OpenUniversiteitNederland

Canonical Forms in Interactive Exercise Assistants

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

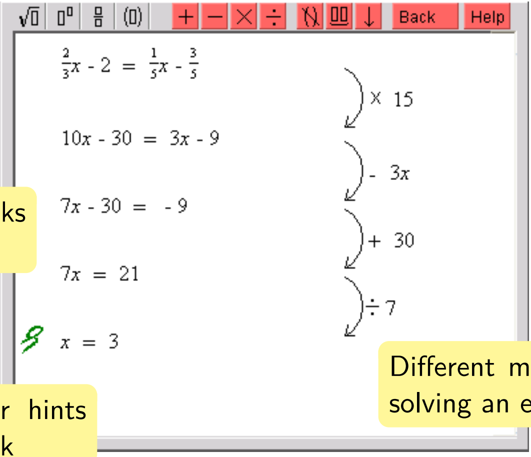
[1] Open Universiteit Nederland
[2] Universiteit Utrecht, The Netherlands

11 July 2009 (MKM'09)
Grand Bend, Canada

DWO Math Environment

- Applet by Freudenthal Institute for linear equations

Buttons for the operations



The tool checks each step

No further hints or feedback

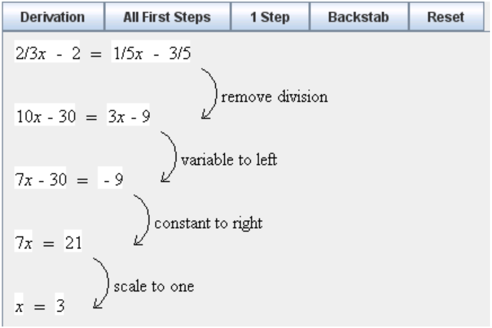
Different modes for solving an exercise

Interactive exercises

Ideally, interactive exercise assistants do more than validating submitted answers:

- Present worked-out examples
- Provide hints how to proceed
- Comment on the direction of a step

A prototype applet of DWO, extended with our feedback services



Finer control over symbolic simplification

A popular approach for exercise assistants is to delegate all computations to a CAS:

- ✓ Gives good instant results
- × Cannot be configured easily for finer control
- × Not designed for interaction with exercise assistants

Finer control over symbolic simplification

A popular approach for exercise assistants is to delegate all computations to a CAS:

- ✓ Gives good instant results
- × Cannot be configured easily for finer control
- × Not designed for interaction with exercise assistants

We follow Beeson's guidelines:

- Cognitive fidelity: software solves problem as student does
- Glassbox computation: you can see how software solves the problem
- Customization of software to level of user

Our approach: strategies for exercises

- Strategies (MKM'08) specify how to solve an exercise incrementally:

$$\begin{aligned} \textit{solveEquation} \;=\; & \textit{repeat}\,(\textsc{Merge} <|> \textsc{Distribute} <|> \textsc{NoDivision}) \\ & <\!\!*\!\!> \textit{try}\ \textsc{VarLeft} <\!\!*\!\!> \textit{try}\ \textsc{ConRight} <\!\!*\!\!> \textit{try}\ \textsc{Scale} \end{aligned}$$

- Feedback can be calculated automatically from a strategy

Our approach: strategies for exercises

- Strategies (MKM'08) specify how to solve an exercise incrementally:

$$
\begin{aligned}
\mathit{solveEquation} \;=\;& \mathit{repeat}\ (\textsc{Merge} <|> \textsc{Distribute} <|> \textsc{NoDivision}) \\
& <\!\!*\!\!> \mathit{try}\ \textsc{VarLeft} <\!\!*\!\!> \mathit{try}\ \textsc{ConRight} <\!\!*\!\!> \mathit{try}\ \textsc{Scale}
\end{aligned}
$$

