Towards a Datatype Defining Rewrite System for Rational Arithmetic

Wijnand K. van Woerkom
10808981

Bachelor thesis
Credits: 18 EC

Bachelor Opleiding Kunstmatige Intelligente

University of Amsterdam
Faculty of Science
Science Park 904
1098 XH Amsterdam

Supervisor
dr. A. Ponse

University of Amsterdam
Faculty of Science
Science Park 904
1098 XH Amsterdam

July 2nd, 2017
Abstract

In [3] the concept of datatype defining rewrite systems (DDRSs) is introduced. A DDRS is an equational specification of a datatype that yields a ground-complete term rewriting system (TRS) when its equations are interpreted from left to right as rewrite rules. In [3] a number of DDRSs are presented for terms in unary, binary, and decimal notation. The goal of the present work is to review whether it is possible to further extend these specifications so that they may model rational arithmetic.

Some research has been done on this topic which we review in Section 2 in particular we compare and contrast the results proven in [2] and [1] regarding the existence of such specifications. In Section 3 we rephrase the approach for computing irreducible fractions proposed in [1] by using conditional rewrite rules, which results in the design for a conditional TRS (CTRS) that may give rise to the aforementioned specification.

Contents

1 Introduction .............................................. 3
   1.1 Datatype defining rewrite systems ....................... 3
   1.2 Extension to rational arithmetic ....................... 4

2 Unconditional rewrite rules for rational arithmetic .... 5
   2.1 Integer arithmetic .................................... 5
      2.1.1 Definitions of sign(x), |x|, and min(x, y) .......... 6
   2.2 Computing irreducible fractions ....................... 8
   2.3 Rational arithmetic ................................... 8

3 Conditional rewrite rules for rational arithmetic .... 10
   3.1 Unary view ........................................... 10
      3.1.1 Integer arithmetic ............................... 10
      3.1.2 Definitions of sign(x), |x|, and x ≤ y ............. 11
      3.1.3 Computing the greatest common divisor and irreducible fractions .......... 13
      3.1.4 Rational arithmetic ............................... 15
   3.2 Binary view ........................................... 16
      3.2.1 Definitions of sign(x) and 0 ≤ x .................. 18
      3.2.2 Computing the greatest common divisor .......... 18
      3.2.3 Computing irreducible fractions .................. 19

4 Conclusion and digression ......................... 20

A Alternative formulations ......................... 22
1 Introduction

In [3] the concept of datatype defining rewrite systems (DDRSs) is introduced. A DDRS is an equational specifications of a datatype that yields a ground-complete term rewriting system (TRS) when its equations are interpreted from left to right as rewrite rules. In [3] a number ofDDRScs are presented for terms in unary, binary, and decimal notation. The goal of the present work is to review whether it is possible to further extend these specifications so that they may model rational arithmetic.

The remainder of this section is spend further introducing the concept of DDRSs and such extensions, as well as discussing both a negative result proven in [2] and a positive result proven in [1] related to this topic. In Section 2 the work presented in [1] is reviewed in more detail with the goal of reconciling the aforementioned results. In Section 3 the approach proposed in [1] for computing irreducible fractions is phrased using conditional rules, which results in the design for a conditional TRS (CTRS) that may produce the intended set of normal forms. The work ends with a conclusion and digression in Section 4.

1.1 Datatype defining rewrite systems

A TRS consists of a signature, a term set induced by the signature and symbols for variables, and rules for rewriting terms. Generally we will use the concepts as defined and used in [5] and [3], but we repeat some of them here for convenience:

• A signature $\Sigma$ consists of a non-empty set of function symbols $F, G, \ldots$, each equipped with a fixed arity $n \in \mathbb{N}$. Nullary function symbols are allowed and are called constant symbols or constants.

• Given a set $\text{Var}$ containing symbols for variables (commonly $x, y, z, \ldots$) the set of terms over $\Sigma$, denoted $\text{Ter}(\Sigma)$, is defined inductively:
  
  (i) $\text{Var} \subseteq \text{Ter}(\Sigma)$.
  
  (ii) If $F$ is an $n$-ary function symbol and $t_1, \ldots, t_n \in \text{Ter}(\Sigma)$ then $F(t_1, \ldots, t_n) \in \text{Ter}(\Sigma)$.

• A rewrite rule for $\Sigma$ is a pair $(l, r) \in (\text{Ter}(\Sigma) - \text{Var}) \times \text{Ter}(\Sigma)$ that satisfies the condition that variables occurring in $r$ also occur in $l$. Such rules are denoted $l \rightarrow r$.

The functions in the signature are frequently denoted using infix or postfix notation, e.g. $4 + 5$ instead of plus(4, 5). Intuitively put, the rewrite rules operate on terms by use of pattern matching and variable substitution, and may be applied in context. The relation given by the union of the rewrite rules is denoted by $\rightarrow$ and its reflexive transitive closure by $\rightarrow^*$. Lastly we repeat some terminology. A term to which no rewrite rules apply is said to be in normal form. If a function symbol occurs in terms that are in normal form we call it a constructor (function). If there is no term $t$ such that there is an infinite rewriting sequence $t \rightarrow r \rightarrow l \rightarrow \ldots$ we say the TRS is strongly terminating (which we will just call terminating), and if for each $t, l, r \in \text{Ter}(\Sigma)$ such that $t \rightarrow l$ and $t \rightarrow r$ there exists $u$ such that $l \rightarrow^* u$ and $r \rightarrow^* u$ we say it is confluent, or ground-confluent if this holds only for terms that do not contain variables. A TRS is (ground-)complete if it is both terminating and (ground-)confluent.
When constructing a DDRSs the choice of intended normal forms is made beforehand, after which a complete TRS is sought that produces them. To exemplify the aforementioned concepts as well as this process we will repeat the construction of a DDRS for natural-number arithmetic.

As mentioned we start by defining our intended set of normal forms. To keep the number of rules required to a minimum we will use $\mathbb{N} = \{0\} \cup \mathbb{N}^+ \cup \mathbb{N}^- = \{S(0), S(x) \mid x \in \mathbb{N}^+\}$.

We want to do natural-number arithmetic and so we set $\Sigma = \{0, S, +, \cdot\}$. A well known set of equations that give rise to a DDRS for this signature and choice of normal forms is

\[
\begin{align*}
x + 0 &= x, \\
x + S(y) &= S(x + y), \\
x \cdot 0 &= 0, \\
x \cdot S(y) &= (x \cdot y) + x.
\end{align*}
\]

Now consider how we might extend our DDRS to integer arithmetic. First we need to enlarge the signature to be able to denote negative numbers; we opt for the minus sign in this case, but note that we could choose to use a predecessor function as well. Next we enlarge our set of normal forms $N := N \cup N^-$ with $N^- = \{-x \mid x \in \mathbb{N}^+\}$. To produce them we add the equations

\[
\begin{align*}
S(-S(x)) &= -x, \\
-0 &= 0, \\
-x &= x, \\
x - y &= -(x \cdot y).
\end{align*}
\]

Ground completeness proofs for these systems can be found in [3]. In the remainder of this work we will just refer to rewriting rules rather than equations.

### 1.2 Extension to rational arithmetic

Having seen DDRSs for integer arithmetic it is tempting to conclude that a similar extension to rational arithmetic is possible. Continuing our example we would expand the signature with a symbol for division, say $\div$, and set $N := N \cup N^\div$ with $N^\div = \{x \div y \mid x \in \mathbb{N}^+, y \in \mathbb{N}^+ - \{S(0)\}, \gcd([x],[y]) = 1\}$, as well as $N^- := N^- \cup \{-x \mid x \in \mathbb{N}^+\}$.

However unlike the other extensions finding these rules has shown to be problematic. The successor function gives very little information about the divisors of a number (as opposed to a constructor like $:\colon b_0$, cf. Section 2.1) and rules that reduce arbitrary rationals to normal form must not be applicable to rationals that are already reduced. As 0 and $\div$ are in the signature we would also require a way of handling division by zero. In [2] a theorem is proven that states under quite general conditions that no DDRS can exist for the kind of normal forms we have just described, and the authors conjecture that no DDRS can exist regardless of the choice of normal forms for some particular specifications of the rationals.

Despite these difficulties and the nonexistence result a TRS is proposed in [1] for reducing rationals to a similar choice of normal forms: fractions $x/y$ where $x$ is a nonzero integer, $y$ a positive integer and $(x, y)$ is a coprime pair. This has the appearance of a contradiction, but there are some differences between the approaches used in [2] and [1]. For instance, the authors of [1] use terms in balanced ternary notation, perform rewriting modulo certain equivalence classes, and take measures to prevent division by zero.

