Open Universiteit

# Software technology for automated feedback generation

*Bastiaan Heeren, ICS Colloquium December 17, 2020*

# Short bio

- Associate professor at the Open University of the Netherlands
- Head of OU's Computer Science Department

- At Utrecht University:
  - PhD in Software Technology (2000-2005)
  - Lecturer (2005-2007)
  - Guest researcher with the Software Technology for Learning and Teaching research group

Bastiaan Heeren

# Intelligent Tutoring Systems

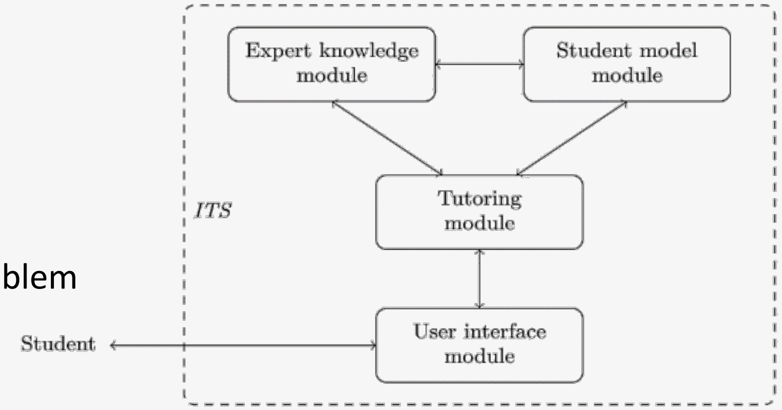
Intelligent Tutoring Systems (ITS): computer systems that provide immediate and customized feedback to learners

Structure:

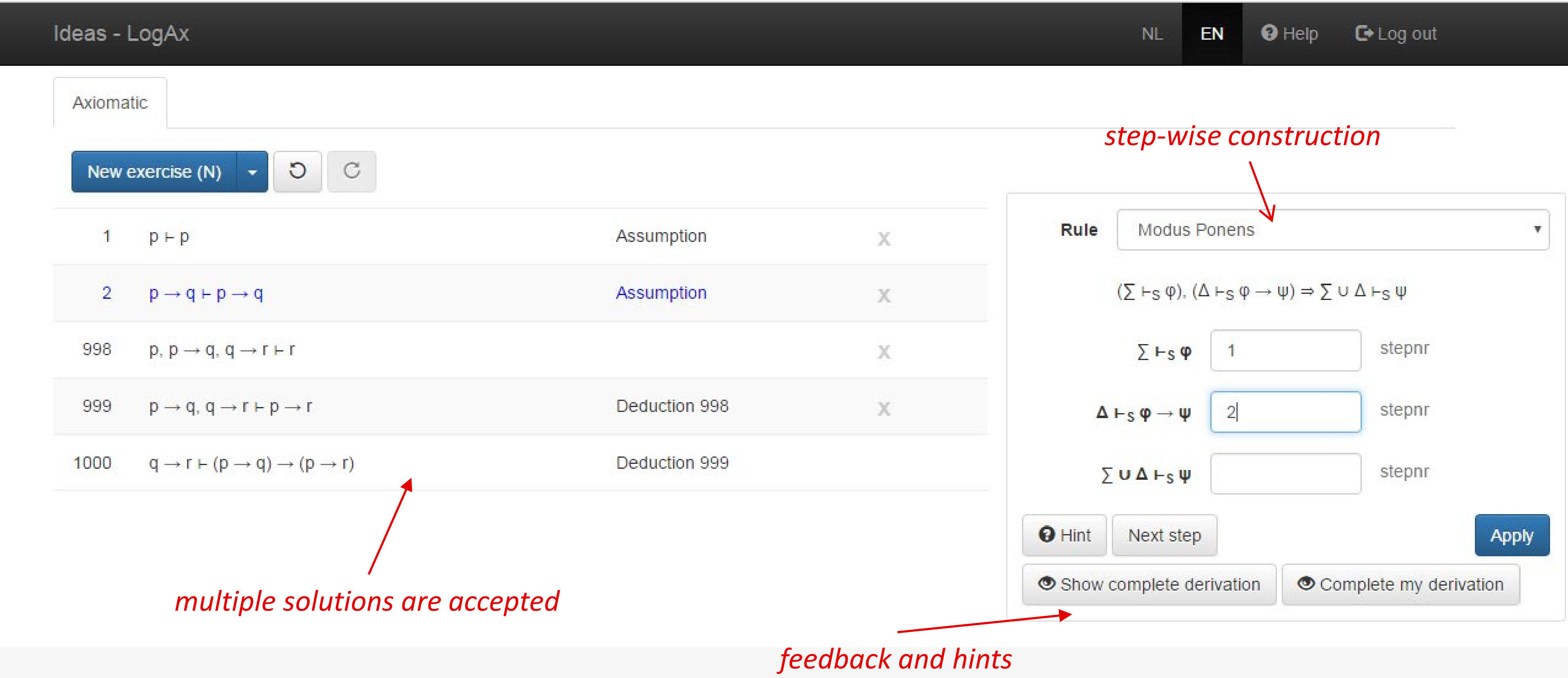
- Classical architecture with four components

