Canonical Forms in Interactive Exercise Assistants

Bastiaan Heeren

Johan Jeuring

Technical Report UU-CS-2009-011
April 2009

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Canonical Forms in Interactive Exercise Assistants

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

[1]School of Computer Science, Open Universiteit Nederland
P.O.Box 2960, 6401 DL Heerlen, The Netherlands
{bhr,jje}@ou.nl
[2] Department of Information and Computing Sciences, Universiteit Utrecht

Abstract. Interactive exercise assistants support students in practicing exercises, and acquiring procedural skills. Many mathematical topics can be practiced in such assistants. Ideally, an interactive exercise assistant not only validates final answers, but also comments on intermediate steps submitted by a student, provides hints on how to proceed, and presents worked-out examples. For these purposes, fine control over the symbolic simplification procedures of the underlying mathematical machinery is needed.
In this paper, we introduce views for mathematical expressions. A view defines an equivalence relation by choosing a canonical form of mathematical expressions. We use views to track and recognize intermediate answers, to help in presenting expressions to a user, and to control the granularity of the steps in worked-out examples. We develop the concept of a view, discuss the laws it satisfies, and show how views are composed, which means that they can be used for multiple exercise classes.

1 Introduction

An interactive exercise assistant supports a student who stepwise solves an exercise. A student gets an exercise, for example about solving a system of linear equations, and takes steps towards the solution. Examples of interactive exercise assistants for mathematics are the Digital Mathematics Environment (DWO) of the Freudenthal Institute [5], MathDox [7], Aplusix [6], MathPert [3], WIMS [8], ActiveMath [9], and many more. Here is an example of a series of (correct) steps a student makes when solving a linear equation:

$$
\begin{array}{lrcll}
 & 1-\frac{4x+2}{3} & = & 3x-\frac{5x-1}{4} & \\
\iff & 12-4(4x+2) & = & 36x-3(5x-1) & \text{times } 12 \\
\iff & 12-16x-8 & = & 36x-3(5x-1) & \text{distribution} \\
\iff & 12-16x-8 & = & 36x-15x+3 & \text{distribution} \\
\iff & 4-16x & = & 21x+3 & \text{merging} \\
\iff & 4-37x & = & 3 & \text{minus } 21x \\
\iff & -37x & = & -1 & \text{minus } 4 \\
\iff & x & = & \frac{1}{37} & \text{divide by } -37
\end{array}
$$

Most interactive exercise assistants would accept this derivation: they check that each step is correct by calculating that the solution of the equation has not changed. The comments on the right-hand side suggest that a single rewrite rule is applied at each step. However, simplification steps are silently performed at all these steps. For instance, unraveling the simplification of the left-hand side after the first step (multiply both sides by 12) gives:

$$
\begin{array}{lll}
 & (1 - \frac{4 \cdot x + 2}{3}) \cdot 12 & \\
\iff & 1 \cdot 12 - \frac{4 \cdot x + 2}{3} \cdot 12 & (a - b) \cdot c = a \cdot c - b \cdot c \\
\iff & 12 - \frac{4 \cdot x + 2}{3} \cdot 12 & \text{constant folding} \\
\iff & 12 - \frac{(4 \cdot x + 2) \cdot 12}{3} & \frac{a}{b} \cdot c = \frac{a \cdot c}{b} \\
\iff & 12 - \frac{12 \cdot (4 \cdot x + 2)}{3} & a \cdot b = b \cdot a \\
\iff & 12 - \frac{12}{3} \cdot (4 \cdot x + 2) & \frac{a \cdot c}{b} = \frac{a}{b} \cdot c \\
\iff & 12 - 4 \cdot (4 \cdot x + 2) & \text{constant folding}
\end{array}
$$

The single step in the first derivation actually consists of around 15 basic rewrite steps. Expanding the steps in this derivation would make it very lengthy.

The first derivation shows a sequence of simplified terms that are in some canonical form. A *canonical form* of a mathematical expression is a standard way of (re)presenting that expression. These canonical forms play an important role in interactive exercise assistants, for instance for simplifying terms. The exercise assistants we have tested all have some notion of canonical forms, but their application is often rather subtle.

Most of the exercise assistants mentioned earlier can perform rewrite steps, followed by automatic simplification to some canonical form, and they can check that a student has not changed the solution of the exercise, which would indicate an error. These tools do not have explicit knowledge about strategies for solving the exercise, however. Therefore, they do not check whether the step made by the student is on the optimal path to the solution, whether the student makes progress, or give hints to students that are stuck. For these purposes we use strategies [10] in our feedback services. A strategy for an exercise describes exactly how to stepwise obtain a solution to an exercise. Strategies can be used to monitor progress, to check whether or not a step submitted by a student follows the strategy, to give hints, and to generate worked-out solutions.