The goal of the present research is firstly to review how these results relate to each other.


2 Unconditional rewrite rules for rational arithmetic

In this section we review the results presented in [1] that are related to the development of rewrite rules for rational arithmetic. In particular, a TRS is provided which is able to reduce fractions to normal form and compute sums and products of rationals. This TRS is the union of four ordered parts: $R_{\text{int}} \subset R_{\text{incr}} \subset R_{\text{f}} \subset R_{\text{rat}}$. Henceforth we will also use these names to refer to the union up to that part, so for instance $R_{\text{f}}$ is also the union $R_{\text{int}} \cup R_{\text{incr}} \cup R_{\text{f}}$. The parts respectively serve to: form a complete TRS for integer arithmetic of terms in balanced ternary notation, define the functions $\text{sign}(x), \text{divides}(x, y)$ over integers, define a function $f : (x, y) \rightarrow x / \gcd(x, y)$ for reducing fractions, and lastly to define rules for rational arithmetic. We will consider these parts sequentially as is done in [1].

The goal of this review is to reconcile the results proven in [2] and [1], as well as explain the workings of the TRS $R_{\text{rat}}$ because this will motivate the approach used in Section 3 and was not done extensively in [1]. Along the way we will suggest some adjustments to the proposed approach, find it to be related to conditional term rewriting, and attempt to verify some of the results.

2.1 Integer arithmetic

As the chosen normal form for rationals makes use of integers the work starts out with incorporating integer arithmetic, the rules for which can be found in Table 1. The signature $\Sigma$ is given by the constant 0, functions $\cdot : t_0, \cdot : t_1 : \mathbb{Z} \rightarrow \mathbb{Z}$, and functions $- : \mathbb{Z} \rightarrow \mathbb{Z}, +, \cdot : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$ for arithmetic. The semantics of the constructors are given by

$$
[x; \cdot_1] = 3 \cdot [x] - 1,
[x; \cdot_0] = 3 \cdot [x],
[x; \cdot_1] = 3 \cdot [x] + 1,
$$

and of the others as usual (e.g. $[x + y] = [x] + [y]$). The full set of rules can be found in Table 1.

Balanced ternary numbers (BTN) are a variation on the common base-3 representation in that $\cdot_2$ is replaced with $\cdot_1$. This has a number of interesting consequences which are discussed in e.g. [4]. The reason Contejean et al. [1] chose BTN over a representation like binary is that BTN gives rise to a complete (rather than ground-complete) TRS. As we will see later the rules for rational arithmetic can be formulated for other number bases, and hence are not fundamentally related to BTN.

The intended set of normal forms may be defined as $N = \{0\} \cup N^1 \cup N^0 \cup N^1$ with

$$
N^1 = \{x; \cdot_1 \mid x \in N\},
N^0 = \{x; \cdot_0 \mid x \in N\},
N^1 = \{x; \cdot_1 \mid x \in N\}.
$$

The system is formulated modulo associativity and commutativity of addition and multiplication, which is an instance of rewriting modulo an equivalence class [5]. In practise this means that in between the application of rewrite rules terms may be switched with terms that are a member of their equivalence class. In the case of rewriting modulo AC of addition this means...
that \( x + y \) may be swapped with \( y + x \), or \( (x + y) + z \) with \( x + (y + z) \).

Adding the following rules seems to lift this condition as the tools AProVE [7] and CSI [6] are able to prove the resulting system is terminating and confluent, respectively. Hence this practise also does not play a significant role in the rules for rational arithmetic.

\[
\begin{align*}
[\text{AC1}] & \quad 0 + x \rightarrow x \\
[\text{AC2}] & \quad (x;\bar{t}1) + (y;\bar{t}0) \rightarrow (x + y);\bar{t}1 \\
[\text{AC3}] & \quad (x;\bar{t}1) + (y;\bar{t}0) \rightarrow (x + y);\bar{t}1 \\
[\text{AC4}] & \quad (x;\bar{t}1) + (y;\bar{t}1) \rightarrow (x + y);\bar{t}0
\end{align*}
\]

### 2.1.1 Definitions of \( \text{sign}(x) \), \( |x| \), and \( \text{min}(x,y) \)

The next part consists of defining a number of functions solely used for the implementation of \( f \). Most notably the signature is extended with \( \text{sign} \), \( \text{abs} : \mathbb{Z} \rightarrow \mathbb{Z} \) and \( \text{min} : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z} \). Their semantics are given by the equations

\[
\begin{align*}
\llbracket \text{abs}(x) \rrbracket &= \llbracket x \rrbracket, \\
\llbracket \text{sign}(x) \rrbracket &= \begin{cases} -1 & \text{if } \llbracket x \rrbracket < 0 \\
0 & \text{if } \llbracket x \rrbracket = 0, \\
1 & \text{if } 0 < \llbracket x \rrbracket
\end{cases},
\end{align*}
\]

\[
\begin{align*}
\llbracket \text{min}(x,y) \rrbracket &= \begin{cases} \llbracket x \rrbracket & \text{if } \llbracket x \rrbracket \leq \llbracket y \rrbracket \\
\llbracket y \rrbracket & \text{otherwise}
\end{cases}.
\end{align*}
\]

It should be noted that the semantics of \( \text{min} \) do not coincide with how it is usually defined; e.g.

it is not commutative and \( \text{min}(2, -3) = 2 \).

In turn a number of auxiliary functions are used to do this, which are indicated by superscript question marks. As an example, the semantics of \( \text{min}^7 \) are

\[
\begin{align*}
\llbracket \text{min}^7(x,y,z,u) \rrbracket &= \begin{cases} \llbracket y \rrbracket & \text{if } \llbracket z \rrbracket > 0 \lor (\llbracket z \rrbracket = 0 \land \llbracket y \rrbracket \geq 0) \\
\llbracket z \rrbracket & \text{otherwise}
\end{cases}.
\end{align*}
\]

Aside from these two functions denoted \( \text{min}^\prime \) and \( \text{min}'' \), are used with semantics

\[
\begin{align*}
\llbracket \text{min}^\prime(x,y) \rrbracket &= \begin{cases} \llbracket x \rrbracket & \text{if } \llbracket \text{min}(x;\bar{t}1, y;\bar{t}1) \rrbracket = \llbracket x;\bar{t}1 \rrbracket \\
\llbracket y \rrbracket & \text{otherwise}
\end{cases},
\end{align*}
\]

\[
\begin{align*}
\llbracket \text{min}''(x,y) \rrbracket &= \begin{cases} \llbracket x \rrbracket & \text{if } \llbracket \text{min}(x;\bar{t}1, y;\bar{t}1) \rrbracket = \llbracket x;\bar{t}1 \rrbracket \\
\llbracket y \rrbracket & \text{otherwise}
\end{cases}.
\end{align*}
\]

Note that for each of these functions (including the ones not listed) conditional equations are used for their semantics. We will see later that this may necessitate the use of auxiliary functions for their implementation. The exception to this observation is \( \text{abs} \), but the use of \( \text{abs}^7 \) could be avoided by replacing \([\text{incr}6],[\text{incr}12]\) with \( \text{abs}(x) \rightarrow x \cdot \text{sign}(x) \).