Behaviour:

- Outer loop: solving one task after another
- Inner loop: the steps for solving one complex, multi-step problem



# Example: axiomatic proofs



# Research motivation

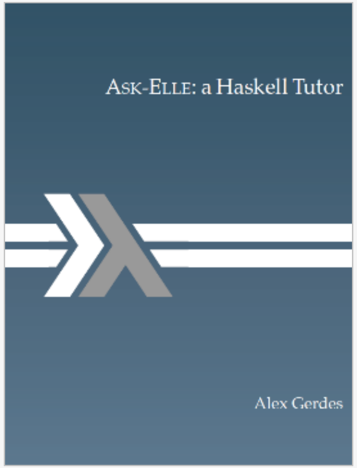
1. Simplify construction of ITSs (which are complex software systems)
2. Represent expert domain knowledge explicitly (for better feedback)
3. Apply approach to a wide range of problem domains

Approach: use software technology for automated feedback generation

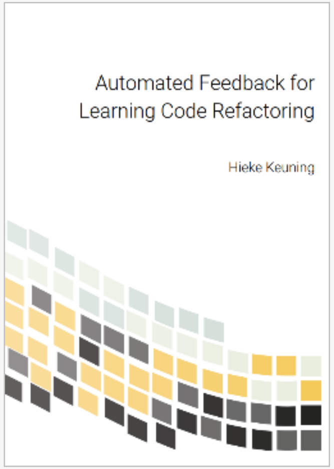
Techniques in this presentation (outline):

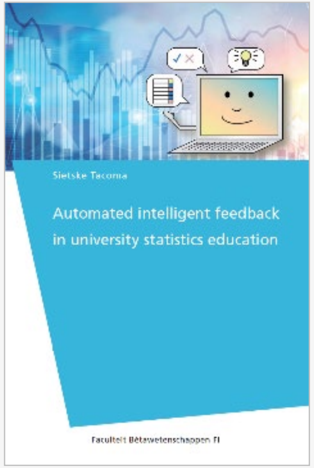
- Rewrite strategies for automated feedback (basics)
- Light-weight rewrite rules
- Generic traversals

# Problem domains








Four recent PhD theses, for different problem domains, all based on the same approach

