!$ $-$	
E	$::=$	Atm	$\triangleright E$
		$e_1 \bowtie_{arith} e_2$	
		$e_1 \bowtie_{rel} e_2$	
		$e_1 \bowtie_{log} e_2$	
		$c()$	
		$c(E_1, \dots, E_n)$	
		$Var \leftarrow E$	
		$Var[E_{idx}]$	
		$Var[[E_{idx}]]$	
		$Var[E_{idx}] \leftarrow E$	
		$Var[[E_{idx}]] \leftarrow E$	
		$setAttr(Var, \{s_1, \dots, s_n\})$	
		$getAttr(Var)$	

$c(\dots) \rightarrow \oplus$ $\bowtie \rightarrow$ **Recycle** $v[i] \rightarrow$ **Select** $v[i] \leftarrow e \rightarrow$ **Update**

We omit usual `if-else` and `while` constructs for brevity.

$$E_{idx} ::= \varepsilon$$

$$| E$$

$$Atm ::= Var \mid Float \mid Int \mid Bool \mid \text{NULL}$$

$$Bool ::= \text{TRUE} \mid \text{FALSE} \mid \text{NA}$$

$$\bowtie_{arith} ::= + \mid - \mid * \mid /$$

$$\bowtie_{rel} ::= > \mid < \mid = \mid \neq \mid \leq \mid \geq$$

$$\bowtie_{log} ::= \& \mid |$$

$$\triangleright ::= ! \mid -$$

$$E ::= Atm \mid \triangleright E$$

$$| e_1 \bowtie_{arith} e_2$$

$$| e_1 \bowtie_{rel} e_2$$

$$| e_1 \bowtie_{log} e_2$$

$$| c()$$

$$| c(E_1, \dots, E_n)$$

$$| Var \leftarrow E$$

$$| Var[E_{idx}]$$

$$| Var[[E_{idx}]]$$

$$| Var[E_{idx}] \leftarrow E$$

$$| Var[[E_{idx}]] \leftarrow E$$

$$| \text{setAttr}(Var, \{s_1, \dots, s_n\})$$

$$| \text{getAttr}(Var)$$

$c(\dots) \longrightarrow \oplus$ $\bowtie \longrightarrow \text{Recycle}$ $v[i] \longrightarrow \text{Select}$ $v[i] \leftarrow e \longrightarrow \text{Update}$

$$Atm ::= Var \mid Float \mid Int \mid Bool \mid NULL$$

Definition

Concrete Vector

A concrete vector $\mathbf{v} \in \mathcal{V}$ is a pair $([e_1, \dots, e_n], a) \in (\mathbb{N} \rightarrow \mathbb{V}_{Atm}) \times \mathcal{A}$: a **total** map from indices to atomic values, plus a set of attribute names.

$$([e_1, \dots, e_n], \mathcal{A}) \equiv [e_1, \dots, e_n]_{\mathcal{A}} \quad ([e_1, \dots, e_n], \emptyset) \equiv [e_1, \dots, e_n]$$

$$Atm ::= Var \mid Float \mid Int \mid Bool \mid NULL$$

Definition

Concrete Vector

A concrete vector $v \in \mathcal{V}$ is a pair $([e_1, \dots, e_n], a) \in (\mathbb{N} \rightarrow \mathbb{V}_{Atm}) \times \mathcal{A}$: a **total** map from indices to atomic values, plus a set of attribute names.

$$([e_1, \dots, e_n], \mathcal{A}) \equiv [e_1, \dots, e_n]_{\mathcal{A}} \quad ([e_1, \dots, e_n], \emptyset) \equiv [e_1, \dots, e_n]$$

- Total means that every index has a value:
 - past the length it is **NULL** (the **empty vector**, not a missing element)
 - in-bounds, missingness is **NA**.

$$Atm ::= Var \mid Float \mid Int \mid Bool \mid NULL$$

Definition

Concrete Vector

A concrete vector $v \in \mathcal{V}$ is a pair $([e_1, \dots, e_n], a) \in (\mathbb{N} \rightarrow \mathbb{V}_{Atm}) \times \mathcal{A}$: a **total** map from indices to atomic values, plus a set of attribute names.

$$([e_1, \dots, e_n], \mathcal{A}) \equiv [e_1, \dots, e_n]_{\mathcal{A}} \quad ([e_1, \dots, e_n], \emptyset) \equiv [e_1, \dots, e_n]$$

- Total means that every index has a value:
 - past the length it is **NULL** (the **empty vector**, not a missing element)
 - in-bounds, missingness is **NA**.
- We use the helper v^* for real readings:

$$v^*(i) \stackrel{\text{def}}{=} \begin{cases} v(i) & \text{if } v(i) \neq \text{NULL} \\ \text{NA} & \text{otherwise} \end{cases} \quad c() \stackrel{\text{def}}{=} [] \quad c(e_1, \dots, e_n) \stackrel{\text{def}}{=} e_1 \oplus \dots \oplus e_n$$

Definition

Recycle(x, y) : $\mathcal{V} \times \mathcal{V} \rightarrow \mathcal{V} \times \mathcal{V}$ is 3 steps:

- **cast** operands to their common type;

Recycle

Definition

Recycle

Recycle(x, y) : $\mathcal{V} \times \mathcal{V} \rightarrow \mathcal{V} \times \mathcal{V}$ is 3 steps:

- **cast** operands to their common type;
- **length**: $m = \max(|x|, |y|)$, decided by the operand lengths alone;

Definition

Recycle

Recycle(x, y) : $\mathcal{V} \times \mathcal{V} \rightarrow \mathcal{V} \times \mathcal{V}$ is 3 steps:

- **cast** operands to their common type;
- **length**: $m = \max(|x|, |y|)$, decided by the operand lengths alone;
- **content**: ρ_c repeats the elements until they fill m :

$$\rho_c(v, l)(i) \stackrel{\text{def}}{=} v(((i - 1) \bmod |v|) + 1) \quad \text{for } i \in [1, l], \quad v \neq []$$

Definition

Recycle

Recycle(x, y) : $\mathcal{V} \times \mathcal{V} \rightarrow \mathcal{V} \times \mathcal{V}$ is 3 steps:

- **cast** operands to their common type;
- **length**: $m = \max(|x|, |y|)$, decided by the operand lengths alone;
- **content**: ρ_c repeats the elements until they fill m :

$$\rho_c(v, l)(i) \stackrel{\text{def}}{=} v(((i - 1) \bmod |v|) + 1) \quad \text{for } i \in [1, l], \quad v \neq []$$

Example

Unequal lengths

$$\text{Recycle}([0, 1], [3, 4, 5, 6]) = ([0, 1, 0, 1], [3, 4, 5, 6])$$

The operator is **Select** : $\mathcal{V} \times \mathcal{V}_{Int} \rightarrow \mathcal{V}$, a **partial** function: the three modes are disjoint, and R rejects a selector that mixes them.

