Karlsruher Institut für Technologie

Masterarbeit

Tools and Algorithms on Real Numbers for Signal Machines

Jakob Dahlum

Abgabedatum: 30.04.2018

Betreuer: Dr. Thomas Worsch

Institut für Theoretische Informatik
Fakultät für Informatik
Karlsruher Institut für Technologie

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, den 30.04.2018

Abstract

Signal machines are a generalization of cellular automata which, instead of using discrete cells, perform computations on the real number line by sending dimensionless signals with varying velocities along the line. Whenever multiple signals collide, we can delete them, change their velocity or create new signals. We present algorithms using this computational model, also called abstract geometrical computation, to perform a variety of tasks. We manipulate and sort intervals, execute a range of arithmetic operators, outline various number representations as well as techniques to switch between them and perform generalizations from the discrete to the continuous realm in other areas like formal languages.

Zusammenfassung

Signalmaschinen sind eine Verallgemeinerung von Zellularautomaten, welche Berechnungen auf dem reellen Zahlenstrahl durchführen, indem sie dimensionslose Signale mit unterschiedlichen Geschwindigkeiten entlang des Strahls senden, anstatt diskrete Zellen zu verwenden. Wenn mehrere Signale kollidieren, können sie entfernt, ihre Geschwindigkeit verändert oder neue Signale erzeugt werden. Wir präsentieren Algorithmen, die dieses Berechnungsmodell, auch genannt abstrakte geometrische Berechnung, für eine Vielzahl von Problemen benutzen. Wir manipulieren und sortieren Intervalle, führen eine Reihe arithmetischer Operatoren aus, entwerfen verschiedene Zahlendarstellungen, sowie Techniken, um zwischen ihnen zu wechseln und führen Verallgemeinerungen vom Diskreten in das Kontinuierliche in anderen Bereichen wie formalen Sprachen durch.

Acknowledgements

I would like to thank Dr. Thomas Worsch for the opportunity to work on such a fresh topic which at times felt more like solving entertaining puzzles than actual work. I appreciate his patience and his input that raised all sorts of new ideas along the way, as well as his useful ideas that helped solving some of the more challenging problems.

Contents

1 Introduction **8**
1.1 Motivation 8
1.2 Related Work 9
1.3 Organization of this Thesis 9

2 Preliminaries **10**
2.1 Cellular Automata 10
2.2 Signal Machines 12

3 Basic Tools **17**

4 Simulation of Cellular Automata **26**

5 Arithmetic on Real Numbers **30**
5.1 Addition and Subtraction 30
5.2 Multiplication and Division 34
5.3 Complex Arithmetic Operators 57

6 Generalized Formal Languages **69**

7 Numbers in Generalized Bases **76**

8 Sorting Intervalls **88**

9 Conclusion **97**
9.1 Future Work 97

List of Figures

1 Moving Intervals . . . 18
2 Stretch and Compress . . . 18
3 Binary Counter . . . 19
4 Storing Signals . . . 21
5 From Interval to Marking (1) . . . 22
6 From Interval to Marking (2) . . . 22
7 Generator . . . 23
8 Binary Representation . . . 25
9 Simulation of a one-dimensional cellular automaton . . . 27
10 Simulation of a one-dimensional cellular automaton in constant space . . . 29
11 Singularity during a simulation in constant space . . . 29
12 Addition . . . 31
13 Flipping the sign . . . 32
14 Subtraction . . . 33
15 Gadgets . . . 34
16 Multiplicative Inverse (1) . . . 37
17 Multiplicative Inverse (2) . . . 38
18 Multiplicative Inverse (3) . . . 39
19 Singularity Problem . . . 40
20 Multiplication (1) . . . 43
21 Multiplication (2) . . . 44
22 Improved Multiplication (1) . . . 46
23 Improved Multiplication (2) . . . 46
24 Improved Multiplication (3) . . . 47
25 Improved Multiplication (4) . . . 49
26 Improved Multiplication (5) . . . 49
27 Improved Multiplication (6) . . . 50
28 Division (1) . . . 53
29 Division (2) . . . 54
30 Improved Division . . . 56
31 Integer Exponentiation . . . 58
32 Factor Decomposition . . . 60
33 Finding Common Factors . . . 61
34 Multiplication of Multiple Factors . . . 63
35 Square Root . . . 65
36 Logarithm . . . 68
37 Mirror Image . . . 70
38 Detecting Length-independent Palindromes . . . 72
39 Detecting Length-dependent Palindromes . . . 73
40 Word Exponentiation . . . 75
41 Interval Number . . . 77
42 Canonical Interval Numbers: Removing and Inserting Boundaries . . . 77
43 Canonicalization First Phase . . . 79
44 Canonicalization Second Phase . . . 80

45 Canonicalization First Phase Iteration (1) 81
46 Canonicalization First Phase Iteration (2) 82
47 Compressing Canonical Numbers Further 86
48 Interval Comparison 89
49 Comparison Conflict (1) 91
50 Comparison Conflict (2) 91
51 Comparison Conflict (3) 92
52 Comparison Conflict (4) 92
53 Bilateral Interval Sort (1) 93
54 Bilateral Interval Sort (2) 94
55 Hourglass Compression 96

1 Introduction

1.1 Motivation

In theoretical computer science cellular automata are a popular model for parallel computation. In contrary to Turing machines which consist of one operating unit, they are an array of many automata, called cells, which compute at the same time. For this, each cell is provided with the information about other cell's states that lie within a certain neighborhood, based on which the next computational step is executed. This allows the simulation of phenomena appearing in nature [10], in which a large number of identical locally operating entities work together to perform complex tasks. Typical examples of this self-organization are flocks of birds or ant colonies, but also, with regard to more primitive organisms, growth patterns of fungi or the growth of bacteria colonies [6]. The domain of human life also provides dynamics like car traffic which can be mapped to and simulated by cellular automata [9].
One disadvantage of cellular automata is that both space and time are discrete. It is possible to let information move from cell to cell with varying speeds as so called signals, but these signals become increasingly erratic the closer we look. There is no possibility to achieve a fluent transition from one cell to the next, forcing the modeled system to be discretized and making it less exact than the natural behavior. Thus, the next logical step is to perform a generalization of cellular automata. There already exist a variety of generalizations like higher-dimensional cellular automata or parallel Turing machines [13], though they do not alter the underlying discrete structure. We now go one step further and replace the cellular structure with the continuous number line of the real numbers. We also permit a continuous flow of time. This allows us to send signals with arbitrary velocities along the number line without losing precision through discretization. On the other hand, there exist no particular automata and there is no clearly defined neighborhood. In addition, there is no singular computational step as the flow of time is continuous. On these grounds not the automata, but the signals themselves are the entities which carry out the computation. One can mark arbitrary points on the number line via stationary signals which do not move and based on their functionality are similar to automata. Because of the missing neighborhood, changes of states can only occur when two or more signals meet at the same position at one point in time. We denote this generalization of cellular automata as signal machines. Since existing works have only sparsely dealt with actual algorithms for signal machines, we want to lay a ground work and especially cover a range of important arithmetic operators.

1.2 Related Work

The field of signal machines is still relatively unexplored. Signal machines were introduced under the umbrella term *abstract geometric computation* by Durand-Lose [3], [4]. Their computing capabilities are further investigated in [5]. The works of Durand-Lose restrict the underlying structure to rational numbers, however, the definitions can also be applied to real numbers. Note, that in the works of Durand-Lose space time diagrams show time flowing from bottom to top whereas we have the time flow from top to bottom. One difficulty that emerges when working on the continuous number line are accumulations of signals, so called singularities, which destroy information or force signal machines to halt. Different approaches to handle singularities can be found in the works of Durand-Lose [3] and Wacker [12].

1.3 Organization of this Thesis

We introduce the relevant definitions for cellular automata and signal machines in section 2. Basic tools which are frequently used for more complex tasks are described in section 3. The proof that signal machines are truly a generalization of cellular automata by demonstrating a simulation of cellular automata by signal machines is described in section 4. A wide range of algorithms for arithmetic operators, from addition to logarithm, are presented in section 5, followed by a generalization of formal languages in section 6. We demonstrate an analogue to the usual positional notation system in the real-valued domain in section 7 and sort intervals by their lengths in section 8. Lastly, we come to a conclusion and present open questions for future work in section 9.

2 Preliminaries

2.1 Cellular Automata

Signal machines are a generalization of cellular automata, so we define the necessary discrete properties of cellular automata first and then generalize them to real-valued properties. For a detailed description of cellular automata, we refer you to the paper by Codd [2].

Definition 1. *One-Dimensional Cellular Automaton* $C = (R, Q, N, \delta)$

Let *space* R be the set $\mathbb{Z}$ of integers and *neighborhood* N be $\{-1, 0, 1\}$. Let Q be a finite *set of states*. Each value $z \in R$ describes a cell which is a finite automaton in a state $q \in Q$. The neighborhood of z corresponds to the cells $z-1, z, z+1$. Let $\delta : Q^N \longrightarrow Q$ be a *local transition function* where Q^N is the *set of all functions* $l : N \longrightarrow Q$. We call l a *local configuration*, which maps a neighborhood to its corresponding states. The local transition function δ assigns a new state to a cell based on the states of its neighborhood.

Cellular automata can be multidimensional and thus operate on other spaces. They may also have other neighborhoods. For this paper it is sufficient to restrict cellular automata to the definition above. All non-defined inputs for δ shall not change a cell's state, in case δ is only partially defined.

Definition 2. *(Global) Configuration* $c : R \longrightarrow Q$

The *global configuration* or just *configuration* c of a cellular automaton assigns a state to each cell in the space R. The configuration given to the cellular automaton as an input is called *initial configuration.* When the calculation is completed, the automaton reaches a so called *final configuration* which does not change when applying the local transition function to any cell.

Definition 3. *Global Transition Function* $\Delta : Q^R \longrightarrow Q^R$

The *global transition function* Δ applies the local transition function δ to each cell of the space R simultaneously.

Definition 4. *Passive Set, Dead State*

The *passive set* is a set $P \subseteq Q$ such that for local configuration $l : N \longrightarrow Q$, if $l(n) \in P \; \forall n \in N$ it is true that $\delta(l) = l(0)$. This means, that if the neighborhood of cell z only consists of cells in states from the passive set, z will not change its state when Δ is applied. A state d is considered *dead*, if $\delta(l) = d$ for all local configurations $l : N \longrightarrow Q$ with $l(0) = d$. That is, a cell in a dead state never changes its state.

Usually cellular automata have finite inputs, that is, a finite number of cells is in a state that is not from the passive set. All other cells can initially be in a state from the passive set or even in a dead state, if the computation is in-place.

Definition 5. *Space Time Diagram*

The *space time diagram* shows a sequence $c_i, i = 0, ..., t_{max}$ of configurations, one underneath the other, where c_0 is the initial configuration and $c_{i+1} = \Delta(c_i)$. The flow of time in this paper will be from top to bottom. Note, that other papers may use the opposite direction.

Definition 6. *(Discrete) Signal*

A *signal* $s : \mathbb{N} \longrightarrow R$ describes information that is encoded in one or more states and moves through space R. At time $t \in \mathbb{N}$, s is in position $r_t \in R$. The speed of a signal has an upper bound defined by the size of the neighborhood. Following our established neighborhood of $\{-1, 0, 1\}$, a signal can move at most one cell when applying the global transition function. A signal can have a rational speed $spd = \frac{u}{v} \in \mathbb{Q}$ by moving a total of u cells in v time steps. Rational speeds usually require multiple states to function. A signal with a speed of zero is called a *stationary signal* or a *marking*.

Definition 7. *Register*

Cells can have multiple states $q_1, ..., q_n$ at once by defining the tuple $(q_1, ..., q_n)$ as one state $q' \in Q$. We say that the automaton has n registers. One can imagine this as parallel tapes the automaton can use for additional information.

One-dimensional cellular automata are Turing-complete [11]. For each Turing machine $\mathcal{T}$ there exists a one-dimensional cellular automaton C as defined above, such that C simulates $\mathcal{T}$ without loss of time.

2.2 Signal Machines

We now leave the discrete room $\mathbb{Z}$ of integers and consider the real number line $\mathbb{R}$. Instead of mapping each number in $\mathbb{R}$ to a state, we restrict ourselves to a finite set of signals that move along the number line. When two or more signals collide, they can change their properties, be removed completely or new signals can be created. Analogously to cellular automata, signals only act locally, that is, they do not know at which part of the number line they are located at and can only interact with other signals upon collision. This is the basis for our computations. Number values are intuitively encoded on the number line by their distance to zero, where numbers left of zero are negative. For this, zero has to always be marked on the number line. We will use copies of the zero signal in some algorithms which we can move to perform calculations, that are dependent on the zero, independently from the actual position of zero on the number line. For this, let 0 be the actual number on the number line and *Zero* be the copy that can be moved. Initially, *Zero* is located at position 0. One can consider *Zero* as the origin of a subspace of $\mathbb{R}$, that defines the value zero at a new position for local calculations. Analogously, 1 is the actual number on the number line and *One* is its copy.

Definition 8. *(Continuous) Signal*

Let *Data* be a finite set, *Speeds* be the real interval $[0, 1] \cup \{\Lambda\}$, where $\Lambda > 1$ and *Directions* be the set $\{-1, 0, 1\}$. We denote $s = (dat, spd, dir)$, $dat \in Data$, $spd \in Speeds \setminus \{\Lambda\}$, $dir \in Directions$, as a *signal.* The information dat carried by s is analogous to the states $q \in Q$ of a cell in cellular automata. A speed of spd means, that s moves spd spacial units during the time $\Delta t = 1.0 \in \mathbb{R}$, where dir describes the direction. So -1 describes a movement to the left and 1 a movement to the right. If $dir = 0$, spd must also be 0, as the signal is stationary.

For simplification, we will give signals unique names such that the declaration of dat is not necessary. In addition, we will combine spd and dir by stating, that a signal s moves with speed spd to the left or right or is stationary. A stationary signal can also be called a *marking* or *boundary.* Boundary can also describe the outer-most signals of a used partial algorithm, like in algorithm 4 (*Storing Signals*) or algorithm 12 (*Addition Gadget*). If no definition of a signal's speed is given, it will always be the default speed of 1.

Definition 9. *Value of an Interval*

Let x, y be two markings, where $x \leq y$. The interval $[x, y]$ can be interpreted as the value $y - x$, which allows an encoding of a number independently from the relative position to 0. We denote the value of the interval $I = [x, y]$ as $|I|$. Note, that this does not allow the encoding of negative numbers.

Definition 10. *Collision, Collision Rule*

A *collision*, also called an *event*, occurs when two or more signals meet at the same position $x \in \mathbb{R}$ at a point in time t_{coll}, except if all involved signals are stationary. Let S, T be multisets of signals. A *collision rule* $r = (S, T)$ at an arbitrary position $x \in \mathbb{R}$ states that when all of the signals of S collide in x, they are removed (also called *destroyed*) and all signals of T are created at x. When a rule (S, T) is applied at time t, then the signals in T cannot be the input for another rule at time t. If such a behavior is desired, however, the two rules can be combined into a new rule. When more signals collide than necessary for a rule, only the signals declared in the rule are destroyed. If no rule fits the signals of a collision, no signal is destroyed or created. There must not be definitions for multiple rules that use the same set S. If rules $r_1 = (S_1, T_1)$ and $r_2 = (S_2, T_2)$ exist, where $S_1 \subset S_2$ and if a collision of all signals in S_2 occurs, then rule r_2 is applied, as it requires a larger set of signals. If rules $r_3 = (S_3, T_3)$ and $r_4 = (S_4, T_4)$ exist, where S_3 and S_4 are not disjoint, additional rules must be defined as follows. Let $S_\cap = S_3 \cap S_4$, $S_\diagdown = (S_3 \cup S_4) \setminus S_\cap$ and let $S_{pow} = \mathcal{P}(S_\cap)$ be the power set of $S_\cap$. The additional rules are $r_{\cap_i} = (S_\diagdown \cup S_{\cap_i}, T_{\cap_i}), S_{\cap_i} \in S_{pow}, |S_{\cap_i}| > 0$. Some of these additional rules may be omitted if it is clear that the corresponding collisions never occur.

As an example, let $s_1 = (\text{“}s_1\text{”}, 1, 1), s_2 = (\text{“}s_2\text{”}, 1, -1), s_1^\star = (\text{“}s_1\text{”}, 0.5, 1)$ be signals and $r = (\{s_1, s_2\}, \{s_1^\star\})$ be a collision rule. Let s_1 be located at position 0 and s_2 be located at position 2 at the initial point in time $t_0 = 0$. At time $t_1 = 1$, s_1 and s_2 collide at position 1 and are both destroyed. At the same time, $s_1^\star$ is created at position 1 and moves to the right with a speed of 0.5. Since s_1 and $s_1^\star$ carry the same information, we will consider them to be the same signal but its speed was reduced during the collision. If a collision only occurs between two signals s, t, where t is stationary, we will synonymously say that “s passes t”, “s reaches t”, “s touches t” and so on. Instead of “signal u is created moving to the right or left” we will also say that “a signal u is sent to the right or left”. If a collision $refl = (\{(\text{“}s\text{”}, spd_s, dir_s), (\text{“}t\text{”}, 0, 0)\}, \{(\text{“}s\text{”}, spd_s, -dir_s), (\text{“}t\text{”}, 0, 0)\})$ occurs, we will say that “s is reflected at t”.

Definition 11. *Kill Signal*

A *kill signal* k is the only type of signal that has the speed $\Lambda > 1$. If a signal s collides with k, then s is destroyed. If specific signals are not supposed to be destroyed by k, they are defined as *immune* and do not interact with k. The 0-signal is always immune.

Kill signals are used to remove signals that are no longer needed. Since signals may move at speed 1, an even faster speed is necessary to reach them. We will exemplarily describe kill signals in some algorithms. However, in general, they can be sent in both directions when the final result is calculated, if necessary.

Definition 12. *Configuration*

The *configuration* $c = \{(s, x) \mid s \in Sigs, x \in \mathbb{R}\}$ with the set $Sigs$ of signals describes the position of all existing signals on the number line at the time t. Let $Conf$ be the set of all configurations. If a collision rule $r = (S, T)$ is applied at t, the impacted signals in S are not listed in c, whereas the signals in T are listed. Analogously to cellular automata we call the configuration at time $t_0 = 0$ the *initial configuration.* When a configuration c_f is reached, such that no more collisions can occur except for destructions caused by kill signals, we call c_f a final configuration. Let c_f be a final configuration, $k_1, ..., k_n, n \in \mathbb{N}$, be kill signals and $s_1, ..., s_m, m \in \mathbb{N}$, be signals that are destroyed at later points in time. We call the set c_r, a subset of c_f without the tuples containing $k_1, ..., k_n, s_1, ..., s_m$ the *result configuration.* If c_f is reached at time t_f, c_r is also reached at t_f. Even though c_f can still change due to kill signals, we consider t_f to be the run time of the algorithm, since all later events do not contribute to the calculation and the speed Λ of the kill signals can be arbitrarily high.

Definition 13. *Signal Machine*

A *signal machine* $\mathcal{S} = (Sigs, Rul)$ consists of the finite set $Sigs$ of possible signals and the finite set Rul of collision rules. The input for $\mathcal{S}$ is an initial configuration and the algorithm's run time is defined by the moment, the result configuration is reached.

Definition 14. *Time Flow Function* $fl : Conf \times \mathbb{R}^+ \longrightarrow Conf$

The *time flow function* fl transfers a configuration c_1 at time t_1 into a configuration c_2 at time t_2, when $t_2 - t_1$ is the second input parameter. If collisions occur in the time interval $(t_1, t_2]$, then collision rules are applied if possible.

Definition 15. *Transition Function* $\Delta : Conf \longrightarrow Conf$

Let c be a configuration at time t_1 and time $t_2 > t_1$ be the time the first collision occurs after t_1. The *transition function* $\Delta(c)$ applies $fl(c, t_2 - t_1)$, that is, all signals are moved throughout the time interval (t_1, t_2) without collision and at the moment t_2 of the next collision, collision rules are applied if possible. The resulting configuration is returned.

Definition 16. *Proxy Signal*

A *proxy signal* is a marking, that carries the information about a former marking that has been destroyed. If no problems occur, the proxy signal may be destroyed at a later time without being used at all. However, if a conflict arises, it can be used to reconstruct the original marking and thus reset a local section of the configuration.

In some algorithms, accumulations of signals can develop, which lead to an infinite number of events within a finite amount of time, even though only a finite number of signals is involved. An easy example goes as follows: Two slow signals are approaching each other and a third signal moves back and forth between the two. When applying the transition function n times for an arbitrarily large $n \in \mathbb{N}$, the point in time when the slow signals collide, will never be reached. A second kind of singularity can occur when there are infinitely many signals in a finite space, which we try to avoid. There are multiple possibilities to handle such singularities. The work of Durand-Lose [3] uses the so called Black Hole Model, while the work of Wacker [12] defines functions to handle singularities of higher orders, that is, accumulations of accumulations. In this paper we will use the following definition of singularities that is based on limit values.

Definition 17. *Singularity*

Let the initial configuration of a computation consist of a finite number of signals. Let $t_c = \{t_{c,i}\}_{i \in \mathbb{N}}$ be an infinite, strictly monotonically increasing sequence of collision times within the finite time interval $[t_{begin}, t_{end}]$. If multiple collisions occur at a time $t_{c,i}$, then that collision time is only included once in t_c, thus $t_{c,i} \neq t_{c,j}$, if $i \neq j$. Since t_c is strictly monotonically increasing and bounded by t_{end}, there exists a limit $t_{lim} = \lim_{i \longrightarrow \infty} t_{c,i}$, which we denote as the moment a *singularity* occurs. A singularity occurs at position x_{lim} at time t_{lim}, if there exists a sequence of collision points $x = \{x_i\}_{i \in \mathbb{N}}$ at times $t_{c,i}$ with the properties

$$\begin{aligned} \lim_{i \longrightarrow \infty} x_i &= x_{lim} \\ \text{and } \lim_{i \longrightarrow \infty} t_{c,i} &= t_{lim}. \end{aligned}$$

Let a singularity S occur at position x_{lim} at time t_{lim} and let s_k, $k = 1, \ldots, m$, be all the signals involved in S, that is, for each position interval $I = [x_{lim} - \epsilon, x_{lim} + \epsilon]$, $\epsilon > 0$, each signal s_k collides an infinite number of times in I within every time interval $[t_{lim} - \delta, t_{lim}]$, $\delta > 0$. At time t_{lim}, all signals s_k are located at x_{lim}. The speed *spd*, direction *dir* and potentially also the carried data *dat* of any such signal $s_k = (dat, spd, dir)$ is not defined at time t_{lim}. We say that the information about these properties *dat*, *spd*, *dir* of signals s_k, $k = 1, \ldots, m$, is destroyed in S. If an algorithm requires any of these properties beyond the singularity, the signal machine halts at time t_{lim}. However, if the point x_{lim} is all the information an algorithm requires, we can continue the computation beyond the singularity by applying the following function. Let Δ_{lim} be a new transition function that, given a configuration c_{begin} at a time t_{begin}, processes the next singularity if it exists. It returns the configuration c_{lim} at time t_{lim} of the next singularity, where all signals involved in the singularity are destroyed and a stationary signal at position x_{lim} marks the result of the computation. If multiple singularities occur at t_{lim} in different places, then each singularity uses its own limit value $x_{lim,j}$. If there is no upcoming singularity, then Δ_{lim} just applies Δ.

In this paper, only algorithms 14 (*Multiplicative Inverse*), 15 (*Multiplication*) and 16 (*Division*) and all more complex algorithm that use these can produce singularities. We provide versions of these algorithms that only produce singularities which we only need to know the position of in order to finish or continue the computation. A possible way to determine when to apply the new transition function is to set a minimal time step Δt_{min}, such that if the next collision occurs earlier than after Δt_{min} time, a singularity is likely to occur.

3 Basic Tools

In this chapter we introduce basic algorithms to manipulate intervals, store signals or transform them into different representations. These algorithms are building blocks that later algorithms use frequently.

Algorithm 1. *Moving Intervals (Fig. 1)*

(*i*) Moving towards a point p (Fig. 1, *right*).

Let p be to the right of interval $x = [a, b]$. Moving x to the right until b and p are in the same place is equivalent to interchanging the intervals $x = [a, b]$ and $[b, p]$. We do this by sending a signal in both directions starting at b and having the signals reflect on the stationary signals a, p. When the reflected signals collide in point c, then interval $x = [c, p]$ ends exactly in p. Analogously, if p is to the left of x, the signals emerge from a instead of b and move point a to point p

(*ii*) Moving beyond a point p (Fig. 1, *left, middle*).

Let p be to the right of $x = [a, b]$. We want to move x to the right until a and p are in the same place. We send two signals to the right starting from a, one signal s with speed 1 and one signal t with speed $1 - \epsilon, \epsilon \in (0, 1)$. When s passes b, its speed reduces to $1 - \epsilon$. When t passes p, its speed increases to 1. When s and t collide in point c, we moved $x = [p, c]$ to the desired location. The case of p being to the left of x is analogous.

Algorithm 2. *Stretch and Compress (Fig. 2)*

For any point p to the right of 0, we can construct $2p$ by sending a signal s from p towards 0 with speed 1, that is reflected at 0, and a signal t to the right with speed $\frac{1}{3}$. The signals collide exactly in point $2p$. We can use this technique to multiply all stationary signals inside the interval $[0, p]$ by a factor of two. Whenever a stationary signal is passed by s, it starts moving to the right with speed $\frac{1}{3}$ and stands still when passed by the reflected s.

For any point p to the right of 0, we can construct $\frac{1}{2}$ exactly like $2p$ by sending t to the left instead of to the right. To divide all stationary signals inside the interval $[0, p]$ by 2, they start moving to the left with speed $\frac{1}{3}$ when passed by s and stand still when passed by its reflection.

To achieve a general stretch by a factor of $a, a > 1$, define the speed of t as $\frac{a-1}{a+1}$ to the right instead of $\frac{1}{3}$. For a general compression by a factor of $a, a < 1$, define the speed of t as $|\frac{a-1}{a+1}|$ to the left instead of $\frac{1}{3}$.

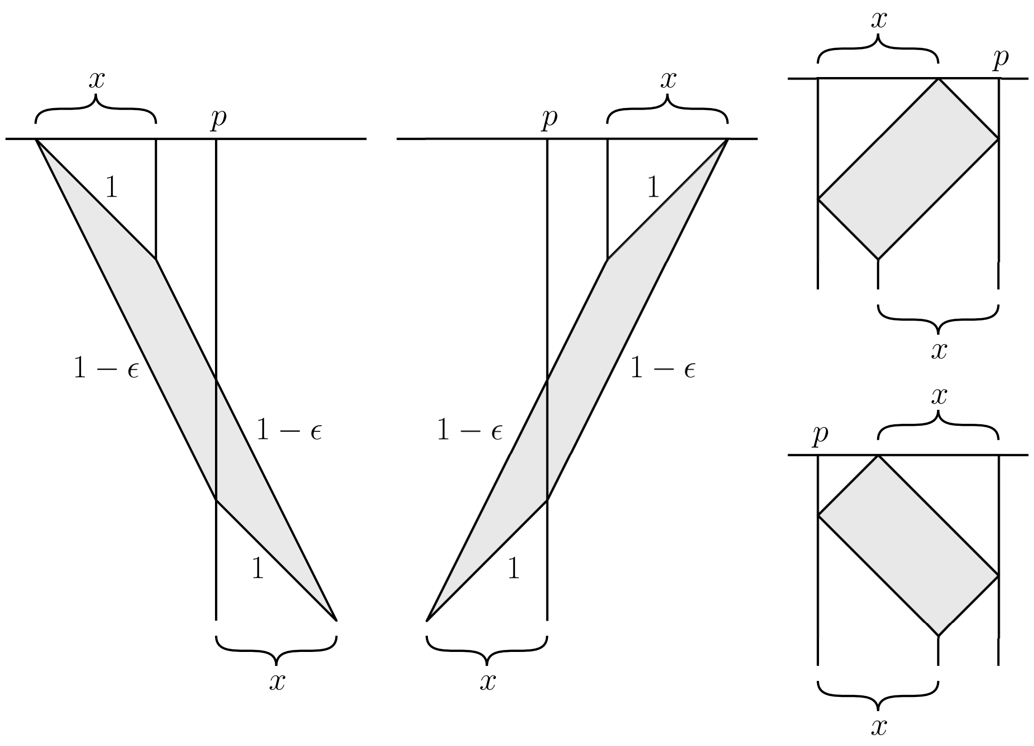


Figure 1: Moving Intervals

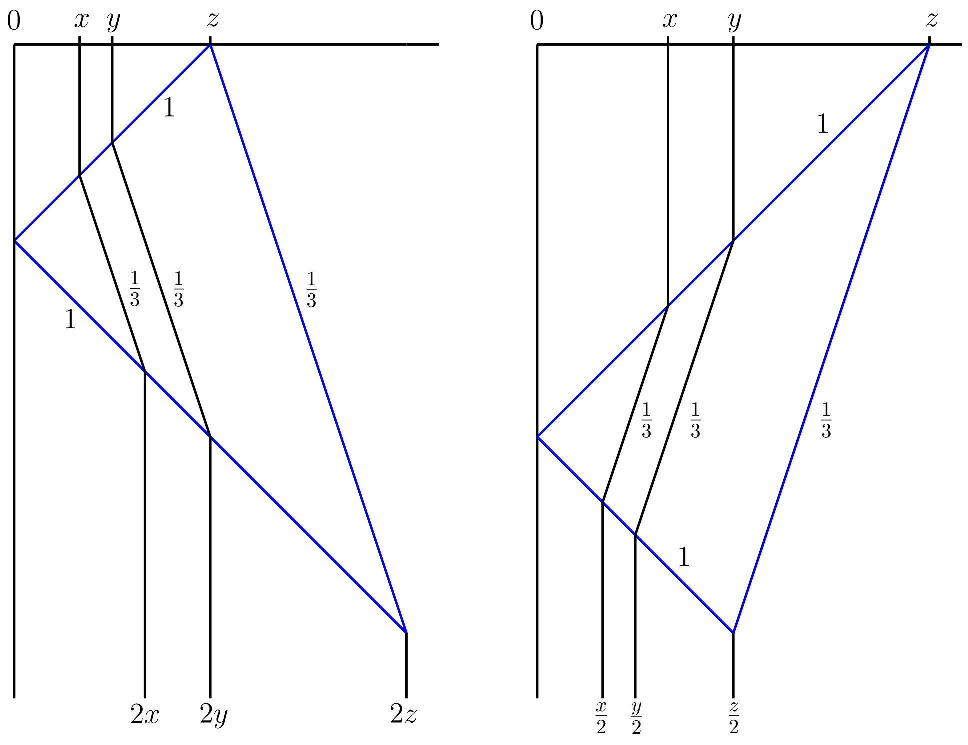


Figure 2: Stretch and Compress

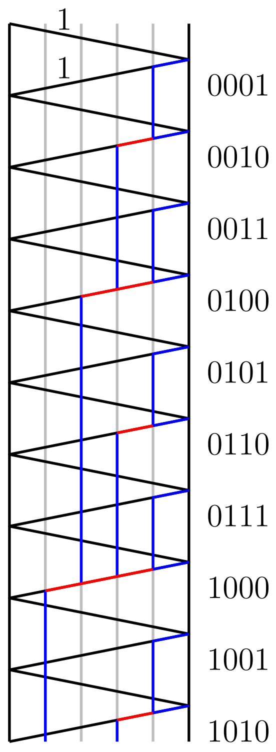


Figure 3: Binary Counter

Algorithm 3. *Binary Counter (Fig. 3)*

Let $[l, r]$ be an interval with stationary boundaries l, r and n stationary signals inside. These n signals will function like bits of a binary number. They can be in the state 0 (*gray*) or state 1 (*blue*). A signal s travels between l and r with speed 1. Whenever it is reflected at r, it changes its state to *blue*. When s is blue and passes a stationary signal in state 0, s returns to the base state and the stationary signal switches to 1. When s is blue and passes a stationary signal in state 1, s changes to a *red* state (carry bit) and the stationary signal switches to 0. Whenever the red signal s passes a stationary signal in state 1, that signal switches to 0. When the red signal s passes a stationary signal in state 0, s returns to the base state and the stationary signal switches to 1.
This algorithm can be used to count an exponential number of steps by using only a linear amount of stationary signals. We can also count a linear amount of steps by letting s mark one unmarked stationary signal per trip. One can easily expand this to counting n^k steps by using k different markings for the stationary signals.