Strategies have to include knowledge about canonical forms of expressions: we do not want to show the basic simplification steps in our hints or worked-out solutions, and we also do not want to force students to perform these simple rewriting steps. In this paper, we investigate the following research questions:

- *Economy of rules (Section 2).* How can we describe rewrite rules on a mathematical domain using a limited set of rules? For example, we want the rewrite rule $\frac{a}{b} + \frac{c}{b} = \frac{a+c}{b}$, but not also $-\frac{a}{b} + \frac{c}{b} = \frac{-a+c}{b}$ and $\frac{a}{b} - \frac{c}{b} = \frac{a-c}{b}$.
- *Canonical form (Section 2).* How can we ensure that we only show intuitive representations of expressions to users in worked-out examples? For example, $a + (-b)$ should be presented as $a - b$. And we should never show -0.

– *Granularity (Section 3).* How can we describe rewrite steps of different granularity, to mimic the typical steps users take? Users with different backgrounds will take steps of different granularity: a university student will usually take fewer steps in a calculation than a 10-year old.
– *Recognizing strategy steps (Section 4).* How can we determine that a student has performed a step that matches the step prescribed by the strategy? A user might have performed a step, but forgotten some of the simplification steps we assume. We want to accept automatic simplification, but we also want to accept partly simplified steps.

In this paper we present so-called *views* [18] to address these questions. Views are used to describe and calculate canonical forms, at each step. Our main contributions are the development of views, and the description of a derivation step in terms of a rewrite rule and a view in which the rule is applied. We use the functional programming language Haskell [14] to explain our ideas, and to show some actual code snippets of our implementation.

2 Views

In this section, we gradually explore the concepts of views and canonical forms. Our views are based on the views proposed by Wadler [18]. His views make it possible to combine pattern matching with abstract data types, and have their origin in research on programming languages. We use views for a very different purpose, namely for rewriting in the context of an interactive exercise assistant. Our views abstract over algebraic laws, and help to hide the underlying representation of mathematical objects.

We start by introducing a representation for mathematical expressions in Section 2.1, which we use in an exercise to perform some basic calculations with fractions. This will be our running example throughout this section. We discuss a number of definitions for matching expressions (Section 2.2), and show how these functions can be combined in Section 2.3. In the last two sections we make the concept of a view more precise with some definitions and properties, and we focus on choosing the canonical form of a view.

2.1 Abstract syntax

We use the following abstract syntax to represent mathematical expressions. Abstract syntax is represented by a data type in Haskell, the programming language in which we have implemented our exercise assistants.

```
data Expr = Nat Integer    | Var String    | Negate Expr
          | Expr :+: Expr | Expr :*: Expr | Expr :-: Expr | Expr :/: Expr
```

Expressions are constructed from the natural numbers (*Nat*) and variables (*Var*), and can be combined into larger expressions using unary negation and the binary operators for addition, multiplication, subtraction, and division. The *Nat*

constructor can only have a non-negative number, and we will maintain this invariant. Hence, the constant value -5 is represented by *Negate* (*Nat* 5). This data type is close to the concrete syntax of mathematical expressions, which makes it suitable for interactive exercise assistants since we can truthfully represent terms that are entered by users of the exercise assistants. The disadvantage of this representation is that it complicates the formulation of rules and strategies. We have to deal with atypical expressions, such as $x + (-2)$ or -0, and we want to avoid reporting these to our users.

In the remainder of the paper, we use the infix constructors surrounded by colons for the abstract representation of mathematical objects. Other representations, such as OpenMath [15] and MathML [17], are quite similar, be it more verbose.

2.2 Matching with views

Consider the exercise of adding two fractions, targeted at primary school pupils. A first step would be to let the fractions have the same denominator, and for this one typically computes the lowest common denominator (*lcd*). Given an expression of type *Expr*, the following function returns its *lcd*:

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

where *lcm* is a predefined function which calculates the lowest common multiple of two integers. The function *lcd* is partial, which is reflected by the *Maybe* type constructor. The function only works for expressions of the same form as the left-hand side pattern: for all other values, the function fails in computing the *lcd* (that is, *Nothing* is returned). In fact, our intuitive definition of *lcd* is unsuitable for our *Expr* data type:

- Suppose we also want to use *lcd* when subtracting one fraction from another, e.g., $\frac{2}{3} - \frac{1}{4}$. This requires an extra case for our definition, in which we match on the constructor :−: at top-level.
- What if the first fraction is negative, as in $-\frac{1}{4} + \frac{2}{3}$? In combination with support for subtraction, this requires a substantial number of new cases.
- The denominator can also be negative ($\frac{1}{-4} + \frac{2}{3}$), leading to even more combinations that have to be considered.

In this scenario, pattern matching is not going to work because the number of cases will grow rapidly. Instead, we introduce *views* [18] to gain the flexibility we are searching for, without obscuring *lcd*'s definition. A view allows us to represent a collection of expressions by means of expressions of a particular canonical form. A view consists of two components: a function for mapping an expression to a canonical form, and a function mapping a canonical form back to an expression. We now introduce the former component, and defer the latter to Section 2.4.