- **positive**: keep the positions listed, in the order given, duplicates included;

The operator is **Select** : $\mathcal{V} \times \mathcal{V}_{Int} \rightarrow \mathcal{V}$, a **partial** function: the three modes are disjoint, and R rejects a selector that mixes them.

- **positive**: keep the positions listed, in the order given, duplicates included;
- **negative**: drop the positions listed, keep the others in increasing order;

The operator is **Select** : $\mathcal{V} \times \mathcal{V}_{Int} \rightarrow \mathcal{V}$, a **partial** function: the three modes are disjoint, and R rejects a selector that mixes them.

- **positive**: keep the positions listed, in the order given, duplicates included;
- **negative**: drop the positions listed, keep the others in increasing order;
- **logical**: recycle the mask to $\max(|v|, |c|)$ and keep where it is **True**.

The operator is **Select** : $\mathcal{V} \times \mathcal{V}_{Int} \rightarrow \mathcal{V}$, a **partial** function: the three modes are disjoint, and R rejects a selector that mixes them.

- **positive**: keep the positions listed, in the order given, duplicates included;
- **negative**: drop the positions listed, keep the others in increasing order;
- **logical**: recycle the mask to $\max(|v|, |c|)$ and keep where it is **True**.

For example, the positive case:

Definition

Positive Selection

$$\text{Select}(v, c) \stackrel{\text{def}}{=} \bigoplus_{i=1}^{|c|} \sigma_i \quad \sigma_i \stackrel{\text{def}}{=} \begin{cases} [v^*(c(i))] & \text{if } c(i) > 0 \\ [] & \text{if } c(i) = 0 \\ [\text{NA}] & \text{if } c(i) = \text{NA} \end{cases}$$

The operator is **Update** $(v, c, v_e) : \mathcal{V} \times \mathcal{V}_{Int} \times \mathcal{V} \rightarrow \mathcal{V}$, a **partial** function: the three modes are disjoint, and R rejects a selector that mixes them.

- **positive**: overwrite the positions c lists with the values v_e ; past $|v|$ the vector grows;

The operator is **Update** $(v, c, v_e) : \mathcal{V} \times \mathcal{V}_{Int} \times \mathcal{V} \rightarrow \mathcal{V}$, a **partial** function: the three modes are disjoint, and R rejects a selector that mixes them.

- **positive**: overwrite the positions c lists with the values v_e ; past $|v|$ the vector grows;
- **negative**: overwrite all but the excluded positions; the length is unchanged;

The operator is **Update** $(v, c, v_e) : \mathcal{V} \times \mathcal{V}_{Int} \times \mathcal{V} \rightarrow \mathcal{V}$, a **partial** function: the three modes are disjoint, and R rejects a selector that mixes them.

- **positive**: overwrite the positions c lists with the values v_e ; past $|v|$ the vector grows;
- **negative**: overwrite all but the excluded positions; the length is unchanged;
- **logical**: recycle the mask to $\max(|v|, |c|)$ and overwrite where it is **True**.

The operator is **Update** $(v, c, v_e) : \mathcal{V} \times \mathcal{V}_{Int} \times \mathcal{V} \rightarrow \mathcal{V}$, a **partial** function: the three modes are disjoint, and R rejects a selector that mixes them.

- **positive**: overwrite the positions c lists with the values v_e ; past $|v|$ the vector grows;
- **negative**: overwrite all but the excluded positions; the length is unchanged;
- **logical**: recycle the mask to $\max(|v|, |c|)$ and overwrite where it is **True**.

For example, the positive case:

Definition

Positive Update

$$\text{Update}(v, c, v_e) \stackrel{\text{def}}{=} r(u_1 \mapsto v'_e(1), \dots, u_k \mapsto v'_e(k)) \quad r = v \oplus \underbrace{[\mathbf{NA}, \dots, \mathbf{NA}]}_{\max(C - |v|, 0)} \quad v'_e = \rho_c(v_e, k)$$

with $C = \max\{c(i) \mid c(i) \neq \mathbf{NA}\}$ and $u_1, \dots, u_k$ the non-zero entries of c .

Four lines, five operators, one shape:

```
x <- c(TRUE, 2L, 3.0)      # Cast, then ⊕: ?  
y <- x + c(10, 20)         # Recycle, then +: ?  
z <- y[c(TRUE, FALSE, NA)] # Select (logical): ?  
z[c(2, 4)] <- c(50, 100)   # Update (positive): ?
```

Deterministic, and not evident. Part 5 has to predict exactly this, without running it.

Four lines, five operators, one shape:

```
x <- c(TRUE, 2L, 3.0)      # Cast, then ⊕: [1, 2, 3] double
y <- x + c(10, 20)         # Recycle, then +: ?
z <- y[c(TRUE, FALSE, NA)] # Select (logical): ?
z[c(2, 4)] <- c(50, 100)   # Update (positive): ?
```

Deterministic, and not evident. Part 5 has to predict exactly this, without running it.

Four lines, five operators, one shape:

```
x <- c(TRUE, 2L, 3.0)      # Cast, then ⊕: [1, 2, 3] double
y <- x + c(10, 20)         # Recycle, then +: [11, 22, 13]
z <- y[c(TRUE, FALSE, NA)] # Select (logical): ?
z[c(2, 4)] <- c(50, 100)   # Update (positive): ?
```

Deterministic, and not evident. Part 5 has to predict exactly this, without running it.

Four lines, five operators, one shape:

```
x <- c(TRUE, 2L, 3.0)      # Cast, then ⊕: [1, 2, 3] double
y <- x + c(10, 20)         # Recycle, then +: [11, 22, 13]
z <- y[c(TRUE, FALSE, NA)] # Select (logical): [11, NA]
z[c(2, 4)] <- c(50, 100)   # Update (positive): ?
```

Deterministic, and not evident. Part 5 has to predict exactly this, without running it.

Four lines, five operators, one shape:

```
x <- c(TRUE, 2L, 3.0)      # Cast, then ⊕: [1, 2, 3] double
y <- x + c(10, 20)         # Recycle, then +: [11, 22, 13]
z <- y[c(TRUE, FALSE, NA)] # Select (logical): [11, NA]
z[c(2, 4)] <- c(50, 100)   # Update (positive): [11, 50, NA, 100]
```

Deterministic, and not evident. Part 5 has to predict exactly this, without running it.

5. The Abstract Domain

Track two positions, summarise the rest.

```
x <- if (runif() >= 0.5) c(0, 0) else c(0, 0, 1, 100) # length 2 or 4
```

Track two positions, summarise the rest.

```
x <- if (runif() >= 0.5) c(0, 0) else c(0, 0, 1, 100) # length 2 or 4
```

The naive element:

$$v = ([2, 4], \langle [0, 0][0, 0] \rangle, [1, 100], \text{AttrEmpty})$$

Positions 3 and 4 both read $[1, 100]$. We lose that $x[3]$ is 1 whenever it exists, and that $x[4]$ is 100 whenever it exists.

