Ten commandments for good default expression simplification

David R. Stoutemyer*

February 12, 2010

Abstract

This article provides goals for the design and improvement of default computer-algebra expression simplification. These goals can also help users recognize and partially circumvent some limitations of their current computer-algebra systems. Although motivated by computer algebra, many of the goals are also applicable to manual simplification, indicating what transformations are necessary and sufficient for good simplification when no particular canonical result form is required.

After motivating the ten goals, the article then explains how the Altran partially-factored form for rational expressions was extended for *Derive* and the computer algebra in Texas Instruments products to help fulfill the goals. In contrast to the distributed Altran representation, this recursive partially-factored semi-fraction form:

- doesn't unnecessarily force common denominators,
- discovers and preserves significantly more factors,
- can represent general expressions, and
- can produce an entire spectrum from fully factored over a common denominator through complete multivariate partial fractions, including a dense subset of intermediate forms.

1 Introduction

Simplicity is the peak of civilization
— Jessie Sampter

First, an explanation for the title: The current obedience to these commandments among computer algebra systems is low enough so that "goals" is a more accurate word than "commandments". However, with apologies to the author of the original Ten Commandments, these goals are called commandments in the title because:

- Moses(**author?**) [10] is cited in this article.
- Ten years later he reappeared in Biblical garb at a computer-algebra conference(**author?**) [11, 12].

---

*DStout at Hawaii dot Edu

- He was present at the Milestones in Computer Algebra conference where a preliminary form of this Ten Commandments article was presented**(author?)** [9].

Computer-algebra programs such as MathPert [1] and the Texas Instruments Student Math Guide program**(author?)** [20] help teach mathematics by having students choose a sequence of elementary transformations to arrive at a result. The transformations can be as elementary as combining numeric sub-expressions, applying 0 and 1 identities, sorting factors or terms, combining similar factors or terms, subtracting an expression from both sides of an equation, or applying a specific differentiation rule. With such **step-oriented derivational** systems, the overall goal is a well-chosen path having several steps at an appropriate tutorial granularity. The interface is oriented around producing and displaying a sequence of equivalent expressions annotated by rewrite rules selected from a context-dependent menu by the user. MathPert and *Derive* also have a "show me" tutorial mode wherein the system automatically chooses and displays a sequence of annotated steps — either uninterrupted or one step per press of the ENTER key.

In contrast, for **result-oriented** computer-algebra systems the overall goal is a satisfying final result in as few steps as possible – preferably one step. The interface is typically oriented around a sequence of input-result pairs. With some changes for annotation, a result-oriented interface could be a special one-step case of the step-oriented interface.

Computer-algebra users generally expect some transformation when they press ENTER. Otherwise they already have the desired result and need at most a system for 2D input and display of mathematical expressions. For example, if the input expression contains an unevaluated integral, derivative, sum or limit, most often users want to have the corresponding result be a closed-form equivalent. Otherwise, in the absence of a transformational function such as expand(...) or factor(...), users haven't indicated a strong preference for any particular form. However, they presumably want the result simpler than the input if that is possible, but not unnecessarily changed beyond that.

Default simplification means what a computer-algebra system does to a standard mathematical expression when the user presses ENTER, using factory-default mode settings, without enclosing the expression in an optional transformational function such as expand(...), factor(...), or simplify(...). Default simplification is the minimal set of transformations that a system does routinely.

Section 2 motivates and presents ten goals that are applicable to this most common case of default simplification.

Section 3 describes how the recursive partially-factored form in *Derive* and in the separate computer algebra in some Texas Instruments products helps meet some of these goals.

Section 4 describes how the form is further extended to partial fractions and to intermediate forms to further meet some of these goals.

Appendix A describes further details for semi fractions.

Appendix B describes further details for ratios of polynomials.

Appendix C describes additional issues for fractional exponents.

Appendix D describes additional issues for functional forms.

Appendix E contains pseudo-code for multiplying and adding partially-factored semi fractions.

2 What should we want from default simplification?

Everything should be made as simple as possible, but not simpler.

— Albert Einstein

2.1 Correctness is non-negotiable.

Nothing is as simple as we hope it will be
— Jim Horning

Definition. The **problem domain** for an expression is the Cartesian product of the default or declared domains of the variables therein, as further restricted by any user-supplied equalities and/or inequalities.

To determine a concise result within the problem domain, we can use transformations that aren't necessarily valid outside that domain. For example, some transformations that are valid for all integers or for all positive numbers aren't valid for more general real numbers, and some transformations that are valid for all real numbers aren't valid for all complex numbers.

There might be some points in the problem domain where some users regard an expression as being undefined. For example, although we can represent $\sin(\infty)$ as the interval $[-1,1]$ and we can represent the expression ± 1 as the multi-interval $\langle -1, 1\rangle$, many users regard such non-unique values as undefined, at least in some contexts such as being the result of a limit. Here are some of the rewrite rules that can define a *domainOfUniqueness*(...) function:

$$\begin{aligned} \text{domainOfUniqueness}(\pm u) &\rightarrow \text{domainOfUniqueness}\,(u)\ \wedge\ u=0,\\ \text{domainOfUniqueness}(\sin(\infty)) &\rightarrow false,\\ \text{domainOfUniqueness}(\mathit{number}) &\rightarrow true,\\ \text{domainOfUniqueness}(\mathit{variable}) &\rightarrow true,\\ \text{domainOfUniqueness}(u+v) &\rightarrow \text{domainOfUniqueness}(u)\wedge\text{domainOfUniqueness}(v). \end{aligned}$$

Most computer algebra systems represent and correctly operate on ∞, $-\infty$, and various complex infinities. Even $0/0$ is representable as the real interval $[-\infty,\infty]$ or the complex interval $[-\infty - \infty\, i, \infty + \infty\, i]$, depending on the problem domain. Nonetheless, many users regard at least some of these as undefined, at least in some contexts, such as solving equations in secondary-school. Here are some of the rewrite rules that can define a *domainOfFiniteness*(...) function:

$$\begin{aligned} \text{domainOfFiniteness}(\ln(u)) &\rightarrow \text{domainOfFiniteness}(u)\wedge u\neq 0,\\ \text{domainOfFiniteness}(\infty) &\rightarrow false,\\ \text{domainOfFiniteness}(u^v) &\rightarrow \text{domainOfFiniteness}(u)\wedge\text{domainOfFiniteness}(v)\\ &\qquad \wedge (u\neq 0\vee v>0)\,. \end{aligned}$$

Some users also regard non-real values as undefined, at least in some contexts, such as when computing limits of real expressions along the real line. Here are some of the rewrite rules that can define a *domainOfRealness*(...) function:

$$\begin{aligned} \text{domainOfRealness}(\mathit{number}) &\rightarrow \mathit{number}\in\mathbb{R},\\ \text{domainOfRealeness}(\mathit{variable}) &\rightarrow \mathit{variable}\in\mathbb{R},\\ \text{domainOfRealness}(\ln(u)) &\rightarrow \text{domainOfRealness}(u)\wedge u>0. \end{aligned}$$

Computer-algebra systems should provide these three functions and an easy way to specify what Boolean combination of them should be redefine the default built-in domainOfDefinition (...) function. For example, educators could use

$$\text{domainOfUniqueness}(u) \wedge \text{domainOfRealness}(u)$$

when teaching limits along the real line,

$$\text{domainOfUniqueness}(u) \wedge \text{domainOfFiniteness}(u)$$

when teaching complex arithmetic, or

$$\text{domainOfFiniteness}(u) \wedge \text{domainOfRealness}(u)$$

when teaching the real solutions of trigonometric equations. (It is no wonder that students *and educators* are confused about the domain of definition!)

In contrast an advanced researcher who believes that everything is defined could use *true*, whereas one who believes that everything but $\pm\infty$ and infinite intervals are defined could use

$$\text{domainOfUniqueness}(u) \vee \text{domainOfFiniteness}(u).$$

The default definition should please the largest possible subset of the users as often as possible for the particular computer algebra system.

The domainOfDefinition (...) function is needed to implement the first three goals below, and all four functions are useful to implementers and users in other ways too. It will not always be possible to simplify the resulting Boolean expression perfectly. For example, it might be a complicated expression that is equivalent to *true*. However, such results are still correct, and will usually evaluate to either *true* or *false* when ground-domain values are substituted for the indeterminates therein.

Definition. The **domain of equivalence** of two expressions is the domain for which they give equivalent values when ground-domain elements are substituted for the variables therein.

> **Goal 1 (Equivalence where defined).** Default simplification should produce an equivalent result within the intersection of the problem domain and the domain of definition.

Some transformations can yield expressions that are not equivalent everywhere. For example with the principal branch, $1/\sqrt{z} - \sqrt{1/z}$ is equivalent to 0 everywhere in the complex plane except where $\arg(z) = \pi$. Along $\arg(z) = \pi$ the expression is equivalent to $2/\sqrt{z}$. Therefore transforming the expression to either result would incorrectly contract the domain of equivalence in the problem domain unless the input includes a constraint that implies one of these two results.

One way to achieve equivalence is to leave the relevant sub-expression unchanged until the user realizes that an appropriate constraint must be attached to the input, then does so. Unfortunately, many users will fail to realize this, and they will judge the system unfavorably as being incapable of the desired transformation.

In an interactive environment, a more kindly route to correctness and favorable regard is for the system to ask the user whether or not $\arg(z) = \pi$, then automatically append a corresponding constraint to the user's input and do the corresponding transformation. If interested, the user can repeat the input with a different combination of replies to obtain another case of the complete general result.

Unfortunately the query might end up being unnecessary. For example, the sub-expression might be multiplied by another sub-expression that subsequently simplifies to 0. If so, users who are conscientious enough to repeat the input with the opposite constraint or reply are likely to be annoyed about being pestered with irrelevant questions. On the other hand, users who don't try the opposite constraint or reply might falsely conclude that the result isn't equivalent to the input without the constraint.

Also, this method can be baffling to a user if the question entails a variable such as a Laplace transform variable that is generated internally rather than present in the user's input. In such situations and for non-interactive situations, an alternative treatment is for the system to assume automatically the reply that seems most likely, such as $\arg(z) \neq \pi$, then append the corresponding constraint to the user's input before proceeding. Thus notified of the assumption, the user can then edit the input to impose a different assumption if desired.

A more thorough method, which doesn't require interaction or risk disdain, is for the system to develop a piecewise result equivalent for all z, such as

$$\frac{1}{\sqrt{z}} - \sqrt{\frac{1}{z}} \quad \rightarrow \quad \begin{cases} \dfrac{2}{\sqrt{z}} & \text{if } \arg z = \pi, \\ 0 & \text{otherwise.} \end{cases}$$

As another example, many algorithms factor out the leading coefficient for monic normalization, such as

$$cx + 1 \quad \rightarrow \quad c\left(x + \frac{1}{c}\right).$$

The left side is defined at $c = 0$, but the right sided isn't. A way to overcome this little-known disadvantage of monic normalization is the admittedly-cumbersome piecewise expression

$$cx + 1 \quad \rightarrow \quad \begin{cases} 1 & \text{if } c = 0, \\ c\left(x + \dfrac{1}{c}\right) & \text{otherwise.} \end{cases}$$

A better way to overcome this disadvantage is to avoid monic normalization when possible in favor of better alternatives such as the primitive normalization.