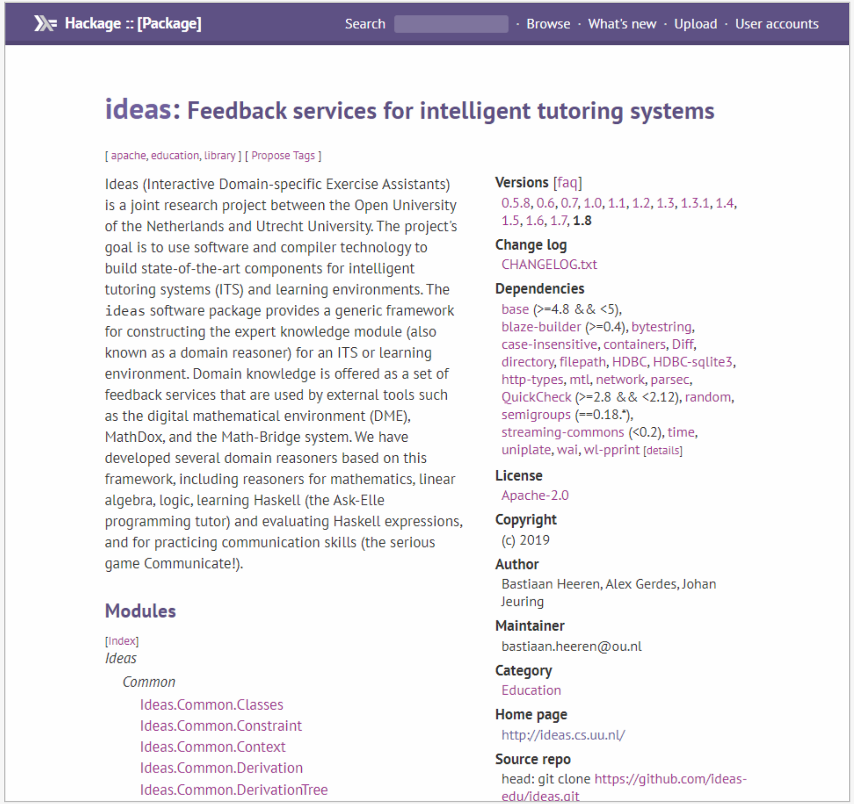
# Ideas framework

Generic framework for constructing domain reasoners

- Developed in Haskell
- Size: 12,397 LOC
- Open source
- Independent of problem domain
- http://ideas.cs.uu.nl/tutorial/

Ideas

Interactive Domain-specific Exercise Assistants


Hackage :: [Package]    Search · Browse · What's new · Upload · User accounts

## ideas: Feedback services for intelligent tutoring systems

[ apache, education, library ] [ Propose Tags ]

Ideas (Interactive Domain-specific Exercise Assistants) is a joint research project between the Open University of the Netherlands and Utrecht University. The project's goal is to use software and compiler technology to build state-of-the-art components for intelligent tutoring systems (ITS) and learning environments. The `ideas` software package provides a generic framework for constructing the expert knowledge module (also known as a domain reasoner) for an ITS or learning environment. Domain knowledge is offered as a set of feedback services that are used by external tools such as the digital mathematical environment (DME), MathDox, and the Math-Bridge system. We have developed several domain reasoners based on this framework, including reasoners for mathematics, linear algebra, logic, learning Haskell (the Ask-Elle programming tutor) and evaluating Haskell expressions, and for practicing communication skills (the serious game Communicate!).

### Modules

[Index]

*Ideas*

*Common*

Ideas.Common.Classes
Ideas.Common.Constraint
Ideas.Common.Context
Ideas.Common.Derivation
Ideas.Common.DerivationTree

**Versions** [faq]
0.5.8, 0.6, 0.7, 1.0, 1.1, 1.2, 1.3, 1.3.1, 1.4, 1.5, 1.6, 1.7, **1.8**

**Change log**
CHANGELOG.txt

**Dependencies**
base (>=4.8 && <5), blaze-builder (>=0.4), bytestring, case-insensitive, containers, Diff, directory, filepath, HDBC, HDBC-sqlite3, http-types, mtl, network, parsec, QuickCheck (>=2.8 && <2.12), random, semigroups (==0.18.*), streaming-commons (<0.2), time, uniplate, wai, wl-pprint [details]

**License**
Apache-2.0

**Copyright**
(c) 2019

**Author**
Bastiaan Heeren, Alex Gerdes, Johan Jeuring

**Maintainer**
bastiaan.heeren@ou.nl

**Category**
Education

**Home page**
http://ideas.cs.uu.nl/

**Source repo**
head: git clone https://github.com/ideas-edu/ideas.git

Open Universiteit

# Rewrite strategies

# Rewrite strategies for automated feedback

- Domain-specific language for specifying problem-solving procedures:
  - describe sequences of rule applications that solve a particular task
  - are formalized by a trace-based semantics (CSP)
  - allow new composition operators (interleaving, topological sorts)

- Problem-solving procedures are used for feedback generation:
  - recognizing the solution strategy
  - detecting detours
  - suggesting subgoals
  - providing next-step hints
  - providing worked-out examples

# Example

Goal: rewrite proposition into negation normal form (NNF)

¬((p ∨ q) ∧ ¬(p ∧ r))

⇔ De Morgan

¬(p ∨ q) ∨ ¬¬(p ∧ r)

⇔ De Morgan

(¬p ∧ ¬q) ∨ ¬¬(p ∧ r)

⇔ Double Neg

(¬p ∧ ¬q) ∨ (p ∧ r)

✓

Rewrite strategy for NNF:

*alternatives*

```
repeat (oncetd (doubleNeg .|. dmOr .|. dmAnd))
```

*top-down application*

*rewrite rule*

# Strategy combinators

| | |
|---|---|
| p .*. q | sequence: first p, then q |
| succeed | always succeeds |
| p .\|. q | choice: p or q |
| p ./. q | preference: p is preferred over q |
| p \|> q | left-biased choice: p or else q |
| fix | fixed-point combinator |