- Feedback can be calculated automatically from a strategy
- Challenges with mathematical domains are:
  1. How to describe the rewrite rules without worrying about the underlying representation
  2. How to show "intuitive" terms only
  3. Granularity of rewrite steps should match users background
  4. How to recognize strategy steps performed by a student

Our approach: strategies for exercises

- Strategies (MKM'08) specify how to solve an exercise incrementally:

$$\begin{aligned} \textit{solveEquation} \;&=\; \textit{repeat}\ (\textsc{Merge} <|> \textsc{Distribute} <|> \textsc{NoDivision}) \\ &\quad <\!\!*\!\!> \textit{try}\ \textsc{VarLeft} <\!\!*\!\!> \textit{try}\ \textsc{ConRight} <\!\!*\!\!> \textit{try}\ \textsc{Scale} \end{aligned}$$

- Feedback can be calculated automatically from a strategy
- Challenges with mathematical domains are:
  1. How to describe the rewrite rules without worrying about the underlying representation
  2. How to show "intuitive" terms only
  3. Granularity of rewrite steps should match users background
  4. How to recognize strategy steps performed by a student
- We propose to use views

Overview

Views and canonical forms

$$\textbf{data}\ \mathit{View}\ a\ b = \mathit{View}\{\mathit{match} :: a \rightarrow \mathit{Maybe}\ b, \mathit{build} :: b \rightarrow a\}$$

Examples:

$$3x - (1 - x) \overset{\mathit{match}}{\rightsquigarrow} [3x, -1, x] \overset{\mathit{build}}{\rightsquigarrow} 4x - 1$$

$$\frac{1}{3} + \frac{1}{4} \overset{\mathit{match}}{\rightsquigarrow} \frac{7}{12} \overset{\mathit{build}}{\rightsquigarrow} \frac{7}{12}$$

Views and canonical forms

$$\textbf{data}\ View\ a\ b = View\{\, match :: a \rightarrow Maybe\ b, build :: b \rightarrow a\,\}$$

Examples:

$$3x - (1 - x) \overset{match}{\rightsquigarrow} [3x, -1, x] \overset{build}{\rightsquigarrow} 4x - 1$$

$$\frac{1}{3} + \frac{1}{4} \overset{match}{\rightsquigarrow} \frac{7}{12} \overset{build}{\rightsquigarrow} \frac{7}{12}$$

- Simplification (first *match*, then *build*) returns the canonical element, and has the following properties:
  - Simplification is idempotent
  - Simplification preserves semantics
- Based on views proposed by Wadler (POPL, 1987)
- Our views abstract over algebraic laws, and hide the underlying representation

Abstract syntax

$$
\begin{aligned}
\textbf{data}\ \mathit{Expr} = {} & \mathit{Nat}\ \mathit{Integer} \\
\mid {} & \mathit{Var}\ \mathit{String} \\
\mid {} & \mathit{Negate}\ \mathit{Expr} \\
\mid {} & \mathit{Expr}\ {:}{+}{:}\ \mathit{Expr} \\
\mid {} & \mathit{Expr}\ {:}{\times}{:}\ \mathit{Expr} \\
\mid {} & \mathit{Expr}\ {:}{-}{:}\ \mathit{Expr} \\
\mid {} & \mathit{Expr}\ {:}{/}{:}\ \mathit{Expr}
\end{aligned}
$$

- We use the functional programming language Haskell
- Close to concrete syntax (including syntactic sugar)
- Similar to OpenMath and MathML, but less verbose

Example: lowest common denominator

Determine the *lcd* of two fractions:

$$
\begin{aligned}
&lcd :: Expr \rightarrow Maybe\ Integer\\
&lcd\ ((a :/: Nat\ b) :+: (c :/: Nat\ d)) = Just\ (lcm\ b\ d)\\
&lcd\ \_ \qquad\qquad\qquad\qquad\qquad\qquad = Nothing
\end{aligned}
$$

Intuitive definition with pattern matching, but not suitable:

- Subtraction: $\frac{2}{3} - \frac{1}{4}$
- Negation: $-\frac{1}{4} + \frac{2}{3}$, or $\frac{1}{-4} + \frac{2}{3}$

Introducing the plus view

Match a plus at top-level:

$$
\begin{aligned}
&\mathit{matchPlus} :: \mathit{Expr} \to \mathit{Maybe}\ (\mathit{Expr}, \mathit{Expr}) \\
&\mathit{matchPlus}\ (a :\!+\!: b) &&= \mathit{Just}\ (a, b) \\
&\mathit{matchPlus}\ (a :\!-\!: b) &&= \mathit{Just}\ (a, \mathit{Negate}\ b) \\
&\mathit{matchPlus}\ (\mathit{Negate}\ a) &&= \mathbf{do}\ (x, y) \leftarrow \mathit{matchPlus}\ a \\
& && \qquad\ \ \mathit{Just}\ (\mathit{Negate}\ x, \mathit{Negate}\ y) \\
&\mathit{matchPlus}\ \_ &&= \mathit{Nothing}
\end{aligned}
$$

- Based on algebraic laws:

$$
\begin{aligned}
a - b &= a + (-b) \\
-(a + b) &= (-a) + (-b)
\end{aligned}
$$

- Not used: law for distribution

Builder for the plus view

$$
\begin{aligned}
&(.\!+\!.) :: \mathit{Expr} \to \mathit{Expr} \to \mathit{Expr}\\
&\mathit{Nat}\ 0\ .\!+\!.\ b = b\\
&a\ .\!+\!.\ \mathit{Nat}\ 0 = a\\
&a\ .\!+\!.\ \mathit{Negate}\ b = a :\!-\!: b\\
&a\ .\!+\!.\ b = a :\!+\!: b
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{plusView} :: \mathit{View}\ \mathit{Expr}\ (\mathit{Expr}, \mathit{Expr})\\
&\mathit{plusView} = \mathit{View}\{\mathit{match} = \mathit{matchPlus}, \mathit{build} = \mathit{uncurry}\ (.\!+\!.)\}
\end{aligned}
$$

- Based on algebraic laws:

$$
\begin{aligned}
0 + a &= a\\
a - b &= a + (-b)
\end{aligned}
$$

- Builder returns intuitive terms
- Similarly, we define views for division, constants, etc.

Improved definition for *lcd*

Composing views with arrow combinators:

$$
\begin{aligned}
(\ggg) &:: \mathit{View}\ a\ b \to \mathit{View}\ b\ c \to \mathit{View}\ a\ c \\
(\ast\!\ast\!\ast) &:: \mathit{View}\ a\ c \to \mathit{View}\ b\ d \to \mathit{View}\ (a, b)\ (c, d) \\
\mathit{second} &:: \mathit{View}\ b\ c \to \mathit{View}\ (a, b)\ (a, c)
\end{aligned}
$$

Improved definition for *lcd*

Composing views with arrow combinators:

$$(\ggg) :: View\ a\ b \rightarrow View\ b\ c \rightarrow View\ a\ c$$
$$(***) :: View\ a\ c \rightarrow View\ b\ d \rightarrow View\ (a, b)\ (c, d)$$
$$second :: View\ b\ c \rightarrow View\ (a, b)\ (a, c)$$

New definition for the *lcd* of two fractions:

$$lcdView :: View\ Expr\ ((Expr, Integer), (Expr, Integer))$$
$$lcdView = \textbf{let}\ v = divView \ggg second\ conView$$
$$\qquad\qquad \textbf{in}\ \ plusView \ggg (v *** v)$$

$$lcd :: Expr \rightarrow Maybe\ Integer$$
$$lcd\ e = \textbf{do}\ ((a, b), (c, d)) \leftarrow match\ lcdView\ e$$
$$\qquad\qquad Just\ (lcm\ b\ d)$$