Addition :

[A1] $a + (b + c) = (a + b) + c$
[A2] $a + b = b + a$
[A3] $0 + a = a$

Multiplication :

[M1] $a{\cdot}(b{\cdot}c) = (a{\cdot}b){\cdot}c$
[M2] $a{\cdot}b = b{\cdot}a$
[M3] $0{\cdot}a = 0$
[M4] $1{\cdot}a = a$
[M5] $a{\cdot}(b + c) = (a{\cdot}b) + (a{\cdot}c)$

Equation :

[E1] $(a = b) = (a + c = b + c)$
[E2] $(a = b) = (a{\cdot}c = b{\cdot}c) \quad (c \neq 0)$

Negation :

[N1] $-(-a) = a$
[N2] $a - a = 0$
[N3] $a - b = a + (-b)$
[N4] $-(a + b) = (-a) + (-b)$
[N5] $-(a{\cdot}b) = (-a){\cdot}b$
[N6] $-(a \,/\, b) = (-a) \,/\, b$

Division :

[D1] $a \,/\, a = 1 \quad (a \neq 0)$
[D2] $a \,/\, 1 = a$
[D3] $a \,/\, (b \,/\, c) = a{\cdot}(c \,/\, b) \quad (c \neq 0)$
[D4] $(a \,/\, b) \,/\, c = a \,/\, (b{\cdot}c)$
[D5] $a{\cdot}(b \,/\, c) = (a{\cdot}b) \,/\, c$
[D6] $(a + b) \,/\, c = (a \,/\, c) + (b \,/\, c)$

Fig. 1. Basic algebraic laws

Let the type *Match a b* be an abbreviation for a partial function from type a to type b:

```
type Match a b = a → Maybe b
```

The intuition is that we view a value of type a in some specific way, and possibly as a value of a different type.

At top-level, *lcd* is expecting an addition, and we can apply some algebraic laws to put an expression into the expected form (if possible). Figure 1 lists a number of basic algebraic laws. The function *matchPlus* tries to match a plus at top-level, and uses laws [N3] and [N4] to do so. If it succeeds, it returns a pair containing the operands of the addition.

```
matchPlus :: Match Expr (Expr, Expr)
matchPlus (a :+: b)    = Just (a, b)
matchPlus (a :-: b)    = Just (a, Negate b)                -- law [N3]
matchPlus (Negate a) = do (x, y) ← matchPlus a
                          Just (Negate x, Negate y)        -- law [N4]
matchPlus _            = Nothing
```

In the case for negation, we call the function recursively on the negated term. If the call succeeds with a pair (x, y), both operands are negated. Preferably, a helper-function is used (instead of the constructor *Negate*) that removes double negations (law [N1]). More laws could be used in the above definition, such as the distribution rule [M5]. The challenge was to define *lcd* for adding fractions. Given our targeted audience, we want this distribution to be performed by the user prior to the addition. Therefore, we do not incorporate the law in *matchPlus*.

In the same fashion, we introduce a function to match a division. Here, we only push negations into the numerator.

```
matchDiv :: Match Expr (Expr, Expr)
matchDiv (a :/: b)     = Just (a, b)
matchDiv (Negate a) = do (x, y) ← matchDiv a
                         Just (Negate x, y)   -- law [N6]
matchDiv _             = Nothing
```

The third match-function alleviates the problems caused by the *Nat* constructor only accepting non-negative constants. This function matches a natural number preceded by one or more negations, and returns an integer value.

```
matchCon :: Match Expr Integer
matchCon (Nat n)     = Just n
matchCon (Negate e) = do c ← matchCon e
                         Just (−c)   -- constant folding
matchCon _           = Nothing
```

Note that $(-c)$ is the primitive negation operation applied to integer c.

2.3 Composing match-functions

With the helper-functions for matching expressions, we can define *lcd*. With some "plumbing" in the *Maybe* monad, this is not too difficult. However, we first present a number of combinators for composing match-functions, which will make it even more straightforward to write *lcd*.

The type constructor *Match* precisely fits the *Arrow* interface [13], which is a general interface to computation. In our case, we modeled partiality by introducing the *Maybe* monad, which turns *Match* into a Kleisli arrow: an arrow of type $a \to m\ b$ for some monad m. The advantage of turning *Match* into an arrow is that this gives us a set of combinators, without too much effort. The combinator $(>\!\!>\!\!>)$, for example, has type $Match\ a\ b \to Match\ b\ c \to Match\ a\ c$, and allows us to sequentially combine two matches: $m >\!\!>\!\!> n$ first matches with m and then with n. Other arrow combinators are $(***)$, which performs two matches in parallel, and *second*, which performs a match on the second component of a pair.

With the arrow combinators, we define *matchTwoFractions*, which views an expression as the sum of two fractions with constants in the denominators.

```
matchTwoFractions :: Match Expr ((Expr, Integer), (Expr, Integer))
matchTwoFractions = matchPlus >>> (matchFraction *** matchFraction)
  where
    matchFraction :: Match Expr (Expr, Integer)
    matchFraction = matchDiv >>> second matchCon
```

For each match-function we have made explicit the laws on which it is based. Therefore, it is easy to determine the laws involved in combinations of match-functions such as *matchTwoFractions*. We give an improved definition for *lcd*:

$$
\begin{aligned}
&\mathit{lcd} :: \mathit{Expr} \to \mathit{Maybe}\ \mathit{Integer} \\
&\mathit{lcd}\ e = \mathbf{do}\ ((a,b),(c,d)) \leftarrow \mathit{matchTwoFractions}\ e \\
&\qquad\qquad \mathit{Just}\ (\mathit{lcm}\ b\ d)
\end{aligned}
$$

2.4 Defining views

This section defines views. We explain how views are used to calculate canonical forms, and which properties they satisfy. The definitions are given in Haskell.

