1 Introduction

This document explains how the rational numbers can be implemented and proofed using the proof-assistant MINLOG. There will be an overview of integers to understand later references to characteristics and how we handle operations between different algebras in MINLOG. Following that, there will be an Overview on the implementation and some proofs of noteworthy characteristics of rationale numbers followed by dyadic numbers. We will look at these definitons and theorems in depth:

```
Integer
  Definition of Algebra and Programm constraints
  Overloading
Rational numbers
  Definition of algebra and program constraints
  Point wise equality of unary Minus
  Compatibility of equivalence
  Upper bound for rational numbers
Dyadic numbers
  Definitions of algebra and program constraints
  Totality of RatTwoCancel
  DyToExpPosLog
  Other Properties of Dyadic Numbers
```

2 Integer

We will look into the definition of integers to understand how they are represented in MINLOG to use them in the definition of rational numbers. Although we will address how the problem of overloading the operations is solved, including the way MINLOG handles operations between elements of different algebras.

2.1 Definition of Algebra and Programm constraints

The algebra of integers is defined using the implementation of positive numbers.

Definition 2.1.

```
(add-algs "int"
   '("IntPos" "pos=>int")
   '("IntZero" "int")
   '("IntNeg" "pos=>int")
)
```

And those operators are defined in the algebra.

Definition 2.2.

- Unary operators:
  IntS, IntP, IntUMinus($\sim$), IntAbs(abs), IntSg(sg)
- Binary operators:
  IntPlus($+$), IntMinus($-$), IntTimes($*$) IntExp($**$), IntMax(max), IntMin(min)
- Binary relations:
  IntLt($<$), IntLe($<=$)

Those definitions are rather simple, but we will use them later to define the algebra, operations and relations for rational numbers.

2.2 Overloading

Those definitions lead us to a problem, we will have with every new algebra, we will define to expand the given algebra. The problem is, that we want to have the same symbols for the operators even if the computation of the operator works different in some algebras. For example the addition for integers and rational numbers.
To solve this problem we are doing two things. First of all we overload the symbol with the new computation rule for the new context. Now we only need to solve the problem which context to use if the two numbers aren't in the same context.
We will compute the lowest common algebra they both share and use the operator in this context. We could although use the highest available one, but that would lead to slower computation, because the computation of some operators are less complex for simpler algebras.
To convert an number to the right context, we define replacement rules. For example the replacement rule from natural numbers to integers:

Example 2.3.

```
(add-item-to-algebra-edge-to-embed-term-alist
  "nat" "int"
  (let ((var (make-var (make-alg "nat") -1 t-deg-one "")))
    (make-term-in-abst-form
      var (make-term-in-app-form
        (make-term-in-const-form
          (constr-name-to-constr "NatToInt"))
        (make-term-in-var-form var)))))

(add-program-constant "NatToInt" (py "nat=>int"))

(add-computation-rules
  "NatToInt Zero" "IntZero"
  "NatToInt(Succ n)" "IntS(NatToInt n)")
```

This is also useful for reading terms that have been parsed by `pp` (pretty print), which converts numbers to the lowest degree algebra, because terms in lower degree algebras are easier readable for humans. For example "1#1" in the algebra "rat" is the same as "1" in the algebra "nat". After reading such a term the elements automatically get transferred to the needed algebra.

3 Rational numbers

Keeping said information in mind, we will proceed to the rational numbers, beginning with the definition.

3.1 Definition of algebra and program constraints

The "rat" algebra is defined with a dependency to the integers and all integers are embedded into this algebra. We define it as:

Definition 3.1.

```
(add-algs "rat"
  '("RatConstr" "int=>pos=>rat")
)
```

We want to be able to write down numbers of this algebra. Therefor we need an symbol to represent the separator of the fraction. The most common used token for that, the "/" is already used for division and we want to use a character we don't necessary need later. So we will use # for representing the fraction. We define the token as following.

Definition 3.2.

```
(add-token
  "#"
  'pair-op (lambda (x y)
    (mk-term-in-app-form
      (make-term-in-const-form (constr-name-to-constr "RatConstr"))
      x y)))
```

We also need the common operations for rational numbers. We define the following ones:

Definition 3.3.

- Unary Operators:
  RatUMinus($\sim$), RatUDiv, RatAbs(abs), RatHalf

- Binary Operators:
  RatPlus($+$), RatMinus($-$), RatTimes($*$), RatDiv($/$), RatExp($**$), RatMax(max), RatMin(min)

- Binary Relations:
  RatEqv(==), RatLt(<), RatLe(<=)

We overloaded the already defined operators, changed the computation rules of some of them and added new operations. We will have a closer look on interesting definitions of some of these operators.

The "+" in the rational numbers is overloading the token, but using an other computation rule, then the "+" operator for integers. We use the well known rules for addition of rational numbers.