Quite often users are interested in only one of the alternatives, which they can then obtain by copying and pasting or by re-simplifying the input or result with an appropriate input constraint. However, such piecewise results can become combinatorially cluttered when combined, so there is still a place for a "query and modify input" mode. These considerations are summarized in the following corollary to the first goal:

> **Goal 2 (Contraction prevention).** If necessary for equivalence within the intersection of the problem domain and the domain of definition, a result should be piecewise or the system should append an appropriate constraint to the input, preferably after querying the user.

2.2 Managing domain enlargement

Some transformations can yield results that are defined where the input is undefined. For example, as persuasively argued in Graham et. al.**(author?)** [3] and Kahan**(author?)** [7], the modern trend is to define 0^0 as 1. However, many users still regard 0^0 as non-unique, non-finite and/or non-real, therefore undefined. Thus for them the transformation $u^0 \rightarrow 1$, hence also

$$\frac{u^\alpha}{u^\beta} \quad \rightarrow \quad u^{\alpha-\beta}$$

for $\alpha \geq \beta$ enlarges the domain of definition wherever expression $u = 0$. The enlargement is a benefit rather than a liability because:

- Unlike the input, the result doesn't suffer catastrophic cancellation for z near π.
- Defining a value at $z = \pi$ turns a *partial* function into a more desirable *total* function, and the value given by the reduced ratio there is the best choice, because it is the omni-directional limit of the input as $z \rightarrow \pi$.
- Removable singularities are often merely a result of an earlier transformation or a modeling artifact that introduced them. For example, perhaps they are a result of a monic normalization or being at the pole of a spherical coordinate system.
- The phrase "removable singularities" implies permission to remove them.
- Goal 1 neither forbids nor requires equivalence equivalence outside the intersection of the problem domain with the domain of definition. Therefore transformations that enlarge the domain of definition make the result *better than equivalent.*

However, there are vocal critics of such gratuitous improvements — particularly at a certain level in the standard mathematics curriculum. To appease these critics, there should be a mode they can activate to force the result to be undefined wherever the input is undefined but still obtain the transformation:

> **Goal 3 (Optional enlargement prevention).** Results should optionally include appropriate constraints if necessary to prevent enlarging the domain of definition within the problem domain.

For example:

$$\frac{z\,(z-\pi)}{z-\pi} \quad \rightarrow \quad z \mid z \neq \pi .$$

Complaints about domain enlargement are an unfortunate consequence of the historical emphasis on equivalence throughout the entire problem domain rather than merely its intersection with the domain of definition. Convenient notations and phrases might help evolve enthusiasm for domain enlargement. Using $\mathrm{dod}\,(\ldots)$ for $\mathrm{domainOfDefinition}\,(\ldots)$, Table 1 lists three proposed relational operators then three corresponding transformational operators that are easily constructable in LaTeX. For example,

$$\frac{z\,(z-\pi)}{z-\pi} \quad \overset{:)}{\longrightarrow} \quad z.$$

Table 1: Notation for changes in dod$(\ldots)$ = domainOfDefinition$(\ldots)$

Example	Can be read as	Analogy
$A \sqsupset B$	A is *more universal* than B	$\mathrm{dod}(A) \supset \mathrm{dod}(B)$
$A \sqsupseteq B$	A is *equivalent to or more universal* than B	$\mathrm{dod}(A) \supseteq \mathrm{dod}(B)$
$A \sqsubset B$	A is *less universal* than B	$\mathrm{dod}(A) \subset \mathrm{dod}(B)$
$A \xrightarrow{:)} B$	*A improves to B*	happy-face emoticon
$A \overset{:)}{\Rightarrow} B$	*A transforms to or improves to B*	equivalent to or better
$A \xrightarrow{:(} B$	*A degrades to B*	sad-face emoticon

2.3 Protecting users from inappropriate substitutions

Seek simplicity, and distrust it
— Alfred North Whitehead

As a corollary to Murphy's law, someone will eventually apply any widely-used result outside the domain of equivalence to the inputs, unless explicitly prevented from doing so or unless the result is universally valid. For example, as discussed by Jeffrey and Norman [6], most publications containing the Cardano formula for the solution of a cubic equation don't state that in contrast to the quadratic formula it isn't always correct for non-real coefficients. Cardano's formula has been misused by many people, including some computer-algebra implementers. (I am guilty.) The consequences can be disastrous.

Implementing the enlargement-prevention goal entails a mechanism for attaching constraints to results. With that mechanism implemented, it is not much additional effort also to propagate to the output any constraints introduced by the user or by the system, and to have the system return the representation for "undefined" whenever a user subsequently makes a substitution that violates result constraints.

The system should determine the domains of definition for sub-expressions and propagate their intersection when expressions are combined. The domains are most generally represented as Boolean expressions involving inequalities, equalities, and type constraints such as $n \in \mathbb{Z}$.

The constraint expression should also be simplified as much as is practical, to make it most understandable. We can omit it if it simplifies to *true*. If it simplifies to *false*, then the algebraic result is undefined everywhere in the problem domain.

Perfect constraint simplification can be quite difficult or undecidable. However, perfection isn't mandatory. The purpose of the constraint is to return the representation for "undefined" if a substitution makes the Boolean constraint simplify to *false*. Otherwise the result is that of the substitution, with a more specialized attached constraint when it doesn't simplify to *true*. A properly-derived result of the form $expression|BooleanConstraint$ is still correct even if the constraint could be simplified to *true* or to *false* but wasn't.

For safety, the output constraint should indicate the basic domain of every variable in the output expression. This can be done by including type constraints of the form *variable* $\in$ *domain*. However, to reduce clutter, the types of variables can often be inferred from constraints. For example, the constraint $x \geq 0$ implies $x \in \mathbb{R}$. In such cases we can omit a type constraint for x if it isn't a more restricted type such as integer. Also, if one type such as $\mathbb{C}$ includes all other possible declared or default numeric types, then declarations of that type can be omitted if arithmetic or comparison operators imply that the variable is numeric.

Despite such economies, constrained results can become distractingly cluttered. Therefore, the default could be to represent any complicated or routine portions of a constraint with an ellipsis that could be expanded by clicking on it. Moreover, there could be an option to hide output constraints. However, they would still be internally attached to results to insure safe substitutions within the computer-algebra system. The constraints would also at least *encourage* safe substitutions if included whenever a result is copied for pasting outside the system.

> **Goal 4** (**Domain propagation**)**.** Domains and constraints should be propagated into results, where they then cause substitution of inappropriate values to return the representation for undefined.

2.4 Disabling default transformations

Simplicity is in the eye of the beholder
– adapted from Margaret Wolfe Hungerford

No matter how modest the set of default transformations, many mathematics educators wish that some of them could be selectively disabled sometimes. At these times, such users would be better served by a step-oriented system. However, even for research or the exposition thereof, we sometimes want to disable transformations that we most often want as default. For example, many users might prefer 2^{9999} to the 3010 digits of the decimal form.

As another example, combining numeric sub-expressions in the coefficients of a truncated power series can mask revealing patterns such as in

$$\frac{1}{2}x^0 + \frac{1\cdot 3}{2\cdot 4}x^2 + \frac{1\cdot 3\cdot 5}{2\cdot 4\cdot 6}x^4 + o(x^4) \quad \mathit{versus} \quad \frac{1}{2} + \frac{3}{8}x^2 + \frac{5}{16}x^4 + o(x^4),$$

where $x^0 \xrightarrow{:)} 1$ is also disabled.

As another example, even though this article is a research rather than educational publication, about 30% of the examples using "$\rightarrow$" in this article are multi-step derivational.

We want to do this stepping in the same software environment that we use for result-oriented computer algebra. Therefore, a compassionate expression simplifier allows selectively disabling such default transformations. Although users should be offered complete control over which transformations are disabled, a menu could also offer common stratified levels such as enabling only arithmetic, only arithmetic together with 0 and 1 identities, etc.

> **Goal 5** (**Optionally disable default transformations**)**.** It should be possible to selectively disable default transformations.

Each Goal is subject to the constraints of all previous goals. For example, disabling default transformations shouldn't compromise the goal of equivalence where the input is defined.

The necessary expression representation and algorithms to support thorough disablement of default transformations are so different from what is best for high-performance result-oriented computer algebra that it is best to implement transformation disablement as a mode that switches to a different data representation and simplifier. For example:

- Fine-grain syntactic control or teaching the laws of signs requires internal representation of negation and subtraction of terms, whereas performance-oriented simplification typically forces signs into the numeric coefficients so that cases for negation and subtraction don't have to be implemented except in arithmetic. A post-simplification pass typically restores subtractions and negations for display.
- Similarly for division versus multiplication by a negative power.
- Teaching the rules for deleting superfluous parentheses requires that they are representable in the step-mode internal representation.
- Fine-grain syntactic control or teaching trigonometry requires internal representations of all the trigonometric functions and their inverses. In contrast, converting them all internally to a lean subset such as sines, cosines, inverse sines and inverse tangents automatically accomplishes many desirable transformations, such as $\tan\theta\cos\theta \xrightarrow{:)} \sin\theta$ and $\arcsin x + \arccos x \xrightarrow{:)} \pi/2$. Tangents, inverse cosines etc. can often be restored for display either optionally or by default when it makes a result more compact.
- Similarly for fractional powers, versus square roots, cube roots etc. For example, allowing students to choose an appropriate transformation to simplify $\sqrt{2}-2^{1/2}$ requires separate internal representations for square roots and fractional powers, which is an unnecessary complication to handle thoroughly for automatic result-oriented simplification. (Unfortunately, a distressing portion of math education is devoted to contending with our many redundant functions and notations rather than learning genuinely new concepts!)

When default transformations are disabled, it is more appropriate to have the interface switch from result mode to a step mode. However, it is important that results obtained in either mode be thoroughly accessible in both modes — that is the reason for wanting both modes in a single product.

2.5 We want candid forms

Cancellation is key.

Definition. A **candid expression** is one that is not equivalent to an expression that visibly manifests a simpler expression class

A candid form is "What You See Is What It Is" (WYSIWII).

Definition. A **misleading expression** is one that isn't candid.

Misleading expressions masquerade as something more complicated than necessary. Try entering the following examples on your computer algebra systems to see if their default results are misleading. If so, how many *optional* transformations did you have to try to obtain a candid result?