Algorithm 4. *Storing Signals (Fig. 4)*

Real numbers allow the storage of an arbitrary finite amount of signals in a constant amount of space. We initialize a signal storage by placing two signals l, r as boundaries and defining an entrance direction. In the figure, signals can enter the storage from the right side through r. When inside, signals reflect each other and are also reflected by l and r, so stored signals will continue to move inside the storage but their order will be secured. One can use this storage gadget as either a stack or a first-in-first-out queue, depending on from which side signals can be taken out of the storage. By default, all signals entering the storage move at speed 1. When an outer signal collides with an inner signal upon entering the storage, the outer signal's speed will be reduced to $\frac{1}{2}$ until it collides again, where it changes its speed back to 1. The only exception is when three signals collide simultaneously. In that case, the signal in the middle becomes stationary and starts moving again after the next collision, unless it is a collision of three signals again. Collisions of four or more signals cannot occur within the storage. The boundaries l, r do not need to be stationary. The entrance boundary (e.g. r in the figure) can have a speed $spd \in [-1, \frac{1}{2})$ with regard to the entrance direction, while the other boundary can have any speed other than -1. A negative speed is interpreted as a speed in the opposite direction with the absolute value.

Algorithm 5. *From Interval to Marking (Fig. 5, 6)*

Since we can describe a real number as an interval or a marking we want to be able to switch between the two. While it is trivial to switch from a marking $m, m \geq 0$ to an interval, since $[0, m]$ is exactly that interval, switching from an interval to a marking depends on the interval's position with regard to the 0. Note that markings can describe negative values while intervals always represent positive values. Let $x = [a, b]$ be the interval that we want to transform into the marking x. Both a and b send signals in both directions. We refer to them as a's and b's left or right signal. The signals emerging from a are shown as solid lines, those of b are shown as dashed lines.
First, consider the left boundary a. The left signal a_l is either reflected at 0 or moves towards negative infinity. Let the reflected signal be $a_{l,0}$. The right signal a_r is reflected at b or 0 or is destroyed by a *negative* signal (*red*). When a_r is reflected at 0, it becomes a *negative* signal and sends a kill signal to the right. When a_r is reflected at b, the stationary signal b is replaced by a proxy signal b' (*green*). Let the reflected signal be $a_{r,b}$. When $a_{l,0}$ and $a_{r,b}$ collide, destroy both, set the marking x and send a signal to the right (*green dashed*) that destroys the proxy signal. Upon destruction, send a kill signal to the right. If the *negative* signal collides with a, use algorithm 1 (*Moving Intervals*) to move interval x to the right of the 0. It may be necessary to use the proxy signal as the interval's right boundary. That is the case when x is to the left of 0 and $|[b, 0]| \geq \frac{1}{2}|x|$. Also, when initiating algorithm 1, send a kill signal to the left.
Now, consider the right boundary b. The left signal b_l will be destroyed by a_r. If b_l reaches 0, send a *negative* signal to the left and a kill signal to the right. If the right signal b_r reaches 0, send a *negative* signal to the left.

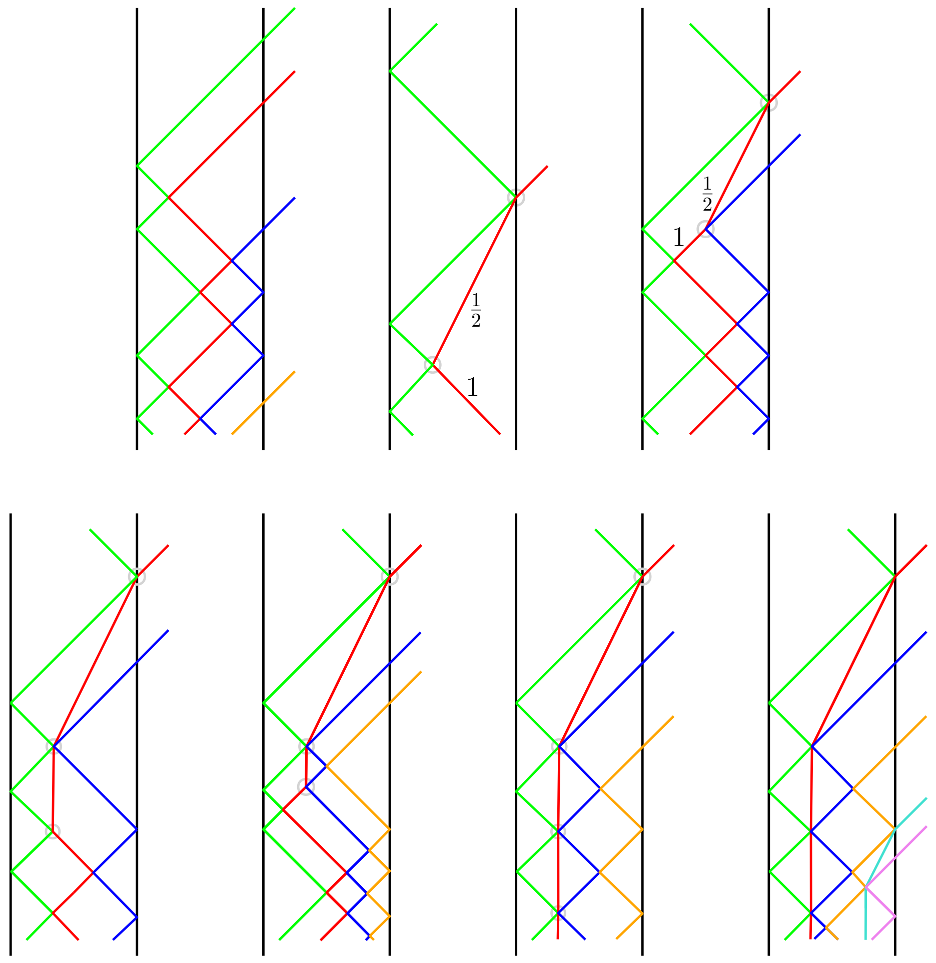


Figure 4: Storing Signals

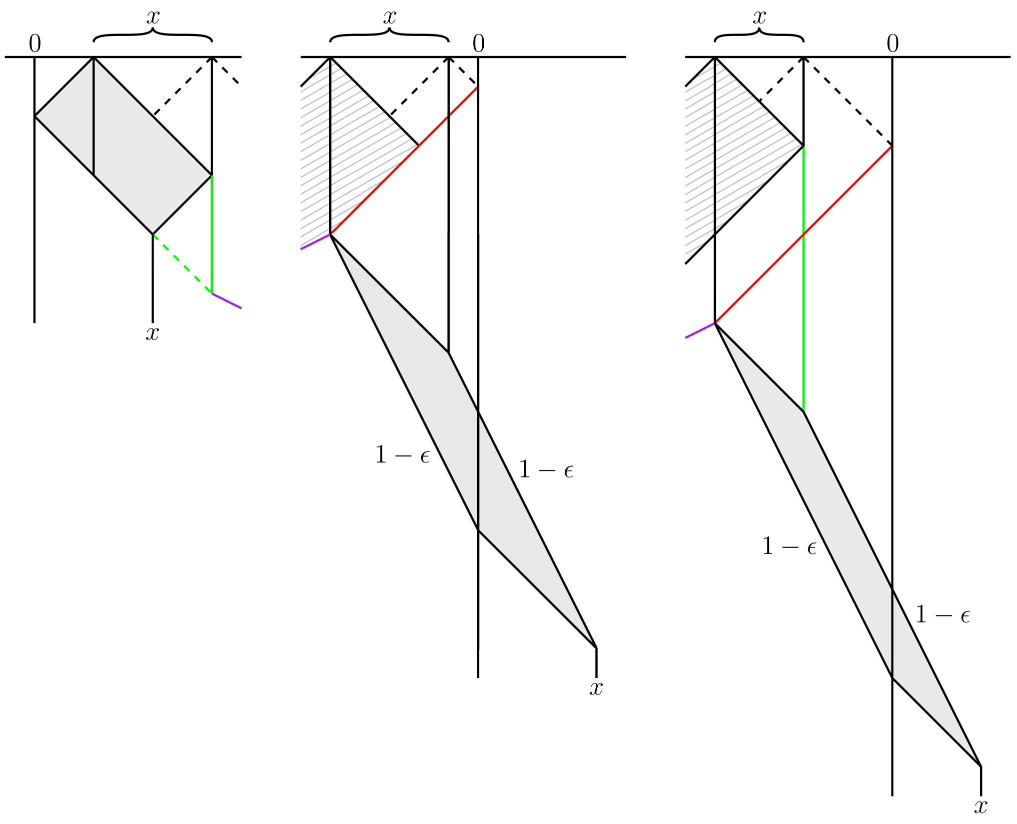


Figure 5: From Interval to Marking (1)

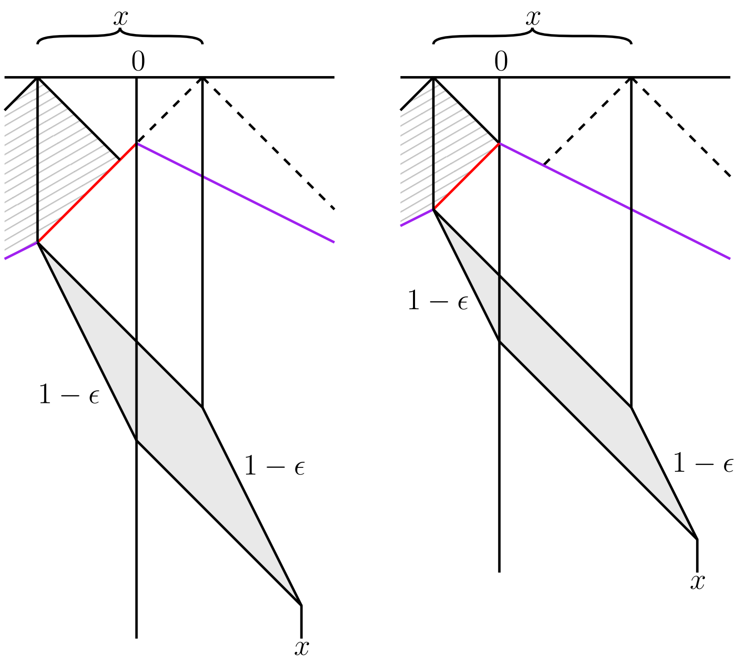


Figure 6: From Interval to Marking (2)

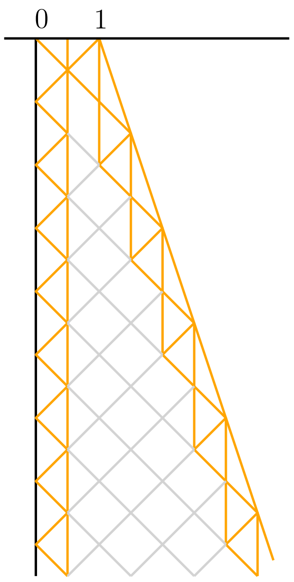


Figure 7: Generator

Algorithm 6. *Generator (Fig. 7)*

(*i*) Stationary generator

Two stationary signals with a distance of $\frac{1}{2}f$ form the generator's boundaries. A signal moves between the two and is reflected at each boundary. Each time it is reflected, a signal is sent in that boundary's direction. These signals are generated in time intervals of length f, which we call the generator's *frequency*. If generated signals are only needed on one side of the generator, simply remove the corresponding collision rules for the other side's boundary.

(*ii*) Moving generator

A generator can create signals in one direction with frequency f while moving in the opposite direction with the speed $\frac{1}{3}$. Let the side of the generator it is moving towards be called the outer side. The generator's boundary on the outer side is moving with speed $\frac{1}{3}$. That boundary starts at point p. Initially, there is a stationary signal at p and a second one with a distance of $\frac{1}{2}f$ to p on the inner side. A signal s moves from p towards the inner side. Whenever s reaches a stationary signal while moving towards the inner side, it is reflected and destroys the stationary signal. When s reaches the outer boundary, it is reflected and creates a stationary signal. Whenever s reaches a stationary signal from any side, a signal is generated that moves towards the inner side.

Algorithm 7. *Binary Representation (Fig. 8)*

We can represent a positive integer in binary by storing signals representing bits in a signal storage. Let k be the integer we want to represent in binary. Create a generator with a left boundary at 0 and a frequency of 1 that only generates signals that move to the right. We will use the inside of the generator as the signal storage. Initially, 0 sends a signal x to the right and k sends a signal y to the left. When x and y pass each other, create a signal z that moves to the left with speed $\frac{1}{2}$. When z is reflected at 0, it becomes x. Signal y is also reflected at 0. When x reaches k, it is replaced by a *red-green* signal that moves to the left and k is destroyed. This is a signal with two main states, *red* and *green*, and which is initially red. When a signal generated by the generator hits the *red-green* signal, it changes its color. When y collides with the *red-green* signal, the *red-green* signal keeps its current color. Also, y is reflected if the signal was green, or reflected and replaced by a signal y' if it was red. When the *red-green* signal reaches the generator, it is stored there. Red stands for $k \bmod 2 = 1$, green stands for $k \bmod 2 = 0$. If y collided with the green signal and thus $k \bmod 2 = 0$, a new stationary signal is created representing $k \longleftarrow k/2$. In this case, the algorithm repeats itself with a halved input. However, if y collided with the red signal and thus $k \bmod 2 = 1$, we need to round down the new value of k. To do this, we create the new stationary signal for k when the next generator signal hits the red signal. Now we encounter another problem, since the signals x and y' are not in sync, so we need to reset the algorithm for the new input. When x collides with y', x is destroyed since we do not have any use for it anymore. Upon creation of y', we also create a signal y'' that moves with speed $\frac{1}{2}$ to the left. Signal y' is reflected at 0. Since y' collides with k at exactly the same time as y'' hits zero, we use this moment to initiate the next iteration of the algorithm. If $k/2 = 0.5$, which means that y collides with the *red-green* signal on the right boundary of the generator, the algorithm ends. In the figure, we show the computation for $k = 12$, which is 1100 in binary. The signals stored inside the generator are green, green, red, red from left to right, so the highest bits are on the right-hand side. Note that the highest bit will always be a 1, so it is not necessary to store it and it can be omitted.
Processing the first digit of an input of size k takes the time $2k$, if the digit is zero, whereas the processing of the next digit already starts at time $\frac{3}{2}k$. If the digit is a one, the processing requires $\frac{5}{2}k$. After each iteration, the size of the input is reduced by its half at the least. Therefore, the total run time is $\mathcal{O}(\frac{5}{2}k + \frac{5}{4}k + \cdots) = \mathcal{O}(5k \sum_i \frac{1}{2^i}) = \mathcal{O}(5k)$.

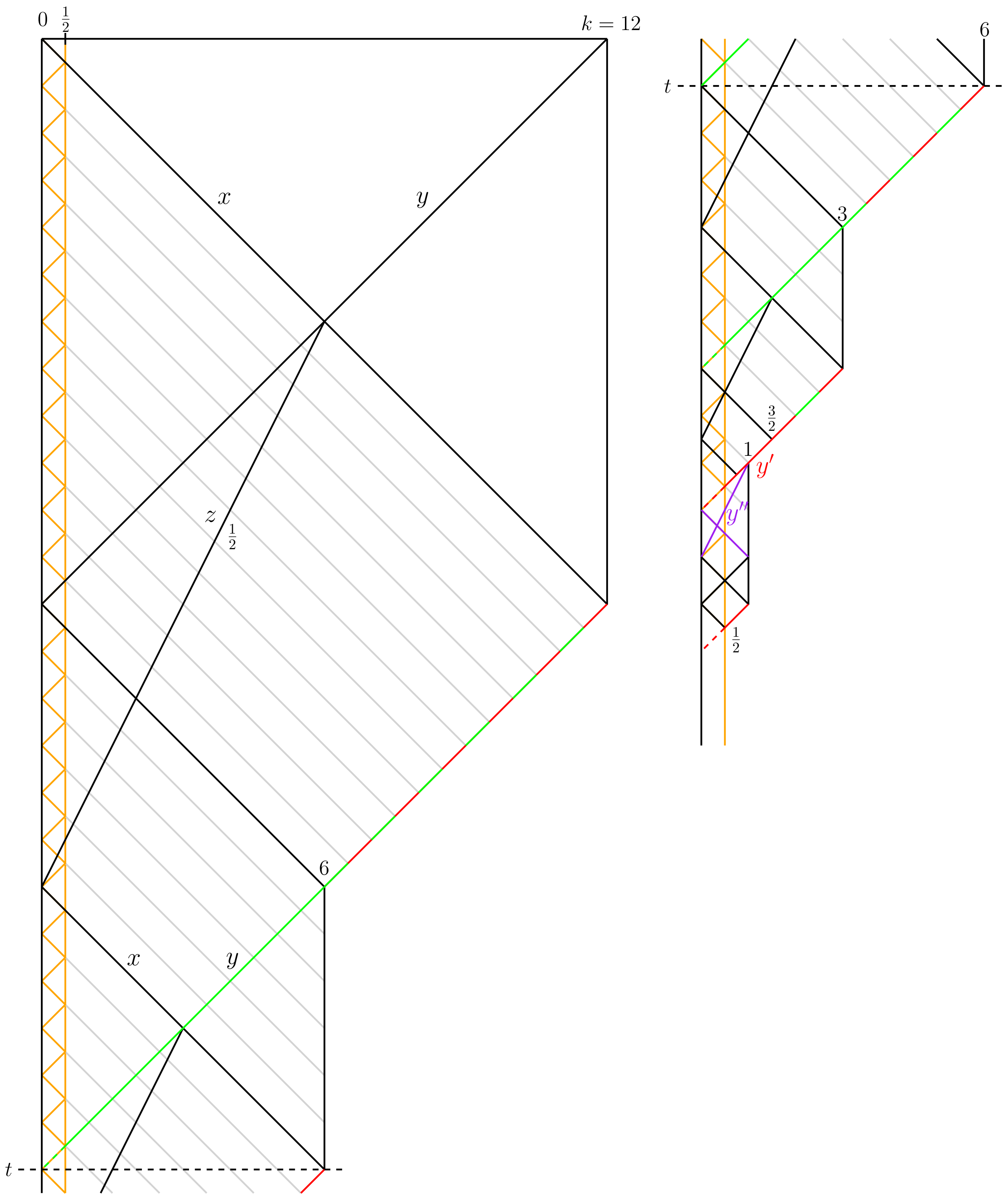


Figure 8: Binary Representation

4 Simulation of Cellular Automata

We stated multiple times that signal machines are a generalization of cellular automata and now propose the following algorithm as a proof that every cellular automaton can be simulated by a signal machine. We only need to simulate cellular automata following definition 1, as these can simulate more complex types of cellular automata themselves, since they are already Turing-complete.

Algorithm 8. *Simulation of a one-dimensional cellular automaton (Fig. 9)*

Let $\mathcal{C}$ be a one-dimensional cellular automaton with the neighborhood {-1,0,1} and input alphabet A. Let $w = w_1 w_2 \cdots w_n$, $w_i \in A$, be the input given to $\mathcal{C}$ and δ be the local transition function of $\mathcal{C}$. We construct a signal machine $\mathcal{S}$ that simulates $\mathcal{C}$ as follows. The input w is encoded in n equidistant markings that represent the w_i, $i = 1, ..., n$. Each of the markings simultaneously spawns a signal to both sides which contain the marking's information. Let $c \in A$ be the information encoded in one of the markings. The next marking on the left carries the information $b \in A$, while the next marking on the right carries the information $d \in A$. When the simulation starts, marking c sends signals c to both sides and so do its neighboring markings with their corresponding signals. When the signals b and d collide with the marking c, we destroy all three and create a marking c', where $c' = \delta(b, c, d)$. Also, signals c' are sent to both sides. This simulates one application of δ on the local configuration (b, c, d) in $\mathcal{C}$. In order to simulate the global transition function Δ of $\mathcal{C}$, we need to consider the outer boundaries of $\mathcal{C}$. If the input of $\mathcal{C}$ is bounded by dead states, we can use stationary kill signals in $\mathcal{S}$ as an equivalent. If $\mathcal{C}$ requires a larger amount of space, we need to create more stationary signals at the outer boundaries. First, we slow down the simulation by a factor of two. This can be achieved by introducing sleeping states. For each state $x \in A$, we define a sleeping state $\tilde{x}$ such that whenever a collision of three signals occurs, a marking in a sleeping state $\tilde{x}$ will only switch to the state x. When a collusion of three signals occurs, where a marking is not in a sleeping state, the new state is immediately switched to its corresponding sleeping state after applying δ. Now that the simulation is slowed down, we can create new stationary signals by having the initial outer markings send a *creation* signal away from the other markings with a speed of $\frac{1}{2}$. Every second time, the *creation* signal is hit by a signal coming from the outermost markings, we create a new marking. To compensate for the slowdown of the simulation we can simply halve the initial distance between markings. We showed that signal machines can simulate one-dimensional cellular automata with neighborhood {-1,0,1}. Since these are Turing-complete, signal machines are Turing-complete as well.

Since the space between signals can be arbitrarily small, it makes sense to alter algorithm 8 in a way, that it only requires a constant amount of space independent from the required space of the cellular automaton. We present an asynchronous approach to handle the varying distances between signals. Sadly, the modified algorithm 8′ is not able to simulate cellular automata that require an unbounded amount of space as this

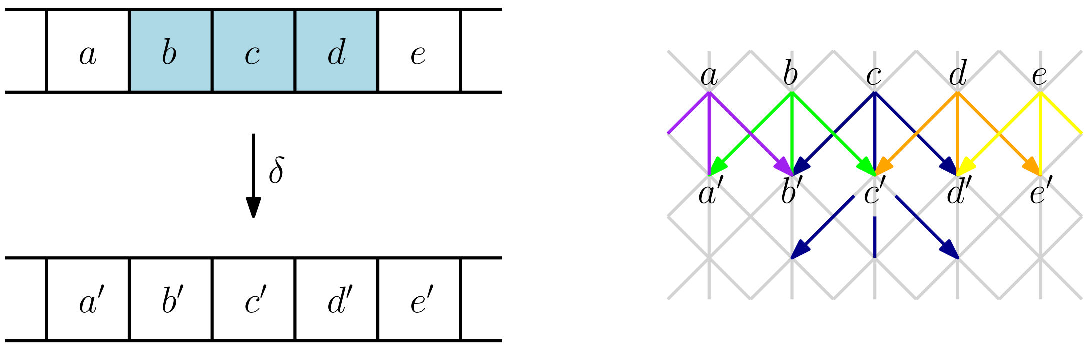


Figure 9: Simulation of a one-dimensional cellular automaton

leads to a singularity that forces the signal machine to halt. It is unclear, if it is possible to simulate all cellular automata in a constant amount of space without running into singularities.

Algorithm 8′**.** *Simulation of a one-dimensional cellular automaton with bounded required space in constant space (Fig. 10, 11)*

Let the cellular automaton C, the input alphabet A, the input $w = w_1 \cdots w_n$ and the local transition function δ be defined like in algorithm 8, but with the sole difference that C requires only a bounded amount of space. We construct a signal machine $\mathcal{S}$ that simulates C in constant space as follows. Let I be an interval of constant size ϵ. Subdivide I into $n+1$ equally sized intervals. The inner n markings represent the input characters w_i. The left and right boundary of I represent the cells in passive states that encompass the input of C. Signals originating from the outer boundaries can create additional signals which corresponds to awakening a cell from its passive state. The state $s = (l, v, r)$ of a signal consists of three components. The value v corresponds to the state of the corresponding cell of C. The information about the neighboring signal's states l and r need to be collected through collisions. Not yet received information is displayed by $\square$. Initially, each inner marking sends a signal $(\square, w_i, \square)$ to the left, the right outer boundary sends $(\square, R, \square)$ to the left and the left outer boundary sends $(\square, L, \square)$ to the right. We denote the outer markings of I and the signals originating from them as *boundary signals* and all other signals as *non-boundary signals.* Consider the collision of two non-boundary signals during which a transfer of information about the signal's states occurs. When a signal collects the information about both neighbor's states, the transition function δ can be applied. In concrete terms, the collision rules have the following forms.

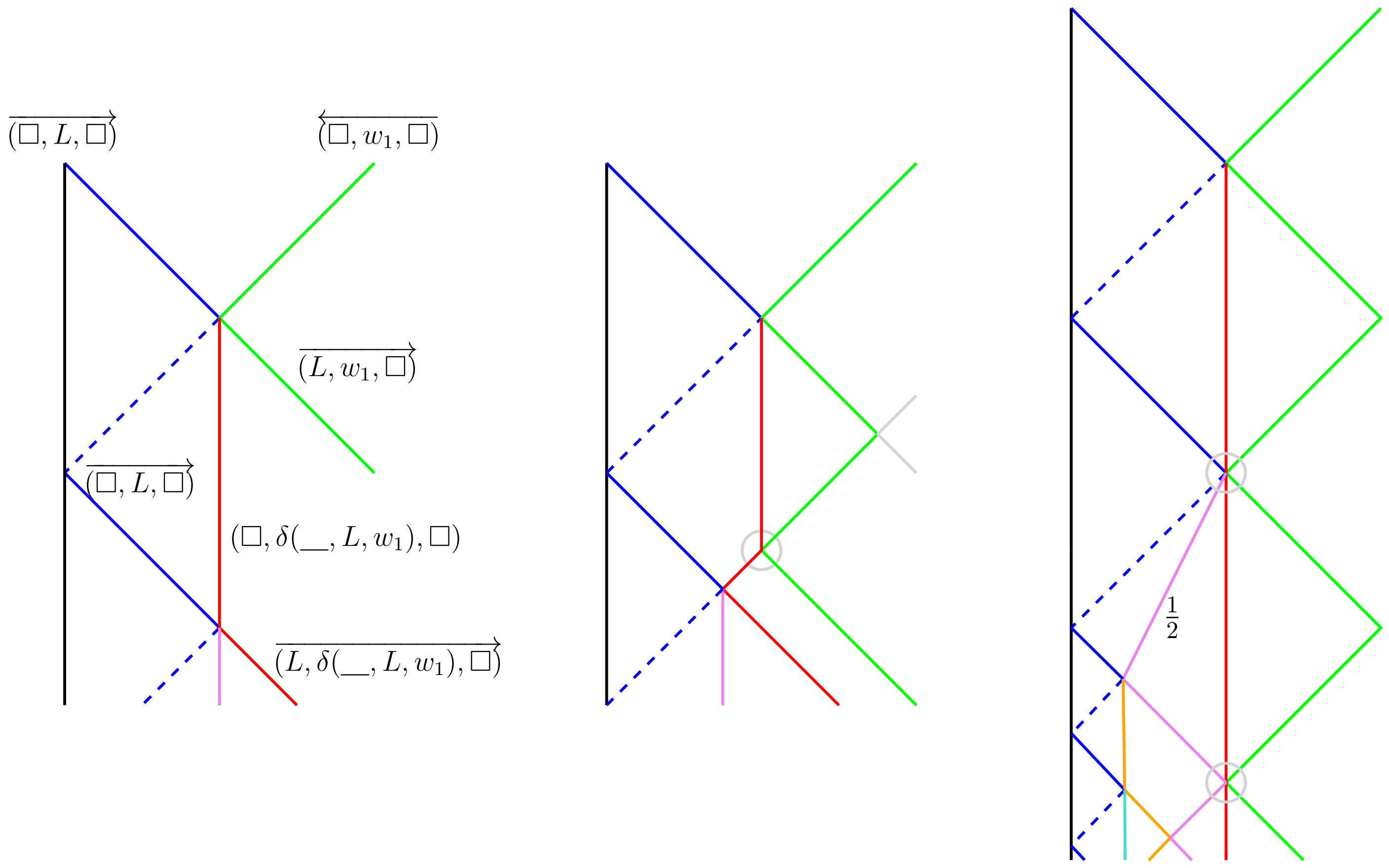
$$\{\overrightarrow{(\square, a, \square)}, \overleftarrow{(\square, b, \square)}\} \Rightarrow \{\overleftarrow{(\square, a, b)}, \overrightarrow{(a, b, \square)}\}$$

$$\{\overrightarrow{(a, b, \square)}, \overleftarrow{(\square, c, \square)}\} \Rightarrow \{\overleftarrow{(\square, \delta(a, b, c), \square)}, \overrightarrow{(b, c, \square)}\}, \text{analogously for} \{\overrightarrow{(\square, a, \square)}, \overleftarrow{(\square, b, c)}\}$$

$$\{\overrightarrow{(a, b, \square)}, \overleftarrow{(\square, c, d)}\} \Rightarrow \{\overleftarrow{(\square, \delta(a, b, c), \square)}, \overrightarrow{(\square, \delta(b, c, d), \square)}\}$$

If three non-boundary signals collide, δ can always be applied for the center signal and it will turn stationary. Stationary non-boundary signals continue to move when they collide with the next signal (*fig 10, center, gray circle*). Now consider collisions of boundary signals (*fig 10, left*), without loss of generality the signal originating at the left boundary. The signal corresponding to the input w_1 (*green*) initially moves to the left, collides with the boundary signal (*blue*), is reflected and collects the information L of its left neighbor. If the collision awakens a cell from its passive state in C, a new stationary signal is created (*red*), which has the value $v = \delta(\_\_, L, w_1)$, where $\_\_$ stands for an arbitrary state or $\square$, as it is not relevant here. If the collision with the boundary signal does not awaken a cell from its passive state, no new signal is created. The boundary signal returns to the boundary marking after each collision (*blue dashed*), where it is reflected. If a boundary signal collides with two non-boundary signals simultaneously (*fig 10, right, upper gray circle*), the left non-boundary signal becomes stationary and the newly created signal moves to the left with the speed of $\frac{1}{2}$. When three non-boundary signals collide (*lower gray circle*), this is not necessary. Since collisions of four or more signals cannot occur, all edge cases are dealt with.