We note for future reference that the implementation used to compute \( \text{min}(x,y) \) is to test whether \( \text{sign}(|y| - |x|) \geq 0 \); if true the term reduces to \( x \) and if not to \( y \). This is done by placing \( \text{abs}(y) - \text{abs}(x) \) in the first argument of \( \text{min}^7 \), of which the sign is computed via rules \([\text{incr}16],[\text{incr}18],[\text{incr}19],[\text{incr}21]\). This value is stored in the last argument and matched against by \([\text{incr}16],[\text{incr}21]\).
\begin{tabular}{|c|c|c|}
\hline
[int1] & 0; 0 \rightarrow 0 & [int8] & -0 \rightarrow 0 \\
[int2] & x + 0 \rightarrow x & [int9] & -(x; 0) \rightarrow (-x); 0 \\
[int3] & (x; 0) + (y; 0) \rightarrow (x + y); 0 & [int10] & -(x; 1) \rightarrow (-x); \bar{1} \\
[int4] & (x; 0) + (y; 1) \rightarrow (x + y); 1 & [int11] & -(x; 1) \rightarrow (-x); 1 \\
[int5] & (x; 0) + (y; \bar{1}) \rightarrow (x + y); \bar{1} & [int12] & x \cdot 0 \rightarrow 0 \\
[int6] & (x; 1) + (y; \bar{1}) \rightarrow (x + y); 0 & [int13] & x \cdot (y; 0) \rightarrow (x \cdot y); 0 \\
[int7] & x - y \rightarrow x + \bar{y} & [int14] & x \cdot (y; 1) \rightarrow x + ((x \cdot y); 0) \\
\hline
\end{tabular}

Table 1: A DDRS specification for $\mathbb{Z}_{\text{abs}}$ denoted by $R_{\text{int}}$, proposed in [1]

\begin{tabular}{|c|c|c|}
\hline
[incr1] & \text{sign}(x) \rightarrow \text{sign}^7(x; 0) & [incr6] & \text{abs}(x) \rightarrow \text{abs}^7(x; 0) \\
[incr2] & \text{sign}^7(0; x) \rightarrow x & [incr7] & \text{abs}^7(x; 0; y; z) \rightarrow \text{abs}^7(x; y; z) \\
[incr3] & \text{sign}^7(x; 0; y) \rightarrow \text{sign}(x; y) & [incr8] & \text{abs}^7(x; 1; y; z) \rightarrow \text{abs}^7(x; y; 0; \bar{1}) \\
[incr4] & \text{sign}^7(x; 1; y) \rightarrow \text{sign}(x; 0; \bar{1}) & [incr9] & \text{abs}^7(x; 1; y; z) \rightarrow \text{abs}^7(x; y; 0; \bar{1}) \\
[incr5] & \text{sign}^7(x; 1; y) \rightarrow \text{sign}(x; 0; \bar{1}) & [incr10] & \text{abs}^7(0; x; 0) \rightarrow 0 \\
& & [incr11] & \text{abs}^7(0; x; 0; \bar{1}) \rightarrow x \\
& & [incr12] & \text{abs}^7(0; x; 0; \bar{1}) \rightarrow -x \\
\hline
[incr13] & \text{min}(x, y) \rightarrow \text{min}^7[|y| - |x|, x, y; 0] \\
[incr14] & \text{min}^7(x; y) \rightarrow \text{min}^7(\text{abs}^7(x; 1; 1; 1), x, y, 0) \\
[incr15] & \text{min}^{7''}(x, y) \rightarrow \text{min}^7(|y; 1| - |x; 1|, x, y, 0) \\
\hline
[incr16] & \text{min}^7(x; 0; y, z, 0) \rightarrow \text{min}^7(x, y, z, 0) \\
[incr17] & \text{min}^7(x; 1; y, z, 0) \rightarrow \text{min}^7(x, y, z, 0; \bar{1}) \\
[incr18] & \text{min}^7(x; 1; y, z, 0) \rightarrow \text{min}^7(x, y, z, 0; \bar{1}) \\
\hline
[incr19] & \text{min}^7(0; x, y, 0) \rightarrow x \\
[incr20] & \text{min}^7(0; x, y, 0; \bar{1}) \rightarrow x \\
[incr21] & \text{min}^7(0; x, y, 0; \bar{1}) \rightarrow y \\
\hline
\end{tabular}

Table 2: The TRS $R_{\text{incr}}$, proposed in [1]
2.2 Computing irreducible fractions

The choice of normal forms implies that rewriting an arbitrary rational denoted \( x/y \) to normal form is effectively to divide both \( x \) and \( y \) by \( \gcd(x, y) \). To do this the signature is extended with a function \( f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z} \), with semantics \( \lceil f(x, y) \rceil = \lceil x \rceil / \gcd(\lceil x \rceil, \lceil y \rceil) \). The rules are listed in Table 3, and are based on an algorithm proposed by J. Stein and detailed in [3], for computing the \( \gcd \) of two numbers in binary representation. This algorithm is an improvement over the well-known algorithm proposed by Euclid because it incorporates division by virtue of right-shifting (i.e. removing a trailing zero from a binary digit). It is based on a set of \( \gcd \) equations which can be found in [3] and in Section 3.2.1 of the present work. In [1] integers are represented in BTN as opposed to binary and so the equations had to be modified to account for this. Unfortunately these modifications are not provided in [1] which makes it somewhat difficult to discern the workings of \( f \) and verify that the rules preserve semantics. Later on in this work we will see how the rules may be derived and the interested reader can skip ahead to Lemma 3.1.3.

A natural question to ask is what the role is of the function \( \min \). We will see that this is to ensure termination as opposed to being necessary for validity. To exemplify this we consider [7] (Table 3) and assume \( \min(x;_1 y;_1) \to ^* x;_1 1 \). This implies \( \min'(x, y) \to ^* x \) and so

\[
f(x;_1 y;_1) \to f(x - \min'(x, y), y;_1) 0 + f(\min(x;_1 y;_1), x - y) \\
\to ^* f(x - x, y;_1) 0 + f(x;_1 1, x - y) \\
\to ^* f(0, y;_1) 0 + f(x;_1 1, x - y) \\
\to 0;_1 0 + f(x;_1 1, x - y) \\
\to ^* f(x;_1 1, x - y).
\]

Conversely when \( \min(x;_1 y;_1) \to ^* y;_1 1 \) then \( \min'(x, y) \to ^* y \) and

\[
f(x;_1 y;_1) \to f(x - \min'(x, y), y;_1) 0 + f(x;_1 1, x - y) \\
\to ^* f(x - y, y;_1) 0 + f(y;_1 1, x - y).
\]

This suggests \( \min \) is used to retain the smallest (in terms of distance to 0) argument and ensure reducing rewriting steps. Regardless of this, APROVE [7] is not able to show \( R \) is terminating.

2.3 Rational arithmetic

The intended set of normal forms for the rationals consists of terms \( x/y \) where \( x \) is an integer, \( y \) is a positive integer, and \( x \) and \( y \) are coprime. The authors of [1] write that “a rational number must have a unique normal form”, as it is their goal to check equality by rewriting techniques. To accomplish this the signature is extended with \( \text{rat} \), \( f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Q} \) and [rat1] (Table 4). Taking the products with \( \text{sign}(y) \) serves to place the minus sign next to the numerator. The semantics are not provided but presumably they overlap with the usual semantics i.e. \( \lceil \text{rat}(x, y) \rceil = \lceil x \rceil / \lceil y \rceil \). The function \( \text{rat} \) is used to denote any fraction and \( / \) denotes irreducible fractions. Division by zero is avoided by introducing a sort for non-null integers although this is left somewhat implicit. To facilitate addition and multiplication on rationals \( + \mathbb{Q}, \cdot \mathbb{Q} : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q} \) are added along with rules [rat2] and [rat3].
We will see that this can be done conveniently with the use of conditional rewrite rules, which in addition helps resolve some of the aforementioned issues with developing rules for the system for addition, and in particular one that operates on both fractions and integers. For instance in the correct normal forms.

Similarly terms of the form \( \frac{x}{y} \) are not rewritten to \( x \). This means normal forms are not unique and the apparent contradiction is in fact not a contradiction. For the purposes pursued in this does not seem problematic because so long as \( \text{rat} \) is used the computations will still result in the correct normal forms.

While this method works for computing irreducible fractions via rewriting, it does not produce the aforementioned desired set of normal forms. Consider for instance the terms \( 0;1 \div (0;1):1 \) and \( (0;1):1 \div (0;1):1 \) representing the fractions \( 1/2 \) and \( 2/4 \) and respectively. By definition they are member of the term set and since no rewriting rules apply to them they are in normal form. However assuming the semantics of \( f \) are as normal we find

\[
[[0;1 \div (0;1):1]] = [[(0;1):1 \div (0;1):1]].
\]

Similarly terms of the form \( x/1 \) are not rewritten to \( x \). This means normal forms are not unique and the apparent contradiction is in fact not a contradiction. For the purposes pursued in this does not seem problematic because so long as \( \text{rat} \) is used the computations will still result in the correct normal forms.

Aside from the matter of nonuniqueness of rationals we would prefer to have just one function for addition, and in particular one that operates on both fractions and integers. For instance in the system \( R_{\text{rat}} \) terms like \( (0;1) + (0;1) \div (0;1) \) are in normal form.

In the next section we will go through the same steps outlined in the previous subsections to develop rules for \( f \) (though, we shall rename it to \( 'r' \)) for other notations like unary and binary. We will see that this can be done conveniently with the use of conditional rewrite rules, which in addition helps resolve some of the aforementioned issues with \( R_{\text{rat}} \).

