Software technology for learning and teaching

Part 2: Rewriting and strategies

Bastiaan Heeren[1] and Johan Jeuring[1,2]

[1] Open Univerity of the Netherlands
[2] Utrecht University

26 January 2015, IPA course, Eindhoven

Open Universiteit
www.ou.nl

Outline of presentation

1. Strategy language
2. Sequential composition
3. Language extensions
   - Initial prefixes
   - Interleaving
   - Left-biased choice
   - Labels
   - Traversals
4. Designing domain reasoners

Outline of presentation

1. Strategy language
2. Sequential composition
3. Language extensions
   - Initial prefixes
   - Interleaving
   - Left-biased choice
   - Labels
   - Traversals
4. Designing domain reasoners

Strategy language

Our approach: to develop a strategy language for expressing cognitive skills for many domains, used to give feedback, hints, and worked-out solutions.

Strategy language with basic rules (r), sequences, and choices:

$$s, t ::= \mathit{succeed} \mid \mathit{fail} \mid \mathit{single}\ r \mid s \lt\!|\!\gt t \mid s \lt\!\star\!\gt t$$

Very similar to (but slightly different from):

- Context-free grammars and their corresponding parsers
- Rewrite strategies
- Communicating sequential processes
- Proof tactics
- Workflows

Requirements for the strategy language

1. Easy to extend the language
2. Give feedback or hints at any time, also for partial solutions
3. Strategies should be compositional
4. Feedback and hints are calculated reasonably efficient
5. Easy to adapt a strategy, or the feedback constructed from a strategy

We need a clear semantics for our strategy language

The language of a strategy

Similar to context-free grammars, we generate the language of a strategy (a set of sentences)

$$
\begin{aligned}
\mathcal{L}(\mathit{succeed}) &= \{\epsilon\} \\
\mathcal{L}(\mathit{fail}) &= \emptyset \\
\mathcal{L}(\mathit{single}\ r) &= \{r\} \\
\mathcal{L}(s \mathbin{<|>} t) &= \mathcal{L}(s) \cup \mathcal{L}(t) \\
\mathcal{L}(s \mathbin{<\star>} t) &= \{xy \mid x \in \mathcal{L}(s), y \in \mathcal{L}(t)\}
\end{aligned}
$$

- Compositional and extensible
- Abstract away from rewrite rules as symbols
- Useful as specification?

Strategy application

Rules and strategies have an effect on the underlying object; they rewrite a term

$$
\begin{aligned}
\mathit{succeed}(a) &= \{\, a \,\} \\
\mathit{fail}(a) &= \emptyset \\
(\mathit{single}\ r)(a) &= r(a) \\
(s \mathbin{<|>} t)(a) &= s(a) \cup t(a) \\
(s \mathbin{<\star>} t)(a) &= \{\, c \mid b \in s(a), c \in t(b) \,\}
\end{aligned}
$$

- Rule application returns a set of results (compositionality)
- What about intermediate terms and the used rules?

Observations

Simplicity of $\mathcal{L}(\cdot)$ is attractive, but:

- Sequences introduce back-tracking
  - Remember that $\mathcal{L}(s <\star> t) = \{xy \mid x \in \mathcal{L}(s), y \in \mathcal{L}(t)\}$
  - Not desirable in tutor (limited look-ahead)
- No easy way to calculate intermediate terms and rules
- Some strategy combinators depend on the current object
  - E.g. $s \triangleright t$: first try s, and only if this fails, use t.

Instead, we use a trace semantics based on firsts and empty.