Derived combinators:

```
try s = s |> succeed
repeat s = try (s .*. repeat s)
```

Finite representation with explicit recursion:

```
repeat s = fix $ \x ->
   try (s .*. x)
```

Advantages:

- Extract rules from strategy
- Customize strategy
- Document/visualise strategy

Open Universiteit

# Light-weight rewrite rules

# Proposition logic

```haskell
data Logic = Logic :&&: Logic    -- conjunction
           | Logic :||: Logic    -- disjunction
           | Not Logic           -- negation
           | Var String          -- variable
```

Representation can be more complex, with nested and parameterised datatypes, e.g.:

$$3x + 9 = 0 \ \vee \ x = 1$$

| | | |
|---|---|---|
| 1 | p ⊢ p | Assumption |
| 2 | p → q ⊢ p → q | Assumption |
| 998 | p, p → q, q → r ⊢ r | |
| 999 | p → q, q → r ⊢ p → r | Deduction 998 |
| 1000 | q → r ⊢ (p → q) → (p → r) | Deduction 999 |

# Rewrite rules

| | | |
|---|---|---|
| ¬¬ϕ | ⇔ | ϕ |
| ¬(ϕ ∧ ψ) | ⇔ | ¬ϕ ∨ ¬ψ |
| ¬(ϕ ∨ ψ) | ⇔ | ¬ϕ ∧ ¬ψ |

```
doubleNeg = rewriteRule "doubleNeg" $
       \phi  ->  Not (Not phi)  :~>  phi

dmAnd = rewriteRule "dmAnd" $
       \phi psi  ->  Not (phi :&&: psi)  :~>  Not phi :||: Not psi
```

*meta-variables are introduced by lambdas*

*left-hand side*

*right-hand side*

How to use such rewrite rules?

# Embedding-projection pair

Approach: conversion from/to a generic Term datatype with support for meta-variables

```
toTerm   :: Logic -> Term
fromTerm :: Term  -> Maybe Logic
```

- From/to should be inverse functions (intuitively)
- Conversion allows generic functions, such as unification and zippers
- Pair can be derived automatically from the datatype definition

Note: more powerful generic programming libraries exist that can guarantee more type safety, with less overhead

# Compiling rewrite rules

```
\phi -> Not (Not phi) :~> phi
```

*values provided by user for problem domain*

Step 1: use two different values (e.g. Var "p" and Var "q"):

```
Not (Not (Var "p")) :~> Var "p"
Not (Not (Var "q")) :~> Var "q"
```

*TCon, TVar, and TMeta are constructors of Term*

Step 2: convert to Term datatype:

```
TCon "Not" [TCon "Not" [TVar "p"]]  :~>  TVar "p"
TCon "Not" [TCon "Not" [TVar "q"]]  :~>  TVar "q"
```

Step 3: find meta-variables by comparing left-hand sides and right-hand sides

```
TCon "Not" [TCon "Not" [TMeta 0]]  :~> TMeta 0
```

# Applying rewrite rules

```
Not (Not (Var “p” :&&: Var “r”))
```

Rewrite rule:

```
TCon “Not” [TCon “Not” [TMeta 0]]
:~> TMeta 0
```

Step 1: convert to Term datatype:

```
TCon “Not” [TCon “Not” [TCon “And” [TVar “p”, TVar “q”]]]
```

Step 2: match with rule’s left-hand side:

```
0 = TCon “And” [TVar “p”, TVar “q”]
```

Step 3: substitute in rule’s right-hand side:

```
TCon “And” [TVar “p”, TVar “q”]
```

Step 4: convert back to Logic:

```
Var “p” :&&: Var “r”
```

# Knuth-Bendix completion

Use case for explicit representation: search for missing rewrite rules (and reach confluence)

Critical pair

$\neg\neg(\phi \land \psi)$

$\phi \land \psi$ $\qquad$ $\neg(\neg\phi \lor \neg\psi)$

$\Leftrightarrow$

$\neg\neg\phi \land \neg\neg\psi$

$\Leftrightarrow$

$\phi \land \psi$

✓

| | | |
|---|---|---|
| $\neg\neg\phi$ | $\Leftrightarrow$ | $\phi$ |
| $\neg(\phi \land \psi)$ | $\Leftrightarrow$ | $\neg\phi \lor \neg\psi$ |