<table>
<thead>
<tr>
<th>[f(0, x) \rightarrow 0]</th>
<th>[f(x, 0) \rightarrow \text{sign}(x)]</th>
<th>[f(x;0, y;0) \rightarrow f(x, y)]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[f(x;0, y;1) \rightarrow f(x, y;1) \div 1]</td>
<td>[f(x;0, y;1) \rightarrow f(x, y;1) \div 0]</td>
<td>[f(x;1, 0;1) \rightarrow f(x;1, y;1) \div 0]</td>
</tr>
<tr>
<td>[f(x;1, 0;1) \rightarrow f(x;1, y;1) \div 0]</td>
<td>[f(x;1, 1;1) \rightarrow f(x - \text{min}^t(x, y), y;1) \div 0 + f(\text{min}(x;1, y;1), x - y)]</td>
<td>[f(x;1, 1;1) \rightarrow f(x + \text{min}^n(-x, y), y;1) \div 0 + f(\text{min}(x;1, (y);1), x + y)]</td>
</tr>
<tr>
<td>[f(x;1, 1;1) \rightarrow f(x;1, y;1) \div 0]</td>
<td>[f(x;1, 1;1) \rightarrow f(x;1, y;1) \div 0]</td>
<td>[f(x;1, 1;1) \rightarrow f(x;1, y;1) \div 0]</td>
</tr>
<tr>
<td>[f(x;1, 1;1) \rightarrow f(x;1, y;1) \div 0]</td>
<td>[f(x;1, 1;1) \rightarrow f(x;1, y;1) \div 0]</td>
<td>[f(x;1, 1;1) \rightarrow f(x;1, y;1) \div 0]</td>
</tr>
</tbody>
</table>

**Table 3:** The TRS \( R_t \), proposed in [1]

| \[\text{rat}(x, y) \rightarrow (\text{sign}(y) \cdot f(x, y)) / (\text{sign}(y) \cdot f(y, x))\] | \[(x / y) \div Q(z / w) \rightarrow \text{rat}((x \div w) + (z \cdot y), y \cdot w)\] | \[(x / y) \div Q(z / w) \rightarrow \text{rat}(x \div w, y \cdot w)\] |

**Table 4:** The TRS \( R_{\text{rat}} \), proposed in [1]
3 Conditional rewrite rules for rational arithmetic

In this section we review ways to define conditional term rewrite rules for rational arithmetic, following the approach proposed in [1]. A complete listing of the rules for the unary case can be found in Table 5, which can be proven to be quasi-decreasing (a property similar to termination, see [5]) up to the second section using AProVE. It is currently unknown whether the full system is quasi-decreasing. As the name \( f \) for the function that helps reducing rationals is not particularly descriptive we will rename it to \( r \), which stands for ‘reduce’. So \( r(x, y) \) may be read as ‘the reduction of \( x \) with respect to \( y \).’

Using conditional rewrite rules allows us to simplify the rules proposed in [1], and makes it so we can use just one function to denote fractions because the reduction rule applies only to those terms not in the intended set of normal forms. After establishing these rules we use them to derive similar rules for terms representing integers in binary notation that are similar to the ones listed in Table 3 (cf. \([r_{10.1}]_{[a]}\)).

3.1 Unary view

To represent integers in unary form we need only one constant, 0, and two functions \(-\) and \( S \). In general this has the effect that less rules are needed to define operations on these terms than for other representations (see [3] for examples). Furthermore it seems as though rules for the unary case can be expanded to operate on terms representing integers in a base-\( n \) representation. For this reason we will consider the unary view before turning to the binary case.

3.1.1 Integer arithmetic

In this section we will pursue the set of normal forms for the rationals defined in the introduction:

\[
N = \{0\} \cup N^+ \cup N^\div \cup N^-
\]

\[
N^+ = \{S(0)\} \cup \{S(x) \mid x \in N^+\}
\]

\[
N^\div = \{x/y \mid x \in N^+, y \in N^+ \setminus \{S(0)\}, \gcd([x],[y]) = 1\}
\]

\[
N^- = \{-x \mid x \in N^+\} \cup \{-x \mid x \in N^+\}
\]

The normal forms for integers are produced by the following set of rules (cf. [3] Table 3).

\[\begin{align*}
[Z_{u1}] & \quad -(-x) \rightarrow x & [Z_{u7}] & \quad x + -y \rightarrow -((-x) + y) \\
[Z_{u2}] & \quad -0 \rightarrow 0 & [Z_{u8}] & \quad x + 0 \rightarrow x \\
[Z_{u3}] & \quad S(-x) \rightarrow -P(x) & [Z_{u9}] & \quad x + S(y) \rightarrow S(x + y) \\
[Z_{u4}] & \quad P(-x) \rightarrow -S(x) & [Z_{u10}] & \quad x \cdot -y \rightarrow -(x \cdot y) \\
[Z_{u5}] & \quad P(0) \rightarrow -1 & [Z_{u11}] & \quad x \cdot 0 \rightarrow 0 \\
[Z_{u6}] & \quad P(S(x)) \rightarrow x & [Z_{u12}] & \quad x \cdot S(y) \rightarrow (x \cdot y) + x \\
[abv1] & \quad x - y \rightarrow x + -y
\end{align*}\]
Two types of functions are used here which may be considered abbreviations. The first type concerns symbols that are not added to the signature. Instead we say each occurrence should be replaced by its meaning, for instance in the system above 1 is not a member of the signature but should be replaced with \( S(0) \). We could add it to the signature and include the rule \( 1 \rightarrow S(0) \), but then we would not be able to use it on the left-hand side of rules (cf. \( \leq_{\text{eq,2}} \)). The other type of abbreviating function symbols is exemplified by the rule for subtraction. While such a symbol is in the signature it can always be rewritten in one step to a term that no longer includes the symbol. Later on we will see that using this shorthand simplifies the notation of some rules, but we could still simply say it is an abbreviation like 1 for which no rule is included. As it seems that termination and confluence are preserved when adding such abbreviation rules we will do so in most cases as intuitively they seem more closely related to computation as opposed to a purely syntactic change like abbreviating \( S(0) \).

It should be noted that there are other choices for our normal forms (see e.g. \[3\] for examples). We opt for this representation because it is common in the literature, its DDRS is relatively short, and as we will see some rules can be easily defined because negative numbers are preceded by an application of \(-\).

### 3.1.2 Definitions of \( \text{sign}(x) \), \( |x| \), and \( x \leq y \)

As we have seen in Section 2 the approach employed in \[1\] makes use of the functions \( \text{sign}(x) \), \( |x| \), and \( \text{min}(x,y) \) and so we will need definitions for these for our choice of normal forms. Such rules need only be defined for terms in the set of normal forms and so our task is to formulate rules for the cases \(-x, 0, S(x)\). If we consider sign, the former two can simply be defined by \( \text{sign}(-x) \rightarrow -\text{sign}(x) \), \( \text{sign}(0) \rightarrow 0 \) but the latter is not so easy. This is because the application of \( S \) could do a number of different things to the sign of \( x \) (i.e. it could keep it -1, change it to 0, change it to 1, or keep it 1). Therefore it does not seem possible to compute \( \text{sign}(S(x)) \) from \( \text{sign}(x) \) which makes it difficult to formulate a reducing rule for this case. Note that if we somehow knew \( S(x) \) was in normal form, the rule would simply be \( \text{sign}(S(x)) \rightarrow 1 \).

Related to sign is checking whether an integer is positive. In \[8\] it is claimed that there is no finite set of unconditional rules that defines such a function \( \text{pos} \) using only the familiar operators \( 0, S, P \) and \( \text{pos} \) such that \( \text{pos}(t) \rightarrow \top \iff 0 < t \). It seems likely that this is the case for \( \text{sign} \) as well, and indeed auxiliary functions are used for this purpose in \[1\].

In \[8\] a solution to the issue with defining \( \text{pos} \) via conditional rules is given. Conditional rules are a simple extension to the kind we have seen. In addition to consisting of a pair \( l \rightarrow r \) a rule may have 0 or more conditions which must be met in order for the rule to apply. We will use the normal variant of such systems (see e.g. \[5\]) which means that conditions are of the form \( t \rightarrow^* u \), where \( u \) is a ground normal form. A conditional rule with one condition is denoted as \( l \rightarrow r \leftarrow t \rightarrow^* u \) and multiple conditions are separated by commas.