Definition 3.4.

```
(add-program-constant "RatPlus" (py "rat=>rat=>rat"))
(add-token-and-type-to-name "+" (py "rat") "RatPlus")

(add-computation-rules "(k#p)+(j#q)" "k*q+j*p#p*q")
```

We also added the equivalence relation to the operations, because the point wise equality doesn't give the equality for rational numbers. For example "2#2 = 1#1" is false, but they are equivalent. We add the equivalent relation with the following definition.

Definition 3.5.

```
(add-program-constant "RatEqv" (py "rat=>rat=>boole"))
(add-token-and-type-to-name "==" (py "rat") "RatEqv")

(add-computation-rules "(k#p)==(j#q)" "k*q=j*p")
```

We will have to show, that this definition is compatible with the point wise equality, because otherwise it would not work on the natural numbers and integers. We will look at this proof later.

3.2 Point wise equality of unary Minus

The first proof we will take a closer look at is the point wise quality of the unary Minus. In MINLOG we define special Operators for the unary version of the subtraction and the division because in minimal logic an operator can only have an arity of one or two, not both at the same time.

Proof explained in natural Language
Let $a, b \in \mathbb{R}$ we want to proof that $a = b \Rightarrow -a = -b$. Because "=" is point wise, we can reduce our problem to the point wise equality of the numerator in the sub algebra of integers. Therefor we can look at the 3 cases numerator is 0, numerator is positive or numerator is negative. All 3 cases follow directly from the definition of the unary minus in the integers.

Proof in MINLOG

```
;; RatUMinusEqP
(set-goal "allnc a^,b^(EqPRat a^ b^ -> EqPRat(~a^)(~b^))")
(assume "a^" "b^" "EqPab")
(elim "EqPab")
(assume "k^1" "k^2" "EqPIntk1k2" "p^1" "p^2" "EqPp1p2")
(elim "EqPIntk1k2")
;; 5-7
(assume "p^3" "p^4" "EqPp3p4")
(ng #t)
(use "EqPRatRatConstr")
(use "EqPIntIntNeg")
(use "EqPp3p4")
(use "EqPp1p2")
;; 6
(use "EqPRatRatConstr")
(use "EqPIntIntZero")
(use "EqPp1p2")
;; 7
(assume "p^3" "p^4" "EqPp3p4")
(ng #t)
(use "EqPRatRatConstr")
(use "EqPIntIntPos")
(use "EqPp3p4")
(use "EqPp1p2")
;; Proof finished.
(save "RatUMinusEqP")
;; (cdp)
```

3.3 Compatibility of equivalence

As already noted after the definition of the rational equivalence, we need to show the compatibility with the point wise equality. This is a very important proof for showing that the Integer and the natural numbers are sub algebras of the rational numbers. Therefor we will look at this proof.

Proof explained in natural Language
Let $a, b, c, d \in \mathbb{R}$. We want to show $a == b \Rightarrow c == d \Rightarrow (a == c) = (b == d)$. We look at the cases for $a == c$:

Case 1 $a == c$ is true: We can use transitivity with c on $a == b$ and get $a == c$ true and $c == b$ true. Now we use transitivity with d on $c == b$ and get $c == d$ true and $d == b$ true. With symmetry we get $b == d$ true.

Case 2 $a == c$ is false: We can use transitivity with c on $a == b$ and get $a == c$ false and $c == b$ false. Now we use transitivity with d on $c == b$ and get $c == d$ true and $d == b$ false. With symmetry we get $b == d$ false.

Therefor our goal is proven.

Proof in MINLOG

```
(set-goal "all a,b,c,d(a==b -> c==d -> (a==c)=(b==d))")
(assume "a" "b" "c" "d" "a=b" "c=d")
(cases (pt "a==c"))
(assume "a=c")
(assert "b==d")
(use "RatEqvTrans" (pt "c"))
(use "RatEqvTrans" (pt "a"))
(use "RatEqvSym")
(use "a=b")
(use "a=c")
(use "c=d")
(assume "b=d")
(simp "b=d")
(use "Truth")
(assume "a=c -> F")
(assert "b==d -> F")
(assume "b=d")
(use "a=c -> F")
(use "RatEqvTrans" (pt "b"))
(use "a=b")
(use "RatEqvTrans" (pt "d"))
(use "b=d")
(use "RatEqvSym")
(use "c=d")
(assume "b=d -> F")
(simp "b=d -> F")
(use "Truth")
;; Proof finished.
(save "RatEqvCompat")
```

3.4 Upper bound for rational numbers

We want to show that every rational number q has an n so that 2^n is an upper Bound of q. This proof will be very useful for later proofs. Also this proof is interesting to handle in a constructive way. The proof consists of two proofs. In the first part we will show that there exists this kind of n. This part is not really constructive and the second proof doesn't rely on this. But we need the first proof the use the ability of MINLOG to extract a program from a proof and use this program to have a computation rule based on the first proof. This

computation rule is sufficient to Proof this property in a constructive manner.