Let us discuss the problem that occurs when the cellular automaton requires an unbounded amount of space. This happens for the exemplary cellular automaton with the states $\{0, 1\}$, the input of one 1 and only zeros otherwise and the transition function that moves the 1 one cell to the left. Figure 11 shows the corresponding signal machine. Each movement of the signal in C corresponds to a collision of the red signal (1) of S with the left boundary signal. As can be seen, the signal machine speeds up the more signals are created. When observing the path of the red signal, a repeating pattern can be seen starting at the second collision. Let the initial size of I be 1 without loss of generality. The second collision occurs at the time $\frac{3}{4}$ at position $\frac{1}{4}$. The red signal then moves to the left with the speed of $\frac{1}{2}$ for $\frac{1}{3}$ time units, pauses for $\frac{1}{3}$ time units, then moves to the left for $\frac{1}{9}$ time units, pauses for $\frac{1}{9}$, moves for $\frac{1}{27}$, an so on. It follows, that at the time $\frac{3}{4} + 2\sum_{i=1}^{\infty} \frac{1}{3^i} = \frac{3}{4} + 2 \cdot (\frac{1}{1-\frac{1}{3}} - 1) = 1\frac{3}{4}$ a singularity occurs. The signal machine cannot handle the singularity and therefore halts. For this specific signal machine, it is possible to destroy signals at the right boundary and thus simulate C without a singularity, as C only requires a constant amount of space when ignoring outer cells in passive states. However, we can alter C so that a second signal is sent to the right, making the singularity in S unavoidable.
It is possible to generate infinitely many signals in a constant amount of space without creating a singularity in any specific point, for example by repeatedly bisecting an interval and all of its subdivisions. In some circumstances, it may even be possible to simulate all cellular automata in a constant amount of time. This may lead to a point in time when an infinite amount of signals exist in I without having a singularity in any particular position. This entirely different kind of singularity would require new definitions to handle the situation. As of now, such an algorithm is unknown.



$\overrightarrow{(\square, L, \square)}$
$\overleftarrow{(\square, w_1, \square)}$
$\overrightarrow{(L, w_1, \square)}$
$\overrightarrow{(\square, L, \square)}$
$(\square, \delta(\_\_, L, w_1), \square)$
$\overrightarrow{(L, \delta(\_\_, L, w_1), \square)}$
$\frac{1}{2}$

Figure 10: Simulation of a one-dimensional cellular automaton in constant space

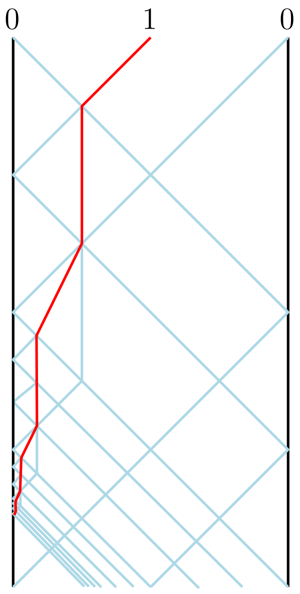


Figure 11: Singularity during a simulation in constant space

5 Arithmetic on Real Numbers

Every computation model should be able to perform a number of arithmetic calculations. We propose algorithms for the basic operations addition, subtraction, multiplication and division, which use real-valued inputs, as well as more complex calculations like the reduction of fractions, rational exponentiation and logarithms. We define two important gadgets that allow us to sum up results as finite or infinite sequences of real numbers. This can lead to singularities, some of which require special on demand algorithms and some of which we can handle as described in definition 17. Note, that a few algorithms like algorithm 22 (*Square Root*) produce approximate results.

5.1 Addition and Subtraction

Algorithm 9. *Addition $x + y, x, y \in \mathbb{R}$(Fig. 12)*

If $|x| = |y|$, destroy y and double the value of x using algorithm 2 (*Stretch and Compress*). If $y = 0$, destroy y and analogously if $x = 0$. Let $|y| < |x|$ without loss of generality. Both x and y send a signal l to the left and a signal r to the right. When l collides with 0, destroy l, send l' to the left and l'' to the right. When r collides with 0, destroy r, send r' to the left and r'' to the right. If r passes x, r becomes s_r, changes its speed to $\frac{1}{3}$ and destroys x. If r'' collides with l, r'' becomes s_r with the speed $\frac{1}{3}$, while l is unchanged. If l'' collides with s_r, destroy both and create the result marking $x + y$ (this is the case $x, y > 0$ or $y < 0 < x$). If l passes x, l becomes s_l, changes its speed to $\frac{1}{3}$ and destroys x. If l' collides with r, then l' becomes s_l with the speed $\frac{1}{3}$. If s_l and r' collide, destroy both and create the result marking $x + y$ (this is the case $x < y < 0$ or $x < 0 < y$). Whenever a result marking is created, send kill signals to both sides.

Algorithm 10. *Flipping the sign (Fig. 13)*

Given a marking x, we want to create a marking with the value $-x$. Choose a number $y, y > |x|, y \in \mathbb{R}^+$. The marking x sends two signals towards 0, one with speed 1 which slows down to speed $\frac{|x|}{y}$ when passing 0, and one with speed $\frac{2|x|}{|x|+y}$. When the two signals collide, destroy both and create the marking $-x$.

Algorithm 11. *Subtraction $x - y, x, y \in \mathbb{R}$ (Fig. 14)*

In general there are three different cases depending on the position of the input markings. For each of the cases, there exists an analogous case which is mirrored at 0, so counting these as well, the total is six. For the edge cases $x = y, x = 0, y = 0$ one can easily define additional signals for a faster computation. In any case, upon the creation of the result marking, kill signals will be sent to both sides to get rid of useless signals.
Case $0 < y < x$*:*
The subtraction is equivalent to the interchanging of intervals $[0, y]$ and $[y, x]$ as seen in algorithm 1 (*Moving Intervals*). The case $x < y < 0$ is analogous.

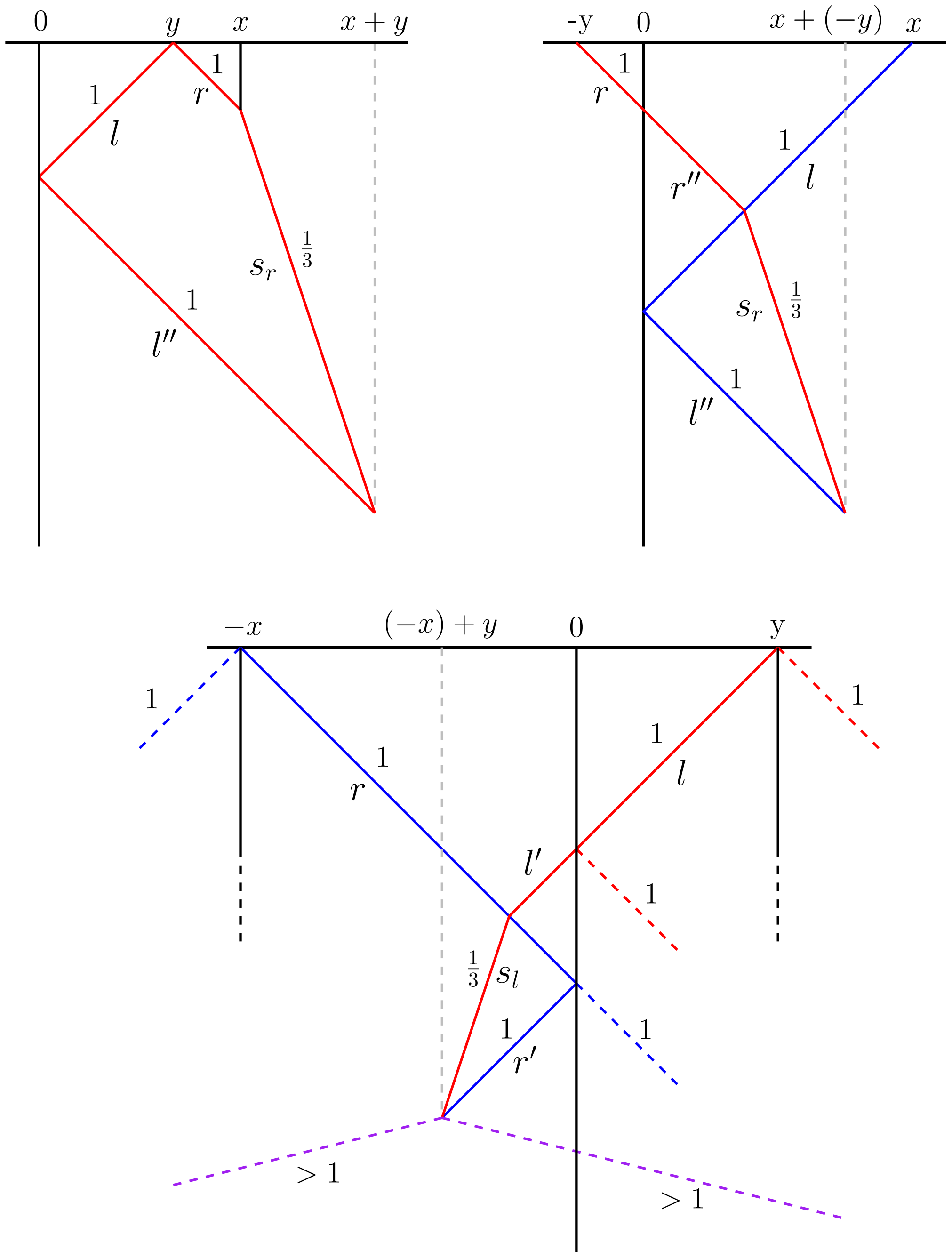


Figure 12: Addition

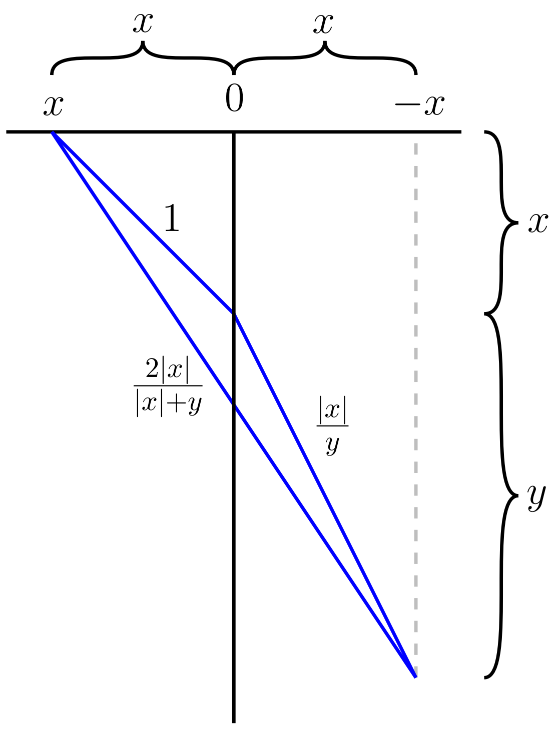


Figure 13: Flipping the sign

Case $0 < x < y$:
Starting at x, a signal is sent in both directions. The left one slows down to the speed $\frac{1}{3}$ when passing the 0, the right one is reflected by y. When the slowed down signal and the reflected signal collide, $x - y$ is reached. The case $y < x < 0$ is analogous.
Case $y < 0 < x$:
Starting at y, a signal is sent to the right that slows down to the speed $\frac{1}{2}$ when passing the 0. Starting at x, a signal is sent to the left which is reflected by y. When the slowed down signal and the reflected signal collide, $x - y$ is reached. The case $x < 0 < y$ is analogous.
To cover all cases simultaneously, we need to combine their collision rules. For example, when x's left signal passes the 0, there will be a signal with speed 1 and a signal with speed $\frac{1}{3}$, both going left.

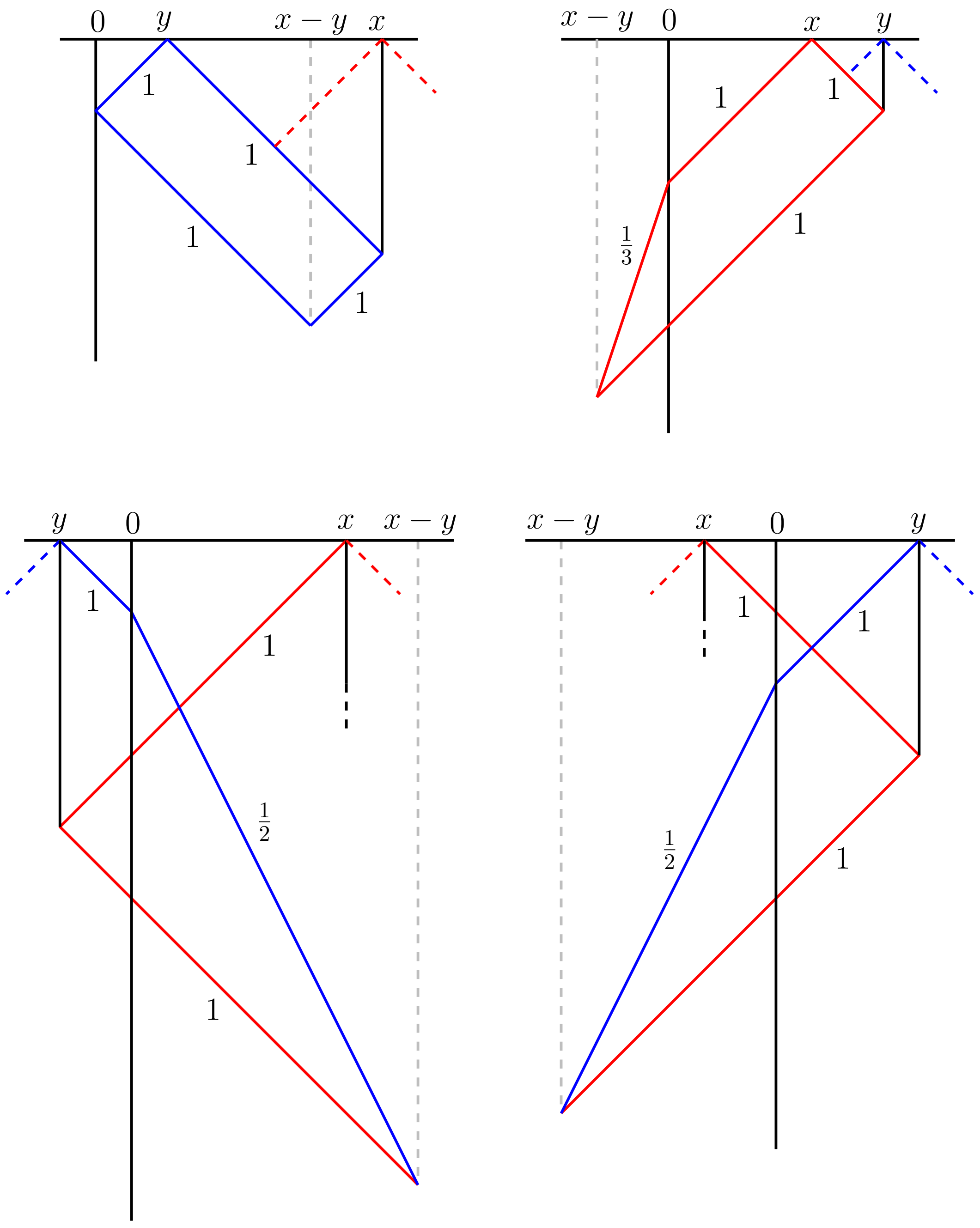


Figure 14: Subtraction

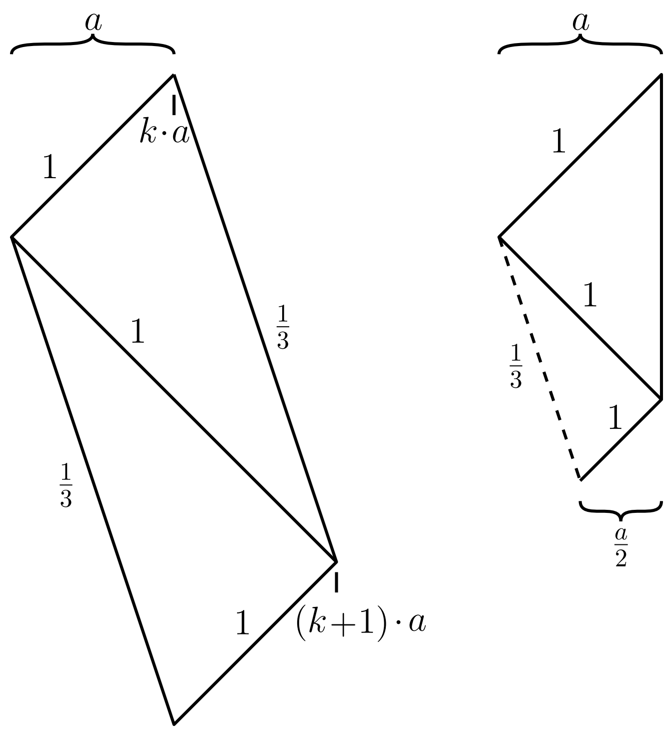


Figure 15: Gadgets

5.2 Multiplication and Division

Algorithm 12. *Addition Gadget (Fig. 15)*

With the gadget with breadth a shown on the left side of the figure one can iteratively add the value a. The right boundary moves with speed $\frac{1}{3}$ to the right and initially sends a signal s with speed 1 to the left which is reflected by both boundaries. When s collides with the left boundary for the first time, the left boundary starts moving to the right with the speed $\frac{1}{3}$. The distance between collision points of s and the right boundary is always a.

Algorithm 13. *Halving Gadget (Fig. 15)*

With the gadget shown on the right side of the figure one can halve the breadth of the addition gadget. The right boundary remains stationary. When the signal s (see algorithm 12) is reflected at the left boundary and collides with the right boundary again, the right boundary continues to move to the right with the speed $\frac{1}{3}$. The left boundary (dashed) remains unchanged.

The algorithms for the multiplicative inverse, multiplication and division each consist of two partial algorithms A and B. A is approaching a specified limit value and sends activation signals to B which accumulates the end result. Depending on the input, A may produce an infinite amount of signals in a finite amount of time. It is not possible to store infinitely many signals within the finite expanse of B. An explanation of the problem will follow algorithm 14 (*Multiplicative Inverse*). To avoid this scenario, we integrate A into B, such that A only computes as far as necessary and that B can request new signals. This way, B will be able to process the incoming signals sufficiently fast. As the resulting algorithms are rather complex, we show them in two parts to ease comprehension. We first specify the base algorithms for multiplicative inverse, multiplication and division (algorithms 14, 15, 16) which can run into the singularity problem. Afterwards, we present the improved versions for multiplication and division that avoid the problem (algorithms 15′, 16′). Even though the multiplicative inverse can be calculated via the division algorithm, the dedicated algorithm is a better introduction to the idea behind the base algorithms as it is simpler. However, to compute the multiplicative inverse without the singularity problem, we use the improved version of the division algorithms. Note that advanced algorithms which use the multiplication and division algorithm may only refer to algorithms 15 and 16 instead of 15′ and 16′ and show rough figures of the base algorithms. It is recommended to still use the improved versions in these cases.

Algorithm 14. *Multiplicative Inverse of $x \in (0,1]$ (Fig. 16, 17, 18)*

Partial Algorithm A (fig. 16, 18 left):

Use the interval [0,x] as the breadth of an addition-gadget. Whenever its signal s collides with its right boundary, a stationary signal is created. Whenever s collides with the left boundary, a signal k (black, dashed) is sent to the left which destroys such a stationary signal. The stationary signals are necessary to reset the addition-gadget to an earlier state. After each complete addition process, an addition signal (*blue*) is sent to the right which partial algorithm B will collect. If the right boundary of the addition-gadget collides with the 1 at the same time an addition process completes, the addition signal will also be a halting signal for B. If the gadget's right boundary collides with the 1 before an addition process completes, a reset signal is sent to the left which uses the last two stationary signals so restart the addition-gadget there. At the moment of this collision with the 1, a halving signal (*red*) will be sent to the right to B. Upon resetting the addition-gadget it will halve its breadth via the halving-gadget before continuing the addition process.

Partial Algorithm B (fig. 17, 18 right):

This partial algorithm uses an addition-gadget with a breadth of 1. The addition process starts when the first signal of A reaches its right boundary (at 1). In the case of an addition signal (*blue*) the usual addition will proceed. In the case of a halving signal (*red*), a halving-gadget will be used instead. While a gadget is performing its task, another signal of A can be caught at the right boundary of the gadget such that the next gadget can begin immediately after the current one finished. If yet another signal reaches the gadget's right boundary while it is already occupied, the new signal will be reflected. We use the inner part of the gadgets as a signal storage for signals sent by A

as described in algorithm 4 (*Storing Signals*), with the difference that the left boundary of the storage is initially the 1, until the gadget's left boundary passes the 1. If a gadget finishes its computation without having any signals left for the next gadget, both boundaries become stationary signals until they are reactivated by more signals sent by A. When the halting signal reaches the right boundary of the gadget, the gadget will destroy itself after finishing the remaining computations and leave its right boundary as the stationary signal of the result $\frac{1}{x}$.

Figure 18 shows the case that several halving-gadgets are used before the first addition-gadget. The left side shows algorithm A, the right side shows B (with a faint A underneath). Each red dot represents a halving signal that immediately reaches B. Since B is always fast enough to process signal sequences with leading halving signals, there is no need to store signals. Once an addition signal (*blue*) reaches the 1, the algorithm proceeds as described above.

Singularity Problem *(Fig. 19)*

Consider a signal storage with a finite breadth, which is entered by an infinite amount of signals $(s_i)_{i\in\mathbb{N}_0}$ in the finite time interval $\delta = [t_0, t_1]$. Let δ and the s_i be defined, such that for each interval $\delta_\epsilon = [t_0, t_1 - \epsilon], \epsilon > 0$, only finitely many signals enter the storage. It follows that a singularity occurs at t_1. We assume that no collision rules are applied at t_1 and move all signals forward as if there was no collision in the first place. In addition, we define a signal named *super signal* which enters the storage exactly at t_1, so it is the last signal entering it. Consider the rightmost signal within the storage (*red*). It is reflected at the right boundary and then collides with the next signal to the left. That signal is then reflected to the left and collides with its left neighbor and so on. This impulse (*red dashed*) moves to the left until it reaches the super signal. At this point in time, a collision should occur that reflects the leftmost signal to the left, which then moves to the left boundary (*purple*). However, since every collision with the super signal results in a singularity, there is no leftmost signal (excluding the super signal). Thus, the purple signal is not defined and the signal machine cannot continue its computation.

Side note: If the super signal had reached the right boundary, the required computation would have been equivalent to the reversal of an infinite sequence. In the figure, a singularity of order two would have occurred at that moment, that is, a singularity of singularities. If the computation had been possible and the storage's boundaries approached each other, their collision would have even resulted in a singularity of order three.

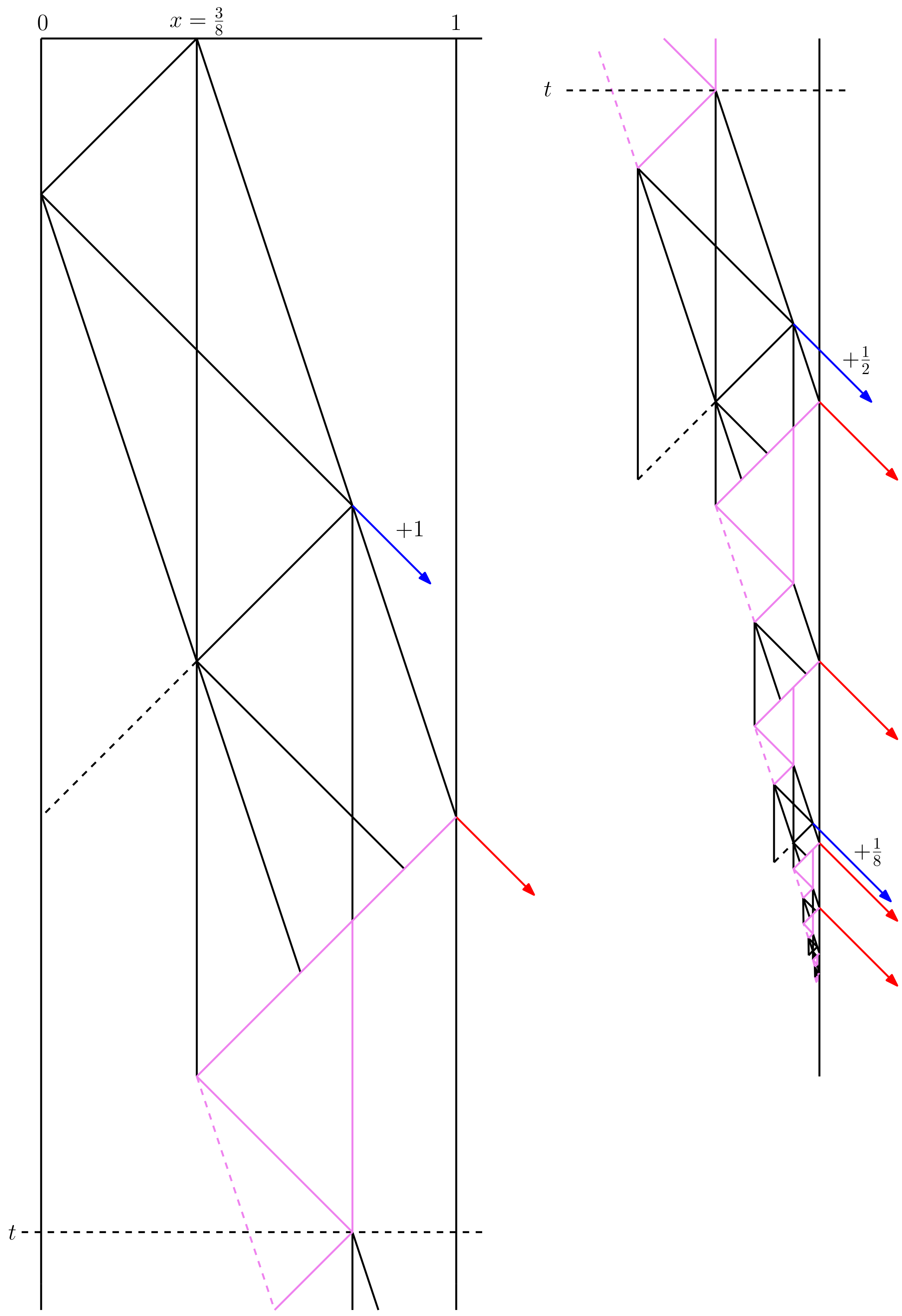


Figure 16: Multiplicative Inverse (1)

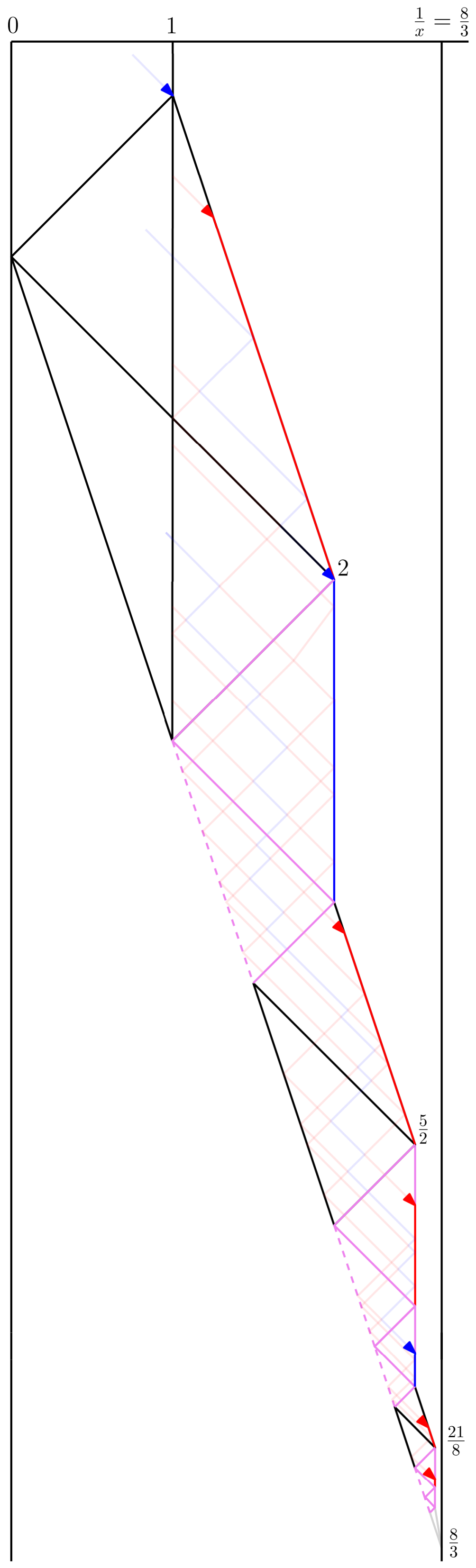


Figure 17: Multiplicative Inverse (2)