- Expressions that are equivalent to 0 but don't automatically simplify to 0. For example,

$$f\left((x-1)(x+1)\right) - f\left(x^2-1\right).$$

- Expressions that contain superfluous variables. For example,

$$2\sinh(x) - e^{x} + e^{-x} + \ln(y)\,, \quad \text{which is equivalent to} \quad \ln(y)\,.$$

- Apparently-irrational expressions that are equivalent to rational expressions. For example,

$$\frac{\sqrt{z}+1}{\sqrt{z}(z+\sqrt{z})}, \quad \text{which is equivalent to} \quad \frac{1}{z}.$$

- Irrational expressions that are equivalent to other irrational expressions containing less nested and/or fewer distinct irrationalities. For example,

$$\sin(2\arctan(z)) + \left(15\sqrt{3}+26\right)^{1/3}, \quad \text{which is equivalent to} \quad \frac{2z}{z^2+1} + \sqrt{3} + 2.$$

- Non-polynomial expressions that can be improved to polynomials. For example,

$$\frac{x^2-1}{x-1}, \quad \text{which improves to} \quad x+1.$$

- Expressions that contain exponent magnitudes larger or smaller than necessary. For example,

$$(x+1)^2 - (x-1)^2 + \frac{y^2-1}{y^2+2y+1}, \quad \text{which is equivalent to} \quad 4x + \frac{y-1}{y+1}.$$

- Expressions that mislead us with disordered terms or factors. For example,

$$x^{18} + x^{17} + x^{16} + x^{15} + x^{14} + x^{13} + x^{19} + x^{11} + x^{10} + x^{9} + x^{8} + x^{2} + x^{6} + x^{5} + x^{4} + x^{3}.$$

  One could easily assume this is a degree 18 polynomial having a minimum exponent of 3. Worse yet, imagine the unlikeliness of noticing otherwise if the expression was several pages long, including several variables and lengthy coefficients. Complying with the traditional ordering for commutative and associative operators described by Moses[10] greatly aids comprehension.

- Expressions that contain i or a fractional power of -1 but are actually real for real values of all variables therein. For example,

$$\frac{i\,((3-5i)\,x+1)}{((5+3i)\,x+i)\,x}, \quad \text{which improves to} \quad \frac{1}{x}.$$

- Non-real expressions that have a concise rectangular or polar equivalent but aren't displayed that way. For example,

$$\frac{(-1)^{1/8}\sqrt{i+1}}{2^{3/4}} + ie^{i\pi/2}, \quad \text{which is equivalent to} \quad -\frac{1}{2} + \frac{i}{2}.$$

  Most users can easily envision a useful geometric image only for rectangular and polar representations of either the form $(-1)^{\alpha}$ or $e^{i\theta}$.

- Expressions that mislead about important qualitative characteristics such as frequencies, discontinuities, symmetries or asymptotic behavior. For example,
$$\frac{\sin(4\theta)}{\cos(2\theta)}, \quad \text{which improves to} \quad 2\sin(2\theta)\,.$$
- Boolean combinations of equalities and inequalities that can be expressed more succinctly. For example,
$$\mathrm{mod}\,(2\lfloor x\rfloor, 2) = 0 \;\wedge\; ((x > 3 \wedge \neg\,(x \le 5)) \,\vee\, x = 5)\,, \quad \text{which is equivalent to} \;\; x \ge 5.$$

Some alternatives can be more candid than others. For example, if an expression is rewritten to eliminate one of its two superfluous variables, that imperfect result is more candid in this regard than the original alternative. It is important for default simplification to be as candid as is practical because:

- The consequences of misleading intermediate or final results can be ruinous. For example, not recognizing that an expression is equivalent to 0 or is free of a certain variable or is a polynomial of a particular degree can lead to incorrect matrix pivot choices or lead to incorrect or thwarted limits, integrals, series, and equation solutions.
- The need for identifying such properties occurs in too many places to require implementers and users to unfailingly employ a simplify(...) function at all of them: Almost every conditional statement in a computer-algebra algorithm tests for some property of an expression. If the property isn't recognized, then an incorrect alternative will be selected or the algorithm will unnecessarily indicate that it can't do that integral, limit, etc.
- If we want a candid result, the easiest way to implement that is to use bottom-up simplification and have every intermediate result be candid. Being able to rely on candid operands greatly reduces the number of cases that must be considered.
- A simplify(...) function probably entails at least one extra pass over the expression after default simplification, which wastes time, code space and expression space compared to making the *first* pass give a candid result. This can make simplification exponentially slower if simplify(...) is used in a function that recursively traverses expression trees. Such recursion is so ubiquitous in computer algebra that this performance penalty precludes using simplify(...) in many of the places that it would be needed to achieve candid results.

In nontrivial cases it is impractical for a single form to reveal *all* possibly-important features of a function. Therefore it is unreasonable to insist that a candid form reveal all such features. However, a candid form at least shouldn't *mislead* us about those features.

> **Goal 6 (Candid results).** Default simplification should be candid for rational expressions and for as many other classes as is practical. Default simplification should try hard even for classes where candidness can't be guaranteed for all examples.

Table 2: Three canonical forms on the main spectrum for rational expressions

Form	Univariate example	Notable advantages
Factored on a common denominator	$\frac{x^3\left(2x+\sqrt{5}-1\right)\left(2x-\sqrt{5}-1\right)}{4\,(x-1)^2\,(x+1)}$	zeros, poles, multiplicities, often less rounding
Expanded on a common denominator	$\frac{x^5-x^4-x^3}{x^3-x^2-x+1}$	degrees of numerator and denominator
Partial fractions	$x^2-\frac{1}{2\,(x-1)^2}-\frac{3}{4\,(x-1)}-\frac{1}{x+1}$	poles, their multiplicities, their residues, asymptotic polynomial

2.6 Canonical forms are necessary options but insufficient defaults

Definition. A **canonical form** is one for which all equivalent expressions are represented uniquely.

With bottom-up simplification of an expression from its simplest parts, merely forcing a canonical form for every intermediate result guarantees that every operand is canonical. This makes the simplifier particularly compact, because there are fewer cases to consider when combining subexpressions.

Table 2 lists examples and informational advantages of three canonical forms for rational expressions. Factored form depends on the amount of factoring, such as square-free, over $\mathbb{Z}$, over $\mathbb{Z}[i]$, with user-specified algebraic extensions, with whatever radicals are necessary and applicable, with a rootOf(...) functional form, or with approximate coefficients. Partial-fraction form similarly depends on the amount of factoring of denominators. Moreover, for multivariate examples there are choices for the ordering of the variables and for which subsets of the variables are factored and/or expanded. Also, reference [15] discusses alternative forms of *multivariate* partial fractions.

There are also canonical forms for some classes of irrational expressions, such as some kinds of trigonometric, exponential, logarithmic, and fractional-power expressions. However:

- No one canonical form can be good for all purposes.
- Any one canonical form can exhaust memory or patience to compute.
- Any and all classic canonical forms can be unnecessarily bulky or unnecessarily different from a user's input, masking structural information that the user would prefer to see preserved in the result. For example, both the factored and expanded forms of candid $(x^{99}-y^{99})(9x+8y+9)^{99}$ are much bulkier. As another example, both the common denominator and complete partial fraction forms of the following candid form are much bulkier:

$$\frac{a^8}{a^9-1}+\frac{b^8}{b^9-1}+\frac{c^8}{c^9-1}+\frac{d^8}{d^9-1}.$$

- Users prefer that default results preserve input structure that is meaningful or traditional to the application, as much as possible consistent with candidness.

Thus canonical forms are too costly and extreme for good default simplification. Given *optional* transformation functions that return desired canonical forms, there is no need for default simplification to rudely force one of them or even the most concise of them.

> **Goal 7** (**Factored through partial fractions**)**.** For rational expressions and the rational aspect of irrational expressions, the set of results returnable by default simplification should include a dense subset of all forms obtained by combining some or all factors of fully factored form or combining some or all fractions of complete multivariate partial fractions.
>
> **Goal 8** (**Nearby form**)**.** Default simplification should deliver a candid result that isn't unnecessarily distant from the user's input.

Good default simplification to a nearby candid form reduces the need for trying a sequence of optional drastic transformation functions with the hope of thereby obtaining a candid result that is probably much further than need be from the input.

2.7 Please don't try my patience!

If a lengthy computation is taking an unendurable amount of time, a user will terminate the computation, obtaining no result despite the aggravating wasted time. Users have to accept that for certain inputs, certain *optional* transformations sometimes exhaust memory or patience. However, if *default* simplification also does this, then the system is worse than useless for those problems because it wastes their time for *no* result. Thus with the availability of various optional transformations such as fully factored over a common denominator through complete or total partial fraction expansion, default simplification should strive to return a candid form that can be computed quickly without exhausting memory. As an associated benefit of thus avoiding costly transformations when we can candidly do so, the result is likely to be closer than most other candid forms to the user's input.

Definition. A **guessed least cost** candid form is one derived such that when more than one supported transformation is applicable, one of those guessed to be least costly is selected.

It is important that the time spent guessing a least-cost alternative is is modest compared to actually doing the transformations. Where practical it is good if the guesses take into account not only the costs of the immediate alternative transformations, but also the cost of likely possible subsequent operations. For example, with B and D being multinomials, for the transformation

$$\frac{A}{B}+\frac{C}{D} \quad \rightarrow \quad \frac{AD+BC}{BD}$$

avoiding gratuitous expansion of BD is likely to reduce the cost of subsequently combining this result with other expressions.

> **Goal 9** (**Frugal**)**.** Default simplification should be as economical of time and space as is practical.

2.8 Idempotent simplification is highly desirable

Definition. An **ephemeral** function or operator is one that can produce a form that default simplification would alter.

For example, most computer-algebra systems have a function that does a transformation such as

$$\text{integerFactor}(20) \quad \rightarrow \quad 2^2 \cdot 5;$$

but if you enter $2^2 \cdot 5$, it transforms to 20.

Results that would otherwise be ephemeral can be protected by returning a different type of expression such as a list of factors. Another alternative is to passively encapsulate $2^2 \cdot 5$ in a special functional form whose name is the null string. For example, many computer-algebra systems use a special form for truncated series results. However, both of these alternatives are a nuisance to undo if you later want the result to combine with ordinary algebraic expressions. For example, such protection might prevent $2 \cdot \text{integerFactor}(20)$ from automatically transforming to either $2^3 \cdot 5$ or to 40. More seriously, such protection might prevent $\text{integerFactor}\,(20) - 20$ from automatically simplifying to 0. Also, the *invisible-function encapsulation* alternative is so visually subtle that many users won't realize there is anything to undo, and they won't notice a dangerously non-candid sub-expression in their result.

This is the **ephemeral form dilemma**: You are *damned if you do* protect results that would otherwise be ephemeral, and you are *damned if you don't.*

There is less need for troublesome protection when ephemeral functions are used at the top level: Users can view a safely unprotected ephemeral result when such functions are used at the top level rather than as a proper sub-expression. Restriction to top-level use can be guaranteed by making the ephemeral function instead be a command, such as

$$\text{integerFactor } 20.$$

Non-ephemeral functions and operators should include the standard named mathematical operators, elementary functions and higher transcendental functions. It would be nice if our default simplifier was flexible enough to recognize and leave unchanged *all* candid expressions. Until then we must accept ephemeral results from some functions that request optional transformations into forms that default simplification doesn't recognize as candid.

Definition. Simplification is **idempotent** for a class of input expressions if simplification of the result yields the same result.

Without this property, a cautious user would have to re-enter such results as inputs until they cycle or stop changing.

Failure of idempotency is usually a sign that a result sub-expression was passively constructed where there should have been a recursive invocation of an internal simplification function. It can be disastrous, because it can cause a non-candid result.

> **Goal 10** (**Idempotency**)**.** Default simplification should be idempotent for all inputs composed of standard named mathematical functions and operators.