Track two positions, summarise the rest.

```
x <- if (runif() >= 0.5) c(0, 0) else c(0, 0, 1, 100) # length 2 or 4
```

The naive element:

$$v = ([2, 4], \langle [0, 0][0, 0] \rangle, [1, 100], \text{AttrEmpty})$$

Positions 3 and 4 both read $[1, 100]$. We lose that $x[3]$ is 1 whenever it exists, and that $x[4]$ is 100 whenever it exists.

A fixed prefix is not enough. When the length has a finite bound, every position is tracked; the summary is only for $u = +\infty$.

Definition

A raw abstract vector in $\tilde{\mathcal{V}}_{\mathcal{B}}^{\#}$ is a 4-tuple

$$v = ([l, u], \langle p_1 \cdots p_k \rangle, s, a) \in \mathbf{Itv}_0 \times \mathcal{B}^{\#*} \times \mathcal{B}^{\#} \times \mathbf{Attr}^{\#}$$

with the invariant:

$$l \leq k \leq u \wedge (u \neq +\infty \Rightarrow (s = \perp_{\mathcal{B}^{\#}} \wedge u = k))$$

Definition

A raw abstract vector in $\tilde{\mathcal{V}}_{\mathcal{B}}^{\#}$ is a 4-tuple

$$v = ([l, u], \langle p_1 \cdots p_k \rangle, s, a) \in \mathbf{Itv}_0 \times \mathcal{B}^{\#*} \times \mathcal{B}^{\#} \times \mathbf{Attr}^{\#}$$

with the invariant:

$$l \leq k \leq u \wedge (u \neq +\infty \Rightarrow (s = \perp_{\mathcal{B}^{\#}} \wedge u = k))$$

For example, setting $\mathcal{B}^{\#} = \mathbf{Itv}$:

- $([1, 2], \langle [1, 1][0, 0] \rangle, \perp_{\mathbf{Itv}}, \mathbf{AttrEmpty}) \xrightarrow{\gamma_{\mathbf{Itv}}^{\#}} \{[1], [1, 0]\}$
- $([1, +\infty], \langle [42, 42] \rangle, [42, 42], \mathbf{AttrEmpty}) \xrightarrow{\gamma_{\mathbf{Itv}}^{\#}} \{[42], [42, 42], [42, 42, 42], \dots\}$
- $([1, +\infty], \langle [42, 42][42, 42] \rangle, [42, 42], \mathbf{AttrEmpty}) \xrightarrow{\gamma_{\mathbf{Itv}}^{\#}} \{[42], [42, 42], [42, 42, 42], \dots\}$

Definition

A raw abstract vector in $\tilde{\mathcal{V}}_{\mathcal{B}}^{\#}$ is a 4-tuple

$$v = ([l, u], \langle p_1 \cdots p_k \rangle, s, a) \in \mathbf{Itv}_0 \times \mathcal{B}^{\#*} \times \mathcal{B}^{\#} \times \mathbf{Attr}^{\#}$$

with the invariant:

$$l \leq k \leq u \wedge (u \neq +\infty \Rightarrow (s = \perp_{\mathcal{B}^{\#}} \wedge u = k))$$

For example, setting $\mathcal{B}^{\#} = \mathbf{Itv}$:

- $([1, 2], \langle [1, 1][0, 0] \rangle, \perp_{\mathbf{Itv}}, \mathbf{AttrEmpty}) \xrightarrow{\gamma_{\mathbf{Itv}}^{\#}} \{[1], [1, 0]\}$
- $([1, +\infty], \langle [42, 42] \rangle, [42, 42], \mathbf{AttrEmpty}) \xrightarrow{\gamma_{\mathbf{Itv}}^{\#}} \{[42], [42, 42], [42, 42, 42], \dots\}$
- $([1, +\infty], \langle [42, 42][42, 42] \rangle, [42, 42], \mathbf{AttrEmpty}) \xrightarrow{\gamma_{\mathbf{Itv}}^{\#}} \{[42], [42, 42], [42, 42, 42], \dots\}$

The last two tuples are the same element.

Definition

Equivalence and minimum representative

- $v_1 \equiv v_2$ if they agree on L, s, a and on the known content up to the shorter sequence, with the longer one's extra positions equal to s .
- $v^\downarrow$ removes the trailing positions of the known content equal to s , keeping at least l .

The last two tuples are the same element.

Definition

Equivalence and minimum representative

- $v_1 \equiv v_2$ if they agree on L, s, a and on the known content up to the shorter sequence, with the longer one's extra positions equal to s .
- $v^\downarrow$ removes the trailing positions of the known content equal to s , keeping at least l .

$([1, +\infty], \langle [42, 42] \rangle, [42, 42], \text{AttrEmpty})$

$([1, +\infty], \langle [42, 42][42, 42] \rangle, [42, 42], \text{AttrEmpty})$

The second spells the same element: its extra $[42, 42]$ is s .

Every element is a class of raw tuples, plus two special elements.

Definition

The abstract domain is the lattice:

$$\mathcal{D}_{\mathcal{V}_B^\#} = \left\langle \mathcal{V}_B^\#, \sqsubseteq, \sqcup, \sqcap, \perp_{\mathcal{V}_B^\#}, \top_{\mathcal{V}_B^\#} \right\rangle$$

where $\mathcal{V}_B^\# = \{[v]_\equiv \mid v \in \tilde{\mathcal{V}}_B^\#\} \cup \{\perp_{\mathcal{V}_B^\#}, \top_{\mathcal{V}_B^\#}\}$ and:

- $\perp_{\mathcal{V}_B^\#} = (\perp_{\text{Itv}_0}, \varepsilon, \perp_{B^\#}, \perp_{\text{Attr}^\#})$
- $\top_{\mathcal{V}_B^\#} = (\top_{\text{Itv}_0}, \varepsilon, \top_{B^\#}, \top_{\text{Attr}^\#})$

$\sqsubseteq$ is containment: every vector the left element represents is represented by the right one.

$$([2, 2], \langle [1, 1][2, 2] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty}) \sqsubseteq ([1, 3], \langle [1, 1][2, 2][3, 3] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$$

$$([1, 3], \langle [1, 1][2, 2][3, 3] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty}) \not\sqsubseteq ([2, 2], \langle [1, 1][2, 2] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$$

The left element of the second line also represents $[1]$ and $[1, 2, 3]$; the right one does not.

$\sqcup$ is the smallest element that contains both.

$$\begin{aligned}
 & ([2, 2], \langle [0, 0][0, 0] \rangle, \perp_{\text{Itv}}, \mathbf{AttrEmpty}) \sqcup ([4, 4], \langle [0, 0][0, 0][1, 1][100, 100] \rangle, \perp_{\text{Itv}}, \mathbf{AttrEmpty}) \\
 &= ([2, 4], \langle [0, 0][0, 0][1, 1][100, 100] \rangle, \perp_{\text{Itv}}, \mathbf{AttrEmpty})
 \end{aligned}$$

Both are below it, and positions 3 and 4 must stay apart: no smaller element contains both. This is the element a fixed prefix could not write.

