REPL-first language design

WGLD meeting 10-6-2021

Tijs van der Storm

CWI
Centrum Wiskunde & Informatica

university of groningen

Read Eval Print Loops (REPLs)

- AKA: consoles, command-line interfaces (CLI), interactive shells

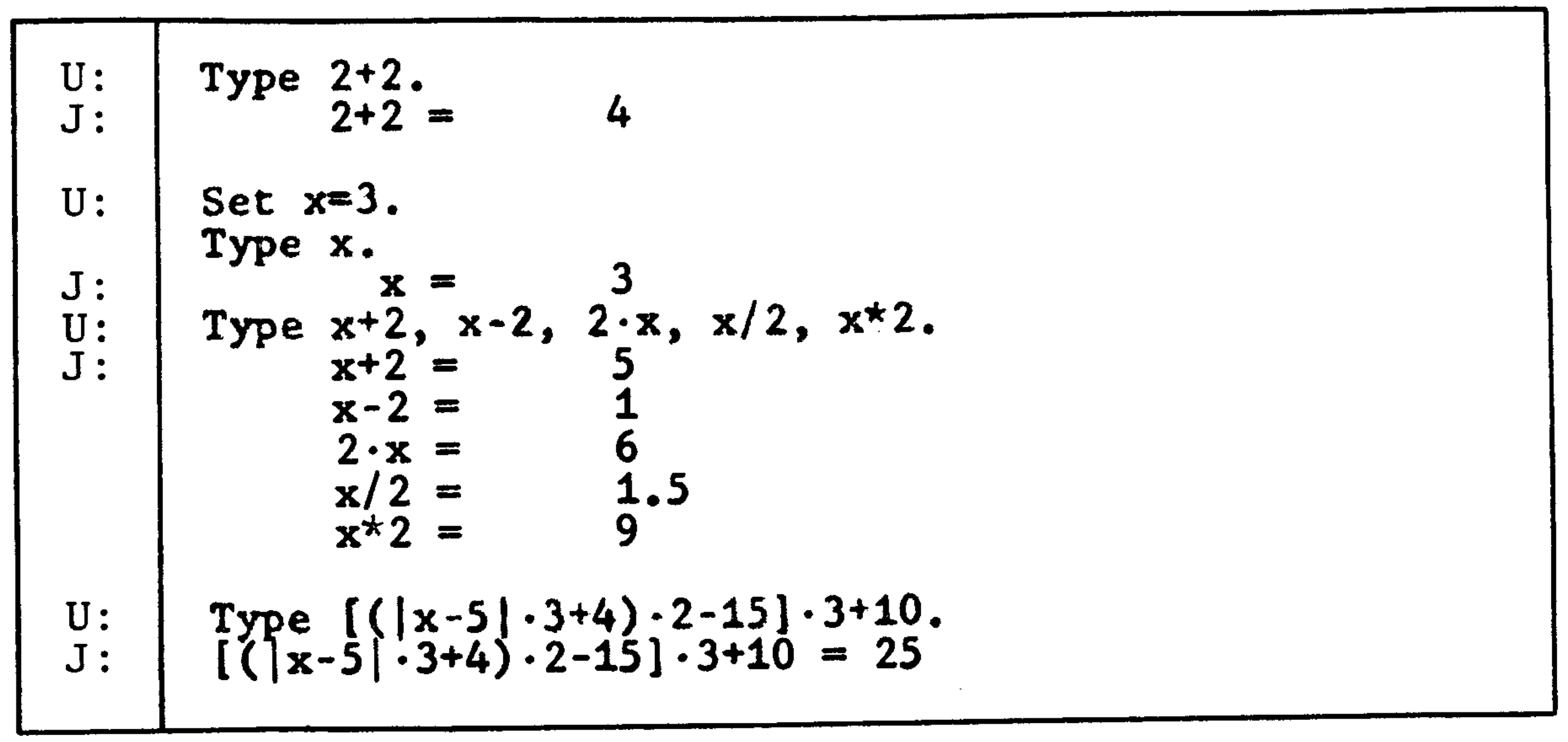
- **“Each S-expression typed in will be evaluated and its value printed out.”**
  (Peter Deutsch on PDP-1 LISP, 1964)