Proof explained in natural Language
Let $p, q \in \mathbb{Z}$. We want to show, that $\exists n \in \mathbb{N} : \frac{p}{q} \leq 2^n$. We choose n as $\log p + 1 - \log q$. Now we have to show, that $\frac{p}{q} \leq 2^{\log p + 1 - \log q}$. This is true as seen by reorganizing the right side:

$$\begin{aligned}
\frac{p}{q} &\leq 2^{\log p + 1 - \log q} \\
\frac{p}{q} &\leq \frac{2^{\log p + 1}}{2^{\log q}} \\
\frac{p}{q} &\leq \frac{p \cdot 2}{q} \\
\frac{p}{q} &\leq 2 \cdot \frac{p}{q} \\
1 &\leq 2
\end{aligned}$$

Before we have a look at the MINLOG code, we need to add, that there are the commands animate and deanimate. Animate adds a computation rule, which is an extracted program from a proof. Deanimate removes the computation rule. We should always deanimate if we don't need the program, because MINLOG checks every computation rule for normalisation or simplification and we don't want to have unnecessary calculations every time we use ng or simp. If you animate a proof, MINLOG will create a command with an c in front of the proof name so we can use the program.

Proof in MINLOG

```
(set-goal "all p,q exl n (p#q)<=2**n")
(assume "p" "q")
(intro 0 (pt "Succ(PosLog p)--PosLog q"))
(use "RatLeTrans" (pt "2**Succ(PosLog p)#2**PosLog q"))
(use "RatLePosExpTwo")
(use "RatLePosExpTwoMinus")
;; Proof finished.
;; (cp)
(save "RatLeBound")

(animate "RatLeBound")

;; RatLeBoundExFree
(set-goal "all p,q (p#q)<=2**cRatLeBound p q")
(assume "p" "q")
(use "RatLeTrans" (pt "2**Succ(PosLog p)#2**PosLog q"))
(use "RatLePosExpTwo")
```

```
(use "RatLeTrans" (pt "(2**(Succ(PosLog p)--PosLog q)#1)"))
(use "RatLePosExpTwoMinus")
(use "Truth")
;; Proof finished.
;; (cp)
(save "RatLeBoundExFree")

(deanimate "RatLeBound")
```

4 Dyadic numbers

We will implement the dyadic numbers. This is the base two representation of the rational numbers. For some proofs it will be way easier to implement them with the dyadic representation. We will use the equivalence of the rational numbers and implement program constraints to check if the numbers are dyadic. We also need to implement operations and proof that their result is also a dyadic number.

4.1 Definitions of algebra and program constraints

We use the algebra of rational numbers and add some program constraints to operate with and on the dyadic numbers:

Definition 4.1.

- Unary Operators:
  RatNum, RatDen, IsTwoExp, Dy, RatCancelTwo, DyNf

- Binary Operators:
  DyPlus($+=$), DyTimes($*=$)

We will have a closer look at more interesting operators, beginning with the boolean operator "IsTwoExp". This operator returns true, if a number has the form 2^n for a $n \in \mathbb{N}$. The computation rule uses "SZero" and "SOne" they yield an representation of numbers equivalent to the binary representation. The implementation of those two program constraint have already been part of the demonstration of the implementation of integer, therefor I will not discus this. The computation rule of "IsTwoExp" essentially reduces the calculation of its value to this implementation. Analogously to the binary numbers, it checks if the only "1" in the representation is the leading one.

Definition 4.2.

```
(add-program-constant "IsTwoExp" (py "pos=>boole"))

(add-computation-rules
  "IsTwoExp 1" "True"
  "IsTwoExp(SZero p)" "IsTwoExp p"
  "IsTwoExp(SOne p)" "False")
```

We have the dyadic normal form, the dyadic form with nominator and denominator don't share 2 as common divider. That means the fraction can not be reduced by 2. The defined times for rational numbers doesn't always return a number in the dyadic normal form, therefor we implement another Multiplication, that holds this property. We also implement a program constraint, that returns the fraction in dyadic normal form.

Definition 4.3.