$\sqcap$ is the largest element contained in both.

$$\begin{aligned}
 & ([2, 2], \langle [1, 1][1, 2] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty}) \sqcap ([1, +\infty], \langle [1, 1][2, 2][3, 3] \rangle, [-1, -1], \text{AttrEmpty}) \\
 &= ([2, 2], \langle [1, 1][2, 2] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})
 \end{aligned}$$

Position 2 keeps only the values in both. Position 3 is impossible: the first element has length 2.

When a position has no common value, the meet is $\perp_{\mathcal{V}_B^\#}$:

$$([3, 3], \langle [1, 1][9, 9][3, 3] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty}) \sqcap ([3, 3], \langle [1, 1][2, 2][3, 3] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty}) = \perp_{\mathcal{V}_B^\#}$$

$x \leftarrow c(x, i)$ in a loop appends one element per iteration: no finite bound.

Definition

Widening

- **length**: $[l', u'] = L_1 \nabla_{\text{Itv}_0} L_2$
- **content**: $p_1(i) \nabla_{\mathcal{B}^\#} p_2(i)$ up to $\max(k_1, k_2)$; only the common prefix if $u' = +\infty \wedge (u_1 = +\infty \vee u_2 = +\infty)$.
- **summary**: $\perp_{\mathcal{B}^\#}$ if $u' \neq +\infty$; otherwise the value widening of s_1 with the join of s_2 and every position dropped from the known content.

$x \leftarrow c(x, i)$ in a loop appends one element per iteration: no finite bound.

Definition

Widening

- **length**: $[l', u'] = L_1 \nabla_{\text{Itv}_0} L_2$
- **content**: $p_1(i) \nabla_{\mathcal{B}^\#} p_2(i)$ up to $\max(k_1, k_2)$; only the common prefix if $u' = +\infty \wedge (u_1 = +\infty \vee u_2 = +\infty)$.
- **summary**: $\perp_{\mathcal{B}^\#}$ if $u' \neq +\infty$; otherwise the value widening of s_1 with the join of s_2 and every position dropped from the known content.

$$x_0 = ([0, 0], \varepsilon, \perp_{\text{Itv}}, \text{AttrEmpty})$$

$$x_1 = x_0 \nabla_{\mathcal{V}_{\text{Itv}}^\#} v_1 = ([0, +\infty], \langle [1, 1] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$$

$$x_2 = x_1 \nabla_{\mathcal{V}_{\text{Itv}}^\#} v_2 = ([0, +\infty], \langle [1, 1] \rangle, [2, 2], \text{AttrEmpty})$$

$$x_3 = x_2 \nabla_{\mathcal{V}_{\text{Itv}}^\#} v_3 = ([0, +\infty], \langle [1, 1] \rangle, [2, +\infty], \text{AttrEmpty}) = x_4$$

Definition

Concretization

The concretization function $\gamma_{\mathcal{V}_B^\#} : \mathcal{V}_B^\# \rightarrow \mathcal{P}(\mathcal{V})$ is defined as:

$$\gamma_{\mathcal{V}_B^\#}([l, u], \mathbf{p}, s, a) \stackrel{\text{def}}{=} \left\{ [c_1, \dots, c_n]_{\mathcal{A}} \in \mathcal{V} \left| \begin{array}{l} n \in \gamma_{\text{Itv}_0}([l, u]), \mathcal{A} \in \gamma_{\text{Attr}^\#}(a), \\ \forall i \in [1, \min(n, |\mathbf{p}|)] : c_i \in \gamma_{\mathcal{B}^\#}(p_i), \\ \forall j \in [|\mathbf{p}| + 1, n] : c_j \in \gamma_{\mathcal{B}^\#}(s) \end{array} \right. \right\}$$

The analysis is concretization-based: it relies solely on γ , since no vector-level best abstraction exists.

A Galois connection would give every concrete set a best abstraction, the least d with $c \subseteq \gamma_{\mathcal{V}_B^\#}(d)$:

$$\alpha_{\mathcal{V}_B^\#}(c) = \sqcap \{d \mid c \subseteq \gamma_{\mathcal{V}_B^\#}(d)\}$$

The two sides of

$$\alpha_{\mathcal{V}_B^\#}(c) \sqsubseteq d \iff c \subseteq \gamma_{\mathcal{V}_B^\#}(d)$$

are:

- **Soundness:** $\alpha_{\mathcal{V}_B^\#}(c) \sqsubseteq d \implies c \subseteq \gamma_{\mathcal{V}_B^\#}(d)$. At $d = \alpha_{\mathcal{V}_B^\#}(c)$ this gives $c \subseteq \gamma_{\mathcal{V}_B^\#}(\alpha_{\mathcal{V}_B^\#}(c))$, so $\alpha_{\mathcal{V}_B^\#}(c)$ is an upper bound of c .
- **Best abstraction:** $c \subseteq \gamma_{\mathcal{V}_B^\#}(d) \implies \alpha_{\mathcal{V}_B^\#}(c) \sqsubseteq d$. Every upper bound is above $\alpha_{\mathcal{V}_B^\#}(c)$: the least one exists and is $\alpha_{\mathcal{V}_B^\#}(c)$.

One set breaks it. Take $\mathcal{C} = \{[1], [1, 2], [1, 2, 3], \dots\}$ and, for every $i \geq 1$,

$$v_i = ([1, +\infty], \langle [1, 1] \dots [i, i] \rangle, [i + 1, +\infty], \text{AttrEmpty})$$

One set breaks it. Take $\mathcal{C} = \{[1], [1, 2], [1, 2, 3], \dots\}$ and, for every $i \geq 1$,

$$v_i = ([1, +\infty], \langle [1, 1] \dots [i, i] \rangle, [i + 1, +\infty], \text{AttrEmpty})$$

- $\mathcal{C} \subseteq \gamma_{\nu_B^\#}(v_i)$ for all i : the first i positions are exact, the tail in $[i + 1, +\infty]$.

One set breaks it. Take $\mathcal{C} = \{[1], [1, 2], [1, 2, 3], \dots\}$ and, for every $i \geq 1$,

$$v_i = ([1, +\infty], \langle [1, 1] \dots [i, i] \rangle, [i + 1, +\infty], \text{AttrEmpty})$$

- $\mathcal{C} \subseteq \gamma_{\mathcal{V}_B^\#}(v_i)$ for all i : the first i positions are exact, the tail in $[i + 1, +\infty]$.
- $v_{i+1} \sqsubseteq v_i$: one more exact position and a tighter summary.