Firsts set

$$
\begin{aligned}
\mathit{firsts}(\mathit{succeed}, a) &= \emptyset \\
\mathit{firsts}(\mathit{fail}, a) &= \emptyset \\
\mathit{firsts}(\mathit{single}\ r, a) &= \{\, r \mapsto \mathit{succeed} \,\} \\
\mathit{firsts}(s \mathrel{<|>} t, a) &= \mathit{firsts}(s, a) \uplus \mathit{firsts}(t, a) \\
\mathit{firsts}(s \mathrel{<\star>} t, a) &= \{\, r \mapsto s' \mathrel{<\star>} t \mid r \mapsto s' \in \mathit{firsts}(s, a) \,\} \\
&\uplus \{\, r \mapsto t' \mid \mathit{empty}(s, a), r \mapsto t' \in \mathit{firsts}(t, a) \,\}
\end{aligned}
$$

- firsts takes a strategy and the current object
- $\uplus$ returns the union of two finite maps
- $r \mapsto s$ and $r \mapsto t$ are merged to form $r \mapsto (s \mathrel{<|>} t)$

Empty property

$$
\begin{aligned}
empty(succeed, a) &= true \\
empty(fail, a) &= false \\
empty(single\ r, a) &= false \\
empty(s \mathbin{<|>} t, a) &= empty(s, a) \vee empty(t, a) \\
empty(s \mathbin{<\star>} t, a) &= empty(s, a) \wedge empty(t, a)
\end{aligned}
$$

- empty checks for successful termination

Traces

Traces can represent unfinished and unsuccessful sequences of steps, for example:

- $a_0 \xrightarrow{r_1} a_1 \xrightarrow{r_2} a_2$
- $a_0 \xrightarrow{r_1} a_1 \checkmark$

$$
\begin{aligned}
steps(s,a) &= \{\,(r,b,t) \mid r \mapsto t \in firsts(s,a), b \in r(a)\,\} \\
traces(s,a) &= \{\,a\,\} \cup \{\,a \checkmark \mid empty(s,a)\,\} \\
&\quad \cup\ \{\,a \xrightarrow{r} x \mid (r,b,t) \in steps(s,a), x \in traces(t,b)\,\}
\end{aligned}
$$

Algebraic laws

Equality:

$$(s = t) = \forall a : \mathit{traces}(s, a) = \mathit{traces}(t, a)$$

Laws:

- Choice is associative, commutative, and idempotent
- Choice has *fail* as its unit element
- Sequence is associative
- Sequence has *succeed* as its unit element
- Sequence has *fail* as its left zero (but not right zero)
- Sequence distributes over choice

Outline of presentation

1. Strategy language
2. Sequential composition
3. Language extensions
   - Initial prefixes
   - Interleaving
   - Left-biased choice
   - Labels
   - Traversals
4. Designing domain reasoners

Sequential composition revisited

Calculating firsts for sequences is not efficient

- Calculating firsts for $(s_1 \lessdot\!\star\!\gtrdot s_2) \lessdot\!\star\!\gtrdot s_3$ requires:
  - firsts for s_1
  - firsts for s_2, if empty s_1
  - firsts for s_3, if empty s_1 and empty s_2
- We introduce prefix combinator $r \rightarrow s$
- Bring strategies to prefix-form
- Use algebraic laws to guide transformation

Prefix combinator

Specification:

$$firsts(r \rightarrow s, a) \;\; = \{ r \mapsto s \}$$

$$empty(r \rightarrow s, a) = false$$

Laws:

- prefix is left-distributive over choice
  $r \rightarrow (s <\!|\!> t) = (r \rightarrow s) <\!|\!> (r \rightarrow t)$
- $single\ r = r \rightarrow succeed$

We show how to transform sequences into prefix-form

Transforming sequence

We can systematically remove sequences:

$$
\begin{array}{lcl}
\mathit{succeed} & <\star> t = & t \\
\mathit{fail} & <\star> t = & \mathit{fail} \\
(s_1 <|> s_2) & <\star> t = & (s_1 <\star> t) <|> (s_2 <\star> t) \\
(r \rightarrow s) & <\star> t = & r \rightarrow (s <\star> t)
\end{array}
$$

$$(s_1 <\star> s_2) <\star> t = s_1 <\star> (s_2 <\star> t)$$

Core grammar for strategies:

$$s, t ::= \mathit{succeed} \mid \mathit{fail} \mid s <|> t \mid r \rightarrow s$$

Outline of presentation

1. Strategy language
2. Sequential composition
3. Language extensions
   - Initial prefixes
   - Interleaving
   - Left-biased choice
   - Labels
   - Traversals
4. Designing domain reasoners

Language extensions

How to extend the strategy language with new combinators?