- Facilitate experimentation, exploration, testing, debugging

- All mainstream languages have them

  - https://www.tiobe.com/tiobe-index/

- Born again as “computational notebooks”



```
U:    Type 2+2.
J:         2+2 =       4

U:    Set x=3.
      Type x.
J:           x =       3
U:    Type x+2, x-2, 2·x, x/2, x*2.
J:         x+2 =       5
           x-2 =       1
           2·x =       6
           x/2 =       1.5
           x*2 =       9

U:    Type [(|x-5|·3+4)·2-15]·3+10.
J:    [(|x-5|·3+4)·2-15]·3+10 = 25
```

JOSS (1964)

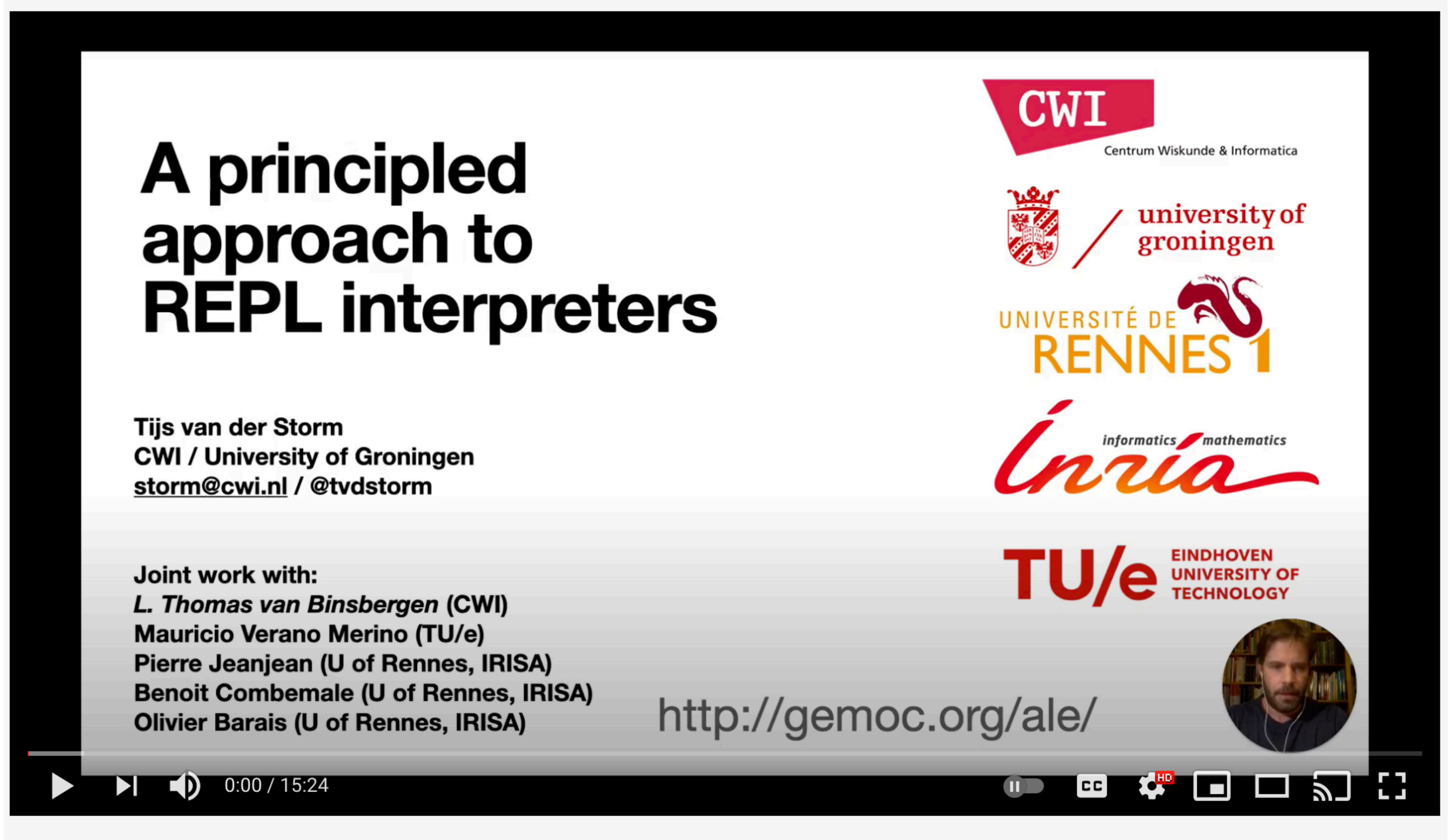