One set breaks it. Take $\mathcal{C} = \{[1], [1, 2], [1, 2, 3], \dots\}$ and, for every $i \geq 1$,

$$v_i = ([1, +\infty], \langle [1, 1] \dots [i, i] \rangle, [i + 1, +\infty], \text{AttrEmpty})$$

- $\mathcal{C} \subseteq \gamma_{\nu_B^\#}(v_i)$ for all i : the first i positions are exact, the tail in $[i + 1, +\infty]$.
- $v_{i+1} \sqsubseteq v_i$: one more exact position and a tighter summary.
- Any upper bound with k exact positions is refined by v_{k+1} : hence no minimum.

One set breaks it. Take $\mathcal{C} = \{[1], [1, 2], [1, 2, 3], \dots\}$ and, for every $i \geq 1$,

$$v_i = ([1, +\infty], \langle [1, 1] \dots [i, i] \rangle, [i + 1, +\infty], \text{AttrEmpty})$$

- $\mathcal{C} \subseteq \gamma_{\nu_B^\#}(v_i)$ for all i : the first i positions are exact, the tail in $[i + 1, +\infty]$.
- $v_{i+1} \sqsubseteq v_i$: one more exact position and a tighter summary.
- Any upper bound with k exact positions is refined by v_{k+1} : hence no minimum.
- The meet would need $\langle [1, 1][2, 2] \dots \rangle$, which no finite tuple holds: no Galois connection.

One set breaks it. Take $\mathcal{C} = \{[1], [1, 2], [1, 2, 3], \dots\}$ and, for every $i \geq 1$,

$$v_i = ([1, +\infty], \langle [1, 1] \dots [i, i] \rangle, [i + 1, +\infty], \text{AttrEmpty})$$

- $\mathcal{C} \subseteq \gamma_{\nu_B^\#}(v_i)$ for all i : the first i positions are exact, the tail in $[i + 1, +\infty]$.
- $v_{i+1} \sqsubseteq v_i$: one more exact position and a tighter summary.
- Any upper bound with k exact positions is refined by v_{k+1} : hence no minimum.
- The meet would need $\langle [1, 1][2, 2] \dots \rangle$, which no finite tuple holds: no Galois connection.

A lattice, not complete: the chain has no infimum, so $\alpha_{\nu_B^\#}(\mathcal{C})$ does not exist. A lower bound fixes only finitely many positions, so the infimum would be $\mathcal{C}$ itself.

6. Abstract Operators

Every abstract operator belongs to one of three families:

- **Data movement**: concatenation and recycling put vectors together or repeat them.
- **Access**: selection reads positions, update writes them.
- **Metadata**: coercion changes the type, attributes annotate the vector.

Every abstract operator belongs to one of three families:

- **Data movement**: concatenation and recycling put vectors together or repeat them.
- **Access**: selection reads positions, update writes them.
- **Metadata**: coercion changes the type, attributes annotate the vector.

Three problems generate every difficulty:

- The zero index is silently dropped.
- Index values are known only at run time.
- A negative index says what is not there.

The type is a finite component: the chain **Bool** < **Int** < **Float**, and the typed domain is $\mathcal{T} \times \mathcal{V}_B^\#$.

$$\begin{aligned}\tau([\text{TRUE}, 3\text{L}, 2.5]) &= (\{\text{Bool}\} \sqcup \{\text{Int}\}) \sqcup \{\text{Float}\} \\ &= \{\text{Int}\} \sqcup \{\text{Float}\} = \{\text{Float}\}\end{aligned}$$

Three arguments, two joins: the abstract type is **Float**.

The type is a finite component: the chain **Bool** < **Int** < **Float**, and the typed domain is $\mathcal{T} \times \mathcal{V}_B^\#$.

$$\begin{aligned}\tau([\text{TRUE}, 3\text{L}, 2.5]) &= (\{\mathbf{Bool}\} \sqcup \{\mathbf{Int}\}) \sqcup \{\mathbf{Float}\} \\ &= \{\mathbf{Int}\} \sqcup \{\mathbf{Float}\} = \{\mathbf{Float}\}\end{aligned}$$

Three arguments, two joins: the abstract type is **Float**.

Attributes are a (**MustHave**, **MayHave**) pair: $\text{setAttr}^\#$ pins it to (A, A) , $\text{getAttr}^\#$ projects.

Example

`m <- matrix(1:2, nrow = 1)` gives dim and class:

$$v = ([2, 2], \langle [1, 1][2, 2] \rangle, \perp_{\text{Itv}}, (\{\mathbf{dim}, \mathbf{class}\}, \{\mathbf{dim}, \mathbf{class}\}))$$

Concatenation joins the lengths and concatenates the known contents.

Example

$$v_1 = ([1, 2], \langle [0, 0][1, 1] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$$

$$v_2 = ([1, 1], \langle [42, 42] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$$

Concatenation joins the lengths and concatenates the known contents.

Example

$$v_1 = ([1, 2], \langle [0, 0][1, 1] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$$

$$v_2 = ([1, 1], \langle [42, 42] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$$

The concretizations are $\gamma(v_1) = \{[0], [0, 1]\}$ and $\gamma(v_2) = \{[42]\}$; the possible concatenations are $\{[0, 42], [0, 1, 42]\}$.

Concatenation joins the lengths and concatenates the known contents.

Example

$$v_1 = ([1, 2], \langle [0, 0][1, 1] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$$

$$v_2 = ([1, 1], \langle [42, 42] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$$

The concretizations are $\gamma(v_1) = \{[0], [0, 1]\}$ and $\gamma(v_2) = \{[42]\}$; the possible concatenations are $\{[0, 42], [0, 1, 42]\}$.

$$v_1 \oplus^{\#} v_2 = ([2, 3], \langle [0, 0][1, 42][42, 42] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$$

$$v_i = ([l_i, u_i], \langle p_{i1} \cdots p_i \mid p_i \mid \rangle, s_i, a_i)$$

$$\text{Recycle}^\# : \mathcal{V}_B^\# \times \mathcal{V}_B^\# \rightarrow \mathcal{V}_B^\# \times \mathcal{V}_B^\#$$

The resulting recycled length is $[l', u'] = [\max(l_1, l_2), \max(u_1, u_2)]$.

$$v'_i = \left([l', u'], \bigcup_{d=l'}^{\max(|\mathbf{p}_1|, |\mathbf{p}_2|)} \rho_f^\#(\mathbf{p}_i, s_i, l_i, d), s'_i, a_i \right)$$

$$v_i = ([l_i, u_i], \langle p_{i_1} \cdots p_{i_{|p_i|}} \rangle, s_i, a_i)$$

$$\text{Recycle}^\# : \mathcal{V}_B^\# \times \mathcal{V}_B^\# \rightarrow \mathcal{V}_B^\# \times \mathcal{V}_B^\#$$

The resulting recycled length is $[l', u'] = [\max(l_1, l_2), \max(u_1, u_2)]$.

$$v'_i = \left([l', u'], \bigcup_{d=l'}^{\max(|p_1|, |p_2|)} \rho_f^\#(p_i, s_i, l_i, d), s'_i, a_i \right)$$

- The target length d lives in $\gamma_{\text{Itv}}([l', \max(|p_1|, |p_2|)]) \rightarrow \text{join over } \rho_f^\#$.

$$v_i = ([l_i, u_i], \langle p_{i_1} \cdots p_{i_{|p_i|}} \rangle, s_i, a_i)$$

$$\text{Recycle}^\# : \mathcal{V}_B^\# \times \mathcal{V}_B^\# \rightarrow \mathcal{V}_B^\# \times \mathcal{V}_B^\#$$

The resulting recycled length is $[l', u'] = [\max(l_1, l_2), \max(u_1, u_2)]$.

$$v'_i = \left([l', u'], \bigsqcup_{d=l'}^{\max(|p_1|, |p_2|)} \rho_f^\#(p_i, s_i, l_i, d), s'_i, a_i \right)$$

- The target length d lives in $\gamma_{\text{Itv}}([l', \max(|p_1|, |p_2|)]) \rightarrow$ join over $\rho_f^\#$.
- At a fixed d , the original length ranges over $[l_i, u_i]$, so $i \bmod k$ changes $\rightarrow$ join over $\rho_c^\#$.

