Abstract Interpretation

The cooler AI

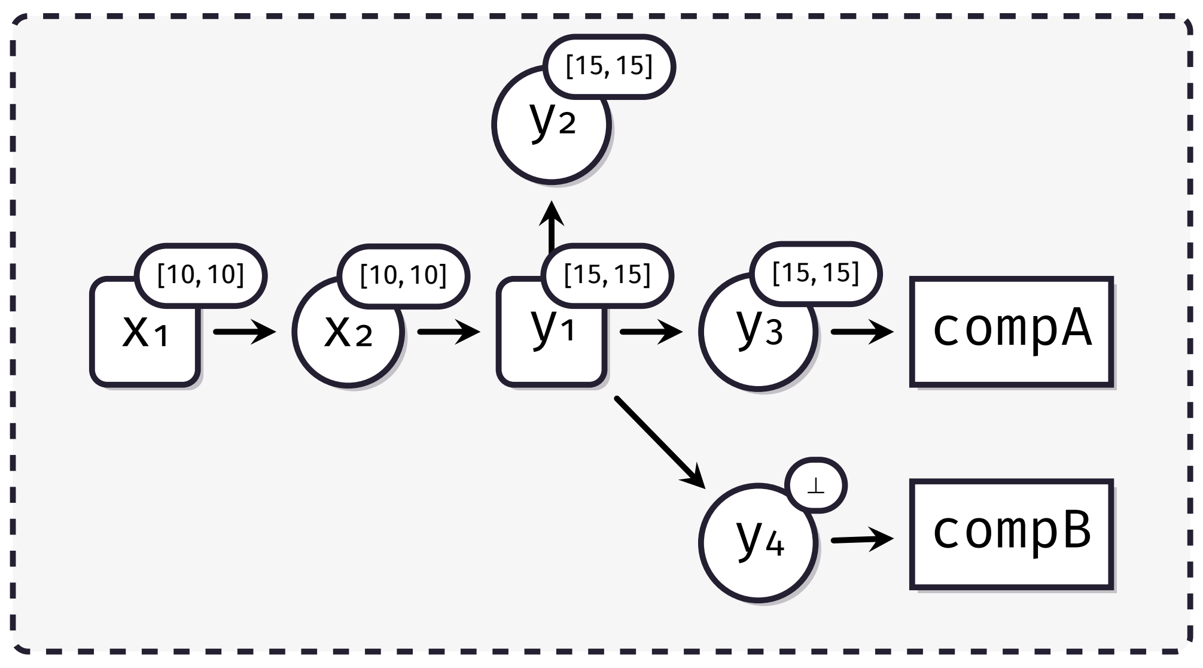
Lukas Pietzschmann, Florian Sihler

Join us and our

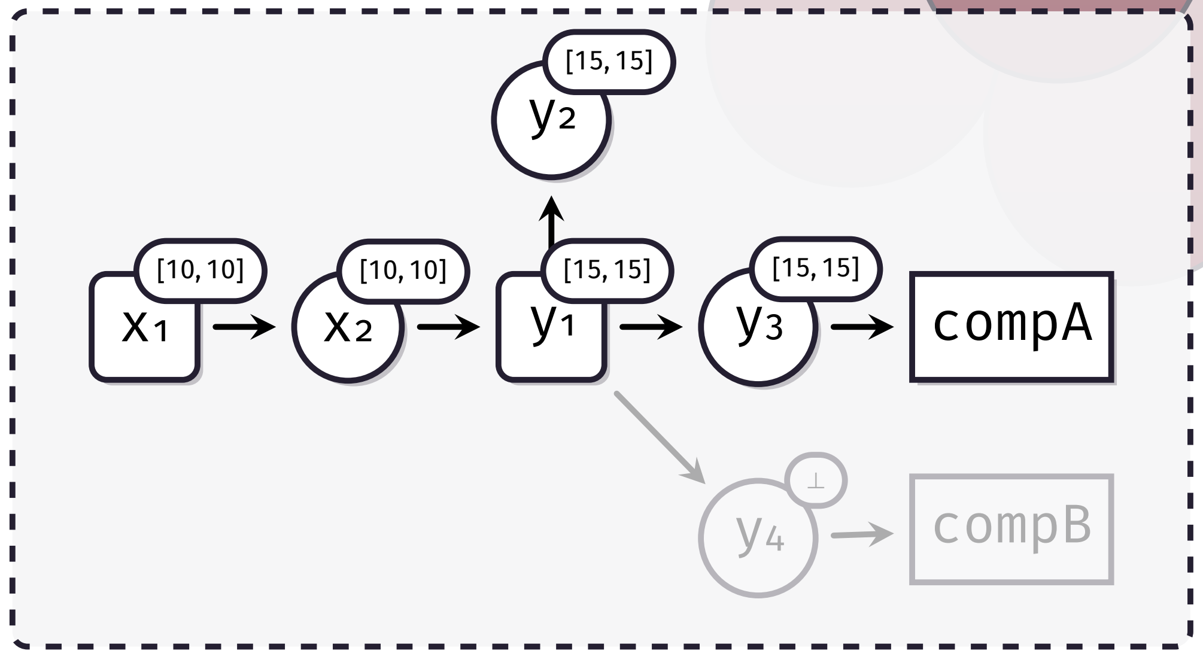
flowR-power

```
x1 ← 10
y1 ← 5 + x2
if(y2 > 10) {
    compA(y3)
} else {
    compB(y4)
}
```

Step 1 To set everything up, we retrieve the code, get its abstract syntax tree, normalize it, and finally build and return the dataflow graph. This graph will then be used as a base for the next step.