3 Recursive partially-factored form

Factors, factors everywhere, with opportunities to share.
— W.S. Brown

The previous section motivated and stated some highly desirable goals for default simplification. This section and Section 4 describe a relatively simple way to accomplish many of these goals for extended rational expressions and for the rational aspects of simplifying irrational expressions.

A major issue for default simplification is the **polynomial-expansion dilemma**: Often polynomial expansion makes expressions bulkier and less comprehensible, but sometimes it is necessary for candidness to enable key cancellations.

Another major issue for default simplification is the **ratio-reduction dilemma**: Computing the greatest common divisor of two multinomials to reduce a ratio is costly, the gcd might end up being 1 anyway, and occasionally reduction increases rather than decreases the bulk. However not doing so can lead to non-candid results that are often bulkier than necessary and more prone to catastrophic cancellation near the removable singularities.

The Altran computer algebra system pioneered an excellent way to greatly ameliorate these two dilemmas. By default, Altran represents rational expressions as a reduced ratio of two polynomials that are individually allowed to range between fully factored and fully expanded. Distributed representation is used for each multinomial factor. This form is candid and the set of result forms that it can produce is a rather dense subset of the spectrum that it spans.

Brown [2] eloquently explains why many factors arise naturally with polynomials and rational expressions during operations such as addition, multiplication, gcd computation, differentiation, substitution, and determinants, then explains why it is important to preserve such factors and how to do so. Hall [4] gives additional implementation details and compelling test results. Those articles are highly-recommended background for this one.

By default, Altran polynomials are expanded only when necessary to satisfy the constraints of the form. The resulting expressions are usually at least partially factored, greatly reducing the total time spent on expanding polynomials and computing their gcds. The results are also usually more compact than a ratio of two expanded polynomials.

Derive also uses partially-factored representation, but with recursive rather than distributed representation for multivariate sums. The opportunities for partial factoring are thereby not confined to the top level. This can dramatically further reduce the result size, its distance from a user's input, the need for polynomial expansion, and the total cost of polynomial gcds.

The computer algebra embedded in the TI-92, TI-89, TI-Interactive, TI-Voyage 200 and TI-Nspire products described at **(author?)** [18] has no official generic name. Therefore it is referred to here as TI-CAS. Unlike *Derive*, the implementation language is C rather than muLISP. Although there are algorithmic differences throughout, both systems use similar algorithms for the recursive partially-factored semi-fraction forms described in this section and Section 4.

3.1 Recursive representation of extended polynomials

The representational statements in the remaining sections are abstract enough so that they apply to both *Derive* and TI-CAS. Reference[16] describes the extremely different concrete data structures used by both systems, and the TI Developers Guide[19] presents more details for the TI-CAS implementation.

In the internal representations, negations and subtractions are represented using negative coefficients, and non-numeric ratios are represented using multiplication together with negative powers. Also, expressions are simplified from the bottom up. Therefore algorithms can rely on operands of functions and operators already being as candid as they can be using the default simplification.

Definition. A **functional form** is the internal representation of almost everything that isn't internally a number, variable, sum, product or rational power.

For example, $\ln(\dots)$ is a functional form. The arguments of functional forms recursively use the same general representation being described here.

Definition. A **generalized variable** is a variable or a functional form.

Definition. A **unomial** is a generalized variable or a rational power thereof.

The exponent can be negative and/or fractional. Exponents that aren't rational numbers are represented internally using $\exp(\dots)$ and $\ln(\dots)$. For example, x^y is represented as $\exp\left(y\ln\left(x\right)\right)$. A post-simplification pass converts this back to x^y for display. This representation automatically achieves certain simplifications such as $4^z - 2^{2z} \to 0$ because of automatic simplifications for $\ln\left(\ldots\right)$ and $\exp\left(\ldots\right)$. However, if I could do it over again I would instead use general expressions as exponents.

A unomial is automatically candid if it is a variable, a power of a variable, or a functional form that that has candid arguments and doesn't simplify to a simpler class. Otherwise we should check for transformations that make the unomial more candid, such as $|x|^2 \mid x \in \mathbb{R} \;\rightarrow\; x^2$.

Definition. A **unomial-headed term** is a unomial or a unomial times a coefficient that is either a number or any candid expression having only lesser generalized variables.

A unomial-headed term is automatically candid if the unomial is a variable or a power thereof. For example, distributing such a unomial over the terms of its candid coefficient that is a sum couldn't enable any cancellations, because all of the terms in the coefficient are dissimilar to each other and have only lesser variables than the distributed unomial. However, if the unomial is a functional form or a power thereof, then we should check for possible cancellation or combination with functional forms in the coefficient. For example with ordering $|x| \succ \operatorname{sign}(x) \succ y$,

$$(\operatorname{sign}(x) + y)\cdot|x| \quad\rightarrow\quad y\cdot|x| + x,$$

which is more candid because the superfluous $\operatorname{sign}(x)$ has been eliminated.

Definition. An **extended polynomial** (in its main generalized variable) is one of :

- a candid constant,
- a unomial-headed term,
- a unomial-headed term plus a candid expression that is a constant or contains only less main generalized variables,
- a higher-powered unomial-headed term plus an extended polynomial having the same main generalized variable.

Such recursively-represented extended polynomials are automatically candid if the unomial for each unomial-headed term is a variable or a power thereof. For example, distributing these distinct unomials over their associated coefficients that are sums cannot enable cancellations: Distributed terms arising from different recursive terms will have distinct leading unomials. However, extended polynomials containing fractional powers or functional forms might require additional checks and transformations to achieve or strive for candidness, as explained in appendices C and D.

Here is an example of a recursive extended polynomial in $\ln(x)$, and y, with ordering $\ln(x) \succ y \succ z$:

$$(2y^{5/2} + 3.27i)\ln(x)^2 + (5z-1)^{-7/3}\ln(x) + (z+1)^{1/2}, \tag{1}$$

which displays as

$$\left(2y^{5/2} + 3.27i\right)\ln(x)^2 + \frac{\ln(x)}{(5z-1)^{7/3}} + \sqrt{z+1}\,.$$

Notice that this is *not* an extended polynomial in z, because of the fractional powers of the *sums* $5z-1$ and $z+1$. However, the two sub-expressions involving z are candid, as required for the entire expression to be candid.

Notice also that in expression (1) *Derive,* TI-CAS and this article display the unomial cofactors *right* of their companion coefficients even though the cofactors are *internally* stored *before* the coefficients. This internal ordering provides faster access to the cofactors, because they are accessed more frequently than the coefficients.

The general-purpose *Derive* and TI-CAS data structures are flexible enough to represent distributed form and mixtures, which are used when needed for algorithmic purposes and for displaying the result of the expand(...) function when expanding with respect to more than one variable. However, expression (1) is as expanded as it can be for recursive representation.

Another example, with $x \succ y$ is

$$\frac{\boldsymbol{x^2}}{y} + \left(y^2 + y + 5\right) + \frac{8y}{\boldsymbol{x}}, \tag{2}$$

which is represented internally as

$$y^{-1}\boldsymbol{x^2} + 8y\boldsymbol{x^{-1}} + y^2 + y + 5. \tag{3}$$

The terms that are 0-degree in the main variable x are artificially grouped as (y^2+y+5) in expression (2) only for emphasis. They are not collected under a single pointer internally.

Notice how internally the term with lead unomial x^{-1} term occurs *between* the term with lead unomial x^2 and the (implicitly) x^0 terms in expression 3. This makes it faster to determine when the reductum of a sum is free of the previously-main variable. A minor disadvantage of this concession to efficiency is that distributing or factoring out negative-degree unomials can change the relative order of terms. For example,

$$x^{-1}\left(x^2 + \mathbf{5}x - \mathbf{7}\right) \quad \leftrightarrow \quad x - \mathbf{7}x^{-1} + \mathbf{5}.$$

The more traditional ordering could be restored for display during a post-simplification pass.

3.2 Recursively partially factored representation

What is good for the goose is good for the goslings.

Distribution and co-distribution over sums is often less costly with recursive form than with distributed form because:

- Unomials are shared by terms that differ only in lesser variables.
- At any one level the term count for multivariate sums tends to be much less, reducing the sorting costs.
- For many purposes distribution is often necessary only with respect to the top-level variable. Such partial distribution is possible for recursive representation, but not for distributed representation.

Nonetheless, co-distribution of two sums having the same main variable can be costly in time and usually also in the resulting expression size. Moreover, it is even more costly to recover the factorization. Therefore, effort is made to avoid such co-distribution wherever candidly possible. One-way distribution of an expression over a sum is less costly and less costly to reverse.

With recursive representation we can employ partial factoring at all levels, with dramatic benefits. Even when some expansion is necessary to enable possible cancellations, the recursive representation might enable us to limit the amount of expansion. For example with recursive form and $x \succ a \succ b$,

$$\begin{aligned} ((a^2-1)^{1000}x+\boldsymbol{b})\boldsymbol{x}-\boldsymbol{bx} &\rightarrow (a^2-1)^{1000}x^2+\boldsymbol{bx}-\boldsymbol{bx} \\ &\rightarrow (a^2-1)^{1000}x^2, \end{aligned}$$

eliminating the superfluous variable b without expanding the coefficient $(a^2-1)^{1000}$.

In contrast, with distributed representation we would have

$$\begin{aligned} ((a^2-1)^{1000}x+\boldsymbol{b)x}-\boldsymbol{bx} &\rightarrow (a^{2000}x-1000a^{1998}x+\cdots+x+\boldsymbol{b)x}-\boldsymbol{bx} \\ &\rightarrow (a^{2000}x^2-1000a^{1998}x^2+\cdots+x^2+\boldsymbol{bx)}-\boldsymbol{bx} \\ &\rightarrow a^{2000}x^2-1000a^{1998}x^2+\cdots+x^2+\boldsymbol{bx}-\boldsymbol{bx} \\ &\rightarrow a^{2000}x^2-1000a^{1998}x^2+\cdots+x^2, \end{aligned}$$

with 1001 terms, from which only the factor x^2 is easily recoverable.

At each recursive level it is helpful to order factors internally by decreasing mainness of their most main generalized variable, with any signed numeric factor *last*. Ties are broken lexically, according to the bases of the factors. This makes the main generalized variable most accessible. Also, when the main generalized variable of a factor is less than for the previous factor, then the previous main generalized variable won't occur from there on.

Ordering functional forms first according to the function or operator is advantageous for a few purposes such as recognizing opportunities for transformations such as, with $x \succ y \succ z$:

$$\begin{aligned} |\boldsymbol{z}|\cdot y\cdot|\boldsymbol{x}| &\rightarrow y\cdot|\boldsymbol{z}|\cdot|\boldsymbol{x}| \\ &\rightarrow y\cdot|\boldsymbol{zx}|, \\ \boldsymbol{\ln(x)}+y+\boldsymbol{\ln(z)} \mid x>0 &\rightarrow \boldsymbol{\ln(x)}+\boldsymbol{\ln(z)}+y \mid x>0 \\ &\rightarrow \boldsymbol{\ln(xz)}+y \mid x>0. \end{aligned}$$

Such transformations can be helpful for limits, equation solving, and to reduce the number of functional forms for display during a post-simplification pass. However, for most default internal simplification it is more helpful to order functional forms according to lexical comparison of their successive arguments, using the function or operator name only as a final tie breaker. This helps group together factors depending on the main variable. Therefore this is the order used by *Derive* and TI-CAS.

