

Data transformations in Vega

Author: Kristýna Petrlíková

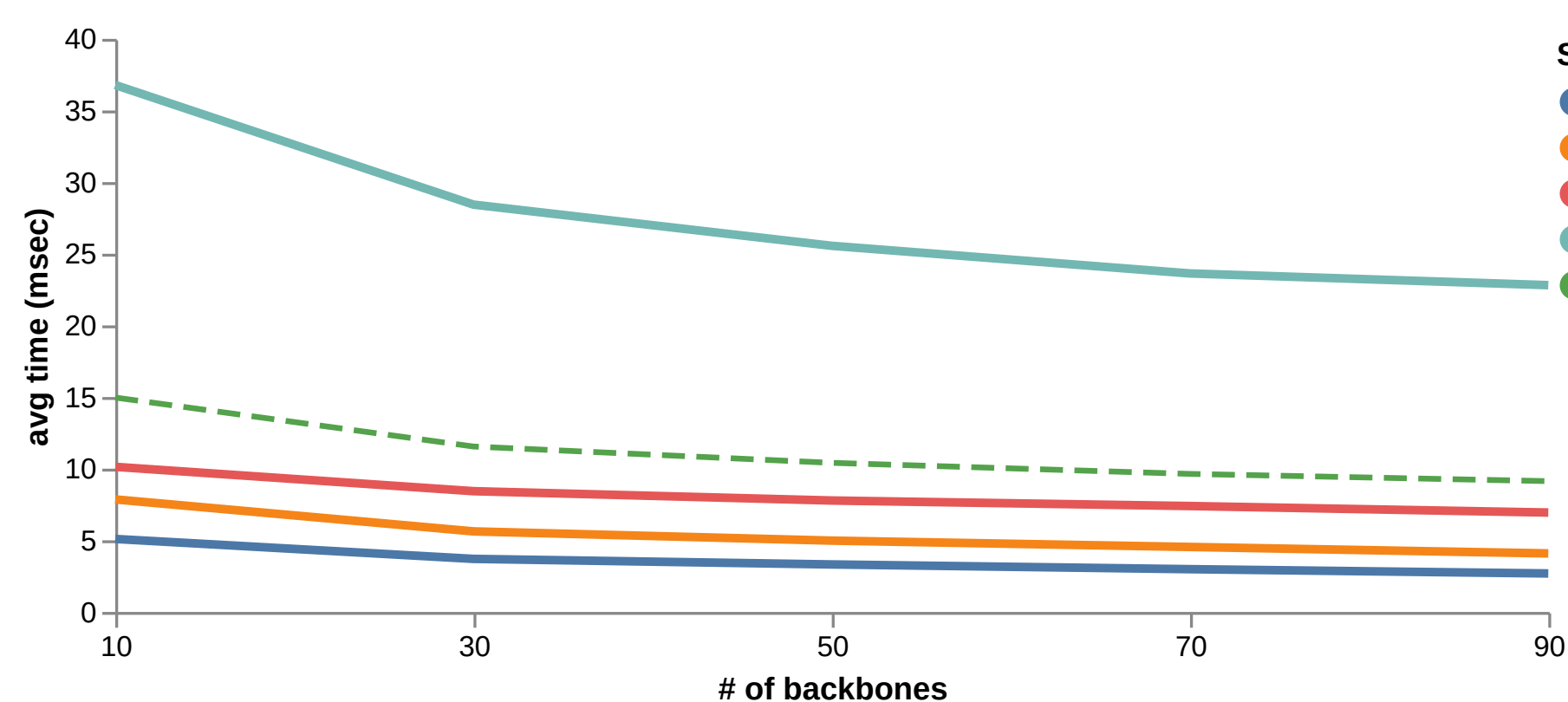
Supervisor: Mgr. Tomáš Petříček, Ph.D.
Charles University

INTRODUCTION

Vega is a popular library for producing visualizations using a declarative, JSON-embedded language. One of Vega's main issues is its lack of useful error reporting. In this work we address the issue of *invalid field access* to data sources. To clearly see where invalid field access might occur, we formalize the semantics of Vega transformations, as their usage is the main source of non-trivial field manipulation.