So far, only functions for matching have been considered. With each partial function from a to b, we associate a *build* function which returns a value in the original domain. A view pairs a *match* and *build* function.

$$\mathbf{data}\ \mathit{View}\ a\ b = \mathit{View}\{\mathit{match} :: \mathit{Match}\ a\ b, \mathit{build} :: b \to a\}$$

For each view we assume that the two functions define a canonical form. We make this idea more precise in the definition of the function *canonical*, which returns the canonical form of an element under a given view:

$$
\begin{aligned}
&\mathit{canonical} :: \mathit{View}\ a\ b \to a \to \mathit{Maybe}\ a \\
&\mathit{canonical}\ \mathit{view}\ a = \mathbf{do}\ b \leftarrow \mathit{match}\ \mathit{view}\ a \\
&\qquad\qquad\qquad\qquad \mathit{Just}\ (\mathit{build}\ \mathit{view}\ b)
\end{aligned}
$$

We apply the *match* function of the view on an element, and on a successful match, we use the *build* function to return to the original domain. For convenience, we also define a simplification function, which returns the value at hand on a failing match:

$$
\begin{aligned}
&\mathit{simplify} :: \mathit{View}\ a\ b \to a \to a \\
&\mathit{simplify}\ \mathit{view}\ a = \mathit{fromMaybe}\ a\ (\mathit{canonical}\ \mathit{view}\ a)
\end{aligned}
$$

The following properties of the *simplify* function should hold for all views, establishing a property for match and build pairs.

Property 1 (Idempotence). For every view v, *simplify* v is expected to be an idempotent function. If this is not the case, we say that view v is improper.

Property 2 (Soundness). Simplification with a view v should preserve the semantics of an object. Let a be some element in the domain of view v, and let $[\![\cdot]\!]$ denote the semantics of that domain. Then $[\![a]\!] = [\![\mathit{simplify}\ v\ a]\!]$.

Because each proper view defines a canonical form, it also defines an equivalence relation. Two elements can be tested for equivalence under a view by comparing their canonical forms. We use *simplify* to do the job:

```
viewEquivalent :: Eq a ⇒ View a b → a → a → Bool
viewEquivalent view x y = simplify view x ≡ simplify view y
```

The overloaded equality operator ≡ belongs to the *Eq* type class, and is normally implemented as equality on the abstract syntax. Hence, if *view* does not apply to x nor y, *viewEquivalent* tests for syntactic equality of x and y.

The functions for matching can be composed using the arrow interface, and likewise, we can compose views. In fact, we use the same interface for the *View* type constructor, which enables us to combine views. The *build* operation is also an arrow since it is an ordinary function, except in the opposite direction. As a consequence, we cannot implement the *pure* function for a view because we cannot automatically compute the inverse of a function. Views are closely related to the bidirectional arrows proposed by Alimarine et al. [1].

2.5 Choosing the canonical form

We continue the example of adding two fractions. Now that we can determine the lowest common denominator of two fractions (*lcd*), we need a rule to scale one of these fractions accordingly. For this purpose, we define *build* functions for *matchCon*, *matchPlus*, and *matchDiv*. The view for positive and negative constants (*conView*) pairs *matchCon* with a function that turns an integer value back into an *Expr* value.

```
conView :: View Expr Integer
conView = View{ match = matchCon, build = buildCon }

buildCon :: Integer → Expr
buildCon n | n ⩾ 0      = Nat n
           | otherwise = Negate (Nat (abs n))
```

This definition results in a proper view, and it respects the invariant imposed by the *Nat* constructor. When defining the builder for the plus view (the counterpart of *matchPlus*), we have another look at the algebraic laws in Figure 1. The matching function maps each of the expressions $3 + (-5)$, $3 - 5$, and $-(-3 + 5)$ to the pair $(3, -5)$. In the definition of the builder, we choose $3 - 5$ as the canonical representation for this pair.

```
(.+.) :: Expr → Expr → Expr
Nat 0 .+. b          = b          -- law [A3]
a     .+. Nat 0      = a          -- law [A3] (and [A2])
a     .+. Negate b = a :−: b      -- law [N3]
a     .+. b          = a :+: b
```

Here, we write the builder as the infix function .+., which should not be confused with the constructor function :+:. With the function *uncurry*, we turn .+. into a function of type $(Expr, Expr) \to Expr$, which we use in the plus view.

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

The builder function of the division view uses law [N6]: we omit its definition.