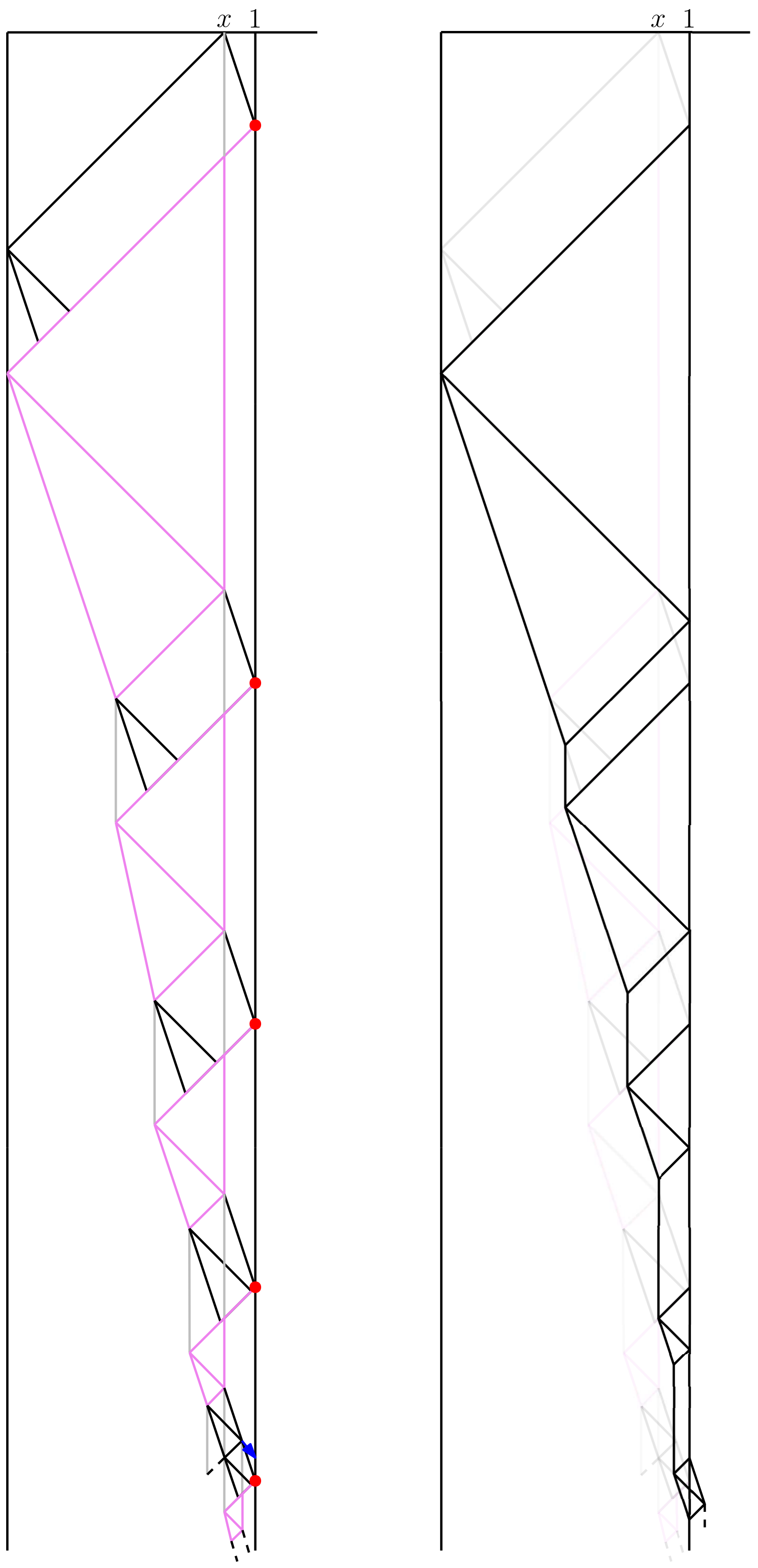


Figure 18: Multiplicative Inverse (3)

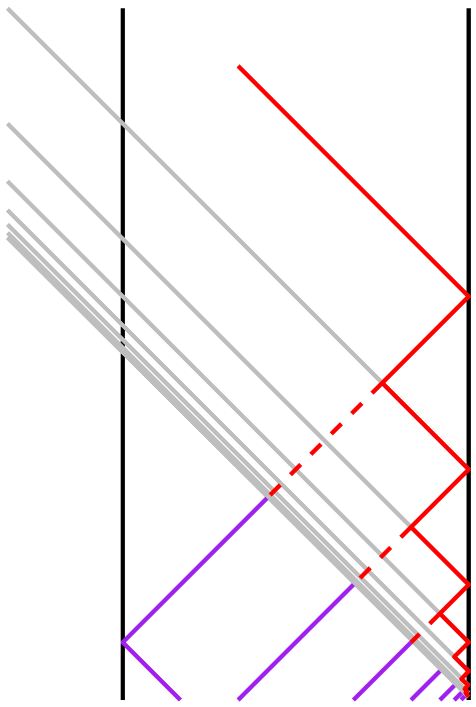


Figure 19: Singularity Problem

Theorem 1. *Algorithm* 14 *(Multiplicative Inverse) is correct, if no singularity problem occurs.*

Proof. The algorithm first finds a maximum $k \in \mathbb{N}_0$, such that $k \cdot x \leq 1$ and $k \leq \frac{1}{x} < k+1$, where k is the number of leading addition gadgets. It follows that $1 - kx < x$ and the first halving gadget reduces the step size of partial algorithm A to $\frac{x}{2}$, as well as the step size of B to $\frac{1}{2}$. If $kx + \frac{x}{2} = 1$, then $k + \frac{1}{2} = \frac{1}{x}$ and the next addition gadget of B determines the end result. If $kx + \frac{x}{2} < 1$, then B adds the value $\frac{1}{2}$ to the intermediate result and the step sizes for A and B are halved again. If $kx + \frac{x}{2} > 1$, the step sizes are halved without adding to the intermediate result. After n halvings we arrive at the following relationship.

$$kx + \alpha_1 \frac{x}{2} + \alpha_2 \frac{x}{4} + \cdots + \alpha_n \frac{x}{2^n} \left\{ \begin{matrix} = \\ < \\ > \end{matrix} \right\} 1 \quad \Longleftrightarrow \quad k + \frac{\alpha_1}{2} + \frac{\alpha_2}{4} + \cdots + \frac{\alpha_n}{2^n} \left\{ \begin{matrix} = \\ < \\ > \end{matrix} \right\} \frac{1}{x} \quad \text{or}$$

$$kx + \sum_i \alpha_i \frac{x}{2^i} \left\{ \begin{matrix} = \\ < \\ > \end{matrix} \right\} 1 \quad \Longleftrightarrow \quad k + \sum_i \frac{\alpha_i}{2^i} \left\{ \begin{matrix} = \\ < \\ > \end{matrix} \right\} \frac{1}{x},$$

where $\alpha_i \in \{0, 1\}, k \leq \frac{1}{x} < k + 1, k \in \mathbb{N}_0$ and $\alpha_i = 1$, if the addition gadget of A after the i-th halving can finish its computation. It follows, that if the right boundary of A reaches the marking 1 and no singularity problem occurs, the right boundary of B reaches the desired result of $\frac{1}{x}$. □

Theorem 2. *The run time of algorithm* 14 *(Multiplicative Inverse) is $\mathcal{O}(\frac{1}{x})$, if no singularity problem occurs.*

Proof. The run time of the whole algorithm is only dependent on partial algorithm B, since A terminates when it sends its last signal to B and B only terminates after processing said signal. Let the first gadget of the sequence be an addition gadget. As seen in figure 17, the left boundary of B moves to the right with the constant speed of $\frac{1}{3}$ from leaving the 0 onward. Partial algorithm B terminates at the latest if the left boundary reaches the value $\frac{1}{x}$. However, as this case leads to a singularity problem, B terminates at an earlier point in time based on the assumption that no such problem occurs. The left boundary of B starts to move at the latest at the time $3 + 1$, resulting in the total run time of $\mathcal{O}(4 + \frac{3}{x}) \subseteq \mathcal{O}(\frac{1}{x})$.
Let us now consider the case that the first j gadgets of the sequence are halving gadgets. After the processing of each of the leading halving signals, B has to pause until A sends the next signal. The individual waiting times are bounded by the lengths of the interrupted addition gadgets of A. The largest possible interrupted gadget exists for $x \approx \frac{1}{2}, x > \frac{1}{2}$. From the launch of the addition gadget to the launch of the following halving gadget, less than $\frac{3}{2}$ time units pass. Since the gadgets halve after each iteration, the total length of the interrupted gadgets is less than $\frac{3}{2} + \frac{3}{4} + \dots = 3\sum_i \frac{1}{2^i} \leq 3$. It follows that the total waiting time of B is bound by a constant and thus the run time for this scenario is $\mathcal{O}(\frac{1}{x})$ also. □

Algorithm 15. *Multiplication $a \cdot b$, $a, b \in \mathbb{R}^+$ (Fig. 20, 21)*

Let $a \leq b$ without loss of generality. If $a = 1$, destroy a and let b be the result.
• Case $a > 1$ (*fig.* 20)
Partial Algorithm A:
At first we compute $\min(a, b) = a$ by having both a and b send signals to both sides (dashed). After the collision of two such signals, the one continuing to the left will reach $\min(a, b)$, while the one continuing to the right will reach $\max(a, b)$, from which a kill signal will be sent to the right.
Upon Determination of $\min(a, b) = a$, the multiplication with the (possibly rounded down) integer value of a begins. Beginning at a, a signal is sent to the left which, starting at One, repeats the subtraction $a \longleftarrow a - 1$ (turquoise boxes, according to algorithm 11 (*Subtraction*)). As long as $a \geq One$ is true after the subtraction, an addition signal (*blue*) is sent to the right to partial algorithm B each time. If $a = One$ after one such subtraction, the original input value of a was an integer. In that case, an addition and halting signal can be sent to B and A terminates. For the initial input $a \notin \mathbb{N}$, its signal will inevitably pass *One* during one of the subtractions. At that moment, a halving signal (*red*) will be sent to B and the value of *One* will be halved (gray triangle). Now the real-valued multiplication begins. We repeatedly compare a and *One*. If $a < One$, which we encounter when during halving of *One*, its right-moving signal passes a, a halving signal is sent to B and *One* is halved once again. If $a > One$, which we encounter when during halving of *One*, its signal with the speed $\frac{1}{3}$ passes a, an addition signal is sent to B. Furthermore, we calculate $a \longleftarrow a - One$ after the completion of *One*'s halving. If

$a = One$, the *One*-signal can be destroyed and a moves to the 0, where it is destroyed as well and sends a halting signal to B.
Partial Algorithm B:

This algorithm functions like partial algorithm B from algorithm 14 (*Multiplicative Inverse*), with the sole difference that the gadget's initial breadth is $\max(a, b)$.
• Case $a < 1$ (*fig. 21*)
Whether this case is occurring can be determined when finding $\min(a, b)$. If a's left signal collides with 0 before reaching *One*, $a < 1$.
Partial Algorithm A:

Analogous to the case $a > 1$, but the integer multiplication is skipped.
Partial Algorithm B:

Since the partial algorithm's initial breadth of $\max(a, b)$ surpasses the result's value, halving signals will be handled differently before the first addition signal reaches B. The signal b will be halved and thus moves closer to 0. This will become the gadget's right boundary at a later point. When the first addition signal reaches b, there is no need for the use of an addition gadget yet, as b is the exact value of the current intermediate result. However, at this point the interval $[0, b]$ is the initial breadth of the gadget and the next addition or halving signal will initiate the usual gadget process. If signals sent by A need to be stored before the gadget is initialized, they will be stored between the boundaries b and $\max(One, a, 0)$, where destroyed signals are counted as 0.

Theorem 3. *Algorithm* 15 *(Multiplication) is correct, if no singularity problem occurs.*

Proof. Let a_{in} and b_{in} be the input values and $a_{in} = \min(a_{in}, b_{in})$ without loss of generality. Let $a_{in} = k + r, k \in \mathbb{N}_0, r \in [0, 1)$. We denote a, b as the signals initialized with the values a_{in}, b_{in} which change their position over time. The algorithm first computes $k \cdot b_{in}$ by repeating $a \longleftarrow a - 1$ until the remainder $a_1 = r < 1$ remains while B repeatedly adds the value b_{in} to the intermediate result. Then A and B halve their step sizes from 1 to $\frac{1}{2}$ and b_{in} to $\frac{b_{in}}{2}$ respectively. These values are also halved after each of the following iterations. During the i-th iteration of the real-valued multiplication of $r \cdot b_{in}$, the algorithm checks whether $a_i \geq \frac{1}{2^i}$. Starting in the first iteration, if $a_1 \geq \frac{1}{2}$, then $a_{in} \geq k + \frac{1}{2}$ and B adds $\frac{b_{in}}{2}$ to the intermediate result which is then $(k + \frac{1}{2}) \cdot b_{in}$. Meanwhile, A computes $a_2 \longleftarrow a_1 - \frac{1}{2}$. If $a_1 < \frac{1}{2}$ the intermediate result remains unchanged as $(k + \frac{1}{2}) \cdot b_{in} > a_{in} \cdot b_{in}$ and let $a_2 \longleftarrow a_1$.
Let res_i be the intermediate result after the i-th iteration. If $a_i \geq \frac{1}{2^i}$, then $a_{in} \geq \frac{res_i}{b_{in}} + \frac{1}{2^i}$ and B computes $res_{i+1} = res_i + \frac{b_{in}}{2^i}$. If $a_i < \frac{1}{2^i}$, then $res_{i+1} = res_i$. All in all, we arrive at the following relationship:
If $a = k + r = k + \sum_i \frac{\alpha_i}{2^i}, \alpha_i \in \{0, 1\}$, where $\alpha_i = 1$, if $a_i \geq \frac{1}{2^i}$, then B computes the value $(k + \sum_i \frac{\alpha_i}{2^i}) \cdot b_{in} = a_{in} \cdot b_{in}$. □

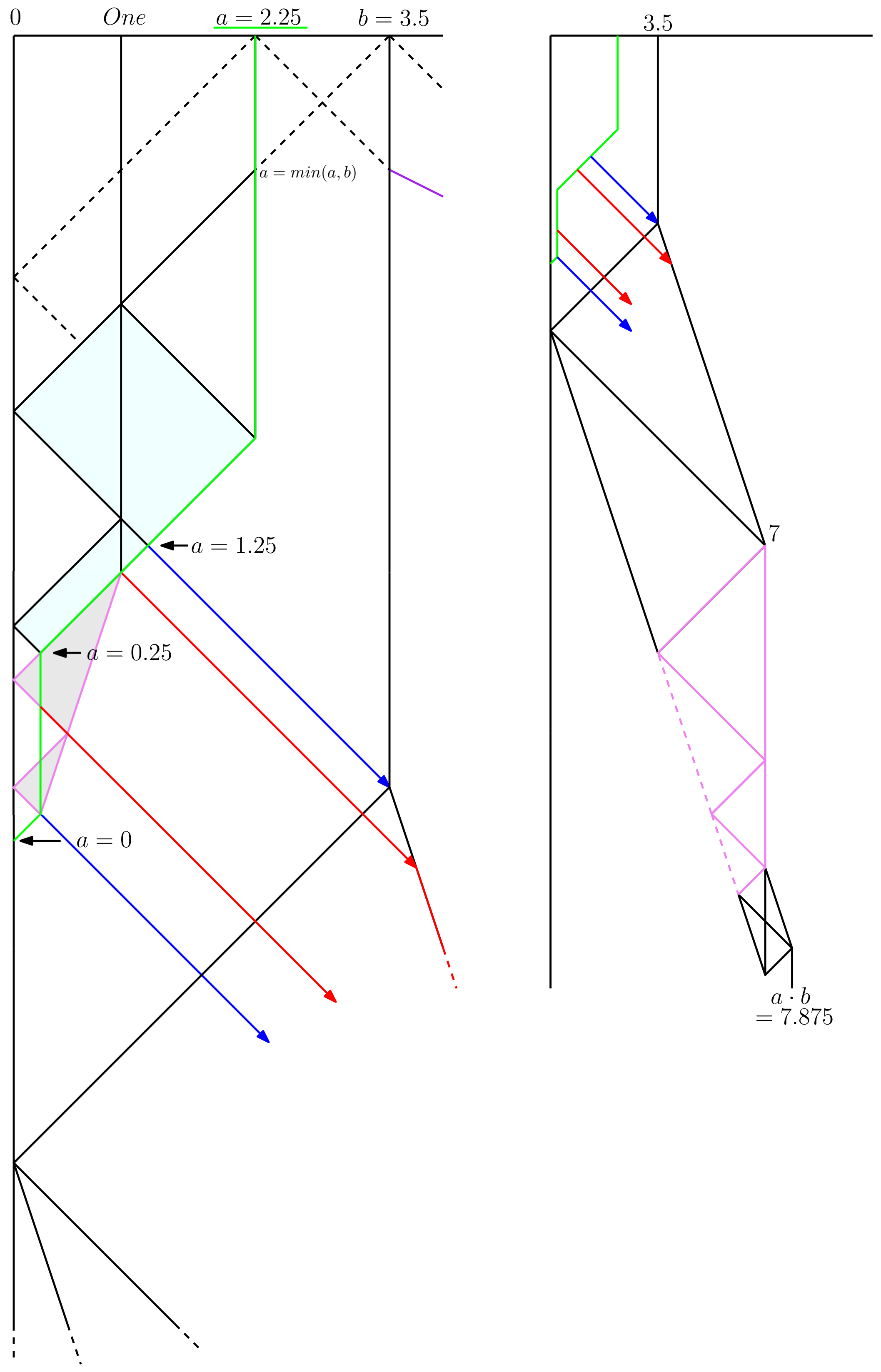
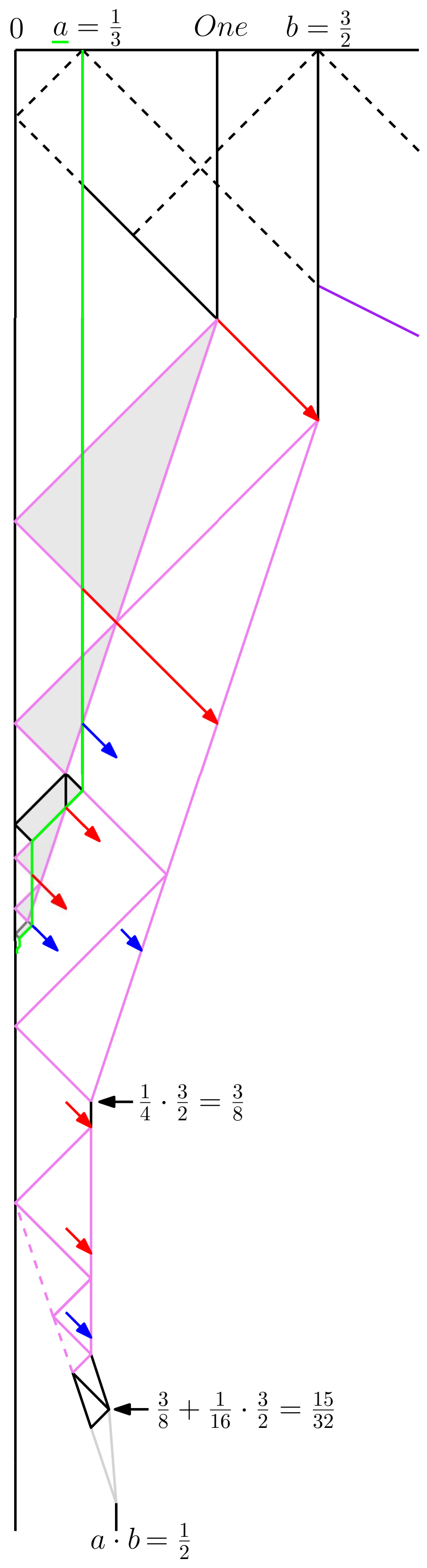


Figure 20: Multiplication (1)



0
$a = \frac{1}{3}$
One
$b = \frac{3}{2}$
$\frac{1}{4} \cdot \frac{3}{2} = \frac{3}{8}$
$\frac{3}{8} + \frac{1}{16} \cdot \frac{3}{2} = \frac{15}{32}$
$a \cdot b = \frac{1}{2}$

Figure 21: Multiplication (2)

Theorem 4. *The run time of algorithm* 15 *(Multiplication) is* $\mathcal{O}(a+b+ab)$*, if no singularity problem occurs.*

Proof. Let $a = \min(a, b)$ without loss of generality. Analogously to algorithm 14 (*Multiplicative Inverse*), the total run time depends solely on the run time of partial algorithm B.
• Case $a > 1$
It takes the time $3b$ until the left boundary of B leaves the 0-marking with the constant speed of $\frac{1}{3}$ to the right. The result is determined at the latest if the left boundary reaches the value $a \cdot b$. However, this would go against the assumption that no singularity problem occurs so the termination happens at some earlier point. The run time is thus $\mathcal{O}(3b + 3ab) \subseteq \mathcal{O}(a + b + ab)$.
• Case $a < 1$
Partial algorithm B starts its first halving after the time $a + b$. Since $a > 0$, the initial gadget breadth of B is determined at the latest after another $3b$ time units. If A's next signal is reflected at the right boundary of B slightly earlier, it takes at most $2 \cdot \frac{b}{2}$ time units until it reaches the right boundary again, as well as at most $\frac{b}{2}$ more units until the left boundary of B leaves the 0-marking. Analogously to the case $a > 1$, the algorithm then terminates after less than $3ab$ more time units resulting in a total run time of $\mathcal{O}(a + 5.5b + 3ab) \subseteq \mathcal{O}(a + b + ab)$. □

Algorithm 15′**.** *Improved Multiplication* $a \cdot b, a, b \in \mathbb{R}^+$ *(Fig. 22, 23, 24, 25, 26, 27)*

The integer multiplication remains unchanged. Let again be $a \leq b$ without loss of generality.
• Case $a \geq 1$:
As soon as the value *One* is halved for the first time and thus the real-valued part of the multiplication starts, partial algorithm A pauses until the now stationary signals are reached by partial algorithm B's first gadget. When this is the case, the computation of A is sheared to the right by $\frac{1}{3}$, that is, each signal of A that is usually stationary now moves with a speed of $\frac{1}{3}$ to the right. The left boundary of B, which also moves to the right with a speed of $\frac{1}{3}$, is used as the *Zero* for A (*Fig. 22*). Due to the shearing we need to redefine how to compute the halving of *One* (if *One* $> a$) and $a \longleftarrow a -$ *One* (if $a >$ *One*). Figure 23 shows these new procedures. The black signal moving to the left is the signal s inside the gadget of B that moves between its boundaries. This signal launches one of the two procedures depending one whether it collides with a or *One* first. If it collides with both at the same time, a and *One* can be destroyed. If $a >$ *One*, an addition signal is sent to the left that is reflected at *Zero* and enters the storage inside B at that moment. If *One* $> a$, a halving signal is sent instead. If $a =$ *One*, an addition and halting signal is sent. An exemplary execution of A is displayed in figure 24. When B begins the processing of the last remaining stored signal, it sends a *demand signal* (*yellow dashed*) which launches the next procedure of A. In the image, blue signals are addition signals (+) and red signals are halving signals ($\frac{1}{2}$, which does not represent a speed here).

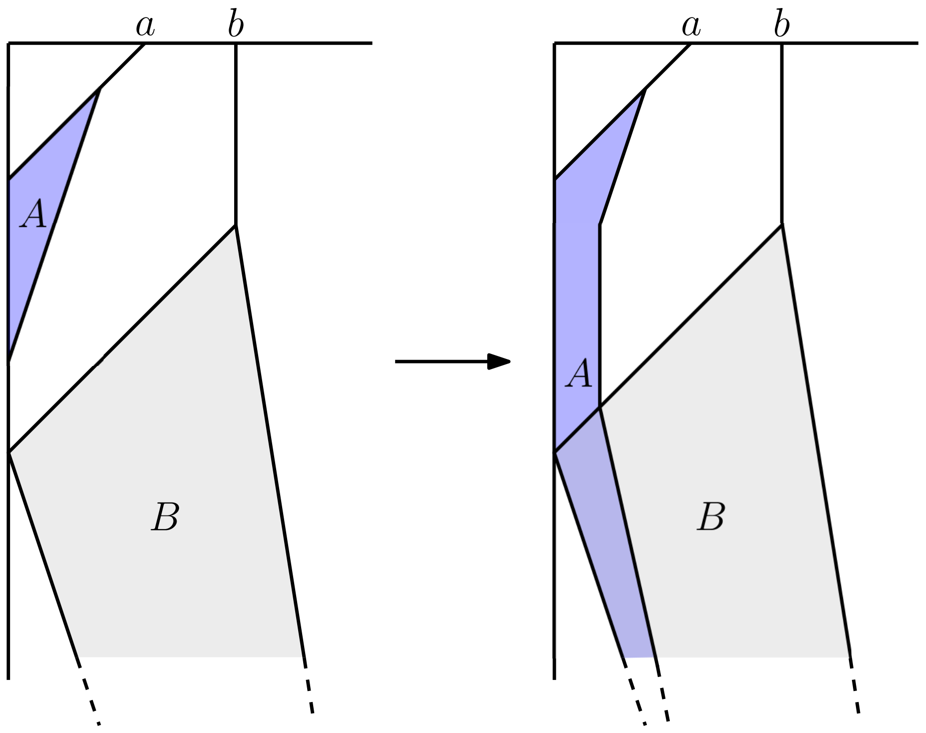


Figure 22: Improved Multiplication (1)

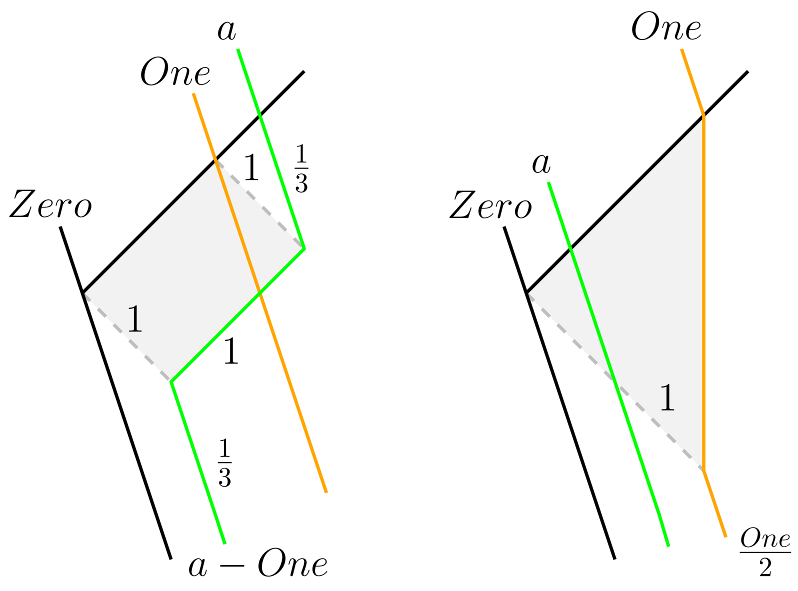


Figure 23: Improved Multiplication (2)

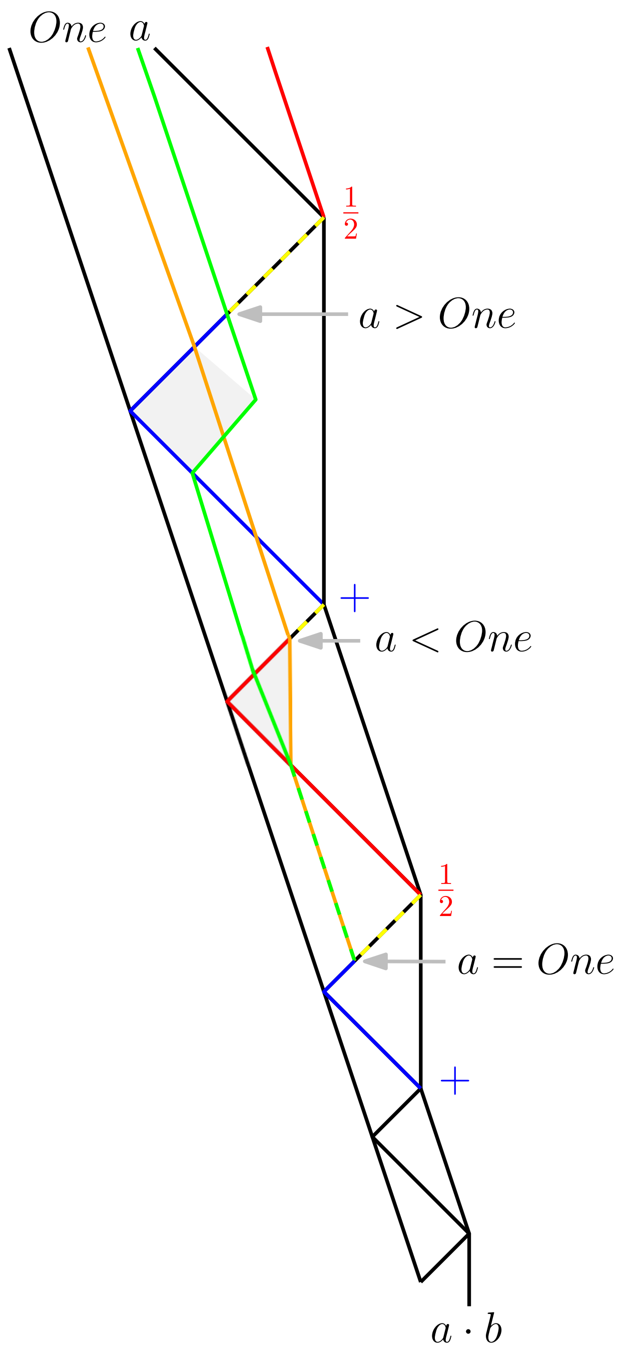


Figure 24: Improved Multiplication (3)

We need to ensure that A remains inside of B at all times such that it computes each step fast enough. For that we only need to consider B's halving gadgets as they are getting thinner and a and *One* are never getting larger. Per calculation step only one of the values a and *One* is decreased depending on which is larger at the beginning of the step. In the case that both signals reside in the right half of the gadget (*fig. 25, top, dashed line*), the signal with the lesser value will leave the gadget (*red dots*). Let us assume that both a and *One* reside in the gadget's left half at the beginning of the step. As seen in figure 25, *bottom*, there always exists a set of procedures such that the two signals reside in the gadget's left half when it finishes and got thinner. If both signals reside in the second quarter from the left at the beginning of the gadget, two consecutive procedures are necessary. As can be seen in the image, there is enough time for them to be executed. Note, that these cases always produce an addition and a halving signal and never two of the same kind. Since both a and *One* must be in the left half of each gadget when it starts, A may have to start the next procedure(s) even though no demand signal was sent. As we proof after this algorithm description, there will at no time be an infinite amount of signals. Plus, the assumption for the initial positions of a and *One* when attaching to B is correct. All in all, the singularity problem does not occur.
Partial algorithm B remains unchanged for the most part in comparison to algorithm 15. The only differences are the demand signals and the marking of the two left quarters. Figure 26 shows how the leftmost quarter can be marked initially (*left figure*) and how the marking is kept throughout the computation (*right figure*) as an example.
• Case $a < 1$ (*fig. 27*):

It is true that $a \cdot b < b$, so we need to determine the initial breadth of B's gadget. Partial algorithm A begins its computation just like in algorithm 15. As soon as an addition signal reaches B, its initial gadget breadth is defined. The first addition gadget is omitted analogously to algorithm 15. When the gadget's breadth is determined, B pauses. When both a and *One* are in the left half of B, A pauses and B is reactivated. When the now stationary signals of A reach the first gadget of B, they are sheared to the right by $\frac{1}{3}$ like in the case above. Figure 27 shows the beginning of the computation for the cases $a < 1 \leq b$ (*left*) and $a \leq b < 1$ (*center*). The signals a (*green*) and *One* (*orange*) are compared. If $One > a$, *One* is halved (*gray triangles*) and a halving signal is sent to B (*red*, the $\frac{1}{2}$ does not stand for a speed here). If $a \geq One$, $a \longleftarrow a - One$ is computed (*tilted rectangle*) and an addition signal is sent to B (*blue*). Partial algorithm B starts at the input b and its halvings are the violet triangles. The dashed lines show some exemplary markings for the two left quarters. The figure on the right-hand side shows the speeds required for their establishment. In the shown case $a < 1 \leq b$, a and *One* reside in the left half of B so early, that B is paused and reactivated at the same time so it sends the first demand signal immediately (*yellow dashed*).