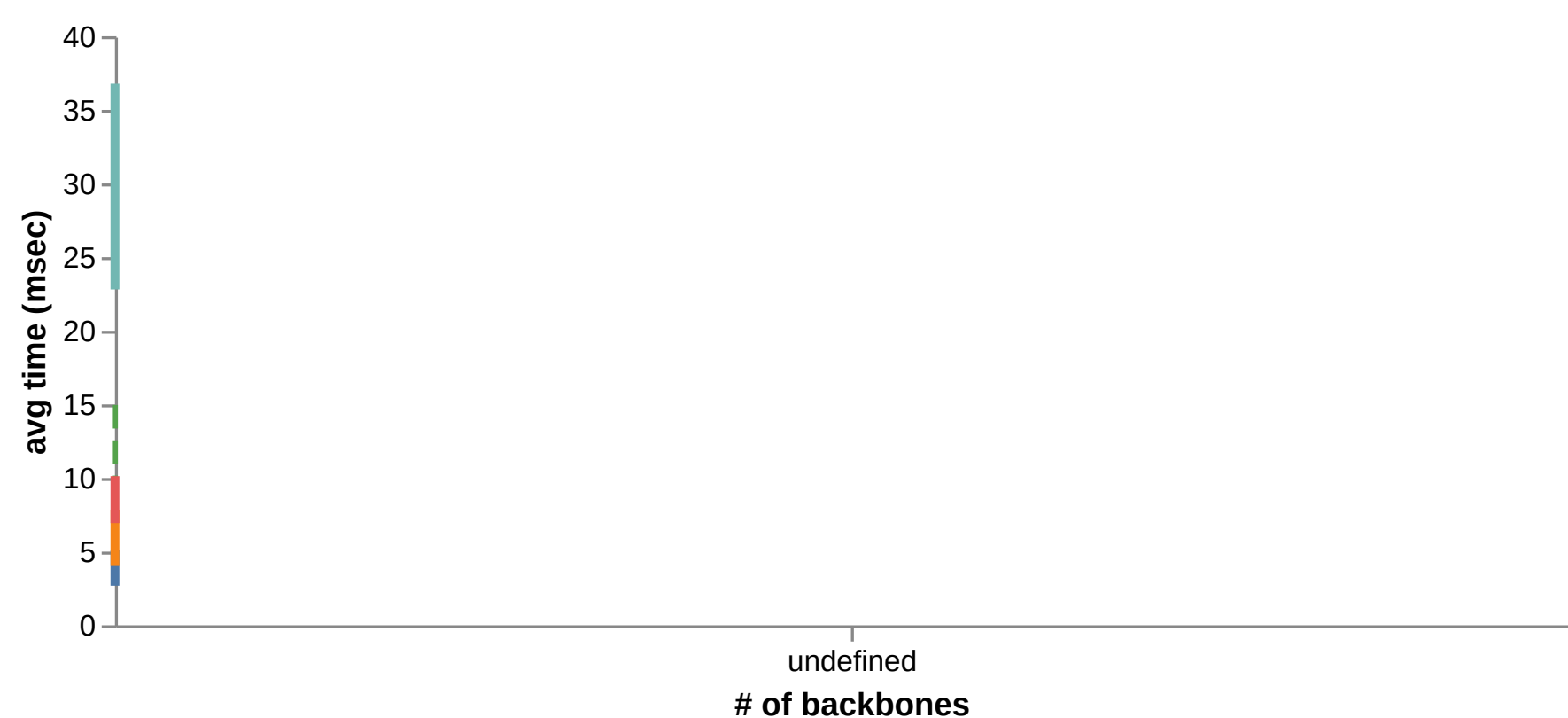
INVALID FIELD ACCESS EXAMPLE

```
"data": [
  { "name": "table1",
    // fields: solver, nbclauses,
      nbbackbones, avg_time
    "values": [...],
    "transform": [
      { "type": "filter",
        "expr": "datum.nbclauses == 411"},
      { "type": "formula",
        "expr": "datum.avg_time * 1000",
        "as": "avg_time_msec" } ] },
```