We conclude this section with a definition for the rule that scales a fraction to a certain denominator, in which we use a composed view both for matching and for building. For this occasion, we make a view that constrains the numerator and the denominator to be constant.

$$
\begin{aligned}
&\mathit{fractionView} :: \mathit{View}\ \mathit{Expr}\ (\mathit{Integer}, \mathit{Integer}) \\
&\mathit{fractionView} = \mathit{divView} >\!\!>\!\!> (\mathit{conView} \ast\!\!\ast\!\!\ast\ \mathit{conView})
\end{aligned}
$$

The rule that scales a fraction can then be defined as follows:

$$
\begin{aligned}
&\mathit{scaleFraction} :: \mathit{Integer} \to \mathit{Expr} \to \mathit{Maybe}\ \mathit{Expr} \\
&\mathit{scaleFraction}\ n\ e = \mathbf{do}\ (a, b) \leftarrow \mathit{match}\ \mathit{fractionView}\ e \\
&\qquad\qquad\qquad\qquad\quad \mathbf{let}\ (c, \mathit{zero}) = n\ \text{`}\mathit{divMod}\text{`}\ b \\
&\qquad\qquad\qquad\qquad\quad \mathit{guard}\ (\mathit{zero} \equiv 0) \\
&\qquad\qquad\qquad\qquad\quad \mathit{Just}\ (\mathit{build}\ \mathit{fractionView}\ (c \ast a, n))
\end{aligned}
$$

We calculate the scale factor (c), and test whether the target value of the denominator (n) is a multiple of the old value (b). Then, we build an expression from the scaled fraction using the same view.

3 Granularity of rewrite steps

In this section, we return to the example of the introduction, and we take a closer look at the size (or granularity) of the rewrite steps in the derivation. For some exercises, the steps that a student is expected to take correspond exactly to the laws that are known for that domain. This is, for instance, the case in most exercise assistants in the area of logic, where propositions have to be manipulated using only a handful of rules, typically the ones appearing in textbooks on this subject. In such a scenario, the granularity of user steps is not an issue. In other cases, terms can be simplified automatically without an interest in intermediate values. For example, when performing Gaussian elimination, the focus of the student should be on applying the elementary row operations, not on simplifying the elements appearing in the matrix. It seems reasonable that a tool performs these simplifications automatically.

In the example of solving a linear equation, we are interested in intermediate results, but the steps should be at a conceptually higher level than the algebraic laws listed in Figure 1. Worked-out examples that are generated by the system should be at the right conceptual level (like the derivation in the introduction), just as hints about the direction to go. We start by making some of our assumptions explicit before we discuss the conceptual level of this exercise.

– Associativity of operators is implicit, meaning that a user cannot and should not distinguish $a+(b+c)$ from $(a+b)+c$. The system can thus minimize the

view	view type	description
plusView	$(Expr, Expr)$	match an addition (:+:) at top-level
divView	$(Expr, Expr)$	match a division (:/:) at top-level
conView	*Integer*	match a natural number, possibly preceded by some negations
sumView	$[Expr]$	order preserving summation ($e_1 + \ldots + e_n$)
productView	$(Bool, [Expr])$	order preserving multiplication ($e_1 \cdot \ldots \cdot e_n$): the Bool indicates negation of the product
rationalView	*Rational*	reduce by folding constants recursively
linearView	$(Rational, Rational)$	normalize a linear expression in x: use all laws to turn the expression into the form $a \cdot x + b$

Fig. 2. Summary of views on expressions

use of parentheses in presenting terms. Commutativity, on the other hand, should be used with care. We want to respect the order in which terms appear as much as possible for a better user experience.
- Constant terms are normalized aggressively: the skills to manipulate fractions and integers are assumed to be present.
- The distribution of multiplication over addition (law [M5]) is an explicit step in the derivation. Laws to manipulate the sign of a term (laws [N1] up to [N6]) can be performed automatically.

Keeping the assumptions above in mind we define four operations to rewrite an equation until it is in a solved form. In an exercise assistant, these operations could be offered to a user as buttons, allowing the student to focus on the strategy, while the tool is doing the calculations. The operations are:

1. Add a term to both sides of the equation ([E1]). The term can be negative, in which case we are actually performing a subtraction.
2. Multiply both sides by a non-zero constant factor ([E2]): since this exercise is restricted to linear equations there is no point in allowing variables to appear in this factor. Division can be mimicked by multiplying by a fraction.
3. Remove parentheses, i.e., apply the distribution law ([M5]). In the remaining part of this section we make more precise where and how this is done.
4. Merge "similar" terms: this too will be made more precise.

3.1 Sum view and product view

We define more views that help to implement the operations on an equation. Figure 2 gives a summary of the views on expressions in this paper. The sum view is similar to the plus view defined earlier, except that we now take associativity of the addition operator into account. The sum view converts an expression to a list of terms. Like the plus view, we push negations inside. For example,

$3x - (1 - \frac{2x}{5})$ is viewed as a list of three elements, namely $[3x, -1, \frac{2x}{5}]$. The function for matching can be defined as[1]:

$$
\begin{array}{l}
\mathit{matchSum} :: \mathit{Match}\ \mathit{Expr}\ [\mathit{Expr}] \\
\mathit{matchSum} = \mathit{Just} \circ f\ \mathit{False} \quad \text{-- laws [A1], [N1], [N3], [N4]} \\
\quad \textbf{where}\ f\ n\ (a :+: b) \quad = f\ n\ a +\!\!+ f\ n \qquad b \\
\qquad\qquad f\ n\ (a :-: b) \quad = f\ n\ a +\!\!+ f\ (\neg\ n)\ b \\
\qquad\qquad f\ n\ (\mathit{Negate}\ a) = f\ (\neg\ n)\ a \\
\qquad\qquad f\ n\ a \qquad\qquad = [\textbf{if}\ n\ \textbf{then}\ \mathit{Negate}\ a\ \textbf{else}\ a]
\end{array}
$$