1. Define in terms of existing combinators:

   $$\textit{options}\ s = s \mathbin{<\!|\!>} \textit{succeed}$$

2. Specify its firsts set and empty property
3. Transform combinator to core language

Some combinators require extensions to the presented trace semantics

Extension 1

Domain: Communication skills

Extension: A player holds a discussion with a patient, possibly about various topic. Players can perform only an initial part of a discussion, and then jump to another discussion.

Combinator: initial prefixes (*inits* s)

Example: If $(a_0 \xrightarrow{r_1} a_1 \xrightarrow{r_2} a_2) \in traces(s, a_0)$
then $\{\, a_0 \checkmark, a_0 \xrightarrow{r_1} a_1 \checkmark, a_0 \xrightarrow{r_1} a_1 \xrightarrow{r_2} a_2 \checkmark \,\}$
$\subseteq traces(inits\ s, a_0)$

Initial prefixes

Specification:

$$firsts(inits\ s, a) \quad =$$

$$empty(inits\ s, a) =$$

Transformation:

$$\begin{aligned} inits\ succeed &= \\ inits\ fail &= \\ inits\ (s <|> t) &= \\ inits\ (r \rightarrow s) &= \end{aligned}$$

Initial prefixes

Specification:

$$firsts(inits\ s, a) = \{ r \mapsto inits\ t \mid r \mapsto t \in firsts(s, a) \}$$

$$empty(inits\ s, a) = true$$

Transformation:

$$\begin{aligned} inits\ succeed &= \\ inits\ fail &= \\ inits\ (s <\!|\!> t) &= \\ inits\ (r \rightarrow s) &= \end{aligned}$$

Initial prefixes

Specification:

$$\mathit{firsts}(\mathit{inits}\ s, a) = \{\, r \mapsto \mathit{inits}\ t \mid r \mapsto t \in \mathit{firsts}(s, a) \,\}$$

$$\mathit{empty}(\mathit{inits}\ s, a) = \mathit{true}$$

Transformation:

$$\begin{aligned}
\mathit{inits\ succeed} &= \mathit{succeed} \\
\mathit{inits\ fail} &= \mathit{succeed} \\
\mathit{inits}\ (s <\!|\!> t) &= \mathit{inits}\ s \quad <\!|\!> \mathit{inits}\ t \\
\mathit{inits}\ (r \to s) &= \mathit{succeed} <\!|\!> (r \to \mathit{inits}\ s)
\end{aligned}$$

Extension 2

Domain: Math

Extension: Some higher-degree equations can be solved by: $AC = BC \Rightarrow A = B \vee C = 0$. A student may switch between the two equations.

Combinator: interleaving ($s <\%> t$)

Example:

If $[r_a, r_b]$ is a sentence of s
and $[r_x, r_y, r_z]$ is a sentence of t
then $s <\%> t$ contains $[r_a, r_b, r_x, r_y, r_z]$, $[r_a, r_x, r_b, r_y, r_z]$, $[r_a, r_x, r_y, r_b, r_z]$, $[r_a, r_x, r_y, r_z, r_b]$, $[r_x, r_a, r_b, r_y, r_z]$, ...

Interleaving

Specification:

$$firsts(s \mathbin{<\!\%\!>} t, a) \quad =$$

$$empty(s \mathbin{<\!\%\!>} t, a) =$$

Transformation:

$$\begin{array}{lll} succeed & \mathbin{<\!\%\!>} t = \\ fail & \mathbin{<\!\%\!>} t = \\ (s_1 \mathbin{<\!|\!>} s_2) & \mathbin{<\!\%\!>} t = \\ (r \rightarrow s) & \mathbin{<\!\%\!>} t = \end{array}$$