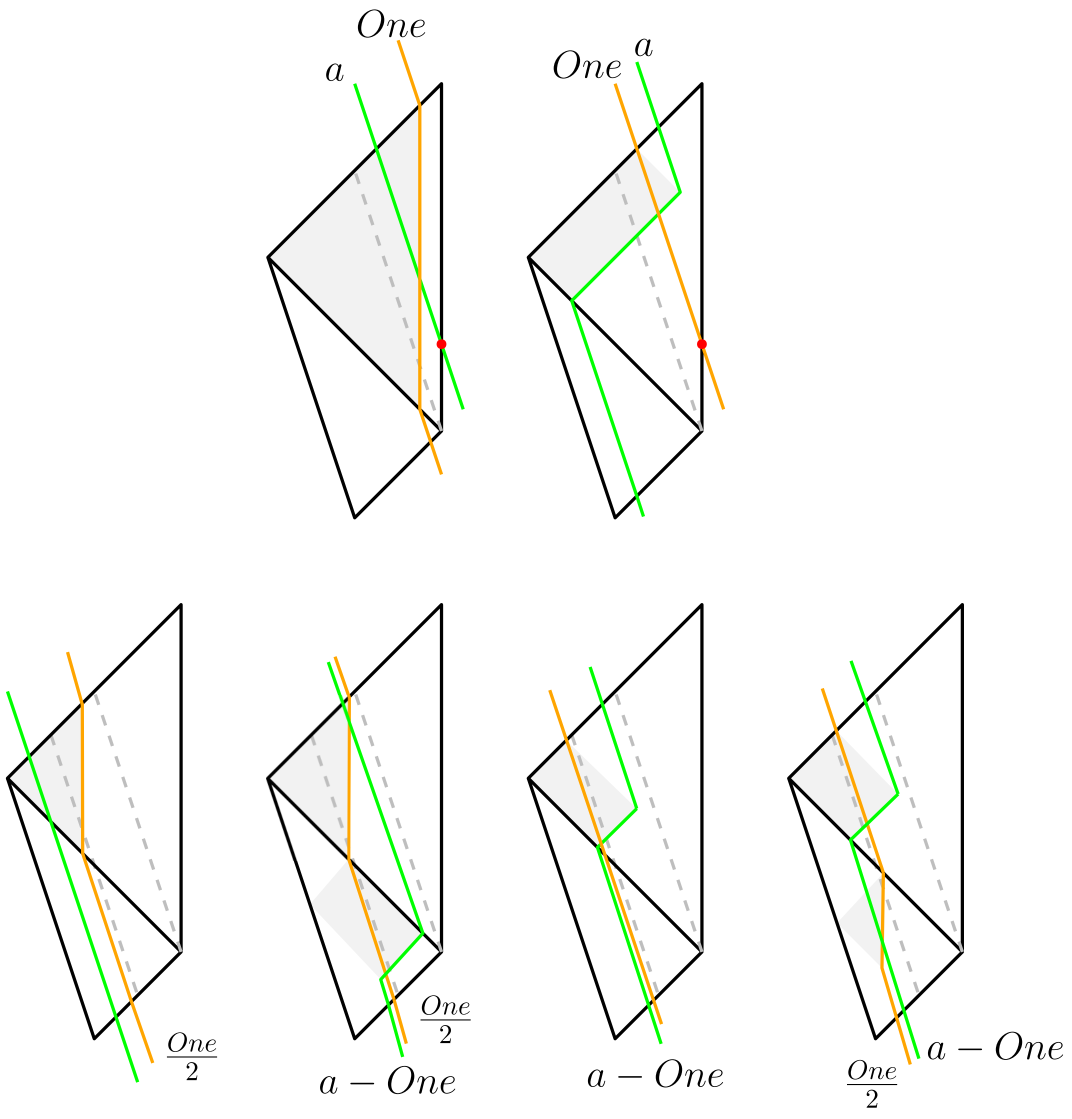


Figure 25: Improved Multiplication (4)

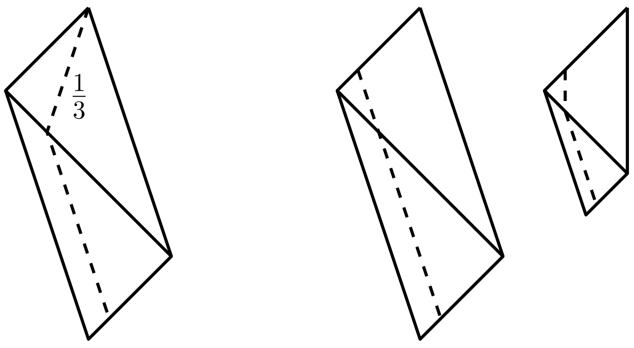


Figure 26: Improved Multiplication (5)

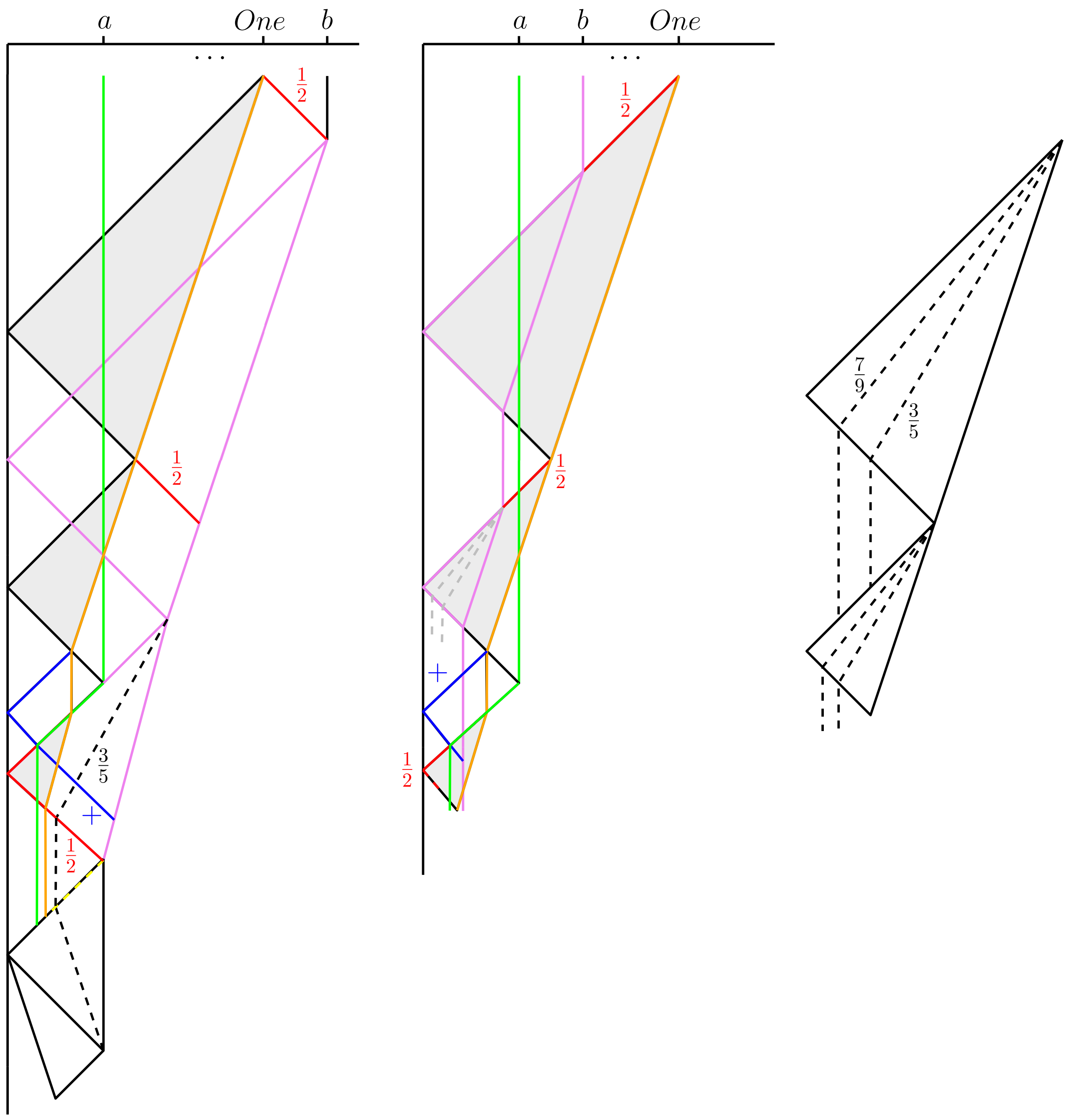


Figure 27: Improved Multiplication (6)

Theorem 5. *Algorithm* 15′ *(Improved Multiplication) is correct.*

Proof. Let a_{in}, b_{in} be the input values and a, b be the corresponding signals that may move over time. We first proof that the signals a and *One* reside in the left half of B when A attaches itself to the left side of B. In the case $a_{in} < 1$ this is true by construction since A does not attach before the property is met. Let now be $a_{in} > 1$. If $b_{in} \geq 2$, the property is fulfilled, since after the integer multiplication, a is always smaller than 1 and $One = \frac{1}{2}$. Let now be $b_{in} \in (1, 2)$. *One* still satisfies the requirement. If $a_{in} \in (1, \frac{3}{2})$, a is less than $\frac{1}{2}$ after the computation $a \longleftarrow a - 1$. If $a_{in} \in [\frac{3}{2}, 2)$, then $a \in [\frac{1}{2}, 1)$ after subtracting 1. This more than halves the value of a and since $a_{in} \leq b_{in}$, the desired property is met. It follows that the signals a and *One* always reside in the left half of B at the time of the attachment and they remain in the left half during the entire computation of B.
Next we show, that at no point in time will there be infinitely many signals present. Let us first consider the case $a_{in} < 1$, before A attaches to B. Since $a_{in} > 0$, *One* passes a after a finite number of halvings, so B also pauses after a finite number of steps. It follows that at most finitely many further steps are required until a and *One* are to the left of the stationary signal marking the middle of B. Now, let us consider the situation that the number of signals increases after A attaches to B (for a general a_{in}). Whenever this happens, A creates two signals while B processes only one gadget, as seen in figure 25, *bottom*. One of the two signals is always an addition signal. When B processes the addition gadget, A can pause, reducing the number of signals by one. As a result, there cannot be an infinite number of signals at any point in time during the execution of algorithm 15′.
Lastly, the relationship between the values of signals of A and B are unchanged in comparison to algorithm 15, so B still terminates when creating the result marking at position $a_{in} \cdot b_{in}$. If a singularity occurs, the result marking is created upon collision of the two boundaries of B following definition 17 (*Singularity*). □

Theorem 6. *The run time of algorithm* 15′ *(Improved Multiplication) is* $\mathcal{O}(a + b + ab)$.

Proof. If $a > 1$, the run time is identical to the run time of algorithm 15, since the run time of partial algorithm B is unchanged and A never terminates later. If $a < 1$, the run time of B is insignificantly longer due to a possible pause of $\epsilon > 0$ time units before the attachment of A. □

Algorithm 16. *Division* a/b, $a, b \in \mathbb{R}^+$, $b \neq 0$ *(Fig. 28, 29)*

• Case $a \geq b$ (*fig. 28*):
Partial Algorithm A:

As the role of numerator and divisor is defined by the input, we do not need to calculate $\min(a, b)$ like in algorithm 15 (*Multiplication*). As long as $a \geq b$ we perform an integer division. For this we repeatedly compute $a \longleftarrow a - b$ (turquoise boxes). All signals for partial algorithm B will emerge from the 0 when a signal is reflected at it. When a passes b, b is halved (*violet, gray triangles*) and the corresponding halving signal is sent

to B, when the violet signal is reflected at 0. When the first halving is happening, the real-valued division starts and this partial algorithm follows A from algorithm 15 (*Multiplication*), with the exception that a is compared to b instead of One (and instead of halving One, we halve b) and the signals for B originate at 0 as mentioned above.
Partial Algorithm B:

The result signal begins at 0 and moves to 1 (initially marked) when the first addition signal of the integer division is created and immediately destroyed at 0 (*blue dot*). Upon reaching 1, the interval $[0, 1]$ is the initial breadth of the gadget collecting A's addition and halving signals and computing the result analogously to algorithm 14 (*Multiplicative Inverse*).

• Case $a < b$ (*fig. 29*):

To differentiate between the two cases, b initially sends a signal to the left. If it collides with a, then the case $a < b$ is executed, or if it collides with 0 first, the case $a \geq b$ is executed. It is true that 1 is too large as the initial breadth of the gadget for partial algorithm B, as it surpasses the result's value. As long as $a < b$, b and the result signal (initially at 1) are halved. As soon as $b \leq a$, an addition signal will be sent to B such that the current intermediate result value can be used as the initial gadget's breadth for B. Notice, that B is closer to the 0 in the figure (*green*).

Theorem 7. *Algorithm* 16 *(Division) is correct, if no singularity problem occurs.*

Proof. Let a_{in}, b_{in} be the input values and a, b be the corresponding signals that may move over time. Let $\frac{a_{in}}{b_{in}} = k + r, k \in \mathbb{N}_0, r \in (0, 1)$. The algorithm first computes k by repeating $a \longleftarrow a - b_{in}$ until a remainder $a - kb_{in} < b_{in}$ is left, while adding up 1 each time. The execution of the real-valued division is very similar to the real-valued multiplication of algorithm 15, but a and b are compared instead of a and One. Additionally, the initial gadget breadth of B (if $a_{in} > b_{in}$) is 1 instead of b_{in}. Analogously to algorithm 15, b is halved after each iteration. During the first iteration, the algorithm checks whether $a_{in} - kb_{in} \geq \frac{b_{in}}{2}$. If true, $\frac{1}{2}$ is added to the intermediate result, as $k + \frac{1}{2} \leq \frac{a_{in}}{b_{in}} \iff kb_{in} + \frac{b_{in}}{2} \leq a_{in}$. Meanwhile, A computes $a \longleftarrow a - \frac{b_{in}}{2}$. If $a_{in} - kb_{in} < \frac{b_{in}}{2}$, the intermediate result stays the same, as $k + \frac{1}{2} > \frac{a_{in}}{b_{in}} \iff kb_{in} + \frac{b_{in}}{2} > a_{in}$. Considering all the subsequent iterations, we end up with the following relationship: $a_{in} - kb_{in} - \sum_i \alpha_i \frac{b_{in}}{2^i} = 0 \iff k + \sum_i \frac{\alpha_i}{2^i} = \frac{a_{in}}{b_{in}}$, where $\alpha_i \in \{0, 1\}, k \in \mathbb{N}_0$ and $\alpha_i = 1$, if $a > \frac{b_{in}}{2^i}$ during the i-th iteration. Consequently, if signal a reaches 0 and no singularity problem occurs, B marks the correct result of $\frac{a_{in}}{b_{in}}$. □

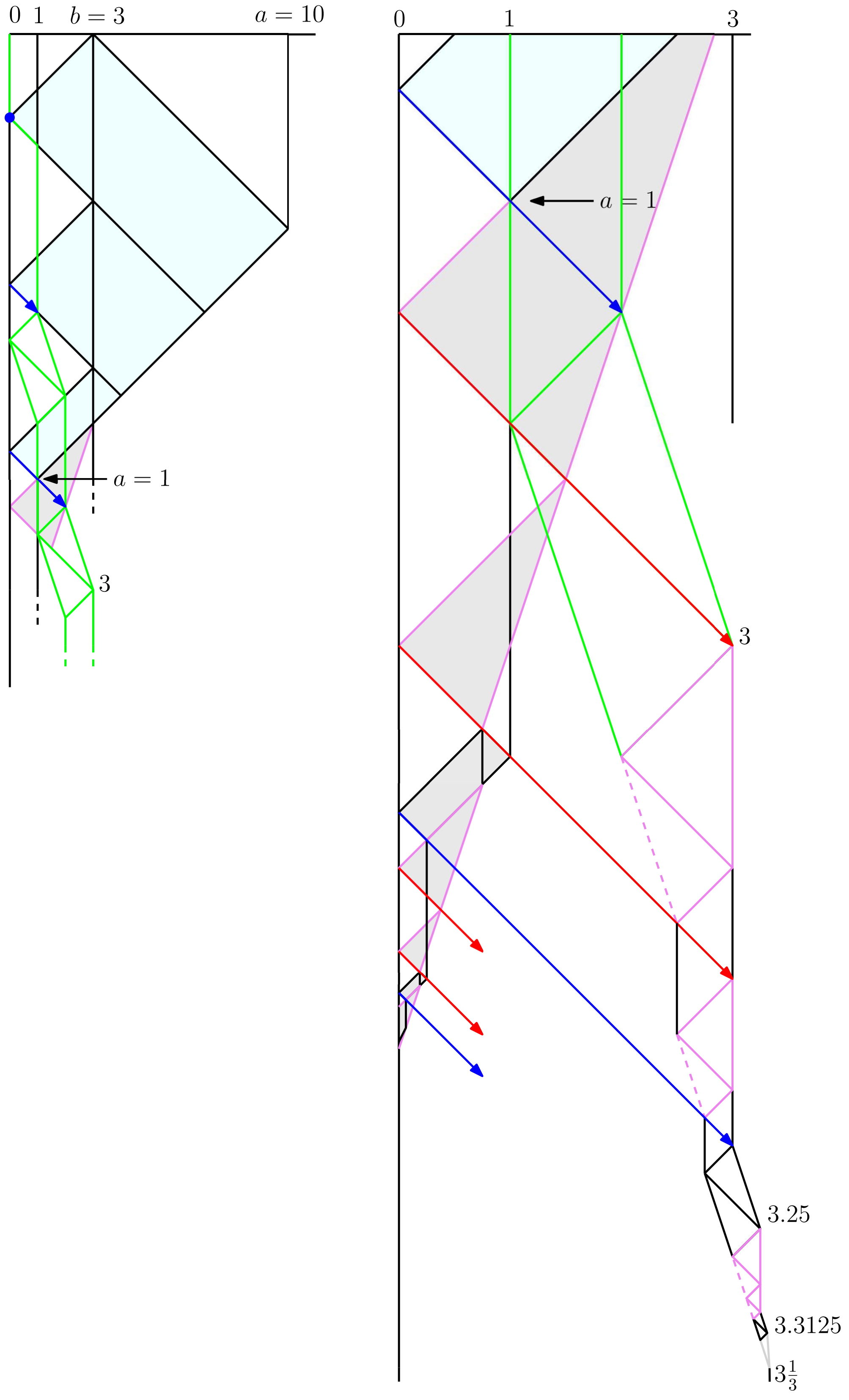


Figure 28: Division (1)

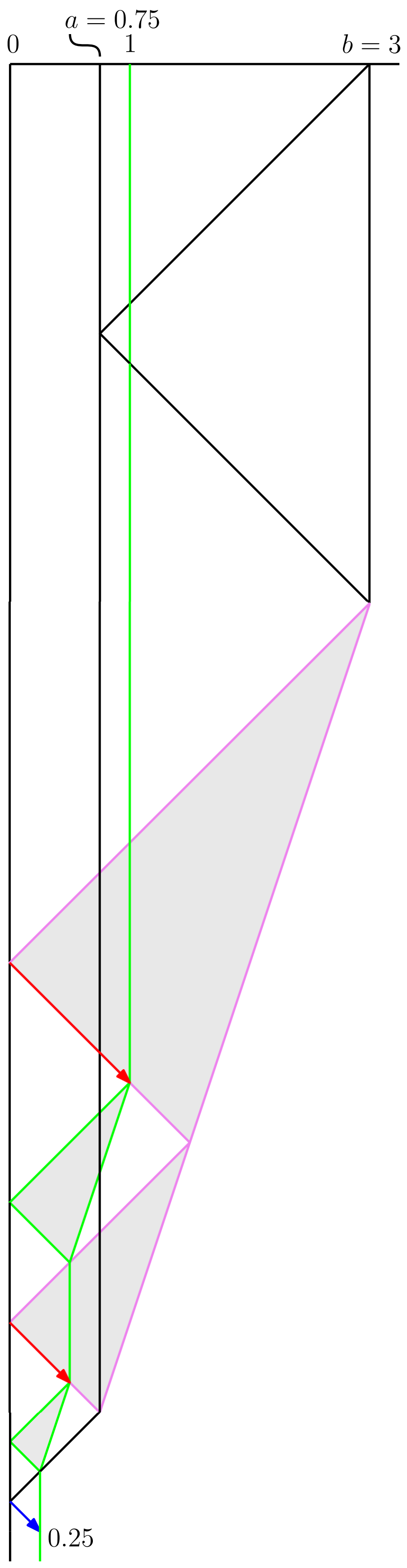


Figure 29: Division (2)

Theorem 8. *The run time of algorithm* 16 *(Division) is* $\mathcal{O}(a + b + \frac{a}{b})$*, if no singularity problem occurs.*

Proof. Analogously to algorithm 14 (*Multiplicative Inverse*), the total run time depends solely on the run time of partial algorithm B.
• Case $a > b$:
Let us first find an upper bound for the run time of A. The first halving of b during the real-valued division starts at the latest at time $2(a - b)$. The total time required for the halvings of b is bounded by $3b$, the total time required for diminishing a (during the real-valued part) is bounded by $b + \frac{b}{2} + \frac{b}{4} + \cdots + \frac{b}{n} < 2b$ for some $n \in \mathbb{N}$. Thus, A terminates earlier than at time $t = 2a + 3b$. Let a signal be sent from A to B at time t. If B processes incoming signals sufficiently fast, this signal reaches B's right boundary earlier than at $t + \frac{a}{b}$, whereupon B requires an insignificant amount of time for the remaining computation. In this case, the total run time is $\mathcal{O}(2a + 3b + \frac{a}{b})$. If B processes incoming signals slower than A sends them, the run time of B for the real-valued part can be estimated by $3\frac{a}{b}$. The left boundary of B leaves 0 at the latest at time $3b + 2$, so in total we get $\mathcal{O}(3b + 2 + 3\frac{a}{b})$. In each scenario, the run time can be described as $\mathcal{O}(a + b + \frac{a}{b})$.
• Case $a < b$:
Note that partial algorithm B is completely enclosed in the interval $[0, 1)$. If $1 < b$, the total run time is approximately the run time of A (plus an $\epsilon > 0$ for the last steps of B after A terminates). The first halving in A occurs at the latest at time $2b$, the following procedure takes at most $5b$ like in the case $a > b$, resulting in a total of $\mathcal{O}(7b)$. If $b < 1$, the total run time can be bounded even stricter by a constant. All in all, the division algorithm has a run time of $\mathcal{O}(a + b + \frac{a}{b})$. □

Algorithm 16′**.** *Improved Division* a/b, $a, b \in \mathbb{R}^+$, $b \neq 0$ *(Fig. 30)*

• Case $a > b$ (*fig. 30*):

Partial algorithm A has a breadth of b and B a breadth of 1 at the beginning of the real-valued division. Since A may calculate faster than B, the singularity problem can occur. We embed A into B similar to the multiplication algorithm, but have to deal with the additional challenge, that B (*green*) may already move to the right before the integer division is completed. For that purpose, B is paused after finishing the processing of the integer division. We can define a stationary signal $\frac{1}{2}$ (*gray dashed*) to mark half of B's gadget's breadth. Partial algorithm A continues its computation until both a and b are to the left of the $\frac{1}{2}$-marking as well as to the left of B. When this is the case, a movement of the signals *Zero*, a and b to the right with speed $\frac{1}{3}$ is initiated (*orange*). The movement ends when *Zero* reaches the left boundary of B. Now A is integrated into B as desired and a reactivation signal can be sent to B (*black dashed*). Afterwards, we proceed just like in the first case of algorithm 15′, when A is already attached to B.
• Case $a < b$:

The execution of this case is close to identical to the multiplication $a \cdot b$ with $a < 1 < b$. The only difference is that the signals *One* and b switch roles. We proceed analogously to algorithm 15′.

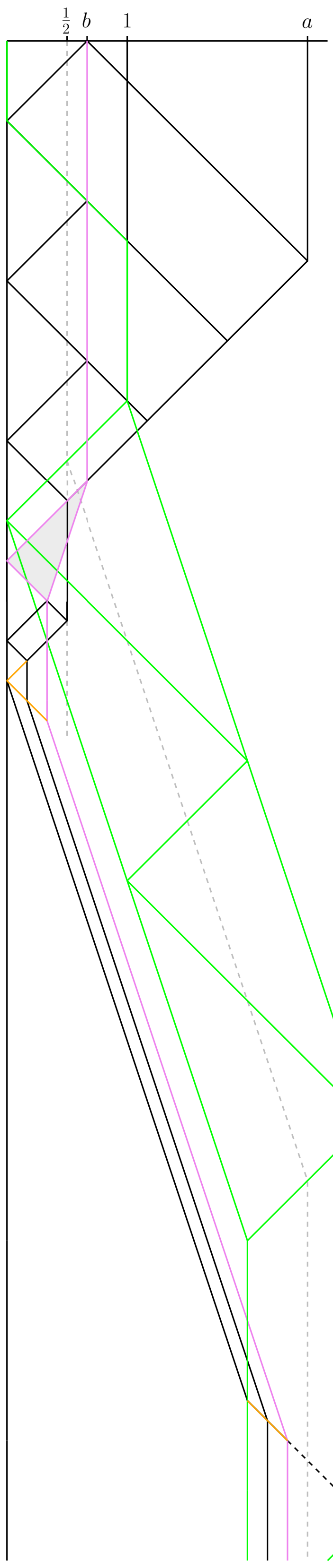


Figure 30: Improved Division

Theorem 9. *Algorithm* 16′ *(Improved Division) is correct and has a run time of* $\mathcal{O}(a+b+\frac{a}{b})$.

Proof. Since the relationship between signal values of A and B has not changed, B still places the result marking at $\frac{a}{b}$. We still need to show that at no point in time are there infinitely many signals. In the case $a < b$, one can argue in the same way as in the proof of correctness for algorithm 15′ (case $a < 1 < b$). Now consider the case $b < a$ for which we can argue in the same way that A only produces finitely many signals up until the moment it pauses and moves towards the right. After A is embedded into B, the two partial algorithms behave like in algorithm 15′, case $a \geq 1$ (but with interchanged names of some signals), so again, at any point in time there exist only a finite amount of signals.
The run times of the altered algorithms only differ in the existence of added pauses which have lengths bounded by $\mathcal{O}(a + b)$. Thus, the total run time is still $\mathcal{O}(a + b + \frac{a}{b})$. □

5.3 Complex Arithmetic Operators

Algorithm 17. *Integer Exponentiation* $x^k, x \in \mathbb{R}^+, k \in \mathbb{N}$ *(Fig. 31)*

Use algorithm 7 (*Binary Representation*) to store k in binary in the signal storage $[0, 0.5]$. This storage will be used as a stack. We will take out bit signals one step at a time. In the case of a 0 bit signal, we square the intermediate result. In the case of a 1 bit signal, we first square the intermediate result and then multiply it with x. This matches the *Exponentiation by Squaring Algorithm.* The initial intermediate result of x sends a fetch signal (*pink*) to the stack. Upon collision with the right-most bit signal inside the stack, the bit signal leaves the stack to the right-hand side and moves to the intermediate result. The fetch signal continues to move left and is destroyed if it collides with another bit signal. If it reaches 0, the stack is empty, and a halting signal is sent to the right. When a bit signal collides with the intermediate result, the squaring and multiplying happens as described in algorithm 15 (*Multiplication*). In the figure, gray triangles represent the rough shape of its partial algorithms and are not to scale. When the calculations induced by a bit signal are completed and a halting signal has not been received, the next fetch signal is sent to the stack. When the halting signal reaches the right boundary of the multiplication process, after completing the calculations induced by the last bit signal, the result is marked as a stationary signal.
The number of multiplications in the exponentiation by squaring algorithm is $\mathcal{O}(\log(k))$. The run time of a multiplication $x \cdot x$ with algorithm 15 is $\mathcal{O}(2x + x^2)$, or linear in the size of the result. Since $\mathcal{O}(x^k + x^{\frac{k}{2}} + x^{\frac{k}{4}} + \cdots) = \mathcal{O}(x^k)$, the total run time is dominated by the last multiplication.