The first parameter of the helper-function f is a boolean indicating whether or not the expression has to be negated. The function *matchSum* is total: for an expression without top-level additions, such as $3(x + 1)$, a singleton list is returned. For the builder of the sum view, we pass (.+.) and addition's neutral element to the *foldl* function, which constructs a left-biased tree.

$$
\begin{array}{l}
\mathit{sumView} :: \mathit{View}\ \mathit{Expr}\ [\mathit{Expr}] \\
\mathit{sumView} = \mathit{View}\ \mathit{matchSum}\ (\mathit{foldl}\ (.+.)\ (\mathit{Nat}\ 0)) \quad \text{-- laws [A1], [A3]}
\end{array}
$$

A list is a natural data structure for viewing associative operators. If we also take commutativity into account, we can sort the list, or use the bag (multi-set) data structure. If the operator is also idempotent, such as logical conjunction, we can turn to sets.

We define the product view similarly. Contrary to the sum view, we propagate negations upwards such that we find negations that cancel each other out (law [N1]). The type signature of the product view is $\mathit{View}\ \mathit{Expr}\ (\mathit{Bool}, [\mathit{Expr}])$. The boolean in the pair indicates whether or not the product has to be negated: we omit its definition, but give some examples instead. Matching the expression $3{\cdot}(-x{\cdot}\frac{1}{5})$ gives the pair $(\mathit{True}, [3, x, \frac{1}{5}])$. Although there is no special notation for the reciprocal function, we can also decompose divisors (but we don't have to), thereby also using law [D3] and taking care of its side-condition. The reciprocal function is its own inverse, and plays the same role as negation did for the sum view. The expression $(1 + 1){\cdot}\frac{x}{4\,/\,7}$ could then be viewed as the pair $(\mathit{False}, [1 + 1, x, \frac{1}{4}, 7])$. The builder of the product view takes care of neutral elements (law [M4]) and absorbing elements (law [M3]).

3.2 Normalizing sums and products

We discuss a normalization procedure for a list of expressions produced by the product view. Constant expressions (i.e., terms without variables) can be reduced to a rational number using constant folding techniques. Let us assume that the rational view (of type *View Expr Rational*) takes care of this. Products are normalized as follows: combine all constant rational numbers, even if they are

[1] Although intuitive, a more efficient definition would avoid having to concatenate lists (+\!\!+) from recursive calls, especially for left-biased abstract syntax trees.

not adjacent in the list. This operation is sound because multiplication is commutative. The order of the other, non-constant elements is left unchanged. The first occurrence of a constant rational number is replaced by the new, combined constant. Let this procedure be:

$$normalizeProduct :: [Expr] \rightarrow [Expr]$$

For instance, consider the list $[1 + 1, x, \frac{2}{8}, 7]$. The rational view is applied to each element, giving $[Just\ 2, Nothing, Just\ (1\ /\ 4), Just\ 7]$. The product of the constants is $7\ /\ 2$ of type *Rational*. We use the rational view to turn this into an expression. This expression is placed in a list before the variable x.

When normalizing sums, we combine constants (using the rational view), but we also merge terms that are "similar". For example, $2x$ and $3x$ should be turned into $5x$ by using the commutative variant of [M5] (from right to left) and constant folding. Product normalization is used for finding similar terms.

Constant folding in sums and products seems straightforward, but preserving the order makes it more involved. We want to emphasize that this is necessary for a tool in order to react naturally on user requests. For example, adding 3 to both sides of the equation $1+x = 2x-1$ would (ideally) result in $4+x = 2x+2$, even though the constants appear at different sides of the addition operator.

3.3 A strategy for solving linear equations

We briefly sketch a strategy for solving linear equations using our strategy combinators [10]. A strategy prescribes the order of rewrite steps in a derivation. If both sides of the equation have the form $a{\cdot}x+b$, then we are done in three steps: move x to the left (law [E1]), move the constant to the right (again law [E1]), and finally scale the equation such that the a on the left-hand side becomes one (law [E2]). Each step can be skipped under certain circumstances. Let this be the basic strategy:

$$basicEquation = try\ varToLeft\ <\!*\!>\ try\ conToRight\ <\!*\!>\ try\ scaleToOne$$

Views are used to implement the rewrite steps of the strategy, i.e., *varToLeft*, *conToRight*, and *scaleToOne*, in the same way as *scaleFraction* was defined in Section 2.5. For more involved equations, we first have to apply the distribution rule (law [M5]), after which we merge "similar" terms, and multiply both sides (law [E2]) to get rid of divisions. The overall strategy, which produces the derivation shown in the introduction, is:

$$\begin{aligned} solveEquation\ =\ & repeat\ (merge\ <|>\ distribute\ <|>\ removeDivision) \\ & <\!*\!>\ basicEquation \end{aligned}$$

4 Recognizing strategy steps

In this section, we briefly discuss how interactive exercise assistants can deal with formulas entered by a student. Such a submitted expression can be an

intermediate answer in a larger derivation. We do not only want to validate that the intermediate term is correct, but we also want to recognize which rewrite rule has been applied. Three terms are involved in such a diagnosis: the term submitted by the student, the previous term in the derivation, and the term that was expected at this point. We use a strategy definition, such as *solveEquation*, to predicted the expected term (possibly more than one).