- DyTimes:

  ```
  (add-program-constant "DyTimes" (py "rat=>rat=>rat"))
  (add-token-and-type-to-name "*=" (py "rat") "DyTimes")

  (add-computation-rules "a*=b" "RatCancelTwo(a*b)")
  ```

- RatCancelTwo:

  ```
  (add-program-constant "RatCancelTwo" (py "rat=>rat"))

  (add-computation-rules
    "RatCancelTwo k" "k#1"
    "RatCancelTwo(0#q)" "0#1"
    "RatCancelTwo(1#SZero q)" "1#SZero q"
    "RatCancelTwo(SZero p#SZero q)" "RatCancelTwo(p\#q)"
    "RatCancelTwo(SOne p#SZero q)" "SOne p#SZero q"
    "RatCancelTwo(IntN 1#SZero q)" "IntN 1#SZero q"
    "RatCancelTwo(IntN(SZero p)#SZero q)" "RatCancelTwo(IntN p#q)"
    "RatCancelTwo(IntN(SOne p)#SZero q)" "IntN(SOne p)#SZero q"
    "RatCancelTwo(IntP p#SOne q)" "IntP p\#SOne q"
    "RatCancelTwo(IntN p#SOne q)" "IntN p\#SOne q"
  )
  ```

4.2 Totality of RatTwoCancel

We want to show totality of the program constraint RatCancelTwo, because it essential for every later proof with dyadic numbers.

Proof explained in natural Language
We want to show that $RatCancelTwo(a)$ is total if a is total. We split our proof

into following three cases:
Case 1: $a = \frac{0}{k}$ for an arbitrary $k \in \mathbb{N}$. Clearly this case holds totality.
Case 2: $a = \frac{p}{k}$ for arbitrary $p, k \in \mathbb{N}$. With an structural induction over the dyadic number p, regarding 1, $SOnep$ and $SZerop$, we can show that this case holds totality.
Case 3: $a = \frac{-p}{k}$ for arbitrary $p, k \in \mathbb{N}$. To show the totality we also need an induction over the structure of dyadic numbers with p.

Proof in MINLOG

```
(set-totality-goal "RatCancelTwo")
(fold-alltotal)
(cases)
  (ind)
    (ind)
      (cases)
        (use "TotalVar")

        (assume "q")
        (use "TotalVar")

        (assume "q")
        (use "TotalVar")
      (assume "p" "IHp")
      (cases)
        (use "TotalVar")

        (assume "q")
        (use "IHp")

        (assume "q")
        (use "TotalVar")
      (assume "p" "IHp")
      (cases)
        (use "TotalVar")

        (assume "q")
        (use "TotalVar")

        (assume "q")
        (use "TotalVar")
    (ng #t)
    (assume "p")
    (use "TotalVar")
    (ind)
      (cases)
```

```
      (use "TotalVar")

      (assume "q")
      (use "TotalVar")

      (assume "q")
      (use "TotalVar")
    (assume "p" "IHp")
    (cases)
      (use "TotalVar")

      (assume "q")
      (use "IHp")

      (assume "q")
      (use "TotalVar")
    (assume "p" "IHp")
    (cases)
      (use "TotalVar")

      (assume "q")
      (use "TotalVar")

      (assume "q")
      (use "TotalVar")
(save-totality)
```

After proofing the totality of dyadic numbers, we will have a closer look at some useful properties.

4.3 DyToExpPosLog

Clearly the denominator of an dyadic number is a power of two. We will proof this to use this useful relation between the logarithm and the denominator of an dyadic number.

Proof explained in natural Language
We want to proof that for all dyadic numbers $\frac{k}{p}$, with $p, k \in \mathbb{N}$ arbitrary numbers, $p = 2^{Log_2 p}$.
We only have to look at p with the following form $p = 2^n$, because that is the form of the denominator of a dyadic number. We use induction over n.
Start of induction $n = 1$: $2^1 = 2 = 2^1$.
Induction hypothesis: We assume that the equality holds for an arbitrary n.

Step of induction $n \mapsto n+1$: $2^{n+1} = 2 \cdot 2^n \overset{IH}{=} 2 \cdot 2^{Log_2(2^n)} = 2 \cdot 2^n = 2^{n+1}$

Proof in MINLOG

```
(set-goal "all k,p(Dy(k#p) -> p=2**PosLog p)")
(assume "k")
(ng)
;; ?^3:all p(IsTwoExp p -> p=2**PosLog p)
(ind)
;; 4-6
(assume "Useless")
(use "Truth")
;; 5
(assume "p" "IH")
(ng)
(use "IH")
;; 6
(assume "p" "IH")
(ng)
(assume "Absurd")
(use "Absurd")

(save "DyToExpPosLog")
```

I won't show the proof of the last two properties, because they don't yield any interesting patterns in the proof. But I want to mention them anyway, because the properties are quite interesting.

4.4 Other properties of dyadic numbers

As mentioned early we defined a new multiplication that holds totality regarding multiplication of dyadic numbers. Actually it even holds the totality for numbers in dyadic normal form. This property is proven in the proof "DyNfDyTimes" in the library.

The other property is, that the dyadic normal form of rational numbers is unambiguous. This property is proven in the proof "RatCancelTwoCompat" in the library.