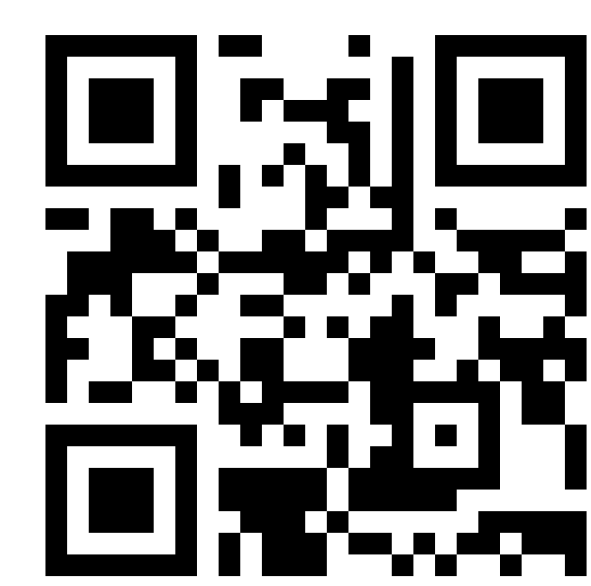
Intended visualization

```
{ "name": "table2",
  "source": "table1",
  // fields: ..., avg_time_msec
  "transform": [
    {
      "type": "aggregate",
      "groupby": ["nbbackbones"],
      "fields": ["avg_time_msec"],
      "ops": ["average"],
      "as": ["average_avg_time_msec"] } ] ],
```