On a submission, we use an equivalence relation to compare the submitted term with both the previous term and the expected term. We could use the semantic interpretation for checking equivalence, but this only establishes the soundness of the step, and ignores the direction in which the student continues the derivation. Using syntactic equality is also not an option since this test does not take minor differences in representation into account.

The equivalence relation should preferably be congruent, that is, compatible with the semantic interpretation of the symbols [2]. If not, it will be hard to predict whether or not two terms belong to the same equivalence class. The views we have seen operate at top-level: they are shallow. As a result, an equivalence relation that belongs to a view (defined in Section 2.4) is often not congruent. For example, the equivalence relations derived from the plus view and the div view are not congruent. To define a congruence relation using views, we have to recursively apply views.

We return to our running example of solving a linear equation. Each linear expression in x can be written as $a \cdot x + b$, where a and b are expressions in which x does not occur. In the remainder we assume that a and b are both constant rational numbers. We introduce two new views:

$$\begin{array}{ll} \mathit{linearView} & :: \mathit{View}\ \mathit{Expr}\ (\mathit{Rational}, \mathit{Rational}) \\ \mathit{equationView} & :: \mathit{View}\ (\mathit{Equation}\ \mathit{Expr})\ \mathit{Rational} \end{array}$$

The linear view returns a pair containing the a and b values. This view can easily be extended to the equation view, which first subtracts one side of the equation from the other, then applies the linear view, and finally divides the b value by $-a$. In fact, the equation view can be used as the semantic interpretation of our exercise. The view is not applicable to non-linear terms, or to terms that are not well-formed (e.g., division by zero).

With the equation view, we check whether or not a submitted term is correct. The linear view is used to test if the two sides of the equation still have the same meaning: if this is the case, we can exclude application of an equation rule ([E1] and [E2]). The derivation in the introduction is correct, and indeed, all equations in the derivation are equivalent under the equation view. In the middle part of the derivation, merging and distribution operations are performed. These operations work on expressions, not on equations. The left-hand sides of these equations ($12-4(4x+2)$, $12-16x-8$, and $4-16x$) and the right-hand sides ($36x-3(5x-1)$, $36x-15x+3$, and $21x+3$) are equivalent under the linear view.

However, if we want to recognize distributions of multiplication over addition (law [M5]), then we need to distinguish $12-4(4x+2)$ from $12-16x-8$. These expressions are equivalent under the linear view. With the help of the sum and

product views, and the normalization functions for sums and products, we can define a congruence relation that distinguishes these terms. The details of this relation are omitted from this paper. Merging alike terms, such as $12 - 16x - 8$ becoming $4 - 16x$, results in an expression from the same equivalence class.

We want some congruence relation for recognizing the steps of a user, but which? From a theoretic point of view, this relation should come from an equational theory based on a collection of laws or axioms. This way, we know which laws are available for testing equivalence. Unfortunately, it is not feasible to automatically derive an equivalence relation from a set of laws. Consequently, we have to restrict ourselves to certain collections.

5 Related work

A popular approach in constructing computer aided assessment (CAA) systems is to delegate all calculations to a computer algebra system (CAS). This approach will give good instant results, since CAS typically have advanced built-in algorithms, and are very good in simplifying complex formulas. These systems are, however, not designed for interaction with a CAA system, and they cannot be configured easily for a finer control of the simplification procedure [11]. This becomes even more of a problem when dealing with interactive exercises.

The purpose of views is related to the design principles of MathPert [3, 4]. We follow the guidelines for cognitive fidelity (the software solves the problem as a student does), glassbox computation (you can see how the software solves the problem), and customization of the software to the level of a user.

Beeson [4] claims that rewriting technology [2] is not enough to implement interactive systems that satisfy the above principles. He concludes that every operation has to be implemented as a function in the underlying programming language. We agree with his claim that rewriting alone is insufficient, however, we believe that a function implementing an operation can be given more structure: it is a rewriting step in the context of a view. The advantage of this separation is that we can still see operations as rewrite steps, but in the context of a view. Views can be reused for different exercise classes, and rewrite rules stay simple.

Interactive exercise assistants like the DWO [5] can be used to stepwise solve exercises. Most of these tools have no knowledge of strategies for solving exercises. As a consequence, intermediate steps are only compared against the final solution, and no hints or worked-out examples can be calculated. Most of these tools perform simplifications automatically, with similar results as we obtain. We have not found descriptions of how these tools implement canonical forms.

Proof planners that use computer algebra systems in their proofs run into similar problems as exercise assistants do: the form of the expression returned by the CAS might not coincide with the canonical form expected by the proof assistant. For example, Sorge [16] uses similar techniques as we do in the proof planner Ωmega. No concept of views is introduced though.

6 Conclusions and future work

In this paper, we have introduced views for specifying canonical forms. A view consists of a function for matching and one for reconstructing. Reconstruction after matching maps an expression to its canonical form, and matching after reconstruction is the identity function. The arrow combinators can be used to compose views, which makes them reusable for multiple exercise classes.