Missing rule:

$\neg(\phi \lor \psi) \quad \Leftrightarrow \quad \neg\phi \land \neg\psi$

# Light-weight rewrite rules

Advantages of explicit representation:

- Knuth-Bendix completion (analysis)
- AC-rewriting
- Rule inversion
- Automated testing
- Documentation (pretty-printing)

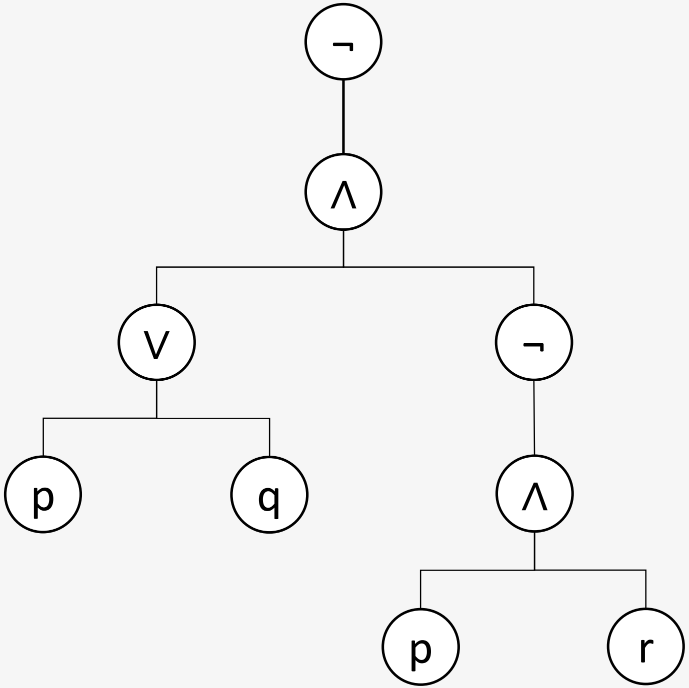
Summary for rewrite rules:

- ✓ Simplify construction (light-weight embedding)
- ✓ Explicit representation (for better feedback)
- ✓ Many problem domains

# Generic traversals

# Tree representation

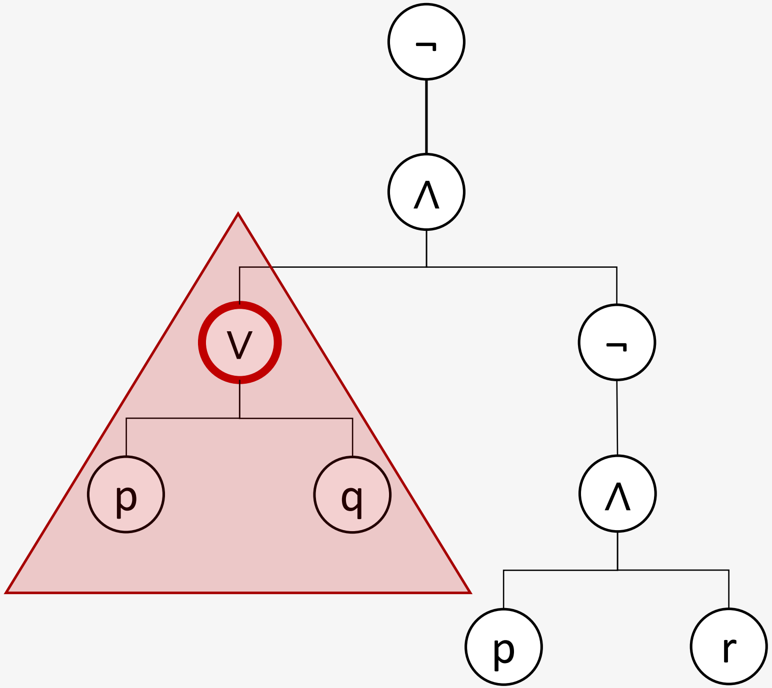
$\neg((p \vee q) \wedge \neg(p \wedge r))$



# Point of focus

¬((p ∨ q) ∧ ¬(p ∧ r))