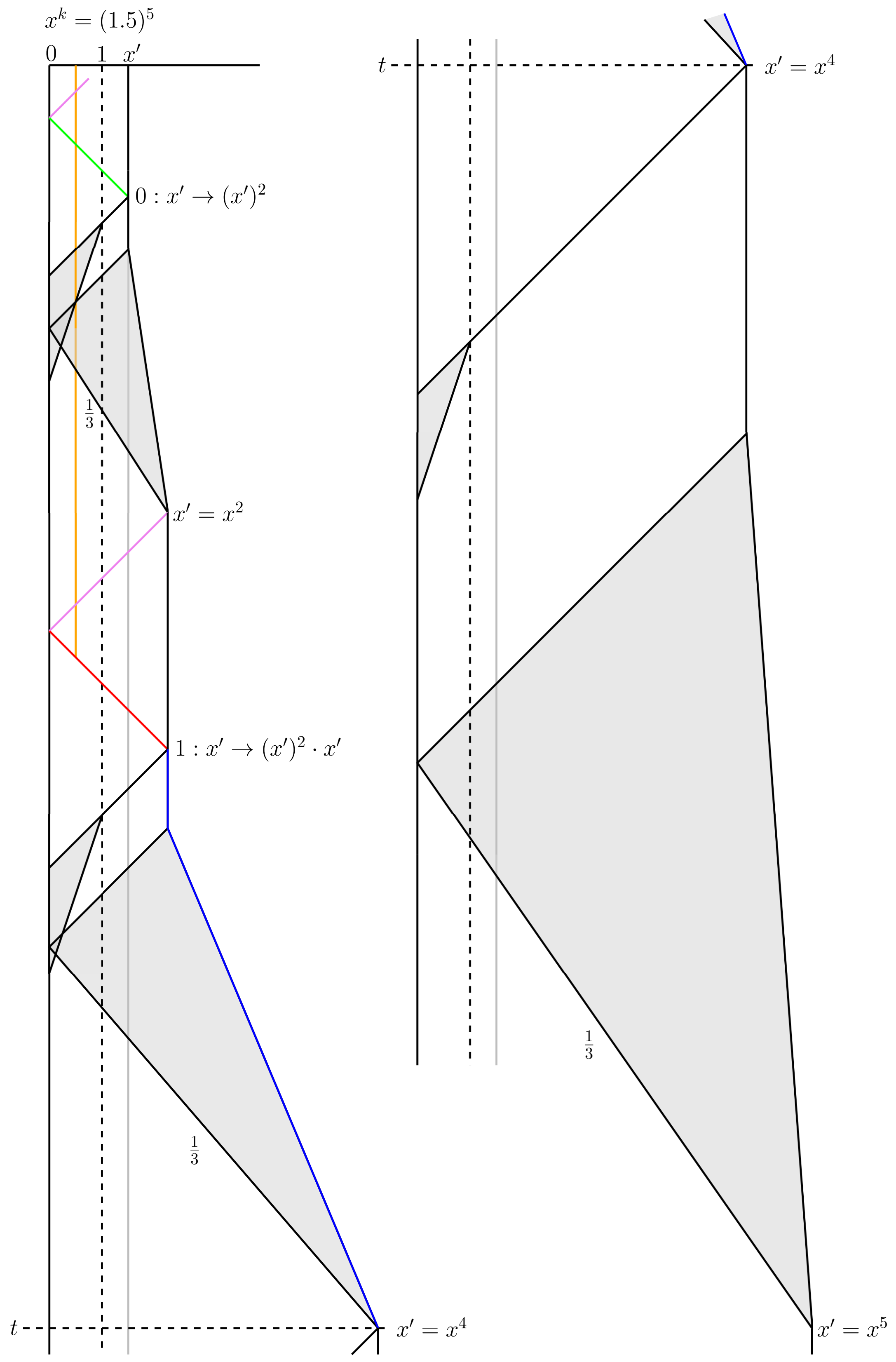


Figure 31: Integer Exponentiation

Algorithm 18. *Factor Decomposition (Fig. 32)*

Given a prime number $f_{max} \in \mathbb{N}$ we define for all prime numbers $f = 2, ..., f_{max}$ the speeds $\alpha_f = \frac{f-1}{2f-1}$. This algorithm finds all prime factors of the input k that are less than or equal to f_{max} and stores them in an ascending order on a stack (see algorithm 4 (*Storing Signals*)). We proceed similarly to algorithm 7 (*Binary Representation*). The right boundary at k now does not send one signal y to the left with speed 1, but a variety of signals with the predefined speeds α_f. If the *red-green* signal, while in the green state, simultaneously collides with a generator signal and a signal with speed α_f, then k is divisible by f. When this happens, the *red-green* signal turns into a *factor-f*-signal (*blue*) and continues to move to the left until it is stored inside the generator. Also, a stationary signal representing $k \longleftarrow k/f$ is created. All other signals with speeds $\alpha_{f'}, f' \neq f$, will just be destroyed upon collision with the *red-green* signal at the wrong time or collision with the *factor-f*-signal. When a factor is found, we use the reset technique from algorithm 7 (*Binary Representation*) and search for a factor of the new value k. If no factor can be found, that is, when either all prime factors are found or remaining factors are larger than f_{max}, the *red-green* signal reaches the 0 and the algorithm terminates.
The run time of the algorithm is the longest when k is a power of two, as this causes the most iterations. It takes the time $\frac{3}{2}k$ to find the first factor and k additional time units until the next iteration with half the size begins. In total, the run time is therefore $\frac{5}{2}k + \frac{5}{4}k + \frac{5}{8}k + \cdots = 5k \sum_i \frac{1}{2^i} \in \mathcal{O}(5k)$.

Algorithm 19. *Finding Common Factors (Fig. 33)*

We use algorithm 18 (*Factor Decomposition*) to find the prime factors $f \leq f_{max}$ of two integers and store them on a stack. On the stack, only factor signals corresponding to the same input integer reflect each other. There are two layers of stacks, so to speak. In the figure, the exemplary integers $84 = 2 \cdot 2 \cdot 3 \cdot 7$ and $30 = 2 \cdot 3 \cdot 5$ are used. Within the stack, the signals are in an inactive state at first (dashed). When an inactive signal reaches the stack's left boundary, it becomes active (solid line). As one integer's factor signals keep reflecting themselves, there is at most one active signal per integer at any given time. When two activated signals (of different integers) collide, their values are compared. If they are not equal, the signal with the lower value is destroyed (*red circles*). If their values are equal, both signals are destroyed and a new signal with this value is created on a third stack layer (*green*). The third layer collects all common factors of the two integers. When a new factor signal is created on the third stack layer, it will move to the stack's right boundary (where it is reflected) without interacting with any other signals. Only from the collision with the boundary onwards will the signal interact with other signals on the third layer, where all of them reflect each other. This simulates entering the stack from the right-hand side. When the *red-green* signal of algorithm 18 (*Factor Decomposition*) reaches the stack, then no more new factors of the corresponding integer will enter the stack. These *no-more-factors* signals are shown in *dark blue* and *yellow* in the figure. When both integer's signals reach the left boundary of the stack, the end of the algorithm is initiated (Fig. 33, *right*). A halting signal, initially in a neutral state (*gray*) moves from the left boundary to the right one. If it collides with one of the

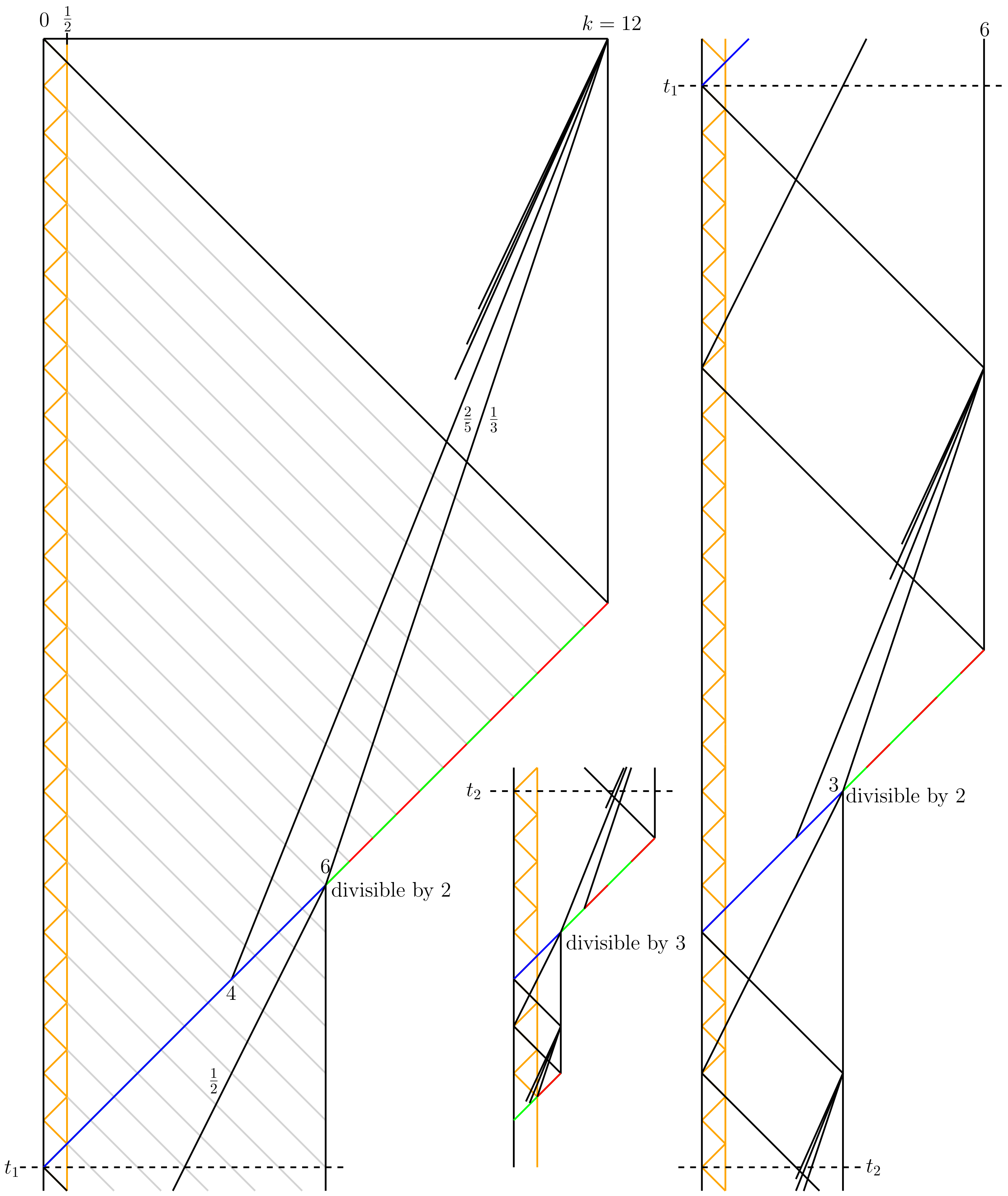


Figure 32: Factor Decomposition

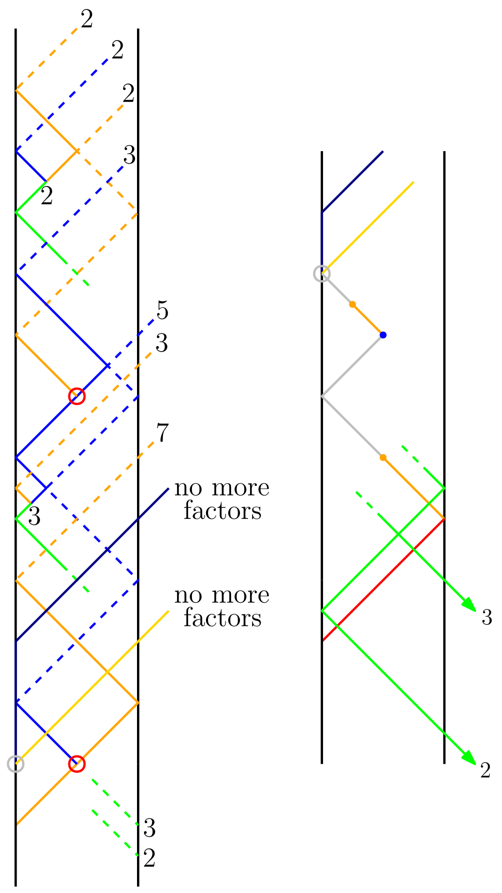


Figure 33: Finding Common Factors

input integer's factor signals while in the neutral state, it switches to a new state corresponding to the integer. In the figure, the neutral signal (*gray*) collides with a factor of the orange integer (*orange dot*) and switches to the "orange state". If the signal then collides with a factor of the other integer (*blue dot*), then the factor comparison has not finished yet. In that case the halting signal moves back to the 0, where it is reflected and switches to the neutral state. When the halting signal reaches the right boundary of the stack, the computation is complete. The halting signal turns into a kill signal (*red*) that moves across the stack to destroy the input integer's factor signals, while releasing the found common factors to the right-hand side of the stack when passing them.
Since the run time of this algorithm is heavily dependent on the breadth of the storage (which can be arbitrarily thin), it is insignificant for other algorithms using it.

Algorithm 20. *Multiplication of Multiple Factors (Fig. 34)*

We use algorithm 19 (*Finding Common Factors*) to find common factors of two integers and now multiply them together to get one large common factor. After releasing the factor signals out of the stack, they need to be turned into markings such that their position corresponds to their value. For this we use one addition gadget per factor (breadth of 1). Since f_{max} is finite, we can define signals to count down the number of required additions to simplify when the gadget can stop adding. Notice, that the computations of markings for higher factors start slightly earlier due to their arrangement in the stack. Signals of different factor's gadgets are not allowed to influence each other so we define gadget signals for each prime number $f = 2, ..., f_{max}$.
One problem remains, as multiple factors can have the same value. In this case only the first of these factors initiates an addition gadget. If another factor of that value reaches this gadget at a later point, it switches to an inactive state and is stored inside the gadget. Analogously to algorithm 14 (*Multiplicative Inverse*), the signal s moving between the boundaries of the gadget creates a stationary signal when colliding with the right boundary and destroys such a signal when colliding with the left boundary, except for when the gadget finishes. In the case of factor f, this leaves the stationary signals $f - 1$ and f. These two markings form a signal storage for all additional signals of factor f. When a factor f is used for a computation, the right-most stored signal can occupy the right boundary of the storage to be used for the next multiplication. If two signals of factor f are required for the computation, another signal can be taken out of the storage. When the storage is empty, its stationary signals can be destroyed.
Back to the main algorithm. Since all factors on the stack from algorithm 19 (*Finding Common Factors*) leave at different times and are sorted, the smallest (*green*) and largest (*red*) factor can be marked. The green factor, when computing the location of its marking finished, begins the multiplication with a factor of the same value or the next in size using algorithm 15 (*Multiplication*). The partial algorithms of the multiplication are hinted at with gray areas in the figure (not to scale). The green mark follows the right boundary of the result computation. If it passes a factor, the mark will be transferred to it as it is the new smallest factor. If the right boundary passes the red mark, the red mark follows the boundary as the result will be the new largest factor. The multiplication can be sped up for smaller factors by precomputing all products less than or equal to f_{max}^2, by defining appropriate signals and speeds. When a multiplication is completed, it sends a signal to the left. When it reaches the green mark, the next multiplication starts. When the green and red mark coincide at the end of a multiplication, the final factor is found and the computation terminates.
As already seen in algorithm 17 (*Integer Exponentiation*), the run time is dominated by the last multiplication, making it linear in the size of the result.

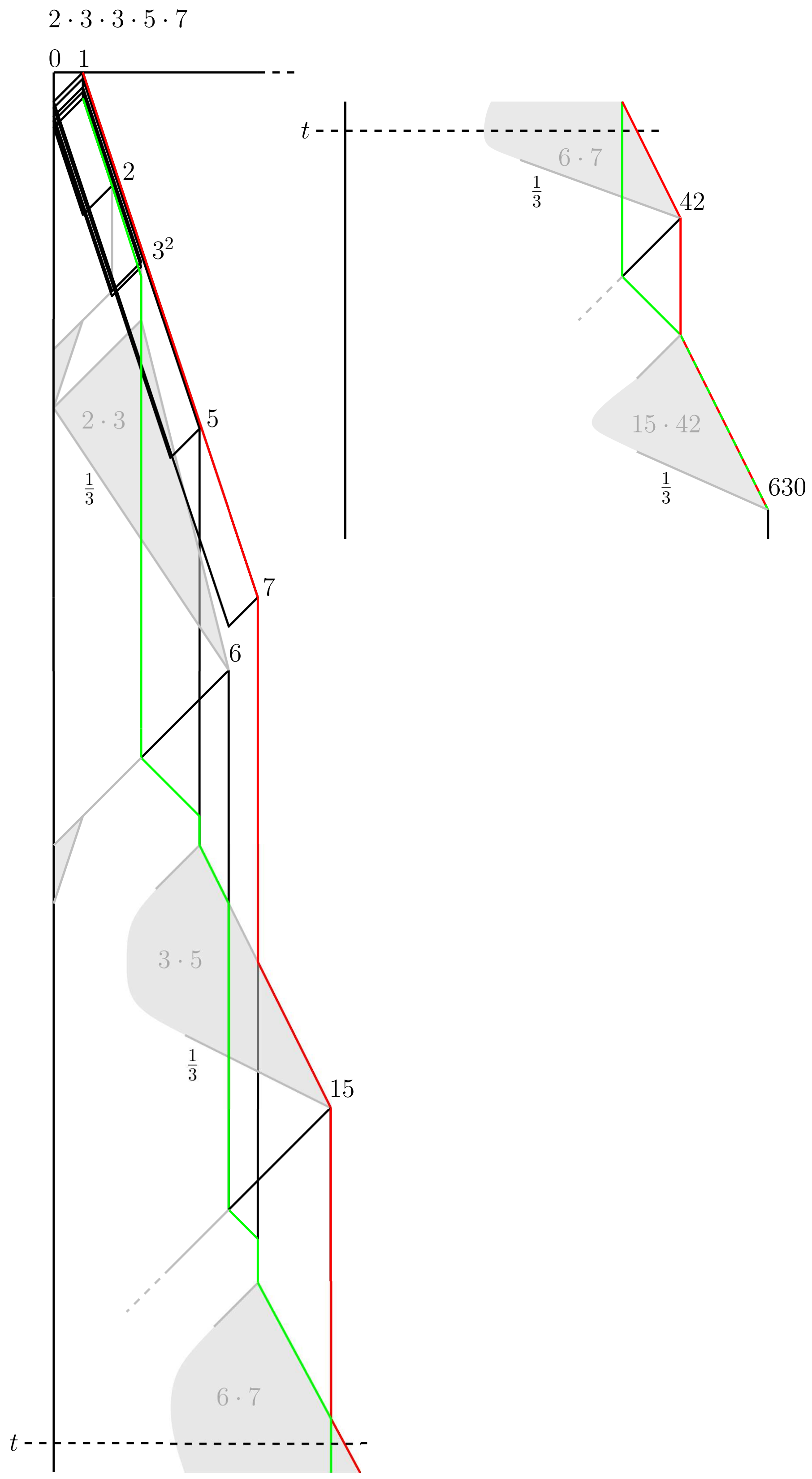


Figure 34: Multiplication of Multiple Factors

Algorithm 21. *Reducing Fractions*

Let u, v be two markings that form the fraction $\frac{u}{v}, u, v \in \mathbb{N}$. Use algorithm 18 (*Factor Decomposition*) to find all prime factors $f \leq f_{max}$ of u and v. Proceed with algorithm 19 (*Finding Common Factors*) to find common factors of the two integers. Then continue with algorithm 20 (*Multiplication of Multiple Factors*) to combine the common factors into one large common factor of u and v. Finally, use algorithm 16 (*Division*) to divide u and v by this factor to receive the fraction $\frac{u'}{v'} = \frac{u}{v}, u' \leq u, v' \leq v$.
Finding the factors takes time $\mathcal{O}(5 \max(u, v))$. The largest possible common factor of u and v is $\min(u, v)$. The final divisions, which are executed in parallel, require the time $\mathcal{O}(u + \min(u, v) + \frac{u}{\min(u,v)})$ for dividing u and time $\mathcal{O}(v + \min(u, v) + \frac{v}{\min(u,v)})$ for dividing v. Let $M := \max(u, v)$ and $m := \min(u, v)$. The total run time of the algorithm is $\mathcal{O}(5M+m+\max(u+m+\frac{u}{m}, v+m+\frac{v}{m})) = \mathcal{O}(5M+2m+\max(u+\frac{u}{m}, v+\frac{v}{m})) = \mathcal{O}(6M+2m+\frac{M}{m}) \subseteq \mathcal{O}(M + m + \frac{M}{m})$.

Algorithm 22. *Square Root (Fig. 35)*

We compute an approximate solution $\sqrt{x}$ through iterative halving of an interval containing the solution. Select the number I of iterations.
Case $x \geq 1$:

Mark the 0 (*orange*) and x (*blue*). Determine the middle of the candidate interval $[0, x]$ (*green*). This value is denoted as the first candidate w_1. Use algorithm 15 (*Multiplication*) to compute w_1^2. The partial algorithms are hinted at by gray areas (not to scale). If $w_1^2 = x$, then $w_1 = \sqrt{x}$ is the exact result. If the multiplication's right boundary reaches x before finishing, then $w_1 > \sqrt{x}$ and the multiplication can be interrupted. The right boundary of the candidate interval (*blue*) takes the position of w_1 and the new middle of the interval represents the next candidate w_2. If the right boundary of the multiplication never reaches x before finishing, then $w_1 < \sqrt{x}$. We set the left boundary of the candidate interval (*orange*) to the position of w_1 and destroy the former marking. Also, w_2 is again computed by determining the middle of the new interval. While checking the candidate w_1, we reduce I by one. If $I > 0$ afterwards, we compute w_2^2 and continue analogously to w_1. We repeat this process until $I = 0$ and return the approximate result w_{I+1}.
Case $x < 1$:

We proceed analogously to the previous case but initialize the right boundary of the candidate interval (*blue*) at 1 instead of x.

Theorem 10. *Algorithm 22 (Square Root) computes $\sqrt{x}$ with an error less than or equal to $\frac{1}{2^{I+1}}|x - \sqrt{x}|$ in the time $\mathcal{O}(Ix + x)$, where I is the number of iterations performed.*

Proof. The run time can be broken down to the three parts halving the interval of candidates, squaring the candidates and signals telling the interval of candidates which half to cut. The halvings require $\frac{3}{2}x + \frac{3}{4}x + \cdots < 3x$ time units in total. Each squaring procedure can be terminated when it leaves the interval $[0, x]$, so we can use an upper bound of

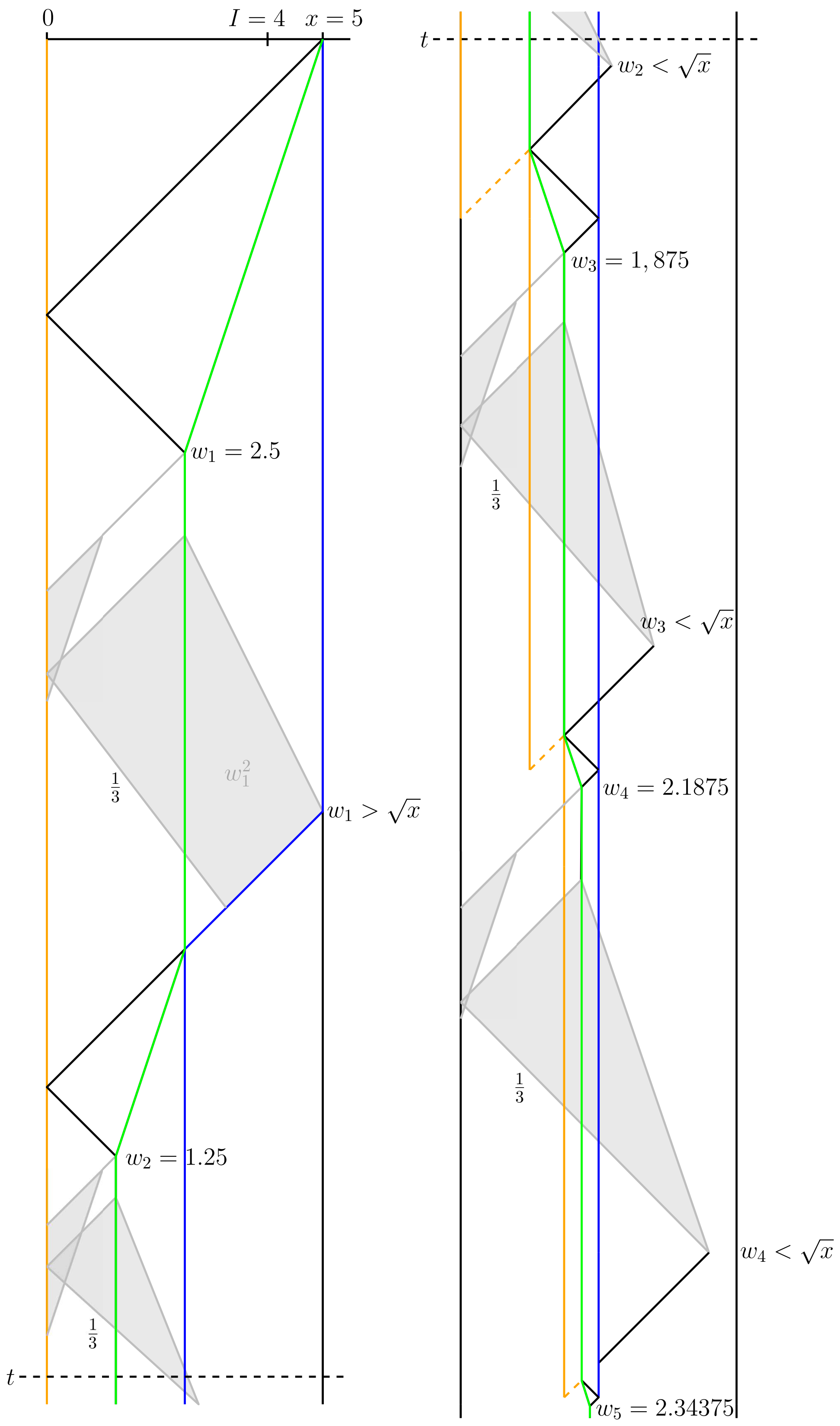


Figure 35: Square Root

$6x$ for each of them. The remaining signals require time less than x per iteration. Since decreasing the number I of remaining iterations happens in parallel to the rest of the algorithm, it is not relevant to the run time. In total, we have $\mathcal{O}(I \cdot 6x + I \cdot x + 3x) \subseteq \mathcal{O}(Ix + x)$. The error of the intermediate results is halved during each iteration and one additional time at the very beginning. After I iterations, the error is therefore less than or equal to $\frac{1}{2^{I+1}}|x - \sqrt{x}|$. □

Algorithm 23. *Rational Exponentiation*

This algorithm computes $x^{\frac{u}{v}} = \sqrt[v]{x}^{u}$, $u, v \in \mathbb{N}$. Reduce the fraction $\frac{u}{v}$ to $\frac{u'}{v'}$ by applying algorithm 21 (*Reducing Fractions*). We compute $\sqrt[v']{x}^{u'}$ as follows. Determine the binary representation of v' using algorithm 7 (*Binary Representation*) to allow a faster exponentiation with the exponent v' (see algorithm 17 (*Integer Exponentiation*)). We proceed analogously to algorithm 22 (*Square Root*), except for exponentiating the candidates w_i with v' instead of squaring them. We use algorithm 17 for the exponentiation but do not need to repeatedly determine the binary representation of v', as we already precomputed it. After calculating $\sqrt[v']{x}$, we use algorithm 17 to exponentiate it with u'. Reducing the fraction takes the time $\mathcal{O}(\max(u, v) + \min(u, v) + \frac{\max(u,v)}{\min(u,v)}) := \mathcal{O}(M + m + \frac{M}{m})$ and the binary representation of v' takes the time $\mathcal{O}(5v')$. As seen in algorithm 22 (*Square Root*), when calculating $\sqrt[v']{x}$, an exponentiation can be terminated when it leaves the interval $[0, x]$, so the run time for the v'-th root is also $\mathcal{O}(Ix + x)$, where I is the number of iterations. The final exponentiation with u' takes the time $\mathcal{O}(\sqrt[v']{x}^{u'})$. In total, we end up with a run time of $\mathcal{O}(M + m + \frac{M}{m} + 5v + Ix + x + \sqrt[v']{x}^{u'})$.

Algorithm 24. *Mantissa Representation*

Let $B \in \mathbb{R}^+$, $B > 1$, be the base for a mantissa representation. If the input $x < B$, $x \in \mathbb{R}^+$, return x. If $x \geq B$, we want to display x in the form of $x = m \cdot B^e$ with mantissa $m \in [1, B)$, base B and exponent $e \in \mathbb{N}$. Let $e = 0$ be marked initially. Use algorithm 16 (*Division*) to repeatedly divide x by B until $x < B$. Increase e by 1 during each division. When $x < B$, set $m \longleftarrow x$. Move the interval $[0, e]$ to the right of m using algorithm 1 (*Moving Intervals*). The result now consists of the markings m and $m + e$.
The run time of the algorithm is dominated by the first division, thus being in $\mathcal{O}(x + B + \frac{x}{B})$.

Algorithm 25. *Logarithm (Fig. 36)*

To determine $\log_B(x)$, we first use algorithm 24 (*Mantissa Representation*) to switch x into its mantissa representation. Now $\log_B(x) = \log_B(m \cdot B^e) = e + \log_B(m)$, where $\log_B(m) \in [0, 1)$. We proceed similarly to algorithm 22 (*Square Root*). Select the number I of iterations. Mark 0 (*orange*) and 1 (*blue*) as the boundaries of the candidate interval and determine the middle k_1 of the interval as the first candidate. We need to test whether $B^{k_1} > m$ or $B^{k_1} < m$ (if they are equal, then w_1 is the exact result). For that we display k_1 as a fraction. Use the auxiliary markings u_i and v_i such that $k_i = \frac{u_i}{v_i}$ during each iteration i. Let $v_i = 2i$ be 2 initially and double it after each iteration. Determine u_i as follows. Let $h_i = \frac{1}{2^i}$ be $\frac{1}{2}$ initially and compute h_{i+1} by halving. Compute $u_i = \frac{k_i}{h_i}$ using algorithm 16 (*Division*). As k_i is now displayed as a fraction, we can use algorithm 23 (*Rational Exponentiation*) to compute B^{k_i}, but can skip the reduction of $\frac{u_i}{v_i}$, as it cannot be reduced further. Analogously to algorithm 22 (*Square Root*), we set the left boundary of the candidate interval (*orange*) to k_i, if $B^{k_i} < m$ or the right boundary (*blue*) to k_i if $B^{k_i} > m$. After I iterations, the approximate result k_{I+1} is returned.
The run time analysis is similar to that of algorithm 22 (*Square Root*). At first, x is transformed into its mantissa representation in $\mathcal{O}(x + B + \frac{x}{B})$. The halvings of the candidate interval take the time $\mathcal{O}(3m)$ and the signals that initiate the halvings require the time $\mathcal{O}(Im)$. The calculation of the rational exponentiation B^{k_i} with $k_i = \frac{u_i}{v_i}$ consists of finding u_i, v_i ($\mathcal{O}(I)$), the binary representation of v_i ($\mathcal{O}(5v_i) = \mathcal{O}(10I)$), the v_i-th root of B ($\mathcal{O}(Im + m)$) and the exponentiation with u_i ($\mathcal{O}(Im + m)$ due to the enclosure in $[0, m]$). All in all, we arrive at a run time of $\mathcal{O}(x + B + \frac{x}{B} + 3m + Im + I \cdot (11I + 2(Im + m))) = \mathcal{O}(x + B + \frac{x}{B} + 3m + Im + 11I^2 + 2I^2m + 2Im) \subseteq \mathcal{O}(x + B + I^2 + I^2x + Ix)$.
Regarding the error δ, when initializing the candidate interval, it is true that $\delta \leq |1 - \log_B(m)|$. Since the interval is halved during each iteration, the error after I iterations is $\delta \leq \frac{1}{2^{I+1}}|1 - \log_B(m)|$.