Interleaving

Specification:

$$\begin{aligned} \mathit{firsts}(s \mathbin{<\%>} t, a) &= \{\, r \mapsto s' \mathbin{<\%>} t \mid r \mapsto s' \in \mathit{firsts}(s, a) \,\} \\ &\uplus \{\, r \mapsto s \mathbin{<\%>} t' \mid r \mapsto t' \in \mathit{firsts}(t, a) \,\} \end{aligned}$$

$$\mathit{empty}(s \mathbin{<\%>} t, a) = \mathit{empty}(s, a) \wedge \mathit{empty}(t, a)$$

Transformation:

$$\begin{aligned} &\mathit{succeed} && \mathbin{<\%>} t = \\ &\mathit{fail} && \mathbin{<\%>} t = \\ &(s_1 \mathbin{<|>} s_2) && \mathbin{<\%>} t = \\ &(r \rightarrow s) && \mathbin{<\%>} t = \end{aligned}$$

Interleaving

Specification:

$$\begin{aligned} \mathit{firsts}(s \mathbin{<\%>} t, a) &= \{\, r \mapsto s' \mathbin{<\%>} t \mid r \mapsto s' \in \mathit{firsts}(s, a) \,\} \\ &\uplus \{\, r \mapsto s \mathbin{<\%>} t' \mid r \mapsto t' \in \mathit{firsts}(t, a) \,\} \end{aligned}$$

$$\mathit{empty}(s \mathbin{<\%>} t, a) = \mathit{empty}(s, a) \wedge \mathit{empty}(t, a)$$

Transformation:

$$\begin{aligned} \mathit{succeed} \mathbin{<\%>} t &= t \\ \mathit{fail} \mathbin{<\%>} t &= \ldots \\ (s_1 \mathbin{<|>} s_2) \mathbin{<\%>} t &= (s_1 \mathbin{<\%>} t) \mathbin{<|>} (s_2 \mathbin{<\%>} t) \\ (r \to s) \mathbin{<\%>} t &= \ldots \end{aligned}$$

Solution: introduce left-interleave $s \mathbin{\%>} t$

Left-interleave

Specification:

$$firsts(s \mathbin{\%\!\!>} t, a) \;= \{\, r \mapsto s' \mathbin{<\!\!\%\!\!>} t \mid r \mapsto s' \in firsts(s, a) \,\}$$

$$empty(s \mathbin{\%\!\!>} t, a) = false$$

Transformation:

$$\begin{aligned}
&succeed & \%\!\!> t &= \\
&fail & \%\!\!> t &= \\
&(s_1 \mathbin{<\!|\!>} s_2) & \%\!\!> t &= \\
&(r \rightarrow s) & \%\!\!> t &=
\end{aligned}$$

Left-interleave

Specification:

$$firsts(s \mathbin{\%\!>} t, a) = \{ r \mapsto s' \mathbin{<\!\%\!>} t \mid r \mapsto s' \in firsts(s, a) \}$$

$$empty(s \mathbin{\%\!>} t, a) = false$$

Transformation:

$$\begin{aligned}
succeed \quad & \mathbin{\%\!>} t = fail \\
fail \quad & \mathbin{\%\!>} t = fail \\
(s_1 \mathbin{<\!|\!>} s_2) & \mathbin{\%\!>} t = (s_1 \mathbin{\%\!>} t) \mathbin{<\!|\!>} (s_2 \mathbin{\%\!>} t) \\
(r \to s) & \mathbin{\%\!>} t = r \to (s \mathbin{<\!\%\!>} t)
\end{aligned}$$

Extension 3

Domain: Propositional logic

Extension: If possible, we use the rewrite rule $\phi \wedge T \Rightarrow \phi$. If not, we succeed.