The solution is provided for a slightly different set of normal forms but the approach is easily
which closely resembles the semantics of \[][\text{eq}, 1][\text{eq}, 2][\text{eq}, 3][\text{eq}, 4]\] These conditions are similar to boolean tests commonly used in programming languages. For 

\[
\text{leq} \ 0 \leq 0 \rightarrow \top
\]

\[
\text{leq} \ 0 \leq -1 \rightarrow \bot
\]

\[
\text{leq} \ 0 \leq S(x) \rightarrow \top \iff 0 \leq x \rightarrow^* \top
\]

\[
\text{leq} \ 0 \leq -S(x) \rightarrow \bot \iff 0 \leq -x \rightarrow^* \bot
\]

where \(\top\) and \(\bot\) abbreviate \(S(0)\) and 0 respectively and serve to represent \textit{true} and \textit{false} (this is more convenient than introducing them as new constants because now we need not define terms like \(\top + \bot\), and \(0 \leq _{\bot} \mathbb{Z} \rightarrow \{\top, \bot\}\). Since we would prefer to avoid the use of auxiliary functions we will switch to conditional rules for the remainder of this work, which in addition will turn out to be useful for reducing fractions.

Note that \([\text{eq}, 3][\text{eq}, 4]\) read as though we may assume its input is in normal form. In fact if a subterm \(S(x)\) of \(0 \leq S(x)\) is not in normal form the rule still works because it would be rewritten to normal form when computing the condition, which is of course sufficient. The relation to whether the term is in normal form becomes apparent when we consider the case in which the condition is not met, i.e. \(0 \leq x \rightarrow^* \bot\). In this case \([\text{eq}, 3]\) \(\leq 0\) and so the normal form of \(x\) is in \(N^\leq\). This means that whether \(x\) is in normal form or not, \(S(x) \notin N\). Likewise if for some term \(0 \leq -S(x)\) we have \(0 \leq -x \rightarrow \top\) then \(-1 \leq -S(x)\). If \(-1 = -S(x)\) we may use \([\text{eq}, 2]\) and if \(-1 < -S(x)\), \(S(x) < 1\) which implies that \(S(x) \notin N\).

Using \(0 \leq _{\bot}\) the definitions for other functions like \text{sign}, \text{abs} easily follow, see the rules below. While functions differ slightly they will serve a very similar purpose.

\[
[\text{leq}, 9] \quad x \leq y \rightarrow 0 \leq y - x
\]

\[
[\text{lt}, 10] \quad x < y \rightarrow S(x) \leq y
\]

\[
[\text{sign}, 5] \quad \text{sign}(-x) \rightarrow -\text{sign}(x)
\]

\[
[\text{sign}, 6] \quad \text{sign}(0) \rightarrow 0
\]

\[
[\text{sign}, 7] \quad \text{sign}(S(x)) \rightarrow 1 \iff 0 \leq x \rightarrow^* \top
\]

\[
[\text{abs}, 7] \quad |x| \rightarrow x \cdot \text{sign}(x)
\]

These conditions are similar to boolean tests commonly used in programming languages. For this reason I may abbreviate conditions such as \(x = y \rightarrow^* \top\) as ‘if \(x \leq y\)’ and analogously \(x \notin y\) for \(\bot\). Furthermore I may abbreviate conjunctions of conditions such as \(0 \leq x \rightarrow^* \top\), \(0 \leq y \rightarrow^* \top\), \(x \leq y \rightarrow^* \top\) in the usual way \(0 \leq x \leq y\). Lastly, in some cases there are multiple rules which may be applied to a term-shape based on specific conditions, which I may abbreviate using the common ‘definition by cases’ notation.

While we have not explicitly defined \text{min} we could do so via rules such as

\[
[\text{min}, 1] \quad \text{min}(x, y) \rightarrow x \iff |x| \leq |y|
\]

\[
[\text{min}, 2] \quad \text{min}(x, y) \rightarrow y \iff |x| \notin |y|
\]

which closely resembles the semantics of \text{min} as used in \cite{1}. Aside from this the workings of \text{min}
and ≤ are similar; compute \(|y| - |x|\) and check whether the result is equal to- or greater than 0. We will not be using these rules for \(\text{min}\) as its usefulness seems to lie in being able to control the way in which the computations are done, which we can now do via conditions on rules. For this purpose having the statement evaluate to \(\top\) or \(\bot\) is sufficient.

### 3.1.3 Computing the greatest common divisor and irreducible fractions

The rules we have seen for \(r\) can be derived using \(\text{gcd}\) equations detailed in [4]. While Contejean et al. did not implement rules for computing the \(\text{gcd}\), we will see this can easily be done and is useful for our purposes (cf. [4]). In this section we will discuss the relevant \(\text{gcd}\) properties and the corresponding rules for \(r\).

In [4] some computational algorithms for finding the \(\text{gcd}\) of two integers are discussed. They are based on a number of equations that are easily translated to a conditional term rewriting setting. For the unary case we will implement the simplest variant of Euclid’s algorithm which uses only subtraction.

The greatest common divisor of two integers \(x, y\) is defined as the largest integer that evenly divides \(x\) and \(y\). Since 0 has arbitrarily large divisors the \(\text{gcd}\) is undefined for \(0 = x = y\). In [4] Knuth notes it is convenient to set \(\text{gcd}(0, 0) = 0\), as we can then deduce the equations

\[
\text{gcd}(x, 0) = |x|, \quad \text{gcd}(-x, y) = \text{gcd}(x, y),
\]

\[
\text{gcd}(x, y) = \text{gcd}(y, x), \quad \text{gcd}(x, 0) = \text{gcd}(x, y).
\]

Using these we can formulate the following conditional rules

\(\text{gcd1}\)

\[
\text{gcd}(-x, y) \rightarrow \text{gcd}(x, y)
\]

\(\text{gcd2}\)

\[
\text{gcd}(x, -y) \rightarrow \text{gcd}(x, y)
\]

\(\text{gcd3}\)

\[
\text{gcd}(0, x) \rightarrow |x|
\]

\(\text{gcd4}\)

\[
\text{gcd}(x, 0) \rightarrow |x|
\]

\(\text{gcd5.i}\)_{i=0}^{|x|}

\[
\text{gcd}(x, y) \rightarrow \begin{cases} 
\text{gcd}(x, y-x) & \text{if } 0 < x < y \\
\text{gcd}(x-y, y) & \text{if } 0 < y < x 
\end{cases}
\]

The strategy is to convert the arguments to positive numbers and subsequently subtract the smallest of the two from the largest until one of them is zero. The conditions in \(\text{gcd5}\) thus serve ensure termination of the resulting system. Presumably this is why \(\text{min}\) is used in [4] as it essentially incorporates the condition in the unconditional rule.
In addition to the rules for \( \text{gcd} \) we can now derive the rules for \( r \) (see Lemma 3.1.3):

\[ \text{[r1]} \quad r(-x, y) \rightarrow -r(x, y) \]

\[ \text{[r2]} \quad r(x, -y) \rightarrow r(x, y) \]

\[ \text{[r3]} \quad r(0, x) \rightarrow 0 \]

\[ \text{[r4]} \quad r(x, 0) \rightarrow \text{sign}(x) \]

\[ \text{[r5.1]}_{i=0} \quad r(x, y) \rightarrow \begin{cases} r(x, y - x) & \text{if } 0 < x \leq y \\ r(x - y, y) + r(y, x - y) & \text{if } 0 < y < x \end{cases} \]

Interestingly enough, it is not necessary to use pattern matching against constructor functions in this formulation, aside from \( - \) and 0. This implies that these rules can work in conjunction with any number representation used for the integers which includes 0 and in which negative numbers are preceded by an application of \( - \), provided the other functions involved are already defined for this notation. Adding pattern matching against \( S \) reduces overlap:

\[ \text{[gcd_{5.i}]}_{i=0} \quad \text{gcd}(S(x), S(y)) \rightarrow \begin{cases} \text{gcd}(S(x), y - x) & \text{if } 0 \leq x \leq y \\ \text{gcd}(x - y, S(y)) & \text{if } 0 \leq y < x \end{cases} \]

\[ \text{[r_{5.i}]}_{i=0} \quad r(S(x), S(y)) \rightarrow \begin{cases} r(S(x), y - x) & \text{if } 0 \leq x \leq y \\ r(x - y, S(y)) + r(S(y), x - y) & \text{if } 0 \leq y < x \end{cases} \]

Furthermore this allows us to use \( 0 \leq x \leq y \) instead of \( 0 < x \leq y \).