Example

Recycle $\mathbf{p} = \langle [1, 1][2, 2] \rangle$ to length 3, with original length 1 or 2:

$$\rho_f^\sharp(\mathbf{p}, \perp_{\text{Itv}}, 1, 3) = \rho_c^\sharp(\mathbf{p}, 1, 3) \sqcup_{\text{Itv}^*} \rho_c^\sharp(\mathbf{p}, 2, 3)$$

Example

Recycle $\mathbf{p} = \langle [1, 1][2, 2] \rangle$ to length 3, with original length 1 or 2:

$$\begin{aligned}\rho_f^\sharp(\mathbf{p}, \perp_{\text{Itv}}, 1, 3) &= \rho_c^\sharp(\mathbf{p}, 1, 3) \sqcup_{\text{Itv}^*} \rho_c^\sharp(\mathbf{p}, 2, 3) \\ &= \langle [1, 1][1, 1][1, 1] \rangle \sqcup_{\text{Itv}^*} \langle [1, 1][2, 2][1, 1] \rangle\end{aligned}$$

Example

Recycle $\mathbf{p} = \langle [1, 1][2, 2] \rangle$ to length 3, with original length 1 or 2:

$$\begin{aligned}\rho_f^\sharp(\mathbf{p}, \perp_{\text{Itv}}, 1, 3) &= \rho_c^\sharp(\mathbf{p}, 1, 3) \sqcup_{\text{Itv}^*} \rho_c^\sharp(\mathbf{p}, 2, 3) \\ &= \langle [1, 1][1, 1][1, 1] \rangle \sqcup_{\text{Itv}^*} \langle [1, 1][2, 2][1, 1] \rangle = \langle [1, 1][1, 2][1, 1] \rangle\end{aligned}$$

$$\mathbf{Select}^\# : \mathcal{V}_B^\# \times \mathcal{V}_{\text{ltv}}^\# \rightarrow \mathcal{V}_B^\#$$

R forbids mixed indices: the sign of the selector chooses the branch.

Positive $\mathbf{Select}_{\text{pos}}^\#$

Indices ≥ 0 or NA. Extracts
by position.

Negative $\mathbf{Select}_{\text{neg}}^\#$

Indices < 0 . Excludes by
position.

Logical $\mathbf{Select}_{\text{log}}^\#$

Mask in $\{\mathbf{NA}, \mathbf{True}, \mathbf{False}\}^+$.
Selects where true.

The selector is filtered by the sign of its concretization: **abstract filtering**, the same move as filtering a guard by the branch condition.

$$\mathbf{p}_2^+ = \langle p \in \mathbf{p}_2 \mid \gamma_{\text{ltv}}(p) \subseteq \mathbb{Z}_{\geq 0} \cup \{\mathbf{NA}\} \rangle$$

$$\mathbf{p}_2^- = \langle p \in \mathbf{p}_2 \mid \gamma_{\text{ltv}}(p) \subseteq \mathbb{Z}_{< 0} \rangle$$

$$\mathbf{Select}^\#(v_1, v_2) = \mathbf{Select}_{\text{pos}}^\#(v_1, v_2^+) \sqcup \mathbf{Select}_{\text{neg}}^\#(v_1, v_2^-)$$

$$\text{Select}^\# : \mathcal{V}_B^\# \times \mathcal{V}_{\text{Itv}}^\# \rightarrow \mathcal{V}_B^\#$$

R forbids mixed indices: the sign of the selector chooses the branch.

Positive $\text{Select}_{\text{pos}}^\#$

Indices ≥ 0 or NA. Extracts
by position.

Negative $\text{Select}_{\text{neg}}^\#$

Indices < 0 . Excludes by
position.

Logical $\text{Select}_{\text{log}}^\#$

Mask in $\{\text{NA}, \text{True}, \text{False}\}^+$.
Selects where true.

The selector is filtered by the sign of its concretization: **abstract filtering**, the same move as filtering a guard by the branch condition.

$$\mathbf{p}_2^+ = \langle p \in \mathbf{p}_2 \mid \gamma_{\text{Itv}}(p) \subseteq \mathbb{Z}_{\geq 0} \cup \{\text{NA}\} \rangle$$

$$\mathbf{p}_2^- = \langle p \in \mathbf{p}_2 \mid \gamma_{\text{Itv}}(p) \subseteq \mathbb{Z}_{< 0} \rangle$$

$$\text{Select}^\#(v_1, v_2) = \text{Select}_{\text{pos}}^\#(v_1, v_2^+) \sqcup \text{Select}_{\text{neg}}^\#(v_1, v_2^-)$$

AdjustForZeros handles the zero index: it drops definite zeros and joins the indices that may land on the same position. A position too wide to enumerate collapses the source to **Squash**(v_1).

Example

$$v_1 = ([2, 3], \langle [10, 10][20, 20][30, 30] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$$

$$v_2 = ([2, 3], \langle [1, 2][0, 1][1, 100] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$$

$$\text{AdjustForZeros}(v_2) = ([1, 3], \langle [1, 2][0, 100][1, 100] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$$

$$\begin{aligned} \text{Select}_{\text{pos}}^{\#}(v_1, v_2) = ([1, 3], \\ \langle [10, 20], [10, 30] \sqcup \alpha_{\text{Itv}}(\text{NA}), [10, 30] \sqcup \alpha_{\text{Itv}}(\text{NA}) \rangle, \perp_{\text{Itv}}, \text{AttrEmpty}) \end{aligned}$$

Position 1 joins 1 and 2: $[10, 20]$; positions 2 and 3 are too wide to enumerate, so each is **Squash**(v_1) $\sqcup \alpha_{\text{Itv}}(\text{NA})$.

Negative selection excludes: when a possibly-deleted set covers a range, the survivors cannot be listed one by one, so they collapse to **SquashExcept**(v_1 , **MustDeleted**).