Summary

Summary:

- A view consists of two functions (*match* and *build*)
- A view specifies a canonical form
- Views can be combined, and they are reusable

The four challenges are:

1. How to describe the rewrite rules without worrying about the underlying representation (discussed)
2. How to show "intuitive" terms only (discussed)
3. Granularity of rewrite steps should match users background
4. How to recognize strategy steps performed by a student

Granularity of rewrite steps

Assumptions about user level for “linear equation” exercise:

- Associativity is implicit (but preserve order if possible)
- Calculating with constants is a prerequisite
- Distribution of $\times$ over $+$ is an explicit step

Granularity of rewrite steps

Assumptions about user level for "linear equation" exercise:

- Associativity is implicit (but preserve order if possible)
- Calculating with constants is a prerequisite
- Distribution of $\times$ over $+$ is an explicit step

Hence, we choose the following operations on an equation:

1. Add term to both sides
2. Multiply both sides
3. Apply distribution law ("remove parentheses")
4. Merge "similar" terms

Last two operations will be made more precise with views

Order preserving summation

- Associativity of addition, but not commutativity
- List is the natural data structure
- **Example:**

$$3x - (1 - x) \overset{\textit{match}}{\rightsquigarrow} [3x, -1, x] \overset{\textit{build}}{\rightsquigarrow} 3x - 1 + x$$

$$
\begin{aligned}
&\textit{matchSum} :: \textit{Expr} \rightarrow \textit{Maybe}\ [\textit{Expr}]\\
&\textit{matchSum} = \textit{Just} \circ f\ \textit{False}\\
&\quad \textbf{where}\ f\ n\ (a :+: b) = f\ n\ a +\!\!+ f\ n\ b\\
&\qquad\quad\ f\ n\ (a :-: b) = f\ n\ a +\!\!+ f\ (\neg\ n)\ b\\
&\qquad\quad\ f\ n\ (\textit{Negate}\ a) = f\ (\neg\ n)\ a\\
&\qquad\quad\ f\ n\ a = [\textbf{if}\ n\ \textbf{then}\ \textit{Negate}\ a\ \textbf{else}\ a]
\end{aligned}
$$

$$
\begin{aligned}
&\textit{buildSum} :: [\textit{Expr}] \rightarrow \textit{Expr}\\
&\textit{buildSum} = \textit{foldl}\ (.+.)\ (\textit{Nat}\ 0)
\end{aligned}
$$

Merging similar terms

- Easier with sum view than with original representation
- Use product view for non-constant terms
- **Example:**

$$\begin{aligned} & 3x - (1 - x) + 4 \\ & \overset{\textit{match}}{\rightsquigarrow} [3x, -1, x, 4] \\ & \overset{\textit{merge}}{\rightsquigarrow} [4x, 3] \\ & \overset{\textit{build}}{\rightsquigarrow} 4x + 3 \end{aligned}$$

- Normalization is again a view: just combine *merge* with the *build* function
- Details can be found in the paper

Recognizing strategy steps

- Syntactic equality is too strict for recognizing input
- Each view defines an equivalence relation
- Use different equivalence relations for recognizing intermediate student answers:
  - Use equation view for semantic equivalence:
    $4x + 3 = 3x + 5 \overset{match}{\rightsquigarrow} 2$
  - Use linear view for normalizing the sides of an equation (in the form $a{\cdot}x + b$):
    $4(3x - 2) \overset{match}{\rightsquigarrow} (12, -8)$
  - The sum view and product view are needed for recognizing the distribution rule

Conclusions

- Views hide the underlying representation by abstracting over algebraic laws
- A view corresponds to a level of detail
- Views are applicable to domains other than mathematics
- Views are reusable, and can be combined with standard rewriting technology giving good results
- Multiple views can coexist in a strategy specification
- The presented techniques resulted in a working prototype for solving linear equations by establishing a binding with the DWO