Combinator: left-biased choice ($s \triangleright t$)

Example: If $\quad traces(s, a_0) = \{ a_0 \}$
then $traces(s \triangleright t, a_0) = traces(t, a_0)$

Left-biased choice

Use a strategy predicate to specify left-biased choice:

- *active* s: strategy s is empty or offers steps (local)
  - Opposite of *active* s is *stopped* s
- *test* s: strategy s can finish successfully (global)
  - Opposite of *test* s is *not* s

Specification:

$$\mathit{firsts}(\mathit{stopped}\ s, a) = \emptyset$$

$$\mathit{empty}(\mathit{stopped}\ s, a) = \neg \mathit{empty}(s, a) \wedge \mathit{steps}(s, a) = \emptyset$$

Then:

$$s \triangleright t = s <|> (\mathit{stopped}\ s <\star> t)$$

Transforming left-biased choice

- Left-biased choice depends on the current object
- In some cases, we can transform strategies with a left-biased choice:

$$(s_1 \triangleright s_2) \mathbin{<\star>} t = (s_1 \mathbin{<\star>} t) \triangleright (s_2 \mathbin{<\star>} t)$$
$$\text{provided that } \forall a : \neg \mathit{empty}(s, a)$$

$$s \triangleright t = s \qquad \text{provided that } \forall a : \mathit{empty}(s, a)$$

Labelled strategies

Labels mark a position in a strategy

$$label\ \ell\ s = Enter\ \ell \ \langle\star\rangle\ s\ \langle\star\rangle\ Exit\ \ell$$

- Labels show up in traces
- Customize reported feedback for a label
- Labels can be used to identify subtasks
- We can collapse, hide, or remove a labelled substrategy (adaptability)

Traversal combinators

Use navigation rules *Left*, *Right*, *Up*, and *Down* for defining all kinds of generic traversals

$$somewhere\ s = s \mathrel{<|>} layerOne\ (somewhere\ s)$$

$$layerOne\ s = Down \mathrel{<\star>} visitOne\ s \mathrel{<\star>} Up$$

$$visitOne\ s = s \mathrel{<|>} (Right \mathrel{<\star>} visitOne\ s)$$

Many more variations:

- left-to-right, right-to-left
- top-down, bottom-up
- full, spine, stop, once

Outline of presentation

1. Strategy language

2. Sequential composition

3. Language extensions

   Initial prefixes

   Interleaving

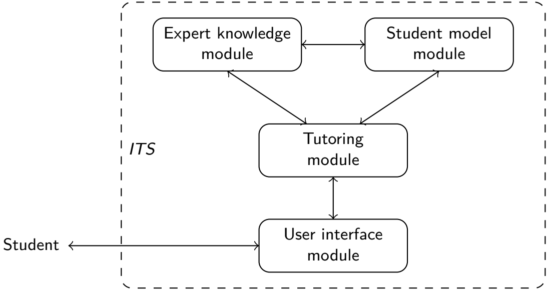
   Left-biased choice

   Labels

   Traversals

4. Designing domain reasoners

Four component ITS architecture



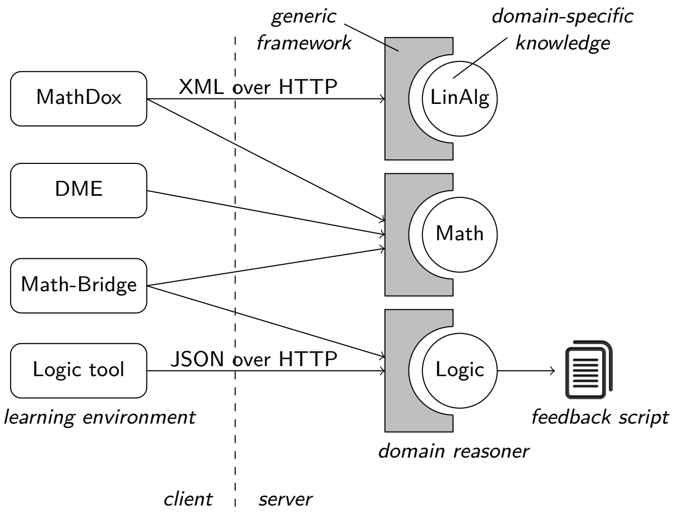
- Traditionally, an ITS is described by four components
- Also: monitoring module for teachers, authoring environment, etc.
- We focus on the expert knowledge module

Designing domain reasoners

- Following Goguadze, we use the term domain reasoner
- Design goals:
  - External, separate component reusable by other learning environments
  - Feedback-oriented (e.g., not a CAS)
  - Support for an exercise class (not one exercise)
  - Calculating feedback is not tied to a particular domain