We have proposed views as a solution to the research questions posed in the introduction. A view defines a canonical form, which is used to show intuitive representations to users. It abstracts over a set of algebraic laws, which we have made explicit for the views introduced in this paper. This is helpful for determining the granularity of a rewrite step, which should correspond to the background of the student, but also for describing rewrite rules without having to worry about slightly different representations. Views help us to recognize expressions entered by students, provide helpful hints on how to proceed with the exercise, and generate worked-out examples with the right level of detail.

Views are useful in any situation where we need canonical forms of expressions: if for some reason $a + (-b)$ is to be preferred over $a - b$, we can define a view that calculates such a canonical form. In a strategy for solving an exercise, multiple views can coexist, for instance to show more detail at the start of a calculation.

The examples presented in this paper are exercises in calculating with fractions, and solving a linear equation. Views are also applicable to exercises outside the domain of mathematics. We are working on interactive exercises assistants for relation algebra and for an introductory programming course, and we believe that views will play a fundamental role within these exercise classes too.

We will proceed our research in the following directions. Multiple domain reasoners for classes of mathematical exercises are to be investigated for the upcoming European MathBridge project. Views will be used for implementing these reasoners. We have integrated our tools with the DWO, such that our step recognition technology can be used. The results are promising, and are expected to be used in mathematics courses in Dutch high schools next year. Further investigation is needed to understand how views can be incorporated in our generalized rewriting framework, in which we use generic programming techniques [12]. More information about our tools can be found on `http://ideas.cs.uu.nl/`.

Acknowledgements. We gratefully acknowledge discussions with Peter Boon and Wim van Velthoven, the authors of the DWO. Our introductory example was taken from the DWO applet on solving linear equations. We thank Alex Gerdes and the anonymous referees for their useful comments. This work was made possible by the support of the SURF Foundation, the higher education and research partnership organization for Information and Communications Technology (ICT), for the NKBW2 project. For more information about SURF, please visit `http://www.surf.nl`.

References

1. A. Alimarine, S. Smetsers, A. van Weelden, M. van Eekelen, and R. Plasmeijer. There and back again: arrows for invertible programming. In *Haskell '05*, pages 86–97. ACM, 2005.
2. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1997.
3. M.J. Beeson. A computerized environment for learning algebra, trigonometry, and calculus. *Journal of Artificial Intelligence and Education*, 1:65–76, 1990.
4. M.J. Beeson. Design principles of MathPert: Software to support education in algebra and calculus. In N. Kajler, editor, *Computer-Human Interaction in Symbolic Computation*, pages 89–115. Springer, 1998.
5. P. Boon and P. Drijvers. Algebra en applets, leren en onderwijzen (algebra and applets, learning and teaching, in Dutch). `http://www.fi.uu.nl/publicaties/literatuur/6571.pdf`, 2005.
6. H. Chaachoua et al. Aplusix, a learning environment for algebra, actual use and benefits. In *ICME 2004: 10th International Congress on Mathematical Education*, 2004. Retrieved from `http://www.itd.cnr.it/telma/papers.php`, May 2008.
7. A. Cohen, H. Cuypers, E. Reinaldo Barreiro, and H. Sterk. Interactive mathematical documents on the web. In *Algebra, Geometry and Software Systems*, pages 289–306. Springer, 2003.
8. X. Gang. WIMS: An interactive mathematics server. *Journal of Online Mathematics and its Applications*, 2001.
9. G. Goguadze, A. González Palomo, and E. Melis. Interactivity of exercises in ActiveMath. In *International Conference on Computers in Education, ICCE 2005*, 2005.
10. B. Heeren, J. Jeuring, A. van Leeuwen, and A. Gerdes. Specifying strategies for exercises. In S. Autexier et al., editor, *MKM '08*, volume 5144 of *LNAI*, pages 430–445. Springer, 2008.
11. J. Kyle and C.J. Sangwin. To simplify or not to simplify: that is the question in the CAA of mathematics. In *3rd International Conference on the Teaching of Mathematics*, 2006.
12. T. van Noort, A. Rodriguez Yakushev, S. Holdermans, J. Jeuring, and B. Heeren. A lightweight approach to datatype-generic rewriting. In *WGP '08*, pages 13–24. ACM, 2008.
13. R. Paterson. Arrows and computation. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, pages 201–222. Palgrave, 2003.
14. S. Peyton Jones et al. *Haskell 98, Language and Libraries. The Revised Report*. Cambridge University Press, 2003. A special issue of the Journal of Functional Programming, see also `http://www.haskell.org/`.
15. The OpenMath Society. The OpenMath Standard. `http://www.openmath.org/standard/index.html`, 2006.
16. V. Sorge. Non-Trivial Computations in Proof Planning. In H. Kirchner and C. Ringeissen, editors, *Frontiers of combining systems: Third International Workshop, FroCoS 2000*, volume 1794 of *LNCS*, pages 121–135. Springer, 2000.
17. W3C. MathML 2.0. `http://www.w3.org/Math/`, 2001.
18. P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *POPL*, pages 307–313, 1987.