jupyter

ObservableHQ

REPL = language extension + “;”

$$[\![p_1 \fatsemi p_2]\!] = [\![p_2]\!] \circ [\![p_1]\!]$$

An associative sequence/ concatenation operator



A principled approach to REPL interpreters

Tijs van der Storm
CWI / University of Groningen
storm@cwi.nl / @tvdstorm

Joint work with:
L. Thomas van Binsbergen (CWI)
Mauricio Verano Merino (TU/e)
Pierre Jeanjean (U of Rennes, IRISA)
Benoit Combemale (U of Rennes, IRISA)
Olivier Barais (U of Rennes, IRISA)

http://gemoc.org/ale/

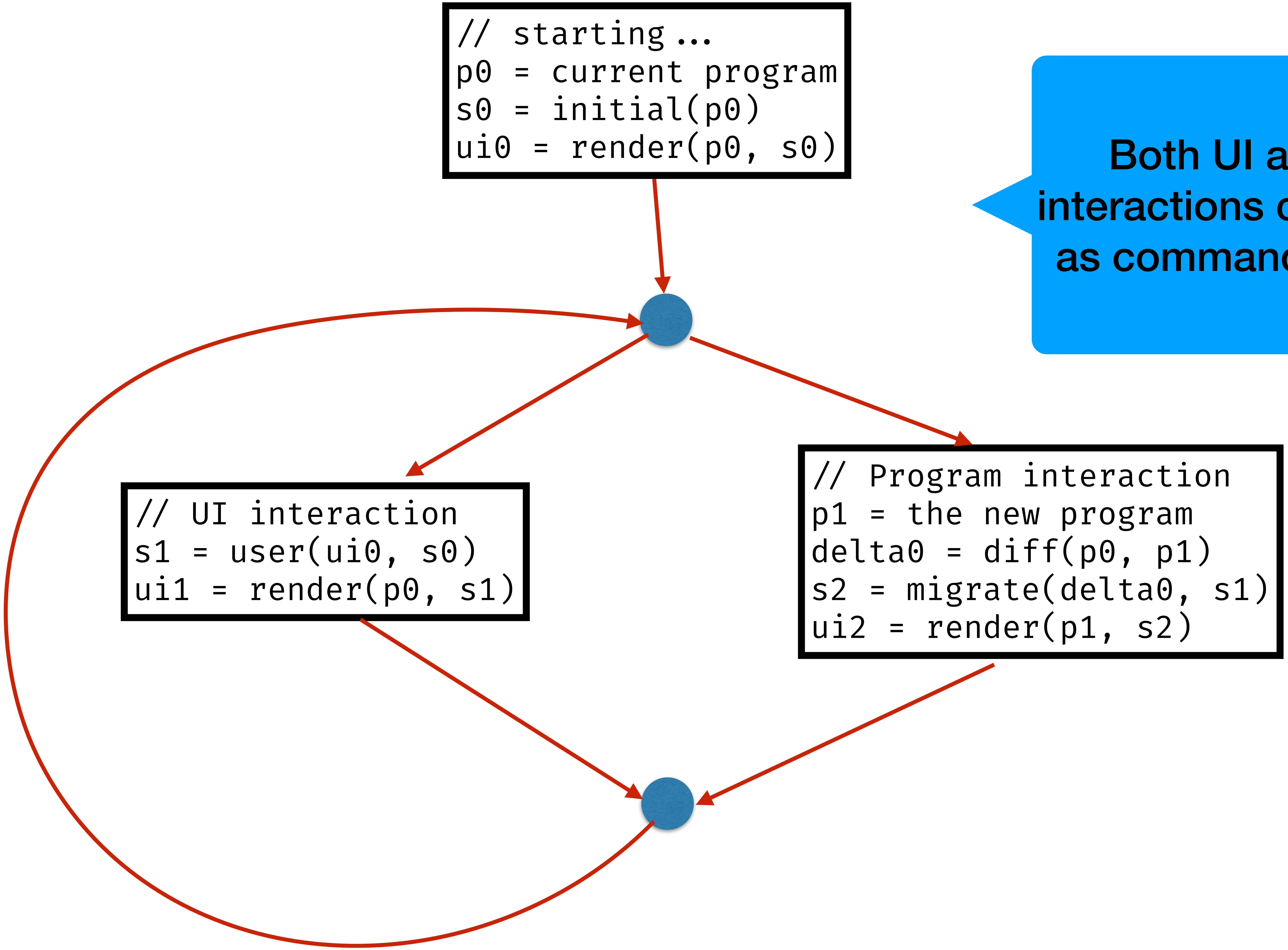
https://www.youtube.com/watch?v=xP4I2DJKHVQ