**Lemma 3.1.1.** These rules are sound with respect to \( \llbracket r(x, y) \rrbracket \).

**Proof.**

\[ \llbracket r(-x, y) \rrbracket = \frac{[-x]}{\text{gcd}([-x], [y])} = -\frac{[x]}{\text{gcd}([x], [y])} = \llbracket -r(x, y) \rrbracket, \]

\[ \llbracket r(x, -y) \rrbracket = \frac{[x]}{\text{gcd}([x], [-y])} = \frac{[x]}{\text{gcd}([x], [y])} = \llbracket r(x, y) \rrbracket, \]

\[ \llbracket r(0, x) \rrbracket = \frac{0}{\text{gcd}(0, [x])} = \frac{0}{[x]} = \llbracket 0 \rrbracket, \]

\[ \llbracket r(x, 0) \rrbracket = \frac{[x]}{\text{gcd}([x], 0)} = \frac{[x]}{[x]} = \llbracket \text{sign}(x) \rrbracket. \]

Assuming that \( 0 < x \leq y \):

\[ \llbracket r(x, y) \rrbracket = \frac{[x]}{\text{gcd}([x], [y])} = \frac{[x]}{\text{gcd}([x], [y] - x)} = \llbracket r(x, y - x) \rrbracket. \]
Assuming that $0 < y < x$:

$$[r(x, y)] = \frac{[x]}{\gcd([x], [y])} = \frac{[x]}{\gcd([x - y], [y])}$$

$$\Rightarrow [r(x, y)] - [r(y, x)] = \frac{[x]}{\gcd([x - y], [y])} - [r(y, x)]$$

$$\Rightarrow [r(x, y)] - [r(y, x - y)] = \frac{[x]}{\gcd([x - y], [y])} - \frac{[y]}{\gcd([x - y], [y])}$$

$$= \frac{[x - y]}{\gcd([x - y], [y])}$$

$$= [r(x - y, y)]$$

$$\Rightarrow [r(x, y)] = [r(x - y, y) + r(y, x - y)].$$

A peculiar consequence of having defined $\gcd(0, 0) = 0$ is that in particular

$$0 = [r(0, 0)] = [0] / \gcd([0], [0]) = 0 / 0.$$  

As it does not seem possible to start a rewriting sequence with a term $r(x, y)$ such that $0 \not\equiv [x]$ or $0 \not\equiv [y]$ that ends in $r(0, 0)$ this peculiarity will have little impact in practise. Nevertheless this matter should be addressed in future work.

We conclude with an example rewriting sequence starting with the term $1 < \gcd(4, 6)$ (an example for $r$ is given in the next subsection). The constants below ten are used here as abbreviations for ease of reading, and some steps are omitted.

$$1 < \gcd(4, 6) \rightarrow 2 \leq \gcd(4, 6) \rightarrow 0 \leq \gcd(4, 6) - 2$$

$$\rightarrow^* 0 \leq \gcd(4 - 4 - 2) \rightarrow^* 0 \leq \gcd(4 - 2 - 2) - 2$$

$$\rightarrow^* 0 \leq \gcd(2 - 2 - 2) \rightarrow^* 0 \leq \gcd(2, 0) - 2$$

$$\rightarrow^* 0 \leq 2 - 2 \rightarrow^* 0 \leq 0$$

$$\rightarrow \top.$$

### 3.1.4 Rational arithmetic

Now we may use $r$ to define rules for reducing fractions. As mentioned there are terms in the sort of the form $x/0$ so we need a way of handling these. We will adopt the same approach as in [1] and do not spend much time worrying about these terms, but it should be noted that there are definitions that can be used which are discussed in [2]. Examples of such definitions are $x/0 = 0$ and $x/0 = a$ (which can be thought of as an error value).
We extend the signature with / : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q} along with the rules:

\[ [Q1] \quad (-x) / y \to -x / y \]
\[ [Q2] \quad x / (-y) \to -(x / y) \]
\[ [Q4] \quad x / 1 \to x \]
\[ [Q5] \quad x / y \to r(x, y) / r(y, x) \quad \text{if} \quad 1 < \gcd(x, y) \]

Rule [Q4] prevents nonuniqueness of the integers and [Q5] that of the rationals. The condition for the simplification rule ensures that it can only be applied to reducible fractions.

To be able to compute arithmetic expressions on all our normal forms (i.e., integers can also be added to rationals) we add the rules

\[ [Q6] \quad x + (y / z) \to (x \cdot z + y) / z \]
\[ [Q7] \quad x \cdot (y / z) \to (x \cdot y) / z \]
\[ [Q8] \quad x / (y / z) \to (x \cdot z) / y \]
\[ [Q9] \quad (x / y) / z \to x / (y \cdot z) \]

It appears these rules are sufficient for ground-completeness, as AProVE can show the union of section 1 and 4 of Table 5 (minus the reduction rule [Q5]) is terminating.

We conclude this section with an example rewriting sequence starting with the term 4/6. As we have seen earlier, \( 1 < \gcd(4,6) \to^{*} \top \) and so the reduction rule [Q5] applies to this term. The other condition checks have been omitted.

\[
4/6 \to r(4,6)/r(6,4) \\
\to^{*} r(4,2)/(r(2,4)+r(4,2)) \\
\to^{*} (r(2,2)+r(2,2))/(r(2,2)+r(2,2)+r(2,2)) \\
\to^{*} (r(2,0)+r(2,0))/(r(2,0)+r(2,0)+r(2,0)) \\
\to^{*} (1+1)/(1+1+1) \\
\to \quad 2/3
\]

As an aside we note that when using these rules in practise it is beneficial to employ a strategy that results in trying unconditional rules before conditional ones. This is because rules like [leq, leq] will work relatively efficiently when the argument is in normal form, but may fail the condition when it is not. In this case the term might repeatedly fail the condition check in between being rewritten to normal form, resulting in many unneeded rewriting steps.

### 3.2 Binary view

Having derived the rules for the unary view we will now turn to the binary variant. In contrast to the previous section we will not consider the rules for integer arithmetic or rational arithmetic separately, because as we have seen the latter appear to work irrespective of our particular choice of integer normal forms.

As we have seen we technically need only define the rules for \( 0 \leq \_ \) and sign (recall that \( \_ \) and binary \( \leq \) are abbreviations) and so we begin by extending the rules for the unary case to operate on binary notation. Nevertheless the use of binary allows us to divide by two (by shifting
The function $f$ is denoted as $r$ and is short for ‘reduce’.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-x \rightarrow x$</td>
<td>[Zu1]</td>
</tr>
<tr>
<td>$0 \rightarrow 0$</td>
<td>[Zu2]</td>
</tr>
<tr>
<td>$S(-x) \rightarrow -P(x)$</td>
<td>[Zu3]</td>
</tr>
<tr>
<td>$P(-x) \rightarrow -S(x)$</td>
<td>[Zu4]</td>
</tr>
<tr>
<td>$P(0) \rightarrow -1$</td>
<td>[Zu5]</td>
</tr>
<tr>
<td>$P(S(x)) \rightarrow x$</td>
<td>[Zu6]</td>
</tr>
<tr>
<td>$0 \leq -S(x) \rightarrow \bot$ if $0 \notin -x$</td>
<td>[lequ1]</td>
</tr>
<tr>
<td>$0 \leq 1 \rightarrow \bot$</td>
<td>[lequ2]</td>
</tr>
<tr>
<td>$0 \leq 0 \rightarrow \top$</td>
<td>[lequ3]</td>
</tr>
<tr>
<td>$0 \leq S(x) \rightarrow \top$ if $0 \leq x$</td>
<td>[lequ4]</td>
</tr>
<tr>
<td>$x \leq y \rightarrow 0 \leq y - x$</td>
<td>[abv2]</td>
</tr>
<tr>
<td>$x &lt; y \rightarrow S(x) \leq y$</td>
<td>[abv3]</td>
</tr>
<tr>
<td>gcd($-x, y$) $\rightarrow$ gcd($x, y$)</td>
<td>[gcd1]</td>
</tr>
<tr>
<td>gcd($x, -y$) $\rightarrow$ gcd($x, y$)</td>
<td>[gcd2]</td>
</tr>
<tr>
<td>gcd($0, x$) $\rightarrow</td>
<td>x</td>
</tr>
<tr>
<td>gcd($x, 0$) $\rightarrow</td>
<td>x</td>
</tr>
<tr>
<td>gcd($x, y$) $\rightarrow \begin{cases} \text{gcd}(x, y - x) &amp; \text{if } 0 &lt; x \leq y \ \text{gcd}(x - y, y) &amp; \text{if } 0 &lt; y &lt; x \end{cases}$</td>
<td>[gcd5, i] $</td>
</tr>
<tr>
<td>r($x, y$) $\rightarrow \begin{cases} r(x, y - x) &amp; \text{if } 0 &lt; x \leq y \ r(x - y, y) + r(y, x - y) &amp; \text{if } 0 &lt; y &lt; x \end{cases}$</td>
<td>[r5, i] $</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(-x) / y \rightarrow -(x / y)$</td>
<td>[Q1]</td>
</tr>
<tr>
<td>$x / (-y) \rightarrow -(x / y)$</td>
<td>[Q2]</td>
</tr>
<tr>
<td>$x / 1 \rightarrow x$</td>
<td>[Q4]</td>
</tr>
<tr>
<td>$x / y \rightarrow r(x, y) / r(y, x)$ if $1 &lt; \text{gcd}(x, y)$</td>
<td>[Q5]</td>
</tr>
<tr>
<td>$x + (y / z) \rightarrow (x \cdot z + y) / z$</td>
<td>[Q6]</td>
</tr>
<tr>
<td>$x \cdot (y / z) \rightarrow (x \cdot y) / z$</td>
<td>[Q7]</td>
</tr>
<tr>
<td>$x / (y / z) \rightarrow (x \cdot z) / y$</td>
<td>[Q8]</td>
</tr>
<tr>
<td>$(x / y) / z \rightarrow x / (y \cdot z)$</td>
<td>[Q9]</td>
</tr>
</tbody>
</table>