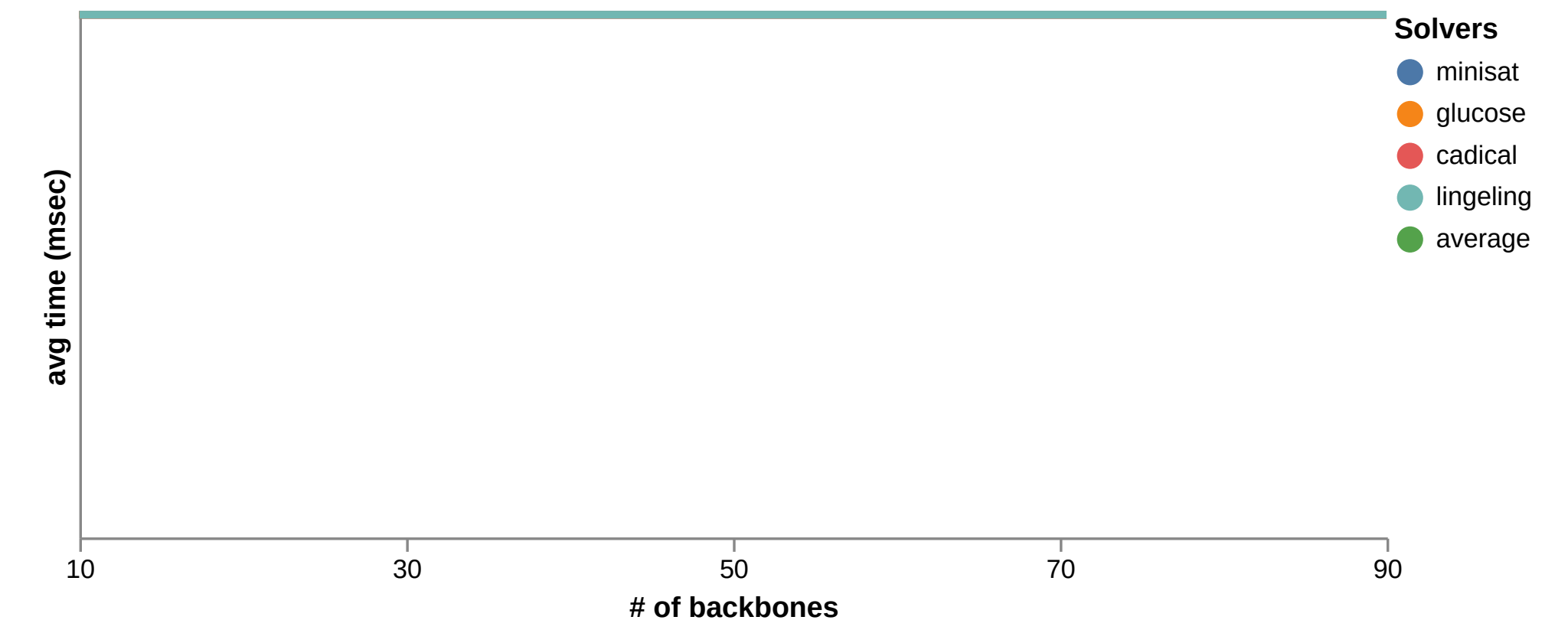


Accessing "nbbackbone" (nonexistent) in a discrete scale: **malformed visualization**, no warnings!

Full example available
in Vega editor at



tinyurl.com/vega-example



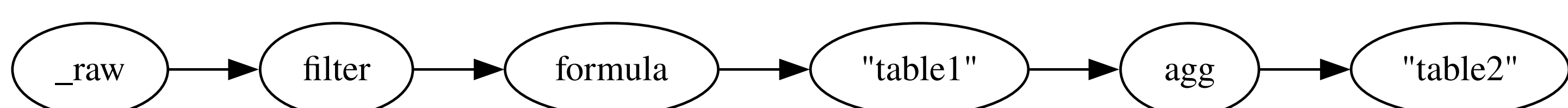
Accessing "avg_time_msec" (no longer exists) of table2 in a continuous scale: **malformed visualization**, strange warning about infinite extent of "avg_time_msec"!

ESSENCE OF VEGA

Access path	$p ::= f_1 \cdot \dots \cdot f_n$ (sequence of field accesses)
Expression	$e ::= \text{datum}.p \mid \text{arithmetics} \mid \text{array} \mid \text{object}$
Operation	$\text{op} ::= \text{sum } p_{\text{in}} \text{ as } f_{\text{out}} \mid \text{count } p_{\text{in}} \text{ as } f_{\text{out}}$
Transformation	$t ::= \text{agg}(p_1, \dots, p_n, \text{op}_1, \dots, \text{op}_k) \mid \text{cross}(f_a, f_b) \mid \text{filter}(e) \mid \text{formula}(e, f_{\text{out}})$
Dataset	$d ::= \text{data}(n_{\text{out}}, n_{\text{in}}, t_1, t_2, \dots, t_k)$
Specification	$\text{spec} ::= d_1, \dots, d_k$

OPERATIONAL SEMANTICS

- We construct a dependency graph modelling Vega's runtime Dataflow graph
 - vertices correspond to streaming data sources
 - edges represent how data flows between the vertices
- The abstract machine is defined on vertices processing streaming *events*: addition, update or removal of a data tuple, or an error
- Every vertex has a state and a list of unprocessed events
- Evaluation ends once all events have been processed



GOALS

- Formalize field manipulation in Vega by giving types to safe Vega specifications
- Formalize Vega's stream-based runtime using small-step operational semantics
- Implement a verbose proof-of-concept typechecker for verifying field access in a subset of Vega specifications

TYPE SYSTEM

$T ::= \text{primitive} \mid \{f_1 : T_1, \dots, f_n : T_n\} \mid T[n]$ (shape)

Typing judgments:

- $T \vdash e : T'$: given a datum of type T , the expression e outputs a new datum of type T' .
- $T \vdash t : T'$: given a dataset of shape T , the transform t produces a dataset of shape T' .
- $\Gamma \vdash d : T$: the dataset d has a shape T in the typing context Γ .
- $\Gamma \vdash s \Rightarrow \Gamma'$: the specification s transforms the context Γ to Γ' .

CONTRIBUTIONS

- Our formalization is *sound*: well-typed specifications don't produce an access-related error during runtime.
- We have introduced a way to *statically check for invalid field access* in a subset of Vega specifications, which leads to **more debuggable visualizations**.
- Our implementation detects nonexistent fields and points out possible sources of NaN in expressions. The Docker image is available at tinyurl.com/epsilon-lyrae.