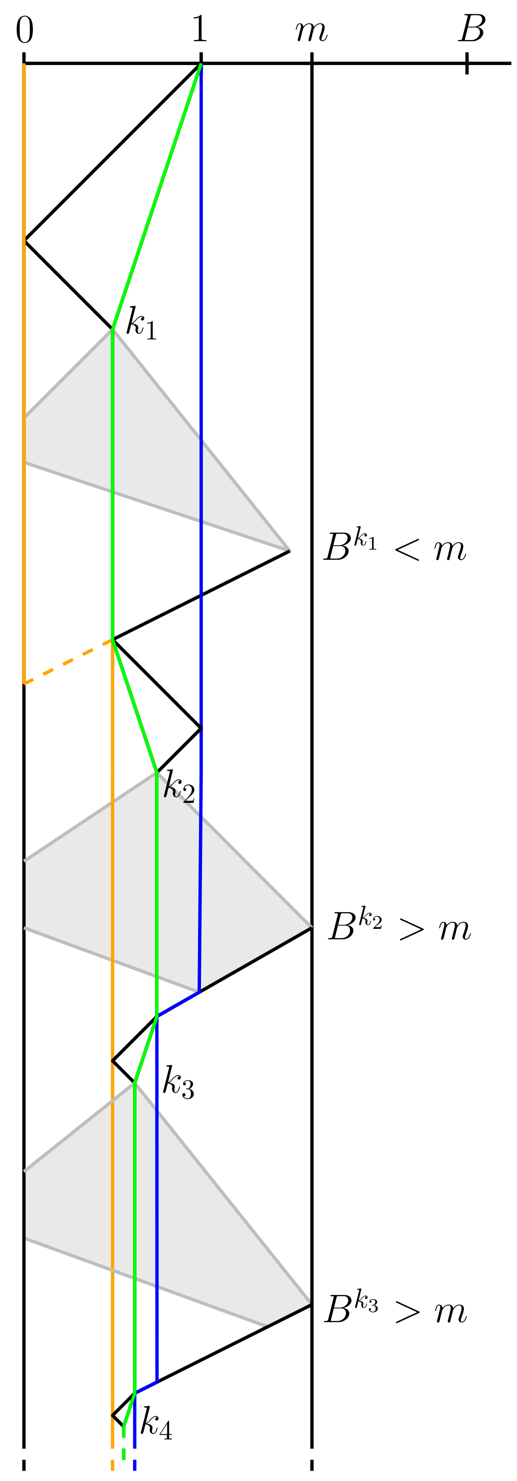


Figure 36: Logarithm

6 Generalized Formal Languages

One important field of theoretical computer science is that of formal languages. This section deals with a generalization of words, giving each symbol a continuous length. Some properties of words become ambiguous under this prospect, like the definition of the palindrome. We provide detection algorithms for two different kinds of generalized palindromes.

Definition 18. *Generalized Words*

Given an input alphabet $A = \{a_1, ..., a_n\}, n \in \mathbb{N}$ a *generalized word* $w = w_1^{l_1} \cdots w_m^{l_m}$ consists of symbols $w_i \in A$ of lengths $l_i \in \mathbb{R}^+$. The value m is called the *discrete length* of w and the value $\sum_{i=1}^{m} l_i$ is called the *continuous length* of w. Elements w_i, w_j of lengths $l_i \neq l_j$ can be treated as distinct symbols, that is, words $u = a_1^1 a_1^2$ and $v = a_1^2 a_1^1$ are considered different even though both their discrete symbols, discrete lengths and continuous lengths are identical.

Algorithm 26. *Mirror Image (Fig. 37)*

Define four signals for each pair (x, y) of possible input symbols: One that moves to the left, one that moves to the right and a reflected version of both. For example, if the input alphabet $A = \{a, b\}$, the possible labels for interval boundaries are aa, ab, ba, bb. At the beginning, each inner interval boundary sends two of the aforementioned signals, one in each direction, which are reflected at the outer boundaries. When two reflected signals with the same label xy collide, a new interval boundary yx is created (*yellow*). If two such reflected signals with the same label collide simultaneously with such an interval boundary, the boundary will be destroyed. When the algorithm starts, the outer boundaries send a signal towards the other side (*black*) which is reflected there and turns into a kill signal for all non-stationary signals. When the kill-signals meet in the middle, they destroy each other and the algorithm terminates. If the input has length n, the algorithm terminates at time $\frac{3}{2}n$.

Theorem 11. *Algorithm* 26 *(Mirror Image) is correct.*

Proof. First, consider one isolated inner boundary. It bisects the input into two intervals. The algorithm then proceeds two swap these intervals in exactly n time. We now need to show that all other signals, that are created, are destroyed again. Consider an arbitrary pair X, Y of inner boundaries which have a distance of Δ to each other. Let the distance between the left outer boundary and X be l and the distance between the right outer boundary and Y be r. The left signal of X moves l units to the left, is reflected and moves a distance l' to the right until an unwanted marking is created. Meanwhile, the right signal of Y moves r units to the right and then a distance r' to the left. At the time $t_1 = l + l' = r + r'$, the collision occurs. We now have a look at the remaining signals.

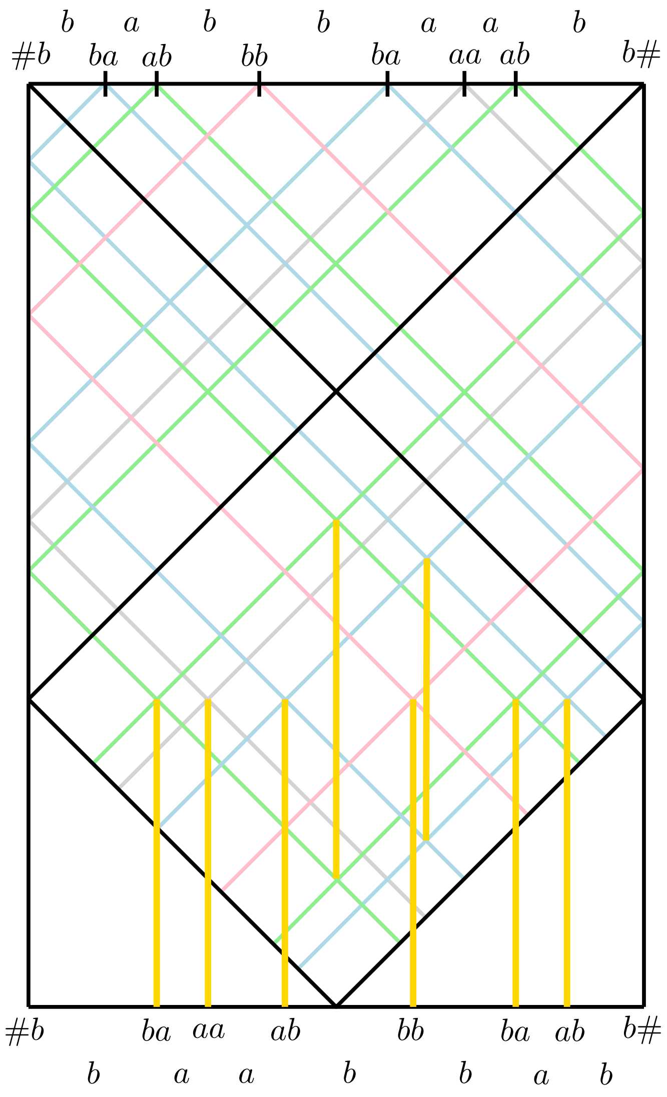


Figure 37: Mirror Image

The left signal of Y moves $l+\Delta$ units to the left, then l'' units to the right, while the right signal of X moves $r+\Delta$ units to the right, then r'' units to the left. Ignoring the first Δ time units of their travel, their pathways are identical to those of the earlier two signals and they meet at the same spot with a delay of Δ at time t_2. It follows that $l' = l''$, $r' = r''$ and $t_2 = l + \Delta + l' = r + \Delta + r'$. For each pair of interval boundaries there exist four relevant collisions. Two of these create the wanted boundaries of the mirror image, the other two cancel each other out as described above. □

Algorithm 27. *Detecting Length-independent Palindromes (Fig. 38)*

This algorithm ignores the lengths of intervals representing input symbols and accepts a word as a palindrome, if the discrete word consisting of the concatenation of the input symbols is a palindrome. For example, the word $a^{1.7}b^{0.9}b^{0.1}a^{1.1}$ is a length-independent palindrome as it turns into the discrete palindrome $abba$. As in algorithm 26 (*Mirror Image*), each interval sends signals in both directions that carry the information about the boundary they originated from. The outer boundaries each send a kill signal towards the middle. All of the signals created at inner or outer boundaries move unhindered in their specified direction until they collide with a kill signal. Whenever two such signals collide, we call that point a *crossing*. A crossing only exists at the moment the signals pass each other.
We now define the notion of *diamond signals*. Initially, each inner interval boundary with the label $xy, x, y \in A, x \neq y$ sends *red* signals in each direction which are reflected at the next crossing. When the reflected red signals collide, they close off a rectangle in the space time diagram which we call a *red diamond*. Also, all inner boundaries with the label $xx, x \in A$ send *green* signals in both directions that behave like red signals. The rectangle in the space time diagram closed off by green signals is called a *green diamond*. Red and green signals are called *diamond signals*. In addition, at each crossing the information carried by the colliding non-diamond signals is compared, unless kill signals are involved. If the signals are carrying mirrored information, that is, they carry the information xy and $yx, x, y \in A$, we call the crossing a *green crossing*. If that is not the case, we call it a *red crossing*. Each red crossing sends red diamond signals. Each green crossing sends green diamond signals, if it is not the end point of a red diamond, and red diamond signals otherwise. In figure 38, on the right side, the purple diamond is a red diamond that originates at a green crossing, which itself is the end point of a red diamond. If the kill signals collide with each other in the end point of a green diamond, the input word is a palindrome. If they collide in the end point of a red diamond, the input word is not a palindrome.

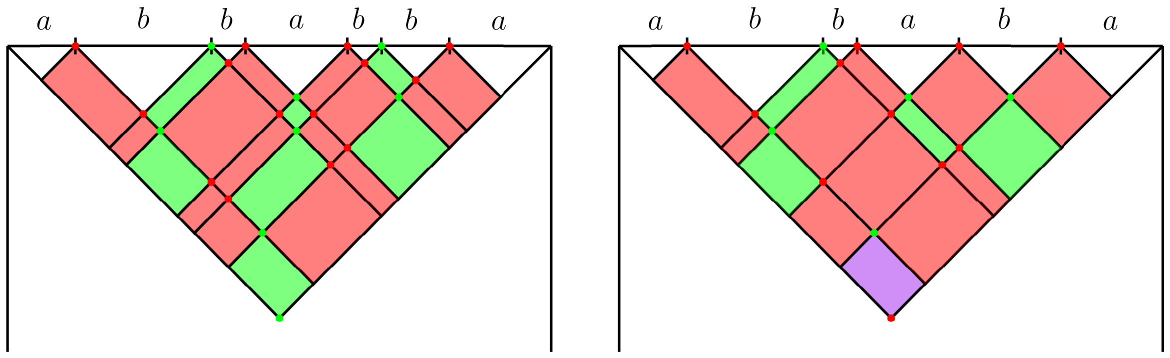


Figure 38: Detecting Length-independent Palindromes

Theorem 12. *Algorithm* 27 *(Detecting Length-independent Palindromes) is correct.*

Proof. We proof by induction, that an input $x_1 \cdots x_n$ is a length-independent palindrome if and only if the last diamond is green. Let $w = x_1 x_2$ be an input. If $x_1 = x_2$, a green diamond is created by construction, if $x_1 \neq x_2$, a red diamond is created by construction. Now consider the crossing at the end of a green diamond and the input $x_1 \cdots x_n$. In the crossing, the signals of the inner boundaries $x_1 x_2$ and $x_{n-1} x_n$ collide. The word $x_2 \cdots x_{n-1}$ is a length-independent palindrome (induction hypothesis). If $x_1 = x_n$, the crossing is green and a green diamond is created. If $x_1 \neq x_n$, the crossing and created diamond are red. Now consider the crossing at the end of a red diamond. The word $x_2 \cdots x_{n-1}$ is not a palindrome (induction hypothesis). Independent from whether $x_1 = x_n$ or not, a red diamond is created. □

Algorithm 28. *Detecting Length-dependent Palindromes (Fig. 39)*

In contrast to algorithm 27 (*Detecting Length-independent Palindromes*), we now only consider input words as palindromes if the input word and its mirror image coincide with both the sequence of their labels, as well as with their interval lengths. For that we modify algorithm 27 as follows. Each inner boundary with the label $xx, x \in A$ and each green crossing that is not the end point of a red diamond, create an additional green stationary signal. If a sole reflected green diamond signal passes this stationary signal, the rectangle in the space time diagram closed off by the diamond signals becomes a red diamond instead of a green diamond. Only if the stationary signal collides with both reflected diamond signals at once, the diamond keeps its green color. The decision whether the input is a palindrome or not is analogous to algorithm 27.

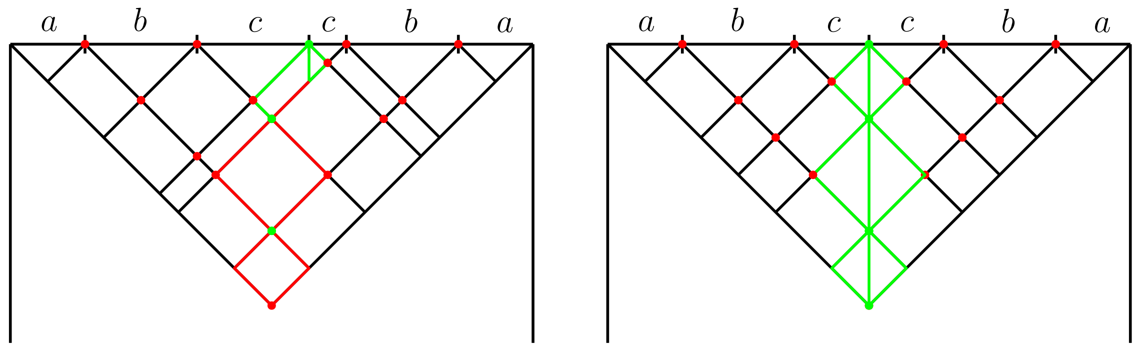


Figure 39: Detecting Length-dependent Palindromes

Theorem 13. *Algorithm 28 (Detecting Length-dependent Palindromes) is correct.*

Proof. We perform an induction analogously to algorithm 27. Inputs which were non-palindromes following algorithm 27 are non-palindromes for this algorithm also. Let $x_1 \cdots x_n$ be an input. Consider the crossing at the end of an emerging diamond in which the signals $s_{1,2}, s_{n-1,n}$ of the inner boundaries x_1x_2 and $x_{n-1}x_n$ collide and which succeeds a green diamond. The word $x_2 \cdots x_{n-1}$ is a length-dependent palindrome (induction hypothesis). We only need to investigate the case $x_1 = x_n$. If the interval lengths $|x_1| \neq |x_n|$, the stationary green signal does not collide with $s_{1,2}, s_{n-1,n}$ simultaneously and the emerging diamond turns red. If $|x_1| = |x_n|$, the three signals meet at the same time and the emerging diamond remains green. Now consider a diamond succeeding a red diamond. The word $x_2 \cdots w_{n-1}$ is not a length-dependent palindrome (induction hypothesis). Independent from the labels x_1, x_n and their lengths, the emerging diamond is red. □

Algorithm 29. *Word Exponentiation w^k, $k \in \mathbb{N}$ (Fig. 40)*

Let $w = w_1w_2 \cdots w_n, w_i \in A$ and $k \in \mathbb{N}$ be the input for the algorithm and let *Zero* and *One* be markings initially. When we calculate $k \longleftarrow k - 1$ using algorithm 11 (*Subtraction*), we use *Zero* instead of 0 and *One* instead of 1. This way, we can move the location of the computation and shorten the distances some signals need to travel.
If $k = 0$, the left boundary at 0 sends a kill signal to the right, that destroys all markings corresponding to w, as well as the *Zero*, *One* and k-signal.
If $k = 1$, the left boundary at 0 sends a kill signal to the right, that only destroys the *Zero*, *One* and k-signal.
If $k \geq 2$, we start by computing $k \longleftarrow k - 1$ as described above. When the new value is determined, a signal s is sent to the left which is reflected at *Zero* and is destroyed upon collision with w's right outer boundary. When s is reflected at *Zero*, *Zero* begins to move to the right with the speed $1 - \epsilon, \epsilon \in (0, 1)$. When s collides with *One* (or k), *One* (or k) also moves to the right with the speed $1 - \epsilon$. At the reflection of s, another signal l is created that moves to the right with the speed $1 - \epsilon$ and which carries the information $\#w_1$. When s collides with an inner boundary or the right outer boundary of w, a copy of that boundary is created and moves to the right with the speed $1 - \epsilon$. When the *Zero* signal

collides with the right outer boundary of w, it becomes stationary and sends a signal t with speed 1 to the right. Since l lays directly on *Zero*, at the time of that collision, l is destroyed and the marking $w_n\#$ is replaced with $w_n w_1$. When signal t collides with a signal with speed $1 - \epsilon$, that signal becomes stationary again. Destroy t when it collides with the right outer boundary of w. Note, that the right outer boundary is also moving to the right during the exponentiation. When t collides with *One*, the next computation of $k \longleftarrow k - 1$ is initiated. As long as the result is $k > 1$, continue analogously to the previous iteration. However, after the first iteration, l carries the label $w_n w_1$ instead of $\# w_1$. If $k = 1$, *Zero*, *One* and k are destroyed and the copy of w produced afterwards is the last one. Essentially, this algorithm is a repeated application of algorithm 1 (*Moving Intervals*), applied to every interval of w.

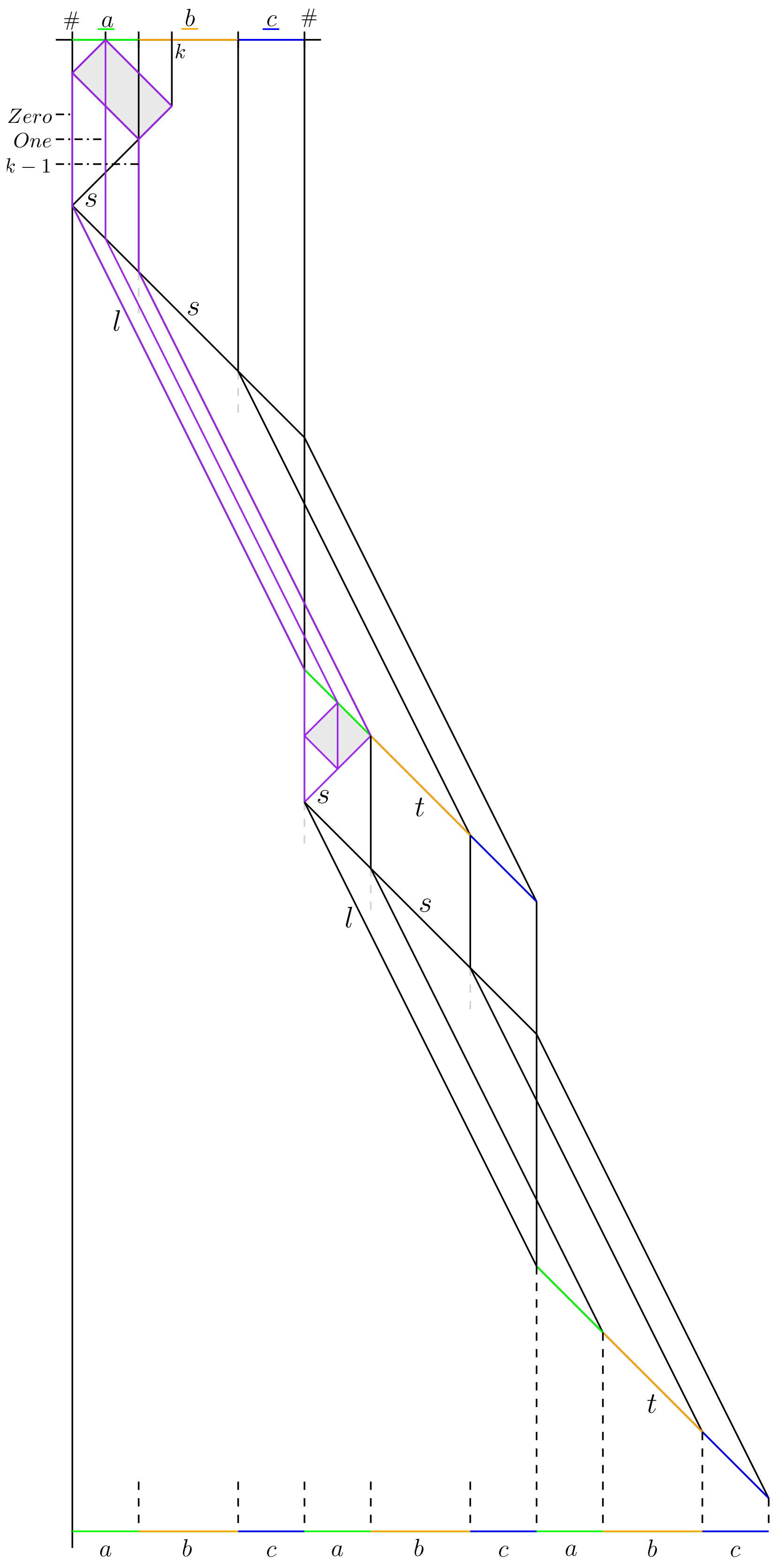


Figure 40: Word Exponentiation

7 Numbers in Generalized Bases

Regarding numbers as the lengths of intervals leads to the issue that the lengths grow linearly with the value of the number, making large numbers cumbersome in this representation. For that reason, we introduce an analogy to the positional notation of decimal and binary numbers, but with a real-valued base B. We restrict the definition to bases $B \geq 2$, as smaller values can lead to undesired behaviors in certain algorithms (an explanation is presented at the end of this section).

Definition 19. *Interval Number to the Base $B \geq 2$*

An *Interval Number* $R = r_0, ..., r_n, n \in \mathbb{N}$, to the base $B \in \mathbb{R}, B \geq 2$, consisting of the intervals $r_i, i = 0, ..., n$, has the value $\bar{R} = \sum_{i=0}^{n} r_i \cdot B^i$ and the length $|R| = \sum_{i=0}^{n} |r_i|$. Interval numbers are not allowed to consist of an infinite amount of intervals r_i, even if their value $\bar{R}$ is finite, as it can result in an infinite run time for some algorithms.

We denote intervals I with $\bar{I} = |I|$ as *unary* or *uncompressed*. Following the definition above, an interval number $I^\star$ to the base B can be found, such that $\bar{I}^\star = \bar{I}$ and $|I^\star| << |I|$. In the following, we also use $|I|$ to describe intervals which have the length $|I|$. The context should make it clear, whether $|I|$ describes a length or an interval. A problem of the definition above is that for each interval I there exists an infinite number of different interval numbers R_j to the base B, such that $\bar{I} = \bar{R}_j$. For example, the unary interval r with $|r| = 6$, can be described in base $B = 2.5$ with the intervals $r_0 r_1$ where $|r_0| = 2.5, |r_1| = 1.4$, or the intervals $r'_0 r'_1 r'_2$ where $|r'_0| = 0.5, |r'_1| = 1, |r'_2| = 0.48$ (see fig. 41). To map each unary interval to a unique interval number to the base B, we introduce a standardized form, the so called *canonical (interval) number.*

Definition 20. *Canonical (Interval) Number*

Given a unary interval I or an interval number I to the base B, the corresponding *canonical (interval) number* is $I' = i'_0, ..., i'_n, n \in \mathbb{N}$, where $|i'_j| = B$ for $j = 0, ..., n-1$ and $|i'_n| \leq B$, such that $\bar{I}' = \bar{I}$.

In figure 41, the interval number in the middle is the canonical representation of the unary interval r, as all but the last intervals are filled up to length B.

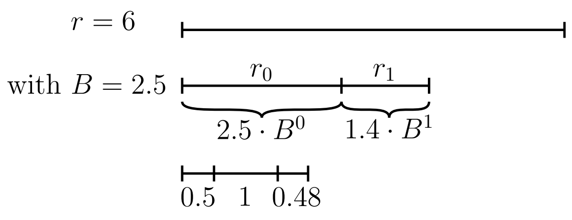


Figure 41: Interval Number

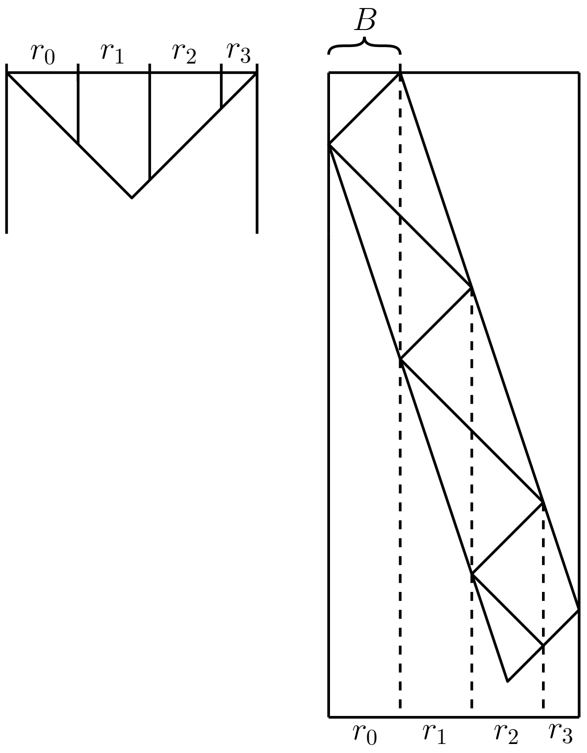


Figure 42: Canonical Interval Numbers: Removing and Inserting Boundaries

Algorithm 30. *Canonical Interval Numbers: Removing and Inserting Boundaries (Fig. 42)*

A canonical number to the base *B* is distinctly defined by its length. Therefore, inner interval boundaries can be removed and inserted at will. To remove all inner boundaries, create kill signals at the outer boundaries that move towards the middle and destroy all inner boundaries and each other. To insert the inner boundaries again, utilize an addition gadget with breadth *B*, that creates a stationary signal after each complete addition. When the gadget's right boundary reaches the number's right boundary, the gadget is destroyed.

Algorithm 31. *Canonicalization (Fig. 43, 44, 45, 46)*

Let $R = r_0, ..., r_n$ be an interval number to the base B. We want to transform R into the canonical number $R' = r'_0, ..., r'_m$, such that $|r'_i| = B, i = 0, ..., m-1, |r'_m| \leq B$. The process of canonicalization consists of two phases.
First Phase (Fig. 43):

In the first phase, we start by dividing r_0 by B and moving the result to r_1 to create a larger interval of significance B^1. Afterwards, that interval will also be divided by B and moved to r_2 to create a larger interval of significance B^2. We continue until, after n iterations, R is transformed into one sole interval $r^\star$ of significance B^n. During the i-th iteration, $i = 1, ..., n$, we also compute $\frac{B}{B^i}$ and align it with the left boundary of the current configuration of R. Starting from that left temporary boundary, the intervals $|B^{1-i}|, |B^{1-i+1}|, ..., |B^1|$ are aligned next to each other. After the n iterations, we computed $r^\star$ with $\bar{r^\star} = \bar{R}$ and if the sequence of the $|B^\alpha|, \alpha = -n+1, ..., 1$, is shorter than $|r^\star|$ we perform more iterations $k = n+1, ..., j$ until we reach a new interval $r^\star$ with

$$|r^\star| \leq \sum_{\alpha=-j+1}^{1} |B^\alpha|$$

Second Phase (Fig. 44):

In the second phase we use the B^α to divide $r^\star$ into m intervals and replace each interval of length $|B^\alpha|, \alpha \leq 0$, with an interval of length B. The overlapping part of $|B^1|$ and $r^\star$ forms the interval r'_m of the canonical number, which can be shorter than B. First, we mark the left boundary of $r^\star$ as *Zero* and with distance of B to the right of *Zero*, create the marking $\tilde{B}$. The interval $[Zero, \tilde{B}]$ will be moved to the left, beyond r'_m, using algorithm 1 (*Moving Intervals*). The left boundary of $|B^0|$ is the first signal left of r'_m to collide with a movement signal and it will follow the movement signal upon collision. It is then reflected at *Zero* and moves back to the left boundary of r'_m. The *Zero* follows the first movement signal and both will become stationary at the left boundary of r'_{m-1}, which is the right-most interval of length B. When *Zero* reaches that left boundary, signals s and t are created. Signal s will be stationary forever, while t follows an interval movement's second signal. The execution of the following iterations $i = 2, ..., j$ is as follows. When the moving left boundary of $|B^{1-i}|$ collides with the left boundary of r'_m, a copy of the interval $[s, \tilde{B}]$ moves to the left beyond r'_{m+1-i}. The left marking of $|B^{1-(i+1)}|$ follows the first movement signal of the copied interval, occupies it and is reflected at s. That first movement signal, which reduces its speed to $1-\epsilon$ when passing s, drags *Zero* with it until the end of the interval movement. The second movement signal, which increases its speed to 1 when passing t, drags t until the end of the interval movement. When the interval movement initiated by the left boundary of $|B^{1-j-1}|$ finishes, R' is constructed and can be moved as a whole such that its left boundary is at 0.