Table 5: Conditional rules for computing irreducible fractions for terms in unary view, with $1 := S(0), \top := S(0), \bot := 0$. The sections correspond with $R_{\text{int}}, R_{\text{incr}}, R_{\ell}$, and $R_{\text{rat}}$ respectively.
numbers ending in an append zero) which gives rise to different formulations for gcd and r that are specific to binary. Deriving these rules will reveal a correspondence between the rules for r that we have seen in the previous section and those used in \([1]\).

3.2.1 Definitions of sign\((x)\) and \(0 \leq x\)

We begin by defining \(0 \leq \_\) on terms representing integers in binary notation.

\[
\begin{align*}
\text{[leq_b1]} & \quad 0 \leq 0 \rightarrow \top \\
\text{[leq_b2]} & \quad 0 \leq -1 \rightarrow \bot \\
\text{[leq_b3]} & \quad 0 \leq x;_b0 \rightarrow 0 \leq x \\
\text{[leq_b4]} & \quad 0 \leq -(x;_b0) \rightarrow 0 \leq -x \\
\text{[leq_b5]} & \quad 0 \leq x;_b1 \rightarrow \top \quad \text{if} \quad 0 \leq x \\
\text{[leq_b6]} & \quad 0 \leq -(x;_b1) \rightarrow \bot \quad \text{if} \quad 0 \leq -x
\end{align*}
\]

In this case 1 abbreviates \(:_b1\). Note that the rules are very similar to the unary case. In a sense only the rule for \(:_b0\) needs to be added; the \(_b1\) case is handled by rules for unary terms (in a way this makes sense as \(x;_b1\) can be considered short for \(S(x;_b0)\)).

3.2.2 Computing the greatest common divisor

A binary representation allows us to use Stein’s algorithm \([4]\) instead of Euclid’s. This algorithm is an upgrade with respect to the one we used for unary in terms of efficiency. The idea makes use of the following equations, in addition to those listed in Section 3.1.3.

\[
gcd(x, y) = \begin{cases} 
\gcd(x/2, y), & \text{if } x \text{ is even, } y \text{ is odd,} \\
\gcd(x/2, y/2) \cdot 2, & \text{if } x, y \text{ are both even.}
\end{cases}
\]

Terms in binary notation are easily divided by two by ‘shifting’ the number one place to the right; i.e. remove a trailing zero. As evidenced by the rules for \(f\) proposed \([1]\), this algorithm can be adapted to other bases. This makes sense as each representation using an append zero function allows one to perform division by removing applications of the append zero function.
Using the equations above we can formulate the following set of rules.

\[
\begin{align*}
[gcd1] & \quad \gcd(-x, y) \rightarrow \gcd(x, y) \\
[gcd2] & \quad \gcd(x, -y) \rightarrow \gcd(x, y) \\
[gcd3] & \quad \gcd(0, x) \rightarrow |x| \\
[gcd4] & \quad \gcd(x, 0) \rightarrow |x| \\
[gcd5] & \quad \gcd(1, x) \rightarrow 1 \\
[gcd6] & \quad \gcd(x, 1) \rightarrow 1 \\
[gcd7] & \quad \gcd(x, 0, y; 0) \rightarrow \gcd(x, y; 0) \\
[gcd8] & \quad \gcd(x, 1, y; 0) \rightarrow \gcd(x, 1; y) \\
[gcd9] & \quad \gcd(x, 0, y; 1) \rightarrow \gcd(x, y; 1) \\
[gcd10.1]_{i=0} & \quad \gcd(x, 1, y; 1) \rightarrow \begin{cases} 
\gcd(x, 1, y - x) & \text{if } 0 < x \leq y \\
\gcd(x - y, y; 1) & \text{if } 0 < y < x
\end{cases}
\end{align*}
\]

While the rules do not correspond exactly to the algorithm as it is described in \[4\], they make use of the same equations and are an improvement on the rules for unary in terms of efficiency.

Note that rules \([gcd5]\) and \([gcd6]\) are technically not needed. This is because terms of the form \(\gcd(1, x)\) and \(\gcd(x, 1)\) match \([gcd10.1]_{i=0}\) if we replace ‘0 . . .’ with ‘0 ≤ . . .’ (in fact, this condition is formulated this way to prevent such rewrite sequences). In practise it seems beneficial to include the rules listed above because it shortens rewriting sequences.

### 3.2.3 Computing irreducible fractions

Using the added rules for \(\gcd\) we can derive the alternatives for \(r\). Similarly as for \(\gcd\) it seems to be beneficial in practise to include rules for the case where one of the arguments of \(r\) equals 1.

\[
\begin{align*}
[r_b1] & \quad r(-x, y) \rightarrow -r(x, y) \\
[r_b2] & \quad r(x, -y) \rightarrow r(x, y) \\
[r_b3] & \quad r(0, x) \rightarrow 0 \\
[r_b4] & \quad r(x, 0) \rightarrow \text{sign}(x) \\
[r_b5] & \quad r(1, x) \rightarrow 1 \\
[r_b6] & \quad r(x, 1) \rightarrow x \\
[r_b7] & \quad r(x; 0, y; 0) \rightarrow r(x, y) \\
[r_b8] & \quad r(x; 0, y; 1) \rightarrow r(x, y; 1; 0) \\
[r_b9] & \quad r(x; 1, y; 0) \rightarrow r(x; 1, y) \\
[r_b10.1]_{i=0} & \quad r(x; 1, y; 1) \rightarrow \begin{cases} 
\frac{r(x; 1, y - x)}{\gcd(x; 1, y - x)} & \text{if } 0 < x \leq y \\
\frac{r(x - y, y; 1)}{\gcd(x - y, y; 1)} + r(y; 1, x - y) & \text{if } 0 < y < x
\end{cases}
\end{align*}
\]

**Lemma 3.2.1.** These rules are sound with respect to \([r(x, y)]\).
Assuming that

\[ [r(1, x)] = \frac{1}{\gcd(1, [x])} = \frac{1}{1} = [1], \]
\[ [r(x, 1)] = \frac{[x]}{\gcd([x], 1)} = \frac{[x]}{1} = [x], \]
\[ [r(x_10, 0, y_10, 0)] = \frac{[x_10]}{\gcd([x_10], [y_10])} = \frac{[x]}{\gcd([x], [y])} = 2 = [r(y, y)], \]
\[ [r(x_11, 0, y_11, 1)] = \frac{[x_11]}{\gcd([x_11], [y_11])} = \frac{[x]}{\gcd([x], [y])} = [r(x, y_11)], \]
\[ [r(x_11, 1, y_10)] = \frac{[x_11]}{\gcd([x_11], [y_10])} = \frac{[x]}{\gcd([x], [y])} = [r(x_11, 1, y_1)]. \]

Proof. Rules that match the unary case are omitted.