It is also helpful to have any unomial factor immediately after all non-unomial factors having the same main generalized variable. That way we can be sure that when the first factor of a product is a unomial, then the rest of the product is free of the unomial's generalized variable. This is a very common case because for fully expanded recursive extended polynomials, non-unomial factors can only occur as the last factor in products.

A post-simplification pass can rearrange the factors to the more traditional display order described by Moses [10].

3.3 Units and unit normal expressions

Polynomials over $\mathbb{Z}, \mathbb{Z}[i], \mathbb{Q}$ and $\mathbb{Q}[i]$ are unique factorization domains. However, to exploit that uniqueness efficiently, in factored expressions we should uniquely represent multinomial factors that differ only by a unit multiple such as -1. This has the additional benefit of making syntactic common factors more frequent, reducing the need for polynomial expansion.

More seriously, not making the numerator and denominator multinomials unit normal in the following example can prevent improving the following expression to 1:

$$\frac{((3-5i)\,z+1)\,((7+i)\,z+i)}{((5+3i)\,z+i)\,((1-7i)\,z+1)}.$$

Definition. The **leading numeric coefficient** of an extended polynomial is recursively:

- the extended polynomial if it is a number,
- 1 if the extended polynomial is a unomial,
- the leading numeric coefficient of the coefficient if the extended polynomial is a unomial-headed term that isn't a unomial,
- otherwise the leading numeric coefficient of the leading term.

Definition. An expression is **unit normal over** $\mathbb{Z}$ if its leading numeric coefficient is positive.

If not, it can be made so by factoring out the unit -1.

Definition. An expression is **unit normal over** $\mathbb{Q}$ **or** $\mathbb{Q}[i]$ if its leading numeric coefficient is 1.

If not, it can be made so by factoring out the leading numeric coefficient, which is a unit in these domains.

Definition. An expression is **unit normal over** $\mathbb{Z}[i]$ if for its leading numeric coefficient c, $-\pi/4 < \arg(c) \leq \pi/4$.

If not, it can be made so by factoring out the unit -1 and/or the unit i. This is one of two alternative definitions motivated and described in more detail in reference [17].

3.4 Recursive factorization of unit quasi content

We can further increase the likelihood of syntactic common factors by factoring out their *quasi content*:

Definition. The **quasi content** of a recursive partially-factored sum is the product of the least powers of all syntactic factors among its terms, multiplied by the gcd of the numeric factors of those terms. The quasi content is computed and factored out level by level, starting with the least main generalized variables.

Definition. The **unit quasi content** of a partially-factored sum is the product of its quasi content and the unit that is factored out to make the sum unit normal.

Definition. A recursive partially-factored sum that has its quasi content factored out at all levels is **quasi primitive**.

Definition. A recursive sum is **unit quasi primitive** if it is unit normal and quasi primitive at every level.

For example with $x \succ a \succ b$,

$$
\begin{aligned}
&-6a^2b\boldsymbol{x} + 6a^2\boldsymbol{x} + 6b^2\boldsymbol{x} - 6\boldsymbol{x} + \mathbf{8}a^2\boldsymbol{(a+b+9)^9} - \mathbf{8}\boldsymbol{(a+b+9)^9} \\
&\quad \xrightarrow{\text{recursive}} \left(\left(\mathbf{-6}b^2 + \mathbf{6}\right)a^2 + \left(\mathbf{6}b^2 - \mathbf{6}\right)\right)x + 8\left(a^2-1\right)(a+b+9)^9 \\
&\quad \xrightarrow{\text{innermost}} \left(-6\left(\boldsymbol{b^2} - \mathbf{1}\right)a^2 + 6\left(\boldsymbol{b^2} - \mathbf{1}\right)\right)x + 8\left(a^2-1\right)(a+b+9)^9 \\
&\quad \xrightarrow{\text{mid-level}} \mathbf{-6}\left(\left(b^2-1\right)\right)\left(\boldsymbol{a^2} - \mathbf{1}\right)x + \mathbf{8}\left(\boldsymbol{a^2} - \mathbf{1}\right)(a+b+9)^9 \\
&\quad \xrightarrow{\text{outermost}} \mathbf{-2}\left(\boldsymbol{a^2} - \mathbf{1}\right)\left(3\left(b^2-1\right)x - 4(a+b+9)^9\right).
\end{aligned}
$$

Not only does this preserve the entered internal factor $(a+b+9)^9$ — it also discovers another internal factor b^2-1 and a top-level factor a^2-1. In contrast, *distributed* partially factored form can't represent the internal factors, and it could discover only the unit -1 and the top-level numeric content of 2. Worse yet, the unavoidable forced expansion of $(a+b+9)^9$ would add many more terms, with many non-trivial coefficients.

Determining minimum degrees of syntactic factors requires a number of base and exponent comparisons that are each bounded by the number of non-numeric syntactic factors in the recursive form. Determining the units to factor out at each level requires less work because only the leading terms at each level must be inspected. Determining the mutual gcd of n numeric coefficients is $n-1$ sequential gcds that start with the gcd of the first two coefficient magnitudes and decrease from there. This is significantly faster than $n-1$ independent gcds between those coefficients, because whenever the net gcd doesn't decrease much, few remainders were required, whereas whenever the net gcd decreases substantially, fewer remainders are required for subsequent gcds.

At each level, if a unit quasi content isn't 1, factoring it out requires effort proportional to the number of top-level terms at that level, plus some possible effort for numeric divisions. Thus the overall cost of making a polynomial unit quasi primitive is at most a few times the cost of thoroughly distributing the units and quasi contents back over the terms as much as is allowed with recursive representation.

3.5 Demand-driven extraction of signed quasi content

We want to represent and operate directly on both expanded and partially-factored recursive expressions. Partially-factored recursive sums are candid regardless of whether they are unit quasi primitive or not. A sum might candidly have unit quasi contents fortuitously factored out at some of the deepest levels but not at shallower levels, with different boundaries for different terms.

The default policy of least guessed cost together with the desire to represent and operate on recursively-expanded extended polynomials as well as on partially-factored ones indicates that we should not automatically unit quasi-primitize a sum result. The next operation, if any, might force partial or total redistribution. Instead we should postpone unit quasi-primitization until we guess that it is the least costly alternative.

For a fully-expanded recursive extended polynomial, sums can occur only at the top level and/or as the last factor in products. Therefore it is helpful to require that all powers of sums and all products containing sums anywhere else are unit quasi-primitive. This makes similar factors more likely and makes it easy to infer that such sums are already unit quasi-primitive. Powers and/or products containing unit quasi-primitive sums are efficiently and candidly multiplied by merging and combining similar factors. Polynomial gcds are required only between factors whose bases are sums raised to oppositely-signed multiplicities. Also, similar factors having such sums as bases can be extracted to reduce the amount of expansion when adding two powers or unit quasi primitive products.

When a sum or a power thereof that isn't the last factor in a product thus implies that all sums in the product are unit quasi primitive, this information can be passed down recursively so that recursions that treat terminal sum-factors will know that they are unit quasi primitive without having to inspect the data to verify that.

It is possible to store information about known unit quasi primitiveness or other factorization levels with each sum. However, consistently managing such information substantially complicates the implementation. Moreover, this approach increases the data size — particularly if done at every recursive level, as it has to be for full effectiveness. Therefore *Derive* and TI-CAS instead re-determine such properties when they can't easily be inferred or passed as a flag to functions that can exploit the information. Even without such information, the time it requires to determine that a sum is already unit quasi primitive is less than the time that it requires to unit quasi-primitize it if it isn't. Moreover, the mean time it requires to determine that a sum *isn't* unit quasi primitive is even less because that fact might be revealed after inspecting only a small portion of the terms — often only one term.

Heuristics can help guide the choice when there is a choice between co-distribution of two sums or making them both unit quasi primitive. For example, If one of the sums is the same as the other but with conjugates or opposite signs for one or more of the coefficients but not all of them, then there is likely to be some beneficial cancellation when these two to conjugate sums are co-distributed.

4 Recursively partially-factored semi-fractions

Fractions, fractions everywhere, with opportunities to share.

Common denominators can enable cancellations that reduce degrees, eliminate denominators, or eliminate variables. For example,

$$\frac{1}{x-1}-\frac{1}{x+1}-\frac{2}{x^2-1}\ \overset{:)}{\longrightarrow}\ 0.$$

However, common denominators can be costly in computing time and in the size of the result. For example,

$$\begin{aligned}&\frac{a}{a-1}+\frac{b}{b-1}+\frac{c}{c-1}+\frac{d}{d-1}\\ &\quad\rightarrow\frac{(((4d-3)\,c-3d+2)\,b-(3d-2)\,c+2d-1)\,a-((3d-2)\,c-2d+1)\,b+(2d-1)\,c-d}{(a-1)(b-1)(c-1)(d-1)}.\end{aligned}$$

This is **the common denominator dilemma**: You might be *damned if you do* force common denominators, but you might be *damned if you don't* force common denominators.

Also, users feel too constrained if all of their default results are forced to have a common denominator. For example, when integration entails distribution of integration over sums, most users prefer to see the result as a corresponding sum — particular in education.

In contrast to Altran, *Derive* and TI-CAS use a candid form for extended rational expressions that also accommodates a sum of an extended polynomial and any number of proper ratios of extended polynomials having denominators whose pairwise gcds don't contain the more main of their two main variables. The denominators do not need to be square-free or irreducible. Moreover, these denominators together with the extended polynomial part and the numerators of the fractions can be partially factored. This *partially-factored semi-fraction form* thus flexibly extends the Altran spectrum through partial fractions. Regarding rational expressions and the rational aspects of irrational expressions, the broad spectrum from factored over a common denominator through partial fractions accommodates most of the result forms often wanted by most users for default simplification.

4.1 Quasi-primitation implies common denominators

Using multiplication and negative exponents to represent ratios leads to the insight that combining expressions over a common denominator is simply quasi-primitation, which is a partial factorization. For example with multinomials B and D, the internal representation of

$$\frac{A}{B}+\frac{C}{D}$$

is $AB^{-1}+CD^{-1}$, for which factoring out the lowest degree of each syntactic factor gives $(AD+BC)B^{-1}D^{-1}$, which is displayed as

$$\frac{AD+BC}{BD}.$$

The only additional responsibility for negative exponents of *sums* is to check for possible gcd cancellations between dissimilar sum factors raised to positive and negative multiplicities.

Definition. Extended rational expressions are composed of extended polynomials and ratios of integer powers of extended rational expressions.

Making an extended rational expression unit quasi primitive and canceling multinomial gcds between numerators and denominators makes it candid: Quasi-primitation recursively factors out the lowest occurring degree of syntactically similar factors, forcing a single common denominator and making the exponents all positive within multinomial factors. This together with the multinomial gcd cancellation guarantees that the total apparent degree for each variable is the actual degree. That in turn guarantees that there are no superfluous variables and that extended polynomials don't appear to be more general extended rational expressions.