Example

$$v_1 = ([4, 4], \langle [10, 10][20, 20][30, 30][40, 40] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$$

$$v_2 = ([2, 2], \langle [-1, -1][-3, -3] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$$

$$\text{MustDeleted} = \{1, 3\}, \quad \text{MayDeleted} = \{1, 3\}, \quad \text{MustNotDeleted} = \{2, 4\}$$

$$\text{Select}_{\text{neg}}^{\#}(v_1, v_2) = ([2, 2], \langle [20, 20][40, 40] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$$

Logical selection reads a boolean mask.

Example

$$v_1 = ([4, 4], \langle [10, 10][20, 20][30, 30][40, 40] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$$

$$v_2 = ([4, 4], \langle \text{True}, \text{False}, \text{NA}, \text{True} \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$$

$$\text{Select}_{\log}^{\#}(v_1, v_2) = ([3, 3], \langle [10, 10], \alpha_{\text{Itv}}(\text{NA}), [40, 40] \rangle, \perp_{\text{Itv}}, \text{AttrEmpty})$$

The mask is recycled with $\rho_f^{\#}$: **True** selects, **False** skips, NA gives $\alpha_{\text{Itv}}(\text{NA})$; the length subtracts the definite **False**.

Both are the positive operator plus a set of positions: the complement of the deleted set, or the positions where the mask is true.

Update: the positive case and its cost

Update is selection plus a write: same filtering, same AdjustForZeros.

- Non-enumerable selector: collapse to **Squash**(v_1) $\sqcup$ **Squash**(v_3); length unbounded.
- Infinite, enumerable: weak update; a position may be written twice.
- Finite: extend to the largest index with NA, then write.

Example

$$v_1 = ([5, 5], \langle [1, 1] \dots [5, 5] \rangle)$$

$$v_2 = ([1, 1], \langle [10, 10] \rangle), \quad v_3 = ([1, 1], \langle [42, 42] \rangle)$$

$$\text{Update}_{\text{pos}}^{\#}(v_1, v_2, v_3) = ([10, 10], \\ \langle [1, 1] \dots [5, 5] \alpha_{\text{ltv}}(\text{NA}) \dots \alpha_{\text{ltv}}(\text{NA}) [42, 42] \rangle)$$

Ten positions, four NA. An update at index i materialises i positions, and widening closes the tail.

Negative update has three sets: not updated for sure, may not be updated, updated for sure. The third writes strongly; the length does not change ($v[-2] \leftarrow 5$ writes, it does not delete).

Example

$$v_1 = ([4, 4], \langle [10, 10][20, 20][30, 30][40, 40] \rangle)$$

$$v_2 = ([1, 1], \langle [-2, -2] \rangle), \quad v_3 = ([1, 1], \langle [99, 99] \rangle)$$

$$\text{Update}_{\text{neg}}^{\#}(v_1, v_2, v_3) = ([4, 4], \\ \langle [10, 10][99, 99][30, 30][40, 40] \rangle)$$

Logical update recycles mask and values: true writes strongly, maybe-true weakly, false keeps.

Example

$$v_1 = ([4, 4], \langle [10, 10][20, 20][30, 30][40, 40] \rangle)$$

$$v_2 = ([4, 4], \langle \text{True}, \text{False}, \text{True}, \text{False} \rangle)$$

$$\text{Update}_{\log}^{\#}(v_1, v_2, \langle [99, 99] \rangle) = ([4, 4], \\ \langle [99, 99][20, 20][99, 99][40, 40] \rangle)$$

The one refusal: if the mask may contain NA and the replacement may be longer than 1 ($u_3 \neq 1$), the concrete program is an error and the abstract returns $\perp_{\mathcal{V}_B^{\#}}$.

Update is selection plus a write; the difficulty is knowing where.

7. Implementation and Validation

The domain is integrated in **flowR**. The orange stage is the contribution of this work:

- Worklist fixpoint: join at merge points, widening $\nabla_{\mathcal{V}_B^\#}$ at loops.
- The visitor decomposes the state into length, known content, summary, attributes and type, then dispatches every CFG vertex to the corresponding abstract operator.
- Outside μR it falls back to $\mathcal{T}_{\mathcal{V}_B^\#}$: sound on μR , conservative elsewhere.



Recycling

```
if (*) v <- c(1, 2, 3) else v <- c(0, 1)
r <- v + c(9, 10, 11)
 $\hookrightarrow r = ([3, 3], \langle [9, 10], [11, 12], [11, 14] \rangle, \perp_{\text{ltv}})$ 
```

Uncertainty

```
v <- *; s <- c(1, 2, 3)
x <- s[v]; y <- x[c(1, 2)]
 $\hookrightarrow * = \top_{\mathcal{V}_{\text{ltv}}^\#}, y = ([2, 2], \langle [1, 3] \sqcup_{\text{ltv}} \alpha_{\text{ltv}}(\text{NA}), [1, 3] \sqcup_{\text{ltv}} \alpha_{\text{ltv}}(\text{NA}) \rangle, \perp_{\text{ltv}})$ 
```

Loop

```
x <- c(1, 2, 3); i <- 1
while (*) { x <- c(x, i); i <- i + 1 }; r <- x
 $\hookrightarrow r = ([3, +\infty], \langle [1, 1], [2, 2], [3, 3] \rangle, [1, +\infty] \sqcup_{\text{ltv}} \alpha_{\text{ltv}}(\text{NA}))$ 
```

- **61 programs**, 6 categories: ¹⁴Creation, ⁹Recycling, ¹²Select, ¹²Update, ⁸Uncertainty, ⁶Loops.

- **61 programs**, 6 categories: ¹⁴Creation, ⁹Recycling, ¹²Select, ¹²Update, ⁸Uncertainty, ⁶Loops.
- 2 checks:
 - the computed abstract value equals the one derived by hand from the definitions;
 - where a concrete run exists, the concrete R run lies in its concretization.

- **61 programs**, 6 categories: ¹⁴Creation, ⁹Recycling, ¹²Select, ¹²Update, ⁸Uncertainty, ⁶Loops.
- 2 checks:
 - the computed abstract value equals the one derived by hand from the definitions;
 - where a concrete run exists, the concrete R run lies in its concretization.
- **Cost**: an update at index i materialises i positions; widening closes the tail.

- **61 programs**, 6 categories: ¹⁴Creation, ⁹Recycling, ¹²Select, ¹²Update, ⁸Uncertainty, ⁶Loops.
- 2 checks:
 - the computed abstract value equals the one derived by hand from the definitions;
 - where a concrete run exists, the concrete R run lies in its concretization.
- **Cost**: an update at index i materialises i positions; widening closes the tail.
- **Limits**: non-relational; no length relations between variables; no interprocedural analysis.

8. Conclusions and Open Problems

Built

- μR : the vector features we analyse, made explicit.
- A parametric domain: length, known prefix, summary, attributes.
- Six abstract operators.
- A prototype in flowR: 61 programs, 61/61.

Proved

- Partial order and meet.
- Widening termination.
- Recycling and positive selection.