IDEAS is a generic framework for developing domain-specific reasoners that offer feedback services to external learning environments: the feedback services are based on the stateless client-server architecture

Proposed design



List of feedback services

outer loop

– examples	predefined example exercises of a certain difficulty
– generate	makes a new exercise of a specified difficulty

List of feedback services

outer loop

– examples	predefined example exercises of a certain difficulty
– generate	makes a new exercise of a specified difficulty

inner loop

– allfirsts	all possible next steps (based on the strategy)
– apply	application of a rewrite rule to a selected term
– diagnose	analyze a student step
– finished	checks whether response is accepted as an answer
– onefirst	one possible next step (based on the strategy)
– solution	worked-out solution for the current exercise
– stepsremaining	number of remaining steps (based on the strategy)
– subtasks	returns a list of subtasks of the current task

List of feedback services

outer loop

– examples	predefined example exercises of a certain difficulty
– generate	makes a new exercise of a specified difficulty

inner loop

– allfirsts	all possible next steps (based on the strategy)
– apply	application of a rewrite rule to a selected term
– diagnose	analyze a student step
– finished	checks whether response is accepted as an answer
– onefirst	one possible next step (based on the strategy)
– solution	worked-out solution for the current exercise
– stepsremaining	number of remaining steps (based on the strategy)
– subtasks	returns a list of subtasks of the current task

meta-information

– exerciselist	all supported exercise classes
– rulelist	all rules in an exercise class
– rulesinfo	detailed information about rules in an exercise class
– strategyinfo	information about the strategy of an exercise class

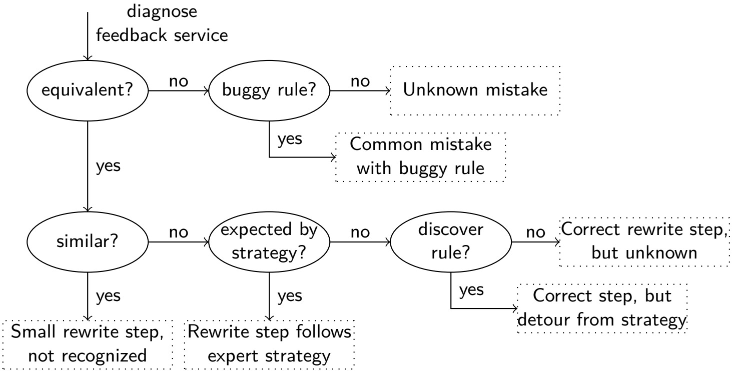
A domain reasoner (for quadratic equations) §4

We have to decide on:

1. A rewrite strategy
2. Rules and buggy rules
   - $(x+y)^2 \not\Rightarrow x^2 + y^2$
3. Equivalence relation
   - $x^2 - 4x + 3 = 0$, $(x-3)(x-1) = 0$, and $x = 3 \vee x = 1$
4. Similarity relation (determines granularity of steps)
   - $x^2 - x = 0 \quad \approx \quad -x + x \cdot x = 0$
5. Solved form
   - does $\sqrt{8}$ require further simplification?

Diagnose feedback service

All these exercise components are used by the diagnose feedback service



List of exercise components

component	description
strategy	rewrite strategy that specifies how to solve an exercise
rules	possible rewrite steps (including buggy rules)
equivalence	tests whether two terms are semantically equivalent
similarity	tests whether two terms are (nearly) the same
suitable	identifies which terms can be solved by the strategy
finished	checks whether a term is in a solved form

List of exercise components §4

component	description
strategy	rewrite strategy that specifies how to solve an exercise
rules	possible rewrite steps (including buggy rules)
equivalence	tests whether two terms are semantically equivalent
similarity	tests whether two terms are (nearly) the same
suitable	identifies which terms can be solved by the strategy
finished	checks whether a term is in a solved form
exercise id	identifier that uniquely determines the exercise class
status	stability of the exercise class
parser	parser for terms
pretty-printer	pretty-printer for terms (inverse of parsing)
navigation	supports traversals over terms
rule ordering	tiebreaker when more than one rule can be used