Step 2 In the second step, the actual abstract interpretation happens. This step gradually decorates the dataflow graph with the domains of the respective variables, by traversing the control flow graph.



Step 3 The last step uses the decorated dataflow graph to determine and track different properties of the original code. We can, e.g., catch endless loops, or, as shown above, ignore unreachable nodes in the dataflow graph.

How Abstract Interpretation fits into *flowR*

flowR *flowR* is a static *dataflow analyzer* and *program slicer* for the R language. Program slicers reduce a given program to a set of statements — the so called slices — that influence a variable at a specific point in the program: the slicing criterion. *flowR* formulates the slicing as a reachability problem on the dataflow graph, which itself is based on the abstract syntax tree of the R program.

Abstract Interpretation Abstract interpretation is all about figuring out *runtime properties* of a given program without actually executing it. These properties include aspects like the sign or nullness of a variable. For now we decided to limit ourselves to determining the *domain of numeric variables*. In other words, we want to know what values a variable can have at a certain point in the program.

Abstract Interpretation in *flowR* Abstract interpretation helps *flowR* to build a more precise dataflow graph by removing certain paths if we can prove that they will never be executed. Imagine a conditional like `if(i < 10) doit() else run()`. If, through abstract interpretation, we know that `i`'s value is always less than `10`, there's no need to include the path stemming from the `if`'s else-case in the slice.

Abstract Interpretation

let $a \in [0, 200] \subset \mathbb{N}_0$
let $b \in [5, 10] \subset \mathbb{N}_0$

```
1 if (a < 100) {
2     a ← a + 1
3 } else {
4     a ← a + 1
5     b ← b + 2
6 }
7 a; b
```

Fig. 1 The source code we want to analyze.

	a	b
start	$[0, 200]$	$[5, 10]$
1	$[0, 99]$	$[5, 10]$
2	$[1, 100]$	$[5, 10]$
3	$[100, 200]$	$[5, 10]$
4	$[101, 201]$	$[5, 10]$
5	$[101, 201]$	$[7, 12]$
7	$[1, 201]$	$[5, 12]$

Fig. 2 The computed domain for both variables at every line.

We assume that the integer variables `a` and `b` have both been defined and initialized elsewhere. When entering the snippet shown above, their values are assumed to fall within the domains $[0, 200]$ and $[5, 10]$. We can read this as follows: `a`'s value will always be greater or equal to 0, and less than or equal to 200.

We then start by traversing the control flow graph in execution order, processing each element of the original program. We're especially interested in (1) operations that change the value of a variable, (2) assignments, and (3) structures that influence the control flow. When encountering an expression of type (1), like addition, subtraction, and other arithmetic operations, we merge the operands' domains according to the given operation. We can observe this in line 2, where `a`'s value was previously determined to be between 0 and 99. Adding 1 to `a` modified its domain by incrementing both the lower and upper bound to 1 and 100. When we encounter an assignment, we use the domain of the assignments source expression and map it to the target variable. We can also see this in line 2, where the domain of `a + 1` — namely $[1, 100]$ — is set as `a`'s new domain. Lastly, if we encounter a structure of type (3), like conditionals or loops, we check how the condition narrows down a variable's domain. In the `if-else` above, it is evident that the condition `a < 100` will always hold in the `if`'s then-case, while conversely, it does not hold in its else-case. In example above, we can use this fact to lower the domains upper bound to the conditions upper bound — namely 99. Figure 2 shows this in line 1, where `a`'s domain $[0, 200]$ got narrowed down to $[0, 99]$.

Besides the domains shown above, we use two special values to indicate two cases. We use $\top$ (top) whenever the domain of a variable is unknown. Or in different words: when the variable could take every possible value. If the opposite is the case — if a variable can't take any value — we use $\bot$ (bottom).

Project Organization

Organization We're doing *weekly meetings*, where we discuss the progress made in the past week, talk about open issues, and prioritize tasks for the next week. We also keep a *record of important things* that came up in our meeting. If there's any spare time, Florian often tells me about new ~~weird~~cool things he learned about the R language.

Tech Stack *flowR* is developed with *TypeScript*, then transpiled to JavaScript with Node.js as its runtime. While we use different libraries for utilitarian tasks — like chai for assertions, mocha for tests, or tslog for logging — all major functionalities, including abstract interpretation, are implemented *by hand*, as there's a big lack of libraries offering these features for the R programming language.

QA To ensure the employment of best practices, we always do *code reviews* on pull requests. This massively helps with keeping the code easy to understand and well readable.

If an error slips through during the review, either *flowR*'s linter, or its extensive suite of over *1000 tests* will probably catch it. Additionally, we consistently introduce new tests promptly whenever there is new code in the project.

We also make heavy use of *assertions* — or how we call them: guards — whenever possible to make sure our mental model aligns with the actual execution of the code.

Documentation *flowR*'s documentation is split into two parts: (1) a *user facing* documentation hosted in a GitHub wiki and (2) a *developer facing* documentation built from inline comments.

Future Work

Well, there's still *a lot to do*! As a first step, we need to implement support for *more control structures*, *operators*, and *function calls*. When this is done and basic cases are handled well, we can start *modifying the dataflow graph* and automatically through that, narrow down the slices *flowR* produces. As a last step, there are many things that can be *optimized*, like the representation of domains. While this is definitely not mission critical, good performance is always a nice thing to have.

And all of this is only just the tip of the iceberg. Abstract interpretation allows for way more than just the tracking of value domain. Obvious extensions of this project could include the tracking of domains for different types, or pointer analysis.