https://doi.org/10.1145/3426428.3426917

REPL = “linguistic Elm architecture”?

- Immediate mode UI programming:
  - init: Model
  - update: Msg x Model -> Model
  - view: Model -> UI
- REPL-first language design
  - init: Program x State
  - exec: Cmd x Program x State -> Program x State
  - view: Program x State -> IDE

As a result, REPL could present a unified view on editing, debugging, executing, visualization, and versioning.



```
// starting...
p0 = current program
s0 = initial(p0)
ui0 = render(p0, s0)
```

```
// UI interaction
s1 = user(ui0, s0)
ui1 = render(p0, s1)
```

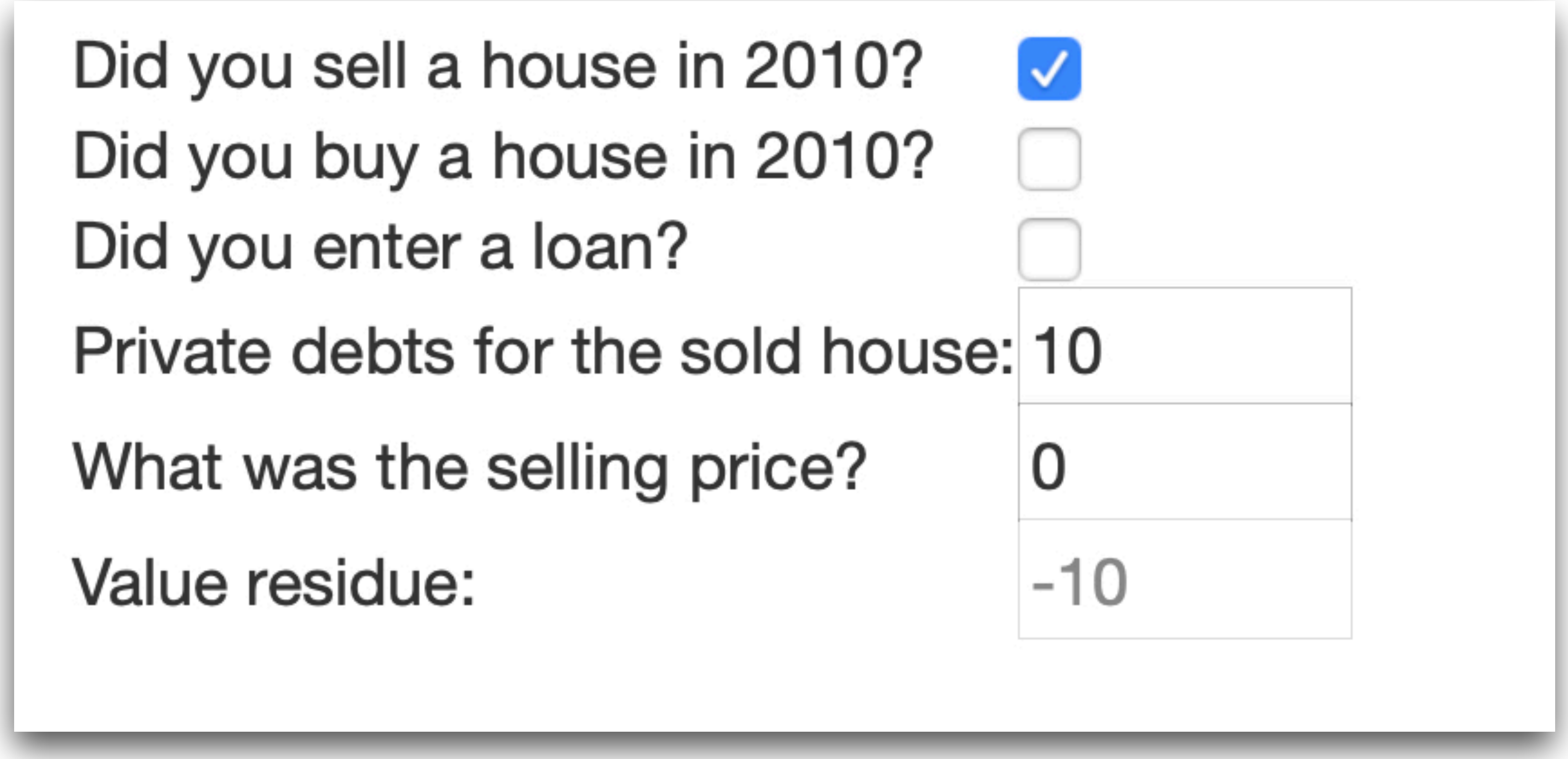
```
// Program interaction
p1 = the new program
delta0 = diff(p0, p1)
s2 = migrate(delta0, s1)
ui2 = render(p1, s2)
```

Both UI and Program interactions can be modeled as commands at the REPL

```
form taxOfficeExample {
  "Did you sell a house in 2010?"
    hasSoldHouse: boolean
  "Did you buy a house in 2010?"
    hasBoughtHouse: boolean
  "Did you enter a loan?"
    hasMaintLoan: boolean

  if (hasSoldHouse) {
    "Private debts for the sold house:"
      privateDebt: integer
    "What was the selling price?"
      sellingPrice: integer
    "Value residue:"
      valueResidue: integer =
       sellingPrice - privateDebt
  }

}
```



Did you sell a house in 2010? ☑

Did you buy a house in 2010? □

Did you enter a loan? □

Private debts for the sold house: 10

What was the selling price? 0

Value residue: -10

One REPL to rule everything

Current status of the QL REPL

- evaluate expressions: gives result
- simulate user input (= assign state variable)
- edit transactions (“semantic deltas”)
- start debugging session: enables the debugger commands
- set breakpoint, step, continue
- various meta commands: load, render, etc.
- backtracking over the “execution” trace (“revert”)

One REPL to rule everything

Current status of the QL REPL

- evaluate expressions: gives result
- simulate user input (= assign state variable)
- **edit transactions (“semantic deltas”)**
- start debugging session: enables the debugger commands
- **set breakpoint, step, continue**
- various meta commands: load, render, etc.
- **backtracking over the execution trace (“back-in-time”, “undo”)**

Demo

Instead of a conclusion…

- Change of perspective: from “state-based” to “change-based”
- Unification: program history (versions) and execution history (trace)
- Elm-like UI architecture for IDEs, with “commands” as core event abstraction
- Exploratory programming: forks in execution/version trace to explore alternatives
- Event sourcing for PLs?
- Collaboration via Operational Transformation?