4.2 Extended Polynomials with rational expressions as coefficients

As discussed in section 3.1, the coefficients in a candid extended polynomial can be any candid expressions having only lesser variables. This includes candid extended rational expressions. Thus recursive form easily accommodates expressions that are extended polynomials having coefficients that are candid extended rational expressions in lesser variables, such as for $x \succ y \succ z$:

$$\left(y^{5/2} + \frac{z^2}{2z+1}y + \frac{7}{z+4}\right)x^2 + \left(\frac{1}{y}\right)x + \left(\frac{3y^3}{y+5}\right).$$

This form is candid despite the lack of a common denominator, because the coefficients (including those of implicit y^0 within the coefficient of x^2 and x^0 for the last term) are all candid expressions in lesser variables.

4.3 Adding improper ratios to extended polynomials

Definition. A **term** is any expression or sub-expression that isn't a sum at its top level.

Definition. A **sum-headed term** is a term whose leading factor is a sum or a power thereof.

It is helpful to order terms in a sum into descending order of their main variables. Among terms having the same main generalized variable it is helpful to order the sum-headed terms first so that the important presence of sum-headed terms in the main generalized variable is more quickly determined. Among sum-headed terms having the same main generalized variable, it is important to order the terms in some well-defined and easily-computed order.

In contrast, rational expressions consisting of a polynomial plus proper-ratio terms are traditionally *displayed* with all the polynomial terms *left* of any proper-ratio terms. A post-simplification pass can be used to display such terms in this more traditional order.

Definition. A **proper term** is one that isn't sum headed or for its main variable the degree of its numerator is less than the degree of its denominator.

An improper sum-headed term can be made proper by using division with remainder with respect to its main generalized variable. This transforms the term into an extended polynomial in that generalized variable plus a proper sum-headed term in that generalized variable.

Regardless of the variables therein, a *proper* term can always candidly coexist with unomial-headed terms and/or a numeric term.

We have already seen that an *improper* term can candidly coexist with a unomial-headed terms having greater main generalized variables. However, if an *improper* term would order before another term, then we should either force a common denominator for those two terms or make the ratio proper to allow possible important cancellations. For example, using the internal ordering of terms with $x \succ y \succ b$,

$$\begin{aligned} x + \frac{\boldsymbol{b}y + \boldsymbol{b} + 1}{y+1} + y - \boldsymbol{b} \quad &\rightarrow \quad x + \frac{y^2 + y + 1}{y+1} \\ \textbf{or} \quad &\rightarrow \quad x + \left(\boldsymbol{b} + \frac{1}{y+1}\right) + y - \boldsymbol{b} \\ &\rightarrow \quad x + \frac{1}{y+1} + y, \end{aligned}$$

either of which eliminates the superfluous variable b.

4.4 Making a ratio proper can introduce removable singularities

Unfortunately, it is a little known fact that making a ratio proper can contract the domain of equivalence by introducing removable singularities if the leading coefficient of the denominator isn't constant. For example, using the internal ordering of terms with $x \succ c$,

$$\frac{x}{cx-1} \quad \overset{:(}{\rightarrow} \quad \frac{1}{c\,(cx-1)} + \frac{1}{c}.$$

At $c = 0$ the left side simplifies to $-x$, whereas the right side is $\pm\infty - \pm\infty$. Also, for approximate arithmetic the right sides is more prone to underflow, overflow and catastrophic cancellation near $c = 0$. Moreover, the right side is less candid because it suggests that c would be a factor in a reduced common denominator.

In contexts such as where integration requires a proper fraction, we can either ask the user if $c = 0$ or use a piecewise result such as

$$\begin{aligned} \int \frac{x}{cx-1}\,dx \quad &\rightarrow \quad \int \begin{cases} -x & \text{if } c = 0 \\ c\left(x + \frac{1}{c}\right) + \frac{1}{c} & \text{otherwise} \end{cases} dx \\ &\rightarrow \quad \begin{cases} -x^2/2 & \text{if } c = 0, \\ \frac{\ln(cx-1)}{c^2} + \frac{x}{c} & \text{otherwise.} \end{cases} \end{aligned}$$

When proper fractions aren't mandatory, default simplification should use the common denominator choice if the leading coefficient of the denominator could be 0 for some values of the variables therein within the problem domain.

If the denominator of a proper sum-headed term has a non-numeric gcd with the denominator of a lesser term. then combining those two terms over a common denominator might improve candidness. For example with $x \succ y \succ c$,

$$\frac{1}{\boldsymbol{c}\,(cx-1)} + \frac{1}{\boldsymbol{c}} \quad \overset{:)}{\rightarrow} \quad \frac{x}{(cx-1)}.$$

The left side falsely suggests that c would be a common factor in the reduced common denominator. Also, only the left side is undefined at $c = 0$. However, forcing a common denominator in such situations could make some results of a properFraction (...) function be ephemeral. Also, it is not as heinous for default simplification to decline an opportunity to remove a removable singularity as it is to introduce one.

4.5 Sums of ratios having the same main variable

Even if their main generalized variables are the same, proper ratios can candidly be merged together as a sum if the gcd of their denominators is numeric. In contrast, there might be important cancellations between sums of *improper* ratios even if the gcd of their denominators is numeric. For example with $x \succ c$:

$$\begin{aligned}\frac{\boldsymbol{c}x+\boldsymbol{c}+1}{x+1}-\frac{\boldsymbol{c}x-\boldsymbol{c}-1}{x-1} &\rightarrow \left(\frac{1}{x+1}+\boldsymbol{c}\right)-\left(-\frac{1}{x-1}+\boldsymbol{c}\right)\\ &\rightarrow \frac{1}{x+1}+\frac{1}{x-1},\end{aligned}$$

which eliminates the superfluous variable c. Therefore a good default is to combine such ratios over a common denominator if it removes a singularity or if making the ratios proper requires a piecewise result. Otherwise make the ratios proper because it is closer to the input and likely to cost less.

There can also be important cancellations between the sum of two ratios A/B and C/D if the gcd G of their denominators is non-numeric. One alternative is to combine such ratios; and that is what *Derive* and TI-CAS do. However, if the main variable of G is the same as that of B and D, then we can instead:

- Split A/B into an extended polynomial part and two proper semi fractions having denominators G and B/G.
- Split C/D into an extended polynomial part and two proper semi fractions having denominators G and D/G.
- Combine the extended polynomial parts and combine the numerators of the fractions having denominator G, then passively merge that result with the passive sum of the proper ratios having denominators B/G and D/G.

If G is small compared to both B and D, then splitting is more likely to give a less bulky result than combining. Here is a borderline example:

$$\begin{aligned}\frac{x^2+x-3}{(x^2-1)(x-2)}-\frac{2x}{(x-2)(x+2)} &\rightarrow -\frac{x^2-x-3}{(x^2-1)(x+2)}\\ \text{or } &\rightarrow \left(\frac{1}{x^2-1}+\frac{1}{\boldsymbol{x-2}}\right)-\left(\frac{1}{\boldsymbol{x-2}}+\frac{1}{x+2}\right)\\ &\rightarrow \frac{1}{x^2-1}-\frac{1}{x+2}.\end{aligned}$$

Notice that $1/(x^2-1)$ wasn't split. There was no need to split it.

Splitting a proper fraction into semi fractions can introduce singularities if the leading coefficient of the given denominator can be 0 in the problem domain. For example,

$$\frac{2x}{c^2x^2-1} \rightarrow \begin{cases}-2x & \text{if } c=0,\\ \dfrac{1}{c\,(cx-1)}+\dfrac{1}{c\,(cx+1)} & \text{otherwise.}\end{cases}$$

Combining fractions is a better default in such cases or when combining fractions eliminates a singularity.

5 Summary

Good default simplification should produce an equivalent result wherever the input is defined in the problem domain. This might require a piecewise result or querying the user and attaching a constraint to their input. Also, users should optionally be able to prevent domain enlargement by having constraints automatically attached to the output. Constraints provided by the user and system should be propagated to the output to prevent substitution of inappropriate values. These goals require implementation of a domainOfDefinition (...) function that is easily customized by users to consider any subset of uniqueness, finiteness and realness.

Also, users should be able to optionally disable default transformations.

Most important, default simplification should try hard quickly to produce a nearby idempotent candid result in the spectrum from fully factored over a common denominator through complete multivariate partial fractions.

Derive and TI-CAS implement a partially-factored semi-fraction form and associated default simplification algorithms that go a long way towards fulfilling these goals.

Appendices

A Very-proper terms

Definition. A sum-headed term of the form $\frac{N}{D^\alpha}$ with D square free, main variable x, and $\alpha \geq 1$ is a **very proper term** if the degree of x in N is less than the degree of x in D.

For reduced ratios having denominators that are numeric or lesser-variable multiples of the same square-free polynomial raised to different powers:

- They can candidly coexist if they are all very proper.
- Otherwise we should either combine the terms or further split them so that all of them are very proper. For example,

$$\begin{aligned} \frac{x+2}{(x+1)^2} - \frac{1}{x+1} \quad &\rightarrow \quad \frac{(x+2)-(x+1)}{(x+1)^2} \rightarrow \frac{1}{(x+1)^2} \\ \textbf{or} \quad &\rightarrow \quad \left(\frac{1}{(x+1)^2} + \frac{1}{x+1}\right) - \frac{1}{(x+1)} \rightarrow \frac{1}{(x+1)^2}. \end{aligned}$$

Further expansion of proper ratios into very-proper ratios often increases total bulk. However, some algorithms such as integration sometimes require very-proper ratios. Default *Derive* and TI-CAS simplification combine ratios for which the gcd of the denominators is non-numeric. Therefore although very-proper fractions are produced by the optional expand(...) function and when needed for purposes such as integration, they can be ephemeral.

B Preserving primitive factors in reduced ratios

Definition. The **term content** of a recursively-represented multinomial is the gcd of its top-level terms, regarded as an extended polynomial in its main generalized variable.

Definition. A recursively-represented multinomial is **recursively term primitive** if its term content is 1 at every recursive level.

To recursively primitize a quasi-primitive polynomial: At each recursive level of each quasi-primitive multinomial factor, starting with the deepest levels, factor out the gcd of the terms. This might entail non-trivial polynomial gcds if any coefficients have multinomial factors. This might also result in a multinomial having partially-factored coefficients.

Most polynomial gcd algorithms and many factoring algorithms either require or benefit from further factoring a quasi-primitive multinomial into a term-primitive polynomial times a term content, and from recursively making that content be term primitive with respect to its main variable, etc. Therefore as a side effect of primitizing a numerator and denominator to assist computing their gcd, the immediate result of the reduction is that every multinomial factor in the reduced result is recursively term primitive with respect to its main variable. This knowledge can save significant time when the ratio is combined with another expression or when further factorization is desired. For example:

- We can skip the primitization step on the numerator or denominator when computing its gcd with another multinomial.
- If two term-primitive multinomials have different main variables, then they are relatively prime, allowing us to avoid computing their gcd.
- If a multinomial is term primitive in a variable and linear in that variable, then the multinomial is irreducible, allowing us to avoid a futile attempt at gcds or further factorization.