List of exercise components

component	description
strategy	rewrite strategy that specifies how to solve an exercise
rules	possible rewrite steps (including buggy rules)
equivalence	tests whether two terms are semantically equivalent
similarity	tests whether two terms are (nearly) the same
suitable	identifies which terms can be solved by the strategy
finished	checks whether a term is in a solved form
exercise id	identifier that uniquely determines the exercise class
status	stability of the exercise class
parser	parser for terms
pretty-printer	pretty-printer for terms (inverse of parsing)
navigation	supports traversals over terms
rule ordering	tiebreaker when more than one rule can be used
examples	list of examples, each with an assigned difficulty
random generator	generates random terms of a certain difficulty
test generator	generates random test cases (including corner cases)

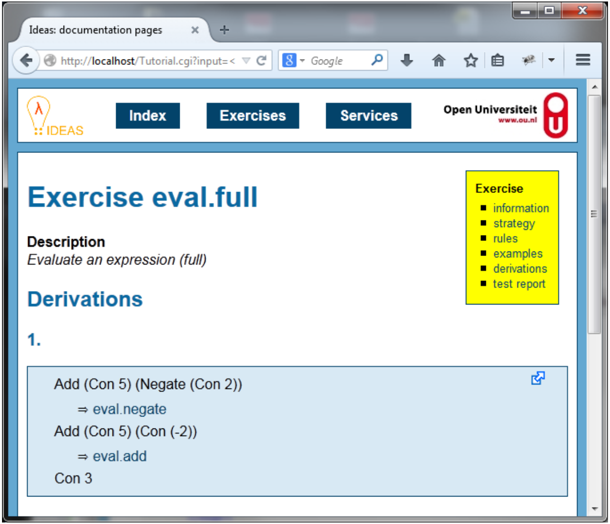
About the Ideas framework

- Latest release: version 1.2 (May 2014)
- Just over 10,000 lines of Haskell code (in 110 modules)
- `http://hackage.haskell.org/package/ideas`

How to interact with a domain reasoner?

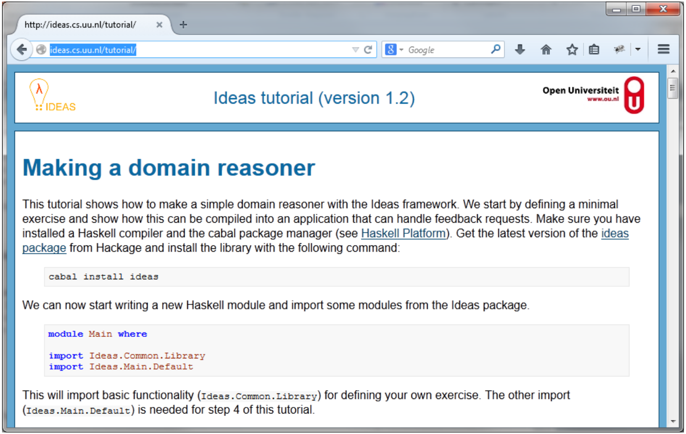
- Use the Haskell interpreter (ghci)
- Compile to a cgi binary (with support for HTML) and deploy on your localhost; use a browser
- Compile and send a request from the command-line (file)

Domain reasoner in browser



Ideas tutorial

Visit `http://ideas.cs.uu.nl/tutorial/`



Lab assignment

Start version has:

- Simple arithmetic expression language
- Two evaluation rules

$$\textbf{data}\ Expr = Add\ Expr\ Expr \mid Negate\ Expr \mid Con\ Int$$

Exercises:

1. Add multiplication to the expression language (and extend the evaluation strategy)
2. Add distribution rules to the strategy
3. Add support for calculating with fractions (e.g. $\frac{5}{7} + \frac{1}{2}$)
   - Find the least common multiple of the denominators
   - Rewrite top-heavy fractions to mixed fractions (e.g. $1\frac{3}{14}$)