Assuming that \(0 < x \leq y\):

\[ [r(x_11, 1, y_1)] = [r(x_11, 1, y_1 - x_11)] = [r(x_11, 1, y - x)] = [r(x_11, y - x)]. \]

Assuming that \(0 < y < x\):

\[ [r(x_11, 1, y_1)] = [r(x_11 - y_1, y_1) + r(y_1, x_11 - y_1)] = [r(x - y, y_1) + r(y_1, x - y)] = [r(x - y, y_1) + r(y_1, x - y)]. \]

We conclude by computing \(r(165, 110)\) (in binary) as an example.

\[
\begin{align*}
10100101, 110111 & \rightarrow 10100101, 110111 \\
& \rightarrow 10100101 - 110111, 110111 \rightarrow 10100101 - 110111 \\
& \rightarrow 10100101, 110111 \rightarrow 110111, 110111 \\
& \rightarrow 110111, 110111 - 110111 \rightarrow 110111, 110111 \\
& \rightarrow 110111, 110111 - 110111 \rightarrow 110111, 110111 \\
& \rightarrow 110111, 110111 - 110111 \rightarrow 110111, 110111 \\
& \rightarrow 1. \end{align*}

This is in line with our expectations as \(165 / \gcd(165, 110) = 165 / 55 = 3\).

4 Conclusion and digression

The purpose of this research was to continue the work towards a DDRS for rational arithmetic, that is, produce a TRS for reducing rationals to normal form and implement common arithmetic operations on these terms. Concretely the goal was twofold; firstly to resolve an apparent contradiction between previous results regarding the use of rewriting rules to produce normal forms for rational numbers, and secondly to review in what way the work proposed in [1] may
aid in constructing a DDRS for rational arithmetic. The first topic was clarified; the TRS $R_{rat}$ does not produce the intended set of normal forms and so the results are not contradictory. With respect to the second goal the approach proposed in [1] was then modified and rephrased using conditional rewrite rules. This resulted in being able to further narrow the set of normal forms of rationals.

At some point during the work a switch was made to CTRSs in part due to the conjecture that an unconditional TRS for this purpose does not exist. However this has not been proven and it is interesting to consider whether we could accomplish this goal by using a different sort of normal form. We discuss two such options.

As reducing fractions closely relates to the concept of prime number decompositions (PNDs), and all natural numbers are uniquely determined by their PND one might wonder whether we can use a representation based on this fact. We may have to use rewriting modulo AC of multiplication but this does not seem unreasonable as a PND is unique modulo AC. The downside to this is that in a sense a different representation for primes would be required. If we use a regular notation like $S$ for the primes (e.g., 4 is denoted by $S(S(0)) \cdot S(S(0))$ and 3 by $S(S(S(0))))$ then we need rules that do not simplify primes but do simplify composite numbers like $S(S(S(S(0))))$, which seems very difficult if not impossible to do. One idea would be to use a successor function over the primes instead, so e.g. $[N(0)] = 2$, $[N(N(0))] = 3$, etc. Each product of such terms is then a unique normal form of an integer. While this would make it easy to reduce fractions via a rule such as $(x \cdot z)/(y \cdot z) \rightarrow x/y$, the semantics of $N$ are unknown which makes it difficult to define addition and other operations.

A second possibility might be found in an algorithm that produces coprime pairs [9]. The scheme is exhaustive, non-redundant, and produces no invalid members which are characteristics needed for constructor functions. It may be interesting to review whether these branches may be used as append functions, over which addition and multiplication can be defined.

Some other topics that may deserve future work are as follows. The system presented in Table 5 can not be shown to be quasi-decreasing by AProVE. Since intuitively it seems likely that this is the case it would seem that AProVE has difficulty ‘understanding’ the consequences of the conditions. Furthermore it is unknown whether the system is ground-confluent.

The present work was devoted largely to translating the approach proposed in [1], and for this purpose conditional rules were used. Little time was spent checking whether the use of conditional rules allow us to forego the use of functions like $r$. An example of this is the absence of the rule $x:40/y:40 \rightarrow x/y$ in the binary case.

Lastly, no time was spent on giving an intuitive explanation of the workings of $r$. Since one of the intended applications of DDRSs is didactics, this should be rectified.

References

A  Alternative formulations

There are a number of alternative ways to define the functions we discussed in Section 3 which we briefly consider here.

A variation on these rules is possible that makes use of the equivalence \( x \leq y \Leftrightarrow -y \leq -x \) and the function \(_\leq 0 : \mathbb{Z} \to \{0, 1\}\).

\[
\begin{align*}
0 \leq -x & \rightarrow x \leq 0 \\
-x & \leq 0 \rightarrow 0 \leq x \\
0 & \leq 0 \rightarrow \top \\
0 & \leq 0 \rightarrow \top \\
1 & \leq 0 \rightarrow \bot \\
0 \leq S(x) & \rightarrow \top \Leftrightarrow 0 \leq x \rightarrow \top \\
S(x) & \leq 0 \rightarrow \bot \Leftrightarrow x \leq 0 \rightarrow \bot
\end{align*}
\]
Note the double occurrence of $0 \leq 0 \rightarrow \bot$; this is because \( \_ \leq 0 \) and \( 0 \leq \_ \) are different functions.

Since we may choose any two distinct terms in the set of normal forms as our choice of \( \top \) and \( \bot \) we could instead let \( \top := S(0) \) and \( \bot := -S(0) \). Then \(|x|\) is also definable by the rule

\[
|x| \rightarrow x \cdot (0 \leq x)
\]

This would in turn give us a simple shorthand for negation

\[
\neg x \rightarrow -x
\]

which lets us define < in a way that generalizes to operate on rationals

\[
x < y \rightarrow \neg (y \leq x)
\]

but this is somewhat unintuitive and it is common practise to set \( \bot := 0 \). For \(|x|\) there are also more alternatives;

\[
|x| \rightarrow \begin{cases} 
    x & \text{if } 0 \leq x \\
    -x & \text{if } x < 0
\end{cases}
\]

Another alternative which is perhaps useful for proving termination or confluence because it makes it clear that \(|\_\|\) does not return a value that is more complex than its argument and reduces the overlap in the patterns:

\[
|\neg x| \rightarrow |x|
\]

\[
|0| \rightarrow 0
\]

\[
|S(x)| \rightarrow S(x) \quad \text{if } 0 \leq S(x)
\]

An alternative formulation for the first rule of sign is

\[
\text{sign}(-x) \rightarrow -1 \quad \text{if } -x < 0
\]

Both the sign and \( \leq \) function are difficult to define without using conditional rules because pattern matching alone does not give enough information. Conditional rules solve this issue essentially by forcing the input to be in normal form. Another boolean function that seems to suffer from this problem is semantic equality. To distinguish between this function and regular equality we write \( \_ \equiv \_ : \mathbb{Z} \times \mathbb{Z} \rightarrow \{\top, \bot\} \). While a rule such as \( S(x) \equiv S(y) \rightarrow x = y \) is clearly valid and reduces complexity, it does not seem possible to find a reducing rule for the case \( -S(x) \equiv S(y) \). A way to define conditional rules such that \( x \equiv y \rightarrow \top \iff \lfloor x \rfloor = \lfloor y \rfloor \) is listed below.

\[
0 \equiv -x \rightarrow 0 \equiv x
\]

\[
0 \equiv 0 \rightarrow \top
\]

\[
0 \equiv S(x) \rightarrow \bot \quad \text{if } 0 \leq x
\]

\[
x \equiv y \rightarrow 0 \equiv y - x
\]
Like for the other rules with conditions, one might wonder what happens to a term $0 \equiv S(x)$ when $0 \leq x \rightarrow \perp$. In this case $\|x\| < 0$ and so the normal form of $x$ is a member of $N^-$. Hence if $x \in N$ then $S(x) \notin N$, and likewise if $x \notin N$ then $S(x) \notin N$. This means there are other rewrite rules that can be applied to $S(x)$ in context, after which rules for $\equiv$ do apply.

Now we could use $\equiv$ to reformulate the condition of the simplification rule as follows.

\[
x \perp y \rightarrow \gcd(x, y) \equiv 1
\]

\[
x \slashbot y \rightarrow r(x, y) \slashbot r(y, x) \quad \text{if} \quad 0 \neq x, x \notin y
\]

Additionally we have a different way of defining negation that shows more likeness to implementations in popular programming languages (like ‘not’ in python):

\[
\text{not } x \rightarrow 0 \equiv x
\]