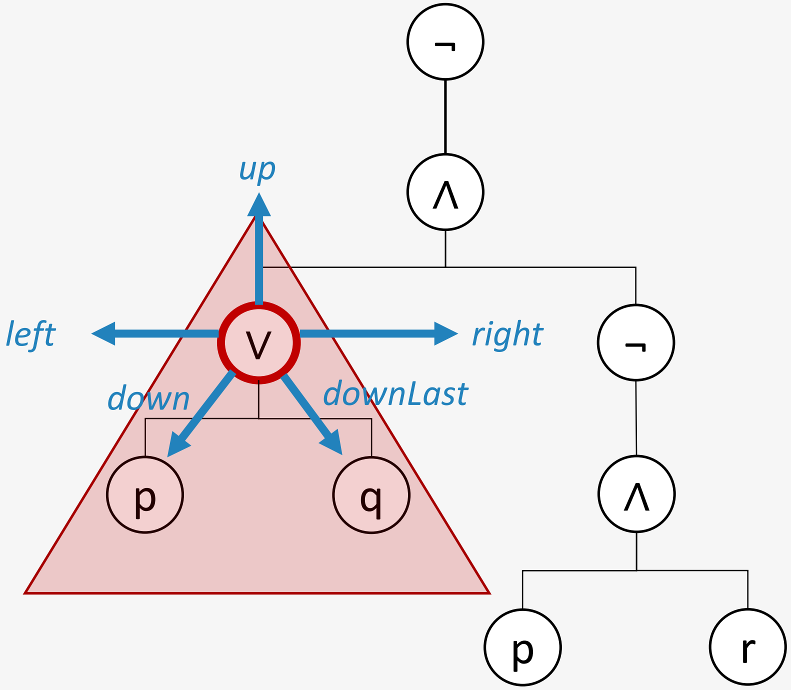
- Implemented as a so-called zipper over the generic Term datatype
- Stored in a Context



# Navigation

Five navigational actions:

- up
- left
- right
- down
- downLast

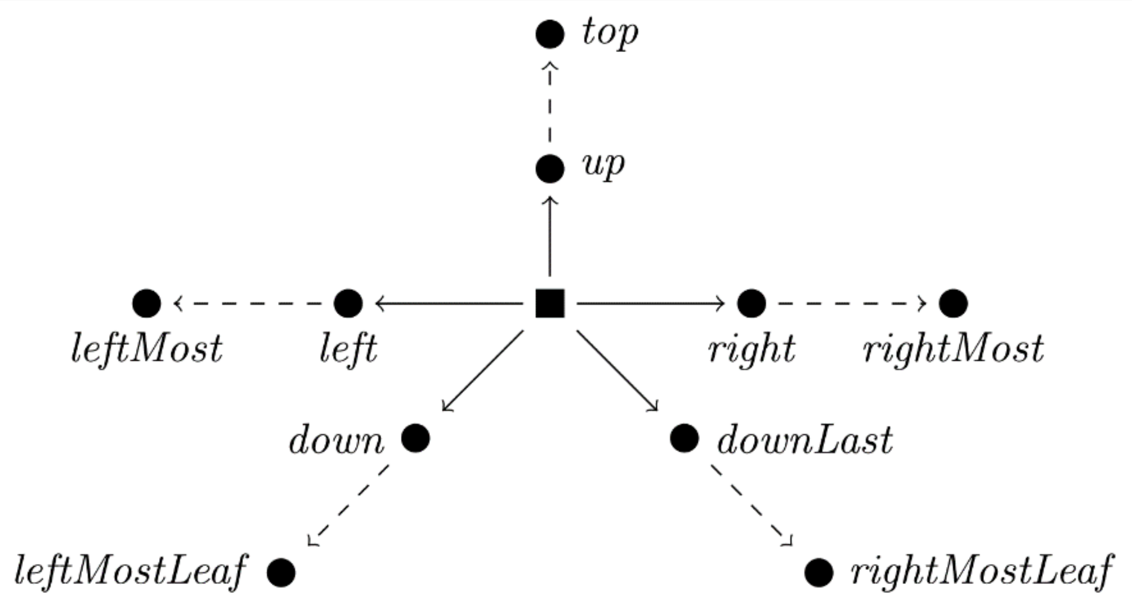



- Actions may fail
- Many useful laws, e.g.:

  left ∘ right ≈ id

  up ∘ down ≈ id

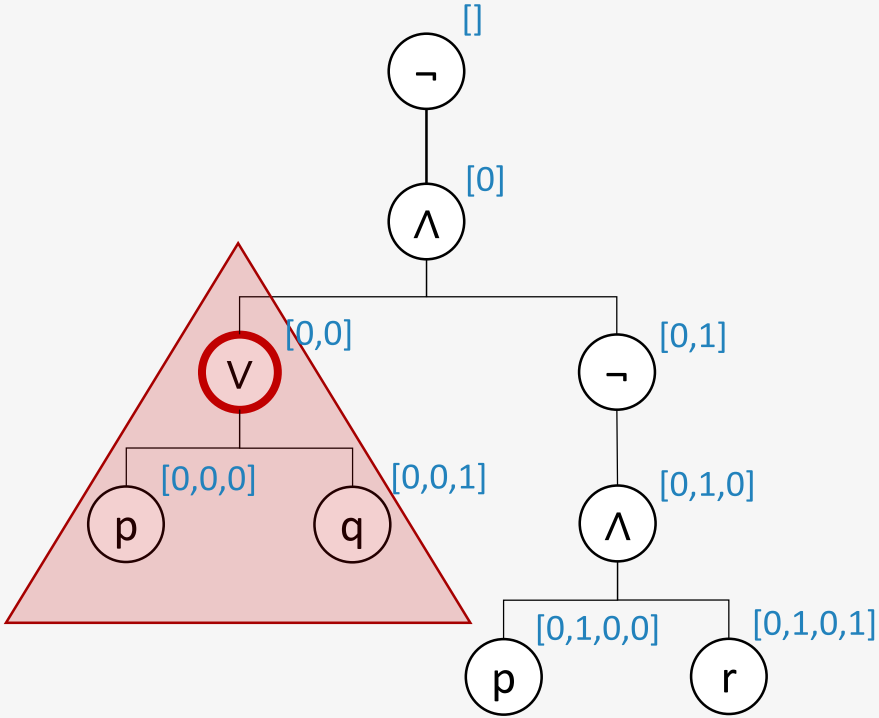
# Navigation (extended)



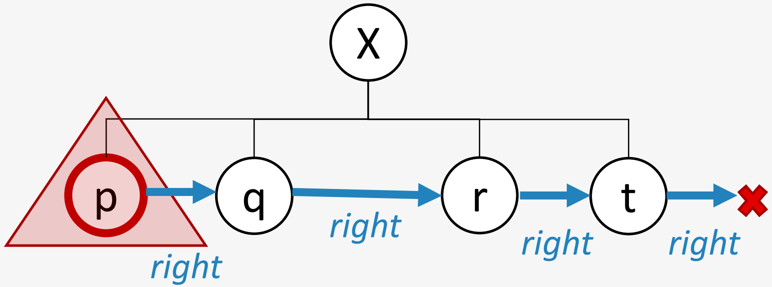

From: Traversals with Class. In Jurriaan Hage and Atze Dijkstra, editors, *Een Lawine van Ontwortelde Bomen: Liber Amicorum voor Doaitse Swierstra*, pages 62-75. 2013.

# Position

- Zippers keep a position for the point of focus
- Position information is useful for generating feedback



# Horizontal visits

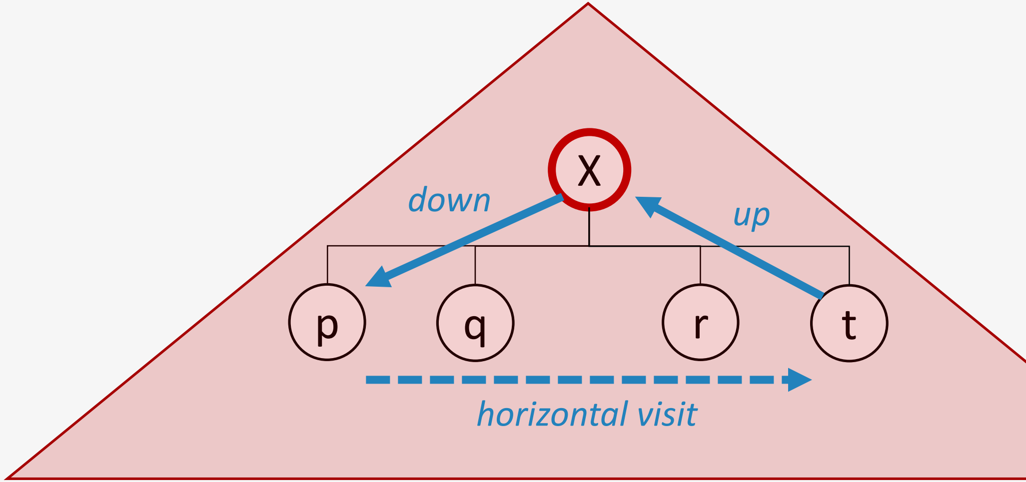



```
visitOne s   = fix $ \x -> s .|. (right .*. x)

visitFirst s = fix $ \x -> s |> (right .*. x)

visitAll s   = fix $ \x -> s .*. (not right |> (right .*. x))
```