Also, primitation further increases the chances of syntactically similar factors that can be combined and shared.

Primitation involves gcds of polynomials having fewer variables than the original quasi-primitive multinomial, and the cost of multinomial gcds generally grows rapidly with the number of variables. For this reason and reasons similar to computation of the numeric content, primitation is often less costly than computing the gcds between the resulting primitive polynomials. For example, it is worth considering the primitated coefficients in order of increasing complexity so that their iteratively updated gcd is likely to approach 1 more quickly. For quasi-primitive multinomials, the term content must be multinomial, and any variable not present in all of the coefficients can't occur in the final content. Therefore the multinomial part of the content is 1 if the intersection of the variables occurring in the multinomial factors of the coefficients, S, is the empty set. Also, we can substitute judicious numeric values for variables not in S without having to lift to restore those variables. For these reasons, recursive primitation can be worth the investment in some circumstances even when not needed for ratio reduction.

The fact that default simplification leaves the numerators and denominators of ratios recursively primitive when they have sums in their denominators means that if the user requests an expanded numerator and/or denominator, it might be ephemeral. However, this is alright, because:

- Term-primitive factorization is generally preferable to an expanded numerator and denominator in most respects.
- In the rare cases where a fully-expanded numerator and/or denominator is helpful, such as facilitating some default and optional transformations for fractional powers and functional

forms as described in appendices C and D, these transformations can be facilitated by a provisional expansion followed by re-primitation if any such transformation then occur.

C Additional considerations for fractional exponents

Hearn and Loos [5] remark that quotients, remainders, gcds and many other polynomial operations can be well defined for fractional exponents of variables. For division and gcds we want nonnegative exponents, and quasi-primitation accomplishes that. As examples of division and gcds for such extended multinomials,

$$\frac{z-1}{z^{1/2}-1} \overset{:)}{\rightarrow} z^{1/2}+1,$$

and

$$\gcd\left(x-1,\, x+2x^{1/2}+1\right) \rightarrow x^{1/2}+1.$$

Polynomial remainder sequence gcd algorithms require no change. However, any polynomial division or gcd algorithm that relies on substituting numbers for variables should first temporarily substitute for any variable x that has fractional exponents in either polynomial, a new variable $t^{1/g}$, where g is the gcd of all the occurring exponents of that variable in both polynomials. (The gcd of two reduced fractions is the gcd of their numerators divided by the least common multiple of their denominators.) Even for all integer exponents this substitution can have the advantage of reducing the degrees, which is important to algorithms that substitute numbers for variables.

Regarding factoring, allowing the *introduction* of fractional exponents of factorization variables makes factoring non-unique and not very useful. For example, we could factor $x-1$ into $(x^{1/2}-1)(x^{1/2}+1)$ or into $(x^{1/3}-1)(x^{2/3}+x^{1/3}+1)$ or into an infinite number of different such products. Instead, we should bias the partially-factored form to expand by default when fractional powers of a variable might thereby be eliminated or reduced in severity.

Common denominators can similarly help eliminate fractional powers. For example,

$$\frac{1}{z^{1/2}-1} - \frac{1}{z^{1/2}+1} \rightarrow \frac{2}{z-1}.$$

Thus it is also worth biasing toward common denominators when fractional powers might thereby be eliminated or reduced in severity.

Collecting similar factors that are fractional powers can enlarge the domain of definition for variables that are real by declaration or default. For example,

$$x^{1/2}x^{1/2} \mid x \in \mathbb{R} \overset{:)}{\rightarrow} x \mid x \in \mathbb{R}$$

enlarges the domain of definition from $x \geq 0$ to all x. Thus with domain-enlargement prevention enabled, the result would instead be $x \mid x \geq 0$. If the user is also using the real branch of fractional powers having odd denominators, such as $(-1)^{1/3} \rightarrow -1$, then we should append the constraint only if for some radicand, fractional powers of that radicand having an even denominator entirely disappear in the result.

Fractional powers of numbers, powers, products and sums involve additional complications that can be superimposed on the algorithms for extended rational expressions over $\mathbb{Z}[i]$. For example, there are additional considerations such as de-nesting and rationalization of denominators or numerators. Also, for internal simplification it is helpful to distribute exponents over products and to multiply the exponents of powers of powers. However, it is not always correct to do so for fractional powers without including a rotational correction factor. Two always-correct principal-branch rewrite rules for exponents are

$$(zw)^{\beta} \;\rightarrow\; (-1)^{(\arg(zw)-\arg(z)-\arg(w))\beta/\pi}\, z^{\alpha}w^{\alpha}, \tag{4}$$

$$(z^{\alpha})^{\beta} \;\rightarrow\; (-1)^{(\arg(z^{\alpha})-\alpha\arg(z))\beta/\pi}\, z^{\alpha\beta}. \tag{5}$$

Depending on any declared realness of z and w or on declared intervals for $\arg(z)$ and $\arg(w)$, the exponent of -1 tends to be quite complicated unless we can simplify it to a constant. Therefore, transformations based on these identities are usually unwise unless that happens, as it always does for $z \geq 0$ or for integer α. To maximize opportunities for exploiting these identities, it is generally best to factor multinomial radicands over $\mathbb{Z}$ or $\mathbb{Z}[i]$. Often, this is enough to extract at least a numeric factor from a radicand.

D Additional considerations for functional forms

It is helpful to force the arguments of a functional form to a particular canonical form that can depend on the set of optional or default rewrite rules for the function or operator.

Fully-expanded arguments are a good choice for functions or operators that have a desired rewrite rule for arguments that are sums or numeric multiples. For example,

$$\exp(u+v) \;\rightarrow\; \exp(u)\exp(v), \tag{6}$$

$$\exp(nu) \;\rightarrow\; (\exp(u))^{n}, \tag{7}$$

$$\sin(u+v) \;\rightarrow\; \sin(u)\cos(u)+\cos(u)\sin(v), \tag{8}$$

$$\sin(2u) \;\rightarrow\; 2\sin(u)\cos(u), \tag{9}$$

$$\int (u+v)\, dx \;\rightarrow\; \int u\, dx + \int v\, dx. \tag{10}$$

Even if the rewrite rule is optional rather than default, expanded arguments relieve users from having to explicitly request the expansion before applying the rewrite rule. Moreover, expanded arguments reveal and suggest the applicability of the optional rules.

For analogous reasons, *a canonical factored form* is a good choice if the function or operator has a rewrite rule for products or powers in one of its arguments, such as

$$|uv| \;\rightarrow\; |u|\cdot|v|, \tag{11}$$

$$|u^{k}| \;\rightarrow\; |u|^{k}. \tag{12}$$

In the absence of either kind of rewrite rule, it is nonetheless helpful to force the arguments to a particular canonical form that can depend on the particular function. Otherwise, opportunities for collecting and canceling similar factors or terms can be missed, leading to a non-candid result. For example, we want

$$f\left(x^{2}-1\right) - f\left((x-1)(x+1)\right) \;\overset{:)}{\Rightarrow}\; 0$$

for any $f(\ldots)$. For the arguments of such functional forms we could choose a canonical form that tends to be compact and not too costly to compute, such as square-free factored form or square-free multivariate partial fractions. However, sub-expressions outside functional forms rarely move inside them. Consequently argument size tends to be small compared to top-level extended rational expressions containing those functional forms. Therefore, a fully factored or fully expanded form over $\mathbb{Z}$ is rarely costly for functional-form arguments. Moreover, for internal representation it is most often helpful instead to move as much of the arguments as possible *outside* functional forms, which increases the chance of the similar terms that can combine or cancel.

Rewrite rules for powers or products of functional forms must be superimposed on the algorithms for the partially-factored semi-fraction form. For example, consider the rules

$$\begin{aligned} \cos(u)^2 &\rightarrow 1-\sin(u)^2, \\ \sin(u)\cos(v) &\rightarrow \frac{\sin(u-v)+\sin(u+v)}{2}. \end{aligned}$$

Opportunities for using such rules are easier to recognize and exploit if the default is biased toward common denominators and expanding products and powers of sums when they contain appropriate sinusoids. For example,

$$\begin{aligned} \frac{\sin(x)}{\cos(x)+1}+\frac{\sin(x)}{\cos(x)-1} &\rightarrow \frac{2\sin(x)\cos(x)}{(\cos(x)+1)(\cos(x)-1)} \\ &\rightarrow \frac{2\sin(x)\cos(x)}{\cos(x)^2-1} \\ &\rightarrow \frac{2\sin(x)\cos(x)}{-\sin(x)^2} \\ &\rightarrow \frac{-2\cos(x)}{\sin(x)}, \end{aligned}$$

which the post-simplification pass could display as $-2\cot(x)$. Even where such rules are optional rather than default, expanding over a common denominator makes the opportunities more obvious to users.

Rewrite rules for sums of functional forms must also be superimposed on the algorithms for the partially-factored semi-fraction form. For example, consider the always-correct principal-value rewrite rule

$$\ln(u)+\ln(v) \rightarrow \ln(uv)+(\arg(u)+\arg(v)-\arg(uv))\,i.$$

Opportunities for using such rules are easier to recognize and exploit if the default is biased toward factoring sums when they contain such functional forms. For example,

$$\begin{aligned} (\ln x)^2+2\ln 2\,\ln x+(\ln 2)^2-1 &\rightarrow (\ln x+\ln 2+1)(\ln x+\ln 2-1) \\ &\rightarrow (\ln(2x)+1)(\ln(2x)-1)\,. \end{aligned}$$

If unomials have functional forms as generalized variables, then rewrite rules between functional forms might require additional checks to make sure that recursive form is candid. For example with $\cos(x) \succ \sin(x) \succ y$,

$$(y+1)\cos^2(x)+y\sin^2(x)$$

complies with recursive representation. However, for candidness it should be transformed to $y\cos^2(x)+1$.

One approach to simplifying expressions containing functional forms is to exploit dependency theorems such as a Risch structure theorem [13]. The idea is: Each time you combine two simplified expressions both containing functional forms, you set up then attempt to solve a system of equations expressing conjectured dependency between the functional forms. If there is no solution, then all of the functional forms are independent and can candidly coexist. Otherwise the solution indicates how to represent a subset of the functional forms in terms of the other functional forms.

By itself, this method doesn't prescribe which subset to use as a basis, so it isn't canonical. Also, two sub-expressions containing functional forms can combine many times during the course of simplifying an input. Consequently there can be many times requiring a complete scan of both operands to set up then solve the equations — perhaps the same set that has already been considered for a different sub-problem.

An approach that tends to avoid these difficulties is instead to use rewrite rules that move as much of the arguments as possible outside the arguments of functional forms, driving them toward canonicality. For example, all of the numbered rewrite rules in this appendix and appendix are of that type.

However, displayed results are often more concise if the *number* of functional forms is reduced by using such rewrite rules in the opposite direction during a post-simplification pass.

Special attention must also be given to infinities and multi-valued expressions. For example, we don't want either $\infty-\infty$ or $\pm1-\pm1$ to simplify to 0.

E Multiplying & adding partially-factored semi fractions

This appendix contains pseudo code for the top-level functions that simplify products and sums of recursively partially-factored semi fractions. This includes polynomial expansion and reduction over a common denominator *versus* merging factors and terms when that is guessed to be more economical.