Empirical only

- Negative and logical selection.
- All updates.
- Attributes.

Where is each gap: domain, semantics, or engineering?

Length relations between variables: reduced product

domain

Branch sensitivity: disjunctive states at merge points

domain

Interprocedural analysis under lazy evaluation

semantics

S3/S4 dispatch modelled, not over-approximated

semantics

The remaining soundness proofs

engineering

Precision and cost on real CRAN packages

engineering

M. Di Agostino, F. Sihler, V. Arceri, O. Gerstl, M. Tichy. *Towards Statically Reasoning about R Vectors*. NSAD '26, pp. 18–25.
doi:10.1145/3840563.3843708



Slides available

Definition

Recycle

Recycle : $\mathcal{V} \times \mathcal{V} \rightarrow \mathcal{V} \times \mathcal{V}$ aligns two vectors to a common length. With $m = \max(|x|, |y|)$, $t = \tau(x) \sqcup \tau(y)$, $x' = \mathbf{Cast}(x, t)$, $y' = \mathbf{Cast}(y, t)$:

$$\mathbf{Recycle}(x, y) \stackrel{\text{def}}{=} ((\rho_c(x'.\text{data}, m), x'.\mathcal{A}), (\rho_c(y'.\text{data}, m), y'.\mathcal{A}))$$

Recycle([], x) = **Recycle**(x , []) $\stackrel{\text{def}}{=} ([], [])$; the cast precedes the cycling and the attributes are preserved.

Definition

ρ_c

$$\rho_c(x, l)(i) \stackrel{\text{def}}{=} x(((i - 1) \bmod n) + 1) \quad \text{for } i \in [1, l]$$

lengthens a data sequence of length n to a target length l : $\lfloor \frac{l}{n} \rfloor$ full copies followed by the first $l \bmod n$ elements. The empty case is handled above, in **Recycle**.

Definition

Select, positive

$$\text{Select} : \mathcal{V} \times \mathcal{V}_{\text{Int}} \rightarrow \mathcal{V}$$

A **partial** function whose three modes are disjoint; **NA** is admitted only here and in the logical mode. For all $c \in \mathcal{V}_{\text{Int}}$ with $\forall i \in [1, |c|] : c(i) \geq 0 \vee c(i) = \mathbf{NA}$:

$$\text{Select}(v, c) \stackrel{\text{def}}{=} \bigoplus_{i=1}^{|c|} \sigma_i \quad \sigma_i \stackrel{\text{def}}{=} \begin{cases} [v^*(c(i))] & \text{if } c(i) > 0 \\ [] & \text{if } c(i) = 0 \\ [\mathbf{NA}] & \text{if } c(i) = \mathbf{NA} \end{cases}$$

with the out-of-bounds helper

$$v^*(i) \stackrel{\text{def}}{=} \begin{cases} v(i) & \text{if } v(i) \neq \mathbf{NULL} \\ \mathbf{NA} & \text{otherwise} \end{cases}$$

which turns the **NULL** a total map stores past the length into **NA**.

Definition

Negative

For all $c \in \mathcal{V}_{Int}$ with $\forall i \in [1, |c|] : c(i) \leq 0 \wedge c(i) \neq \mathbf{NA}$:

$$\mathbf{Select}(v, c) \stackrel{\text{def}}{=} [v(u_1), \dots, v(u_k)]$$

with the exclusion set $O = \{-c(1), \dots, -c(|c|), 0\}$, and $(u_1, \dots, u_k)$ the increasing sequence of $[1, |v|]$ without the elements of O .

Definition

Logical

For all $c \in \mathcal{V}_{Bool}$, with $c' \stackrel{\text{def}}{=} \rho_c(c, \max(|v|, |c|))$ (so a **True** past $|v|$ gives **NA**):

$$\mathbf{Select}(v, c) \stackrel{\text{def}}{=} \bigoplus_{i=1}^{|c'|} \sigma_i \quad \sigma_i \stackrel{\text{def}}{=} \begin{cases} [v^*(i)] & \text{if } c'(i) = \mathbf{True} \\ [] & \text{if } c'(i) = \mathbf{False} \\ [\mathbf{NA}] & \text{if } c'(i) = \mathbf{NA} \end{cases}$$

Notation

Positional update

$$r[u_i \mapsto w_i \mid i \in \{1, \dots, k\}]$$

r with the value at each u_i replaced by w_i ; positions not listed are unchanged.

Definition

Positive

For all $c \in \mathcal{V}_{Int}$ with $\forall i \in [1, |c|] : c(i) \geq 0 \vee c(i) = \mathbf{NA}$, the update is $\mathbf{Update}(v, c, v_e) \stackrel{\text{def}}{=} r[u_i \mapsto v'_e(i)]$ with

$$C \stackrel{\text{def}}{=} \max\{c(i) \mid c(i) \neq \mathbf{NA}\} \quad r \stackrel{\text{def}}{=} v \oplus \underbrace{[\mathbf{NA}, \dots, \mathbf{NA}]}_{\substack{\max(C - |v|, 0) \\ \text{times}}}$$

$v'_e \stackrel{\text{def}}{=} \rho_c(v_e, k)$ where $k = |\{i \in [1, |c|] \mid c(i) \neq 0\}|$, and $(u_1, \dots, u_k)$ are the non-zero values of c in increasing order. With $v = [10, 20, 30]$, $\mathbf{Update}(v, [5], [99]) = [10, 20, 30, \mathbf{NA}, 99]$.

Definition

Negative

For all $c \in \mathcal{V}_{Int}$ with $\forall i : c(i) \leq 0 \vee c(i) = \mathbf{NA}$: $\mathbf{Update}(v, c, v_e) \stackrel{\text{def}}{=} v[u_i \mapsto v'_e(i)]$ with $v'_e \stackrel{\text{def}}{=} \rho_c(v_e, k)$, where $(u_1, \dots, u_k)$ is $[1, |v|]$ without the elements of $O = \{-c(1), \dots, -c(|c|), 0\}$; the length of v is invariant.

Definition

Logical

For all $c \in \mathcal{V}_{Bool}$ with $\forall i : c(i) \neq \mathbf{NA} \vee |v_e| = 1$, with $c' \stackrel{\text{def}}{=} \rho_c(c, \max(|v|, |c|))$ and $(u_1, \dots, u_k)$ the positions where c' is **True** ($k = |u|$), the update is

$$\mathbf{Update}(v, c, v_e) \stackrel{\text{def}}{=} r[u_i \mapsto v'_e(i)] \quad v'_e \stackrel{\text{def}}{=} \rho_c(v_e, k)$$

where r extends v with $\max(\max\{u_i\} - |v|, 0)$ copies of **NA**. The padding is $\max\{u_i\} - |v|$, **not** $k - |v|$: for $v = [1, 2]$ and mask **[False, False, False, True]** R gives $[1, 2, \mathbf{NA}, 9]$, while $k - |v| = 0$ would leave position 3 as **NULL**.