- Approach: define traversals as (normal) strategy combinators
- Idea: also parameterize "next" function to also support right-to-left visits

# One-layer visits

```
layer s = down .*. s .*. up

layerOne s = layer (visitOne s)
```



# Traversals

```
somewhere s = fix $ \x -> s .|. layerOne x

oncetd s    = fix $ \x -> s |> layerOne x          -- top down

oncebu s    = fix $ \x -> layerOne x |> s          -- bottom up
```

- Also: full traversals, spine traversals, innermost, outermost, etc.

# Example trace

¬((p ∨ q) ∧ ¬(p ∧ r))

⇔ De Morgan

¬(p ∨ q) ∨ ¬¬(p ∧ r)

⇔ De Morgan

(¬p ∧ ¬q) ∨ ¬¬(p ∧ r)

⇔ Double Neg

(¬p ∧ ¬q) ∨ (p ∧ r)

Corresponding trace:

| | |
|---|---|
| De Morgan | at [] |
| down | |
| De Morgan | at [0] |
| up | |
| down | |
| right | |
| Double Neg | at [1] |
| up | |

Summary for traversals:

- ✓ Simplify construction (traversals are first-class strategy combinators)
- ✓ Explicit representation (for better feedback)
- ✓ Many problem domains

Open Universiteit

# Conclusion

# Trends and challenges

- Authoring intelligent tutoring system
  - Literature reports 200-300 authoring hours for 1 hour of instruction
  - We believe software technology can help
- Data-driven intelligent tutoring system
  - Use AI techniques to generate feedback from collected data
  - Raises questions about the role of expert domain knowledge
- Further adaptation and personalization
  - Models for mastery learning (e.g. Bayesian knowledge tracing)
- Designing tools for less-structured problem domains
  - For example, domains of software design and learning languages

# Conclusion

- Rewrite strategies are used for feedback generation
- Rewrite rules can be embedded by using datatype-generic programming techniques
- Generic traversals can be composed from navigational actions and strategy combinators
- The presented approach can be applied to a wide range of problem domains

✉ Bastiaan.Heeren@ou.nl

Project website: http://ideas.cs.uu.nl/

# Related publications


- Thomas van Noort, Alexey Rodriguez, Stefan Holdermans, Johan Jeuring, and Bastiaan Heeren. A Lightweight Approach to Datatype-generic Rewriting. In *Proceedings of the ACM SIGPLAN Workshop on Generic Programming*, WGP '08, pages 13-24, 2008. ACM.
- Johan Jeuring, José Pedro Magalhães, and Bastiaan Heeren. Generic Programming for Domain Reasoners. In Zoltán Horváth, Viktória Zsók, Peter Achten, and Pieter W. M. Koopman, editors, *Proceedings of the Tenth Symposium on Trends in Functional Programming, TFP 2009*, pages 113-128, 2009. Intellect.
- Thomas van Noort, Alexey Rodriguez yakushev, Stefan Holdermans, Johan Jeuring, Bastiaan Heeren, and José Pedro Magalhães. A Lightweight Approach to Datatype-generic Rewriting. *J. Funct. Program.*, 20(3-4):375-413, 2010.
- Bastiaan Heeren and Johan Jeuring. Feedback services for stepwise exercises . *Science of Computer Programming* , 88:110-129, 2014. Software Development Concerns in the e-Learning Domain.
- Josje Lodder, Bastiaan Heeren and Johan Jeuring. A Domain Reasoner for Propositional Logic. *Journal of Universal Computer Science*, 22(8):1097-1122, 2016.
- Bastiaan Heeren and Johan Jeuring. An Extensible Domain-Specific Language for Describing Problem-Solving Procedures. In Elisabeth André, Ryan Baker, Xiangen Hu, Ma. Mercedes T. Rodrigo, and Benedict du Boulay, editors, *Proceedings of Artificial Intelligence in Education: 18th International Conference, AIED 2017*, pages 77-89, 2017. Springer International Publishing.
- Josje Lodder, Bastiaan Heeren, Johan Jeuring, and Wendy Neijenhuis. Generation and Use of Hints and Feedback in a Hilbert-Style Axiomatic Proof Tutor. *International Journal of Artificial Intelligence in Education*, 2020.