Definition. A term is **polynomially expandable** if it is sum headed and has as least one factor that is either a sum containing the main generalized variable or a positive integer power of such a sum, and contains no negative integer powers of a sum containing that generalized variable.

Operators such as "^", "·" or "+" in quotes designate nearly-passive construction of the corresponding data structures from the operands: The only simplifications for such nearly-passive operations are that 0 and 1 identities are exploited and operands are merged lexically.

When quotes are omitted from the operators, it designates an invocation of the corresponding active simplification function. For brevity, additional considerations for domain enlargements and for irrational, multi-valued, or functional-form operands are omitted. Also, experienced implementers will notice places where efficiency can be improved at the expense of brevity. For example, some recursion and redundant computations can be avoided by swapping operands rather than recurring and by using *knownFactorLevel* parameters that pass information about unit quasi primitiveness and primitiveness.

Variables that aren't formal parameters are local. Numeric elements can be any mixture of elements of $\mathbb{Z},\mathbb{Q},\mathbb{Z}[i],\mathbb{Q}[i]$, and approximate numbers. They could also be multi-intervals as recommended in reference [14], but they aren't in the current implementation.

Pseudo code is omitted for auxiliary functions having obvious roles.

```
function $E_1 \cdot E_2$:
{ if $E_1$ is numeric, then
  { if $E_2$ is numeric, then
       return their numeric product;   // Floating-point is locally infectious
    if $E_2$ is a non-sum or a unit quasi-primitive sum, then
       return leadFactor($E_2$)“·”($E_1$·remainingFactors($E_2$);
    if it seems significantly easier to distribute $E_1$ than to make $E_2$ unit quasi primitive,
       return distribute($E_1$,$E_2$,mainVariable($E_2$));    //E.g: common denominators are hard
    return $E_1$· unitQuasiPrimitate($E_2$);
  }
  if $E_2$ is numeric, then return $E_2 \cdot E_1$;
  $L_1 \leftarrow$ leadFactor($E_1$);                    // leadFactor($E_1$) $\rightarrow E_1$ if $E_1$ is a non-product
  $R_1 \leftarrow$ remainingFactors($E_1$);              // remainingFactors($E_1$) $\rightarrow 1$ if $E_1$ is a non-product
  $L_2 \leftarrow$ leadFactor($E_2$);    $R_2 \leftarrow$ remainingFactors($E_2$);
  if $L_1$ is similar to $L_2$, then                // The lead bases are identical
    return leadBase($L_1$)$^{\text{leadExponent}(L_1)+\text{leadExponent}(L_2)}$·($R_1 \cdot R_2$);
  if leadBase($L_2$) $\succ$ leadBase($L_1$) then return $E_2 \cdot E_1$;
  $P \leftarrow R_1 \cdot E_2$;
  if $L_1$ is a unomial then return $L_1$“·”$P$;
  // $L_1$ is a sum or a power of a unit quasi-primitive sum:
  $x \leftarrow$ mainVariable($L_1$);
  if $L_1$ and/or $P$ are not unit quasi primitive and it seems significantly harder
       to make them so than to co-distribute $L_1$ with $P$ with respect to $x$, then
    return coDistribute($L_1, P, x$);
  if $P$ isn't unit quasi primitive, then return $L_1$· unitQuasiPrimitize($P$);
  if $L_1$ isn't unit quasi primitive, then return unitQuasiPrimitize($L_1$) ·$P$;
  if $L_1$ has a negative power of a sum and $P$ has a positive power of a sum or vice versa,
  { if context doesn't guarantee that $P$ is primitive, then
       $P \leftarrow$ primitize($P$);                  // $P \leftarrow$ primitize(content($P$))·primitivePart($P$)
    if context doesn't guarantee that $L_1$ is primitive, then
       $L_1 \leftarrow$ primitize($L_1$);
    return productOfPrimitives($L_1, P$);      // Henrici-Brown algorithm [8]
  }                                            // See also references [2] and [4]
  if $L_1$ is a product, then return $L_1 \cdot P$;       // Primitation might make $L_1$ be a product
  return $L_1$“·”$P$;
} // end function ·
```

```
function $E_1 + E_2$:
{ if $E_1$ is numeric, then
   { if $E_2$ is numeric, then return their numeric sum;
      return leadTerm($E_2$) + ($E_1$ + reductum($E_2$));
   }
   if $E_2$ is numeric, then return $E_2 + E_1$;
   $L_1 \leftarrow$ leadTerm($E_1$);   $R_1 \leftarrow$ reductum($E_1$);      // reductum($E_1$) $\rightarrow 0$ if $E_1$ is a non-sum
   $L_2 \leftarrow$ leadTerm($E_2$);   $R_2 \leftarrow$ reductum($E_2$);      // leadTerm($E_2$) $\rightarrow E_2$ if $E_2$ is a non-sum
   if $L_1$ is similar to $L_2$, then                  // The lead factors are identical
      return (leadFactor($L_1$)$\cdot$(remainingFactors($L_1$) + remainingFactors($L_2$)) + ($R_1 + R_2$);
   if leadFactor($L_2$) $\succ$ leadFactor($L_1$), then return $E_2 + E_1$;
   $S \leftarrow R_1 + E_2$;
   if $L_1$ is unomial-headed, then return $L_1$" + "$S$;
   // $L_1$ is a unit quasi-primitive sum-headed term:
   $x \leftarrow$ mainVariable($L_1$);
   if $L_1$ has a negative power of a sum containing $x$, then
   { $D_1 \leftarrow$ passive product of denominator factors in $L_1$ depending on $x$;
      return ratioPlus($L_1, D_1, x, S, false$);
   }
   if $S$ is unit quasi primitive, then        // $L_1$ is already unit quasi primitive
   { if the syntactic content of $L_1$ and $S$ is 1, then
      { if $L_1$ is polynomially expandable, then return expand($L_1, x$) + $S$;
         return $L_1$" + "$S$;
      }
      $G \leftarrow$ syntacticNumericContent($L_1, S$);
      return $G\cdot(L_1/G + S/G)$;
   }
   if $L_1$ is polynomially expandable and it seems easier to do so than make $S$
      unit quasi primitive, then return expand($L_1, x$) + $S$;
   return $L_1$ + unitQuasiPrimitate($S$);
} // end function +
function ratioPlus($L_1, D_1, x, E, DoMakeProper$):
{ // See invocation in function "+" for the roles of $L_1, D_1, x$ and $E$.
  if $E$ is zero, then
  { if $DoMakeProper$ and $L_1$ is improper, then return makeProper($L_1$);
    return $L_1$;
  }
  $L_2 \leftarrow$ leadTerm($E$);
  if the denominator of $L_2$ is free of $x$, then
    return $L_2$ " + " ratioPlus($L_1, D_1, x$, reductum($E$), $true$);
  $D_2 \leftarrow$ product of denominator factors in $L_2$ depending on $x$;
  $G \leftarrow$ gcd($D_1, D_2$);          // Reference [4], but recursively implemented
  if $G$ is numeric, then
  { $S \leftarrow$ ratioPlus($L_1, D_1, x$, reductum($E$), $true$);
    if $S$ is zero, then return $L_2$;
    if $L_2$ is improper, then return makeProper($L_2$) + $S$;
    return $L_2$ " + " $S$;
  }
  if it seems easier to split denominator $G$ from $L_1$ and from $L_2$ than to combine them,
      and splitting doesn't introduce new singularities in the problem domain, then
    return reductum($E$) + splitThenAdd($L_1, L_2, D_1, D_2, G, x$);
  return reductum($E$) + $G^{-2}$ " · " combineOverCommonDenominator($L_1/G, L_2/G$);
} // Use the Henrici-Brown algorithm [8] for the common denominator.
```

Acknowledgments

Thank you Albert Rich for being so patient when I struggled so long experimenting with alternative data structures and partially-factored semi fractions. Thanks also to Sam Rhoads for helping with LaTeX. I also thank the MICA conference organizers and program committee for inviting me to give a talk, which encouraged me to organize and record the ideas presented here. Thanks in advance to any reviewer who makes suggestions that lead to improvement of the article.

References

[1] Beeson, M: Design principles of MathPert: Software to support education in algebra and calculus, in *Computer-human Interaction in Symbolic Computation*, N. Kajler editor, Springer-Verlag, pp. 89-115.

[2] Brown, W.S: On computing with factored rational expressions. *Proceedings of EUROSAM '74, ACM SIGSAM Bulletin* 8 (3), August, Issue Number 31, pp. 26-34.

[3] Graham, R.L., Knuth, D.E., and Patashnik, O., *Concrete Mathematics*, 2nd edition, Addison-Wesley, 1989, p. 162.

[4] Hall, A.D: Factored rational expressions in ALTRAN. *Proceedings of EUROSAM '74, ACM SIGSAM Bulletin* 8 (3), August, Issue Number 31, pp. 35-45.

[5] Hearn, A.C. and Loos, R.G.K: Extended polynomial algorithms. *Proceedings of ACM 73*, pp. 147-52.

[6] Jeffrey, D.J. and Norman, A.C: Not seeing the roots for the branches. *ACM SIGSAM Bulletin* 38 (3) pp. 57-66.

[7] Kahan, W., Branch cuts for complex elementary functions, in *The State of the Art in Numerical Analysis*, editors A. Iserles and M.J.D. Powell, Clarendon Press, 1987, pp. 206-207.

[8] Knuth, D.E: The Art of Computer Programming, Volume 2, 3rd edition, Addison-Wesley, Section 4.5.1.

[9] Milestones in Computer Algebra conference, 2008,
`http://www.orcca.on.ca/conferences/mica2008/`

[10] Moses, J: Algebraic simplification: a guide for the perplexed, *Proceedings of the second ACM symposium on symbolic and algebraic manipulation*, pp. 282-304.

[11] Moses, J., Photograph, *ACM SIGSAM Bulletin* 15 (4), November 1981, p. 9.

[12] Moses, J: Algebraic computation for the masses. (abstract) *Proceedings of the 1981 ACM symposium on symbolic and algebraic computation*, p. 168.

[13] Risch, R.H., Algebraic properties of the elementary functions of analysis. *American Journal of Mathematics* 101, pp. 743-759.

[14] Stoutemyer, D.R: Useful computations need useful numbers. *Communications in Computer Algebra* 41 (3-4), December 2007, pp. 75-99.

[15] Stoutemyer, D.R: Multivariate partial fraction expansion. *ACM Communications in Computer Algebra* 42, No. 4, December 2008, pp. 206-210.

[16] Stoutemyer, D.R: Ways to implement computer algebra compactly. Applications of Computer Algebra 2008, `http://www.risc.uni-linz.ac.at/about/conferences/aca2008/`, submitted for publication.

[17] Stoutemyer, D.R: Unit normalization of multinomials over Gaussian integers. *ACM Communications in Computer Algebra* 43, No. 3, September 2009, pp. 73-76.

[18] TI computer-algebra for calculators and computers,
`http://education.ti.com/educationportal`

[19] TI Developers Guide, `http://education.ti.com/educationportal`

[20] TI Student Math Guide application (SMG), `http://education.ti.com/educationportal`