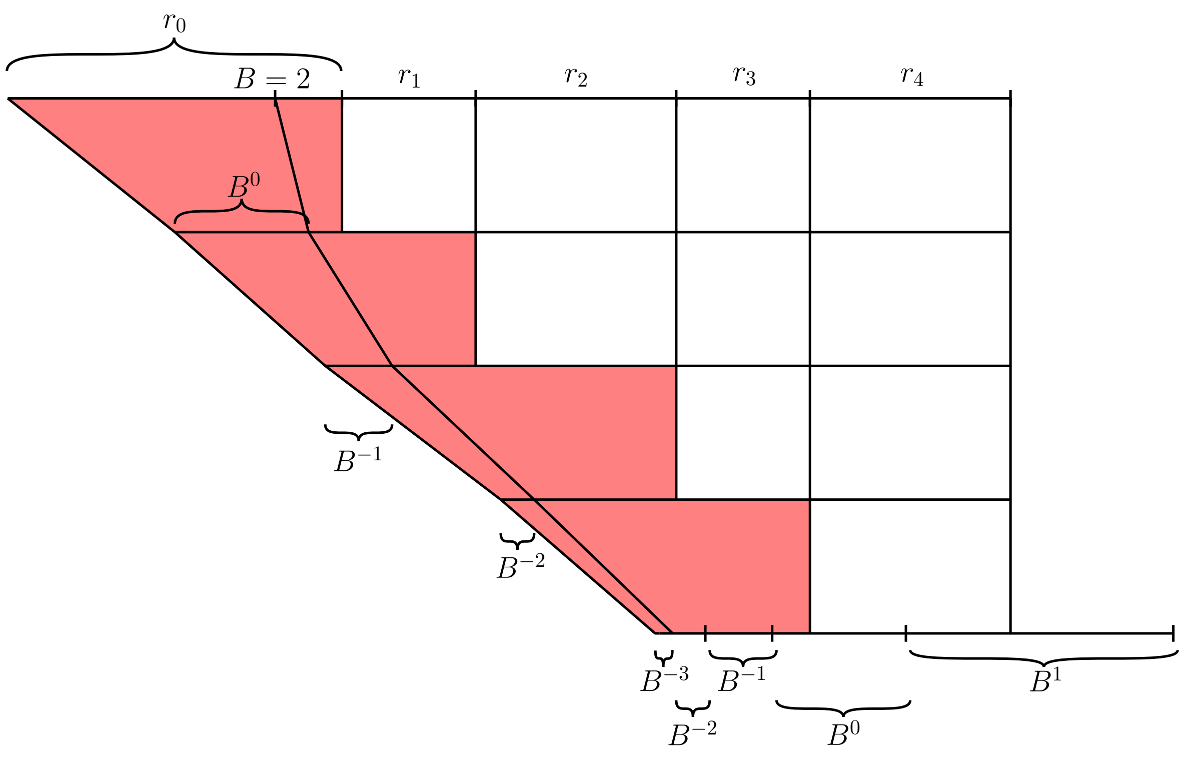


Figure 43: Canonicalization First Phase

Detailed description of an iteration in the first phase (Fig. 45, 46):

We first define the following term Γ_i.

$$\Gamma_i = \frac{\frac{\frac{\frac{r_0}{B}+r_1}{B}+r_2}{\dots} + r_{i-1}}{B}$$

Initially, *Zero* (*purple*), *One* (*blue*) and *Base* (*green*) are marked. At the beginning of iteration i, $|B^{1-i}|$ (*red, light blue*) and Γ_i (*orange*) are computed in parallel using algorithm 16 (*Division*). The interval $[Zero, \Gamma_i]$ is moved to r_i (*gray box*), such that $\Gamma_i + r_i$ is ready for the next iteration. The left movement signal, upon collision with *Zero*, creates a signal f that moves to the right until it reaches the right boundary of $|B^1|$, which is at the end of the sequence $|B^{1-i}|, |B^{1-i+1}|, ..., |B^0|, |B^1|$. Additionally, after the movement of Γ_i, a stationary marking g (*orange*) is created at its left boundary. Signal f initiates a movement of all colliding signals to the right, including *Zero* but excluding g and the boundaries of $r_{i+1}, ..., r_m$. When the moving *Zero* signal collides with g, it becomes stationary, destroys g and sends a signal h with speed 1 to the right, as well as a signal e with the slower movement speed $1 - \epsilon$ to the right. The signals *One*, *Base* and $|B^{1-i}|$ become stationary when they collide with h. Signal e increases its speed to 1 when colliding with $|B^{1-i}|$. All remaining moving signals of $|B^{1-i+1}|, ..., |B^1|$ become stationary when colliding with e. This way, we arrange the sequence $|B^{1-i}|, ..., |B^1|$ to the right of *Zero*. The next iteration can start with a slight offset: as soon as $|B^{1-i}|$ becomes stationary, the computation of $\frac{|B^{1-i}|}{Base} = |B^{1-i-1}|$ can begin. As soon as *Base* becomes stationary, the computation $\frac{\Gamma_i + r_i}{Base}$ can begin.

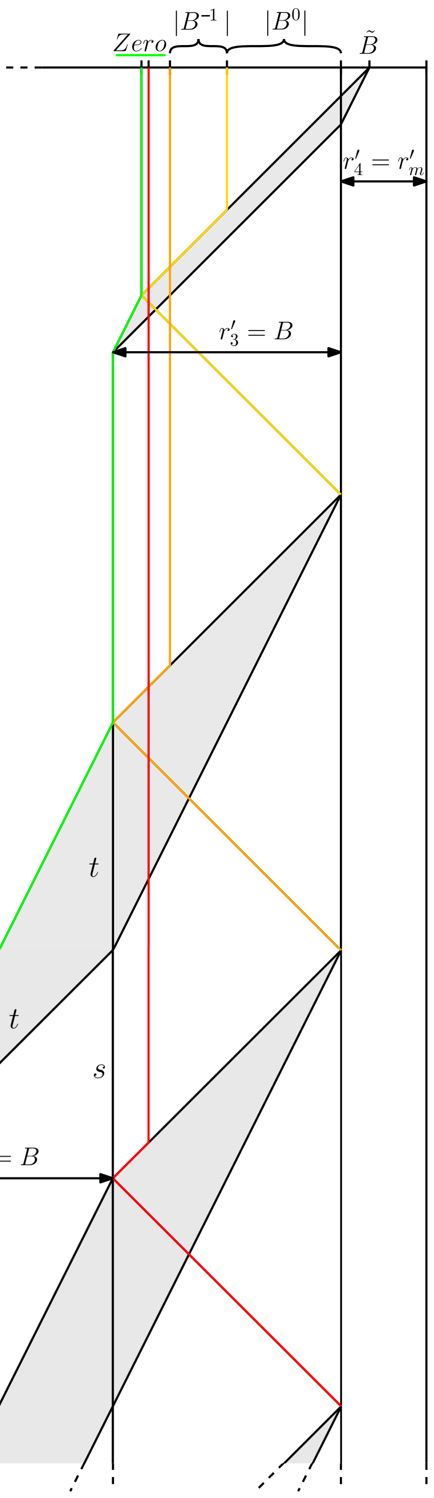


Figure 44: Canonicalization Second Phase

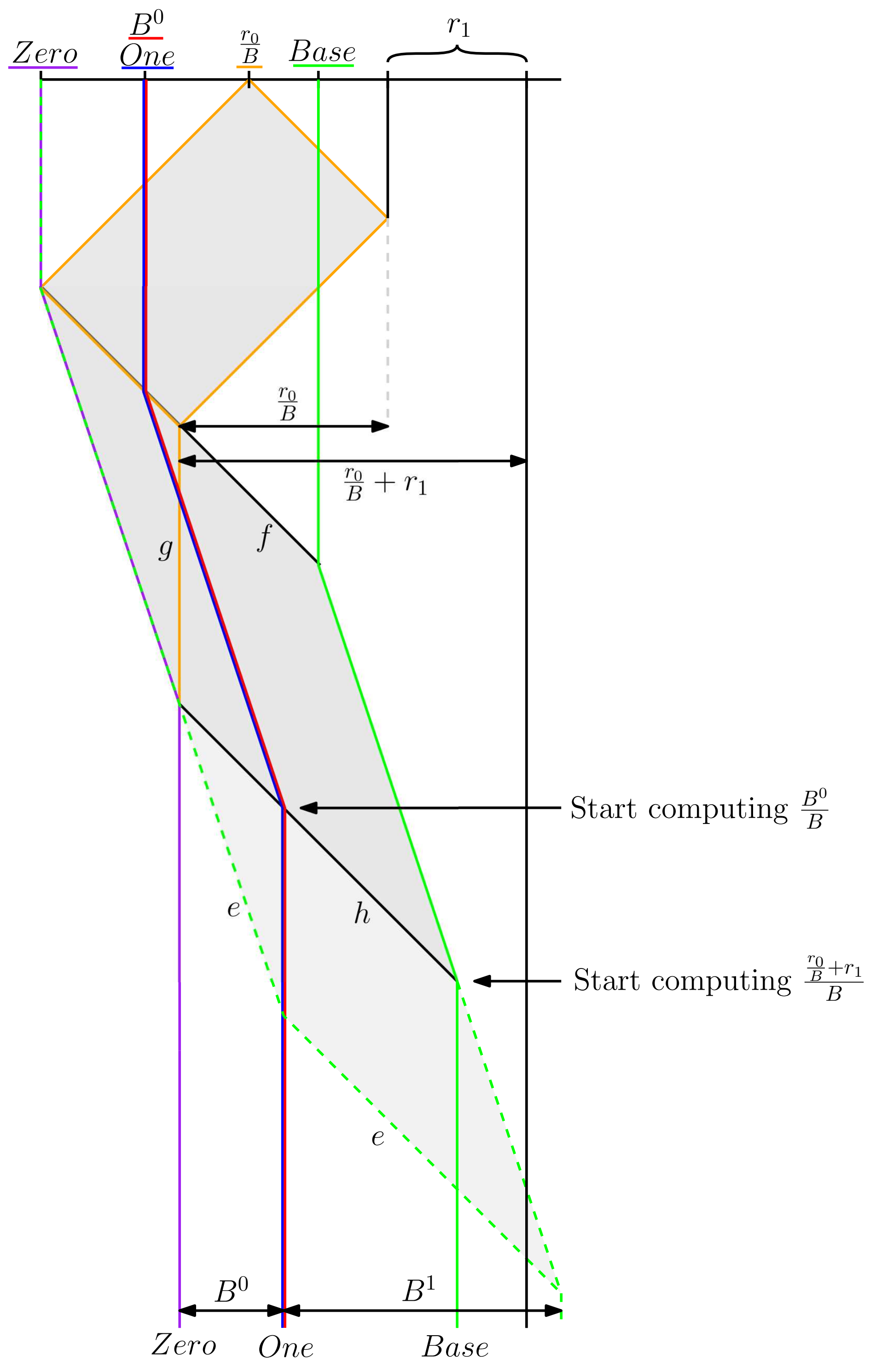


Figure 45: Canonicalization First Phase Iteration (1)

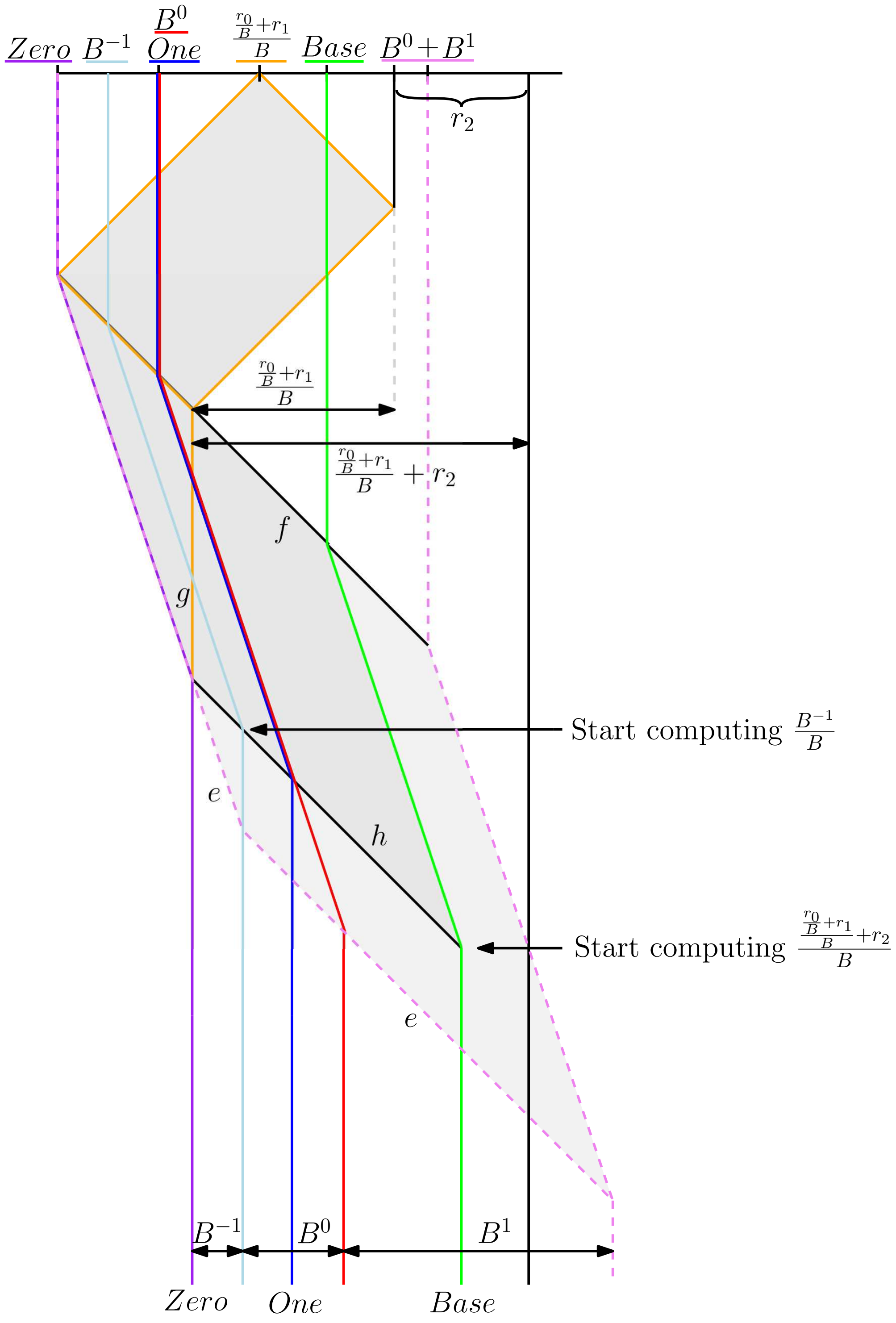

B^0
$Zero$
B^{-1}
One
$\frac{\frac{r_0}{B}+r_1}{B}$
$Base$
B^0+B^1
r_2
$\frac{\frac{r_0}{B}+r_1}{B}$
$\frac{\frac{r_0}{B}+r_1}{B}+r_2$
f
g
Start computing $\frac{B^{-1}}{B}$
e
h
Start computing $\frac{\frac{\frac{r_0}{B}+r_1}{B}+r_2}{B}$
e
B^{-1}
B^0
B^1
$Zero$
One
$Base$

Figure 46: Canonicalization First Phase Iteration (2)

Theorem 14. *Algorithm* 31 *(Canonicalization) transforms the input R into a canonical number R', such that $\bar{R} = \bar{R'}$.*

Proof. The first phase of the algorithm performs the following conversion.

$$\begin{aligned}
\bar{R} &= r_0 B^0 + r_1 B^1 + \cdots + r_n B^n \\
&= (\frac{r_0}{B} + r_1)B^1 + r_2 B^2 + \cdots + r_n B^n \\
&= (\frac{\frac{r_0}{B} + r_1}{B} + r_2)B^2 + r_3 B^3 + \cdots + r_n B^n \\
&= \cdots = (\frac{\frac{\frac{\frac{r_0}{B}+r_1}{B}+r_2}{\cdots} + \cdots + r_{n-1}}{B} + r_n)B^n \\
&= \frac{\frac{\frac{\frac{\frac{r_0}{B}+r_1}{B}+r_2}{\cdots}+\cdots+r_{n-1}}{B} + r_n}{B^{j-n}} B^j = \bar{r^\star}
\end{aligned}$$

After the first phase we arrive at $r^\star$ with $\bar{R} = \bar{r^\star}$. The second phase then performs the following conversion.

$$\bar{r^\star} = (B^{-j+1} + B^{-j+2} + \cdots + B^0 + (|r^\star| - \sum_{i=-j+1}^{0} B^i))B^j$$

$$= B \cdot B^0 + B \cdot B^1 + \cdots + B \cdot B^{j-1} + (|r^\star| - \sum_{i=-j+1}^{0} B^i) \cdot B^j = \bar{R'}$$

$$\text{with } (|r^\star| - \sum_{i=-j+1}^{0} B^i) \leq B \text{ by construction}$$

All in all, we get a canonical number R' with $\bar{R} = \bar{R'}$. □

Theorem 15. *Turning the interval number $R = r_0, ..., r_k$ to the base B into the canonical number $R' = r'_0, ..., r'_{k'}$ using algorithm* 31 *(Canonicalization) takes the time $\mathcal{O}(\max(|R'|, k \cdot \max(B, |R|))$.*

Proof. Each of the first phase's k iterations consists of three parts. The divisions $\frac{\Gamma_i + r_i}{B}$ and $\frac{B^\alpha}{B}$, which happen in parallel, and the movement of intervals in preparation for the next iteration. The division $\frac{\Gamma_i + r_i}{B}$ requires the time $\mathcal{O}(\Gamma_i + r_i + B + \frac{\Gamma_i + r_i}{B})$, the division $\frac{B^\alpha}{B}$ requires the time $\mathcal{O}(B^\alpha + B + \frac{B^\alpha}{B}) = \mathcal{O}(B)$, since $B^\alpha \leq B$. The movement of intervals takes the time $\mathcal{O}(\sum B^\alpha) \subseteq \mathcal{O}(|R|)$. Since $\Gamma_i + r_i \leq |R|$, the total run time of the first phase is $\mathcal{O}(k \cdot \max(B, |R|))$. Now onto the second phase. The movements of the copies of the filled intervals start with a delay of $2B$ between each and require the time $\mathcal{O}(k'B) \subseteq \mathcal{O}(|R'|)$ each, as well as in total. The final alignment of the canonical number with the 0 requires the time $\mathcal{O}(|R'|)$. All in all, the run time of the canonicalization algorithm is $\mathcal{O}(\max(|R'|, k \cdot \max(B, |R|)))$. □

Regarding a unary number $r = \sum_{i=1}^{k} B^i$ and its canonical form r', the ratio of their required spaces is as follows.

$$\frac{r'}{r} = \frac{kB}{\sum_{i=1}^{k} B^i} = \frac{k}{\sum_{i=0}^{k} B^i} = \frac{k}{\frac{B^k-1}{B-1}} = \frac{k(B-1)}{B^k-1}$$

The length of the canonical number is roughly the logarithm of the unary number's length.

Algorithm 32. *Basic Arithmetic with Canonical Numbers*

Let $R_1 = r_{1,0}, r_{1,1}, ..., r_{1,n}$ and $R_2 = r_{2,0}, r_{2,1}, ..., r_{2,m}$ be canonical numbers of base B and $R_1 \leq R_2$ without loss of generality. The left boundaries of both numbers are at 0.

(*i*) *Addition* $R_1 + R_2$

Analogously to the first phase of algorithm 31 (*Canonicalization*), transform R_1 into one interval $r_1^\star$ of significance B^m. Move $r_1^\star$ to the right, such that its left boundary coincides with the left boundary of $r_{2,m}$ and for the next step, treat this boundary as if it was *Zero*. Add up $r_1^\star$ and $r_{2,m}$ using algorithm 9 (*Addition*). If the resulting interval is longer than B, divide the interval into one interval of length B and one interval c with the remaining length. Divide c by B and treat the left boundary of c as *Zero* for the division using algorithm 16 (*Division*).

(*ii*) *Subtraction* $R_2 - R_1$

Compute $r_1^\star$ analogously to (*i*) and subtract it from $r_{2,m}$. If $r^\star > r_{2,m}$, a negative number is created to the left of the temporary *Zero*. Treat this interval $\tilde{c}$ as a positive number, multiply it with B and subtract the result from $r_{2,m-1}$.

(*iii*) *Multiplication* $R_1 \cdot B$

Copy the interval $r_{1,0}$ and move all intervals $r_{1,0}, ..., r_{1,n}$ to the right by B beyond the copy of $r_{1,0}$. The resulting interval number has the value $|R_1| \cdot B + B$, so we need to use (*ii*) to subtract B from it.

(*iv*) *Division* R_1/B

Move the intervals $r_{1,1}, ..., r_{1,n}$ to the left by B to reach 0. The original interval $r_{1,0}$ is destroyed. The resulting interval number has the value $|R_1|/B - B$, so we need to us (*i*) to add B to it.

(*v*) *General Multiplication and Division*

Compute $r_1^\star$ and $r_2^\star$ of significance B^k such that both numbers only consist of one interval each. Use these intervals as input for algorithms 15 (*Multiplication*) and 16 (*Division*) and use algorithm 31 (*Canonicalization*) to turn the result into a canonical number. Note, that through the calculation of $r_1^\star$ and $r_2^\star$, the sequence $|B^{-k}|, ..., |B^1|$ is already present and can be reused for the canonicalization.

The run times of (*i*),...,(*v*) are dominated by the required conversion following the first phase of the canonicalization and thus lie in $\mathcal{O}(k \cdot \max(B, |R_i|))$.

Algorithm 33. *Compressing Canonical Numbers Further (Fig. 47)*

This algorithm replaces the filled up intervals of length B with intervals of length $\epsilon \in (0, \frac{B}{4}]$. Initially, ϵ and B are marked and the left boundary l of r_n "knows" that to the right of it lies the last interval of the canonical number $R = r_1, ..., r_n$. The interval $[0, \epsilon]$ is used as an addition gadget. Each inner interval boundary of R sends a signal to the left that is interpreted by the gadget as an addition signal, analogously to algorithms 14, 15, 16, but with an entrance from the right-hand side. The left signal of l does not lead to another addition but becomes a stationary signal s_1 when colliding with the gadget and sends a signal t to the right. In the figure, ϵ has its maximum value so the signal collides with the gadget exactly at the time it finishes its computation. Additionally, l sends a signal to the right at the beginning, which is reflected at the right boundary of R and destroys the boundary. When the reflected signal collides with t, both are destroyed and a stationary signal s_2 is created.
For decompression, the addition gadget from above repeats its addition process until its right border reaches s_1. At the beginning of the process and after each complete addition except for the last one, an addition signal is sent to the right that is collected by another addition gadget with a breadth of B. This second gadget places a stationary signal after each addition. When the gadget finishes its computation, the interval $[s_1, s_2]$ can be moved behind the last stationary signal it created.

The definition of interval numbers allows the use of empty intervals, for example $R = \sum_{i=0}^{k} r_i B^i, r_i = 0$ for $i = 0, ..., k-1, r_k > 0$. If each empty interval is represented by a stationary signal, the processing of a sequence of empty intervals requires an infinite amount of collision rules, one of for each possible length of the sequence. We present a more elegant representation of empty intervals by allocating a constant amount ϵ of space which consists of as many intervals as the sequence of empty intervals is long. Choose ϵ smaller than the length of the smallest non-empty interval of the interval number. As an example, let $\bar{R} = 2 \cdot B^0 + 0 \cdot B^1 + 0 \cdot B^2 + 0 \cdot B^3 + 0 \cdot B^4 + 1 \cdot B^5$. Ignoring the empty intervals, this interval number consists of the intervals $[0, 2]$ and $[2, 3]$. We now add another interval $[2, 2+\epsilon], \epsilon < 1$. To encode the four empty intervals, we subdivide this interval into the subintervals $[2, 2+\frac{1}{2}\epsilon], [2+\frac{1}{2}\epsilon, 2+\frac{3}{4}\epsilon], [2+\frac{3}{4}\epsilon, 2+\frac{7}{8}\epsilon], [2+\frac{7}{8}\epsilon, 2+\epsilon]$, or in the general case at position x with n empty intervals $[x + (1-\frac{1}{2^i})\epsilon, x + (1-\frac{1}{2^{i+1}})\epsilon]$ for $i = 0, ..., n-2$ and $[x + (1-\frac{1}{2^{n-1}})\epsilon, x+\epsilon]$. Since we forbid an infinite sequence of empty intervals, the empty intervals can be processed one after the other without any singularities. When an algorithm reaches the point $x + \epsilon$, it has to move back to x, as this is where the next non-empty interval begins.

There are other possible canonical representations of interval numbers than the one we described in definition 20 and some may be more useful in some contexts than others. We now describe two other possibilities.

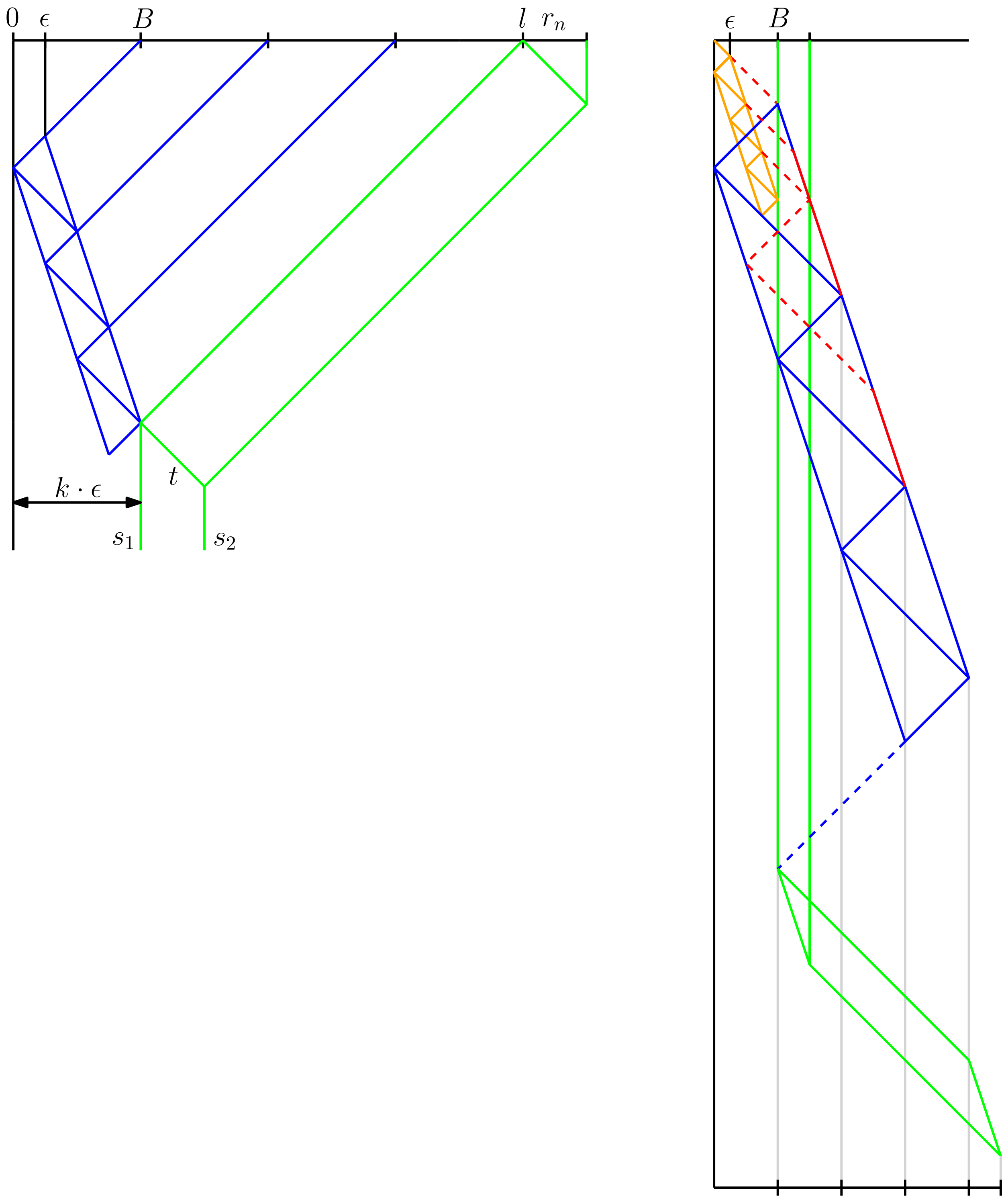


Figure 47: Compressing Canonical Numbers Further

Definition 21. *Squared Canonical (Interval) Number to the Base $B \geq 2$*

Let $\bar{R} = \sum_{i=0}^{n} r_i B^i$ be the value of an interval number to the base B. The *squared canonical (interval) number R''* corresponding to R has the form $\bar{R}'' = \sum_{i=0}^{m} r_i'' B^{2i}$ with $|r_i''| = B^{1-i}, i = 0, \cdots, m-1, |r_m''| \leq B^{1-m}$ and $\bar{R} = \bar{R}''$.

Given a squared canonical number R'' to the base B, it is true that $|R''| \leq \sum_{i=-\infty}^{1} B^i = \sum_{j=-1}^{\infty} \frac{1}{B^j} = B + 1 + \sum_{j=1}^{\infty} \frac{1}{B^j} = B + 1 + \frac{1}{B-1}$, so R'' can be displayed in a constant amount of space independent from its value $\bar{R}''$. If R is already canonical, the new coefficients can be calculated by $r_i'' = \frac{r_i'}{B^i}$. A non-canonical number can be transformed into a squared canonical number similarly to algorithm 31 (*Canonicalization*) by performing divisions by B^2 instead of B.

Definition 22. *Minimal Canonical (Interval) Number to the Base $B \geq 2$*

Let $\bar{R} = \sum_{i=0}^{n} r_i B^i$ be the value of an interval number to the base B. The *minimal canonical (interval) number R^-* corresponding to R has the form $\bar{R}^- = \sum_{i=0}^{m} r_i^- B^i$ with $|r_i^-| = 0, i = 0, \cdots, m-1, 0 < |r_m^-| \leq B$, so $\bar{R}^- = r_m^- \cdot B^m = \bar{R}$. The empty intervals are encoded in the interval $[0, \epsilon]$ as described above.

We now give an explanation why we only allow bases $B \geq 2$ for interval numbers. When adding two numbers, one expects the sum to have at most one more digit than the larger of the two numbers. Thus, when adding two canonical interval numbers, we do not want to have the effect, that two or more intervals of higher significance are created. Consider two canonical numbers consisting of $k+1$ filled-up intervals of length B. When adding up the two numbers, a new interval of significance B^{k+1} is created. We forbid that this carryover is larger than $B \cdot B^{k+1}$ as this would lead to the creation of yet another interval of significance B^{k+2}. We therefore stipulate that $\sum_{i=0}^{k} B \cdot B^i \leq B \cdot B^{k+1}$.

$$\sum_{i=1}^{k+1} B^i = \frac{B^{k+2}-1}{B-1} - 1 = \frac{B^{k+2}-B}{B-1} = \frac{B^{k+2}-B}{B^{k+2}-B^{k+1}} B^{k+1} \leq B^{k+2}$$

$$\Longleftrightarrow \frac{B^{k+2}-B}{B^{k+2}-B^{k+1}} \leq B$$

$$\Longleftrightarrow B^{k+2} - B \leq B^{k+3} - B^{k+2}$$

$$\Longleftrightarrow B^{k+3} - 2B^{k+2} + B \geq 0$$

$$\Longleftrightarrow B^{k+2} - 2B^{k+1} + 1 \geq 0$$

$$\Longleftrightarrow (B-2)B^{k+1} \geq -1$$

$$\Longleftrightarrow B - 2 \geq -B^{-(k+1)}$$

It is apparent, that the requirement is fulfilled for $B \geq 2$, independent from k. If $B < 2$, there exist numbers with k intervals that do not fulfill our requirement.

8 Sorting Intervalls

Sorting numbers is a popular problem in computer science as having a sorted input can speed up a large number of algorithms. Comparison-based sorting algorithms like Quick Sort or Heap Sort cannot be faster than the time $\mathcal{O}(n \log n)$, while sorting integers is possible in $\mathcal{O}(n)$ with algorithms such as LSD Sort. In this chapter we describe how to sort intervals by lengths and exploit the properties of real numbers to compress large inputs of intervals such that the time required to sort the compressed input becomes infinitesimally small.

Definition 23. *Red Activation Signal*

When a *red activation signal* reaches a separating marking between two intervals, a comparison of the intervals starts using algorithm 34 (*Interval Comparison*), if the adjacent intervals have not been compared before and the marking is still in its initial position.

Definition 24. *Green Activation Signal*

When a *green activation signal* reaches a separating marking between two intervals, a comparison of the intervals starts using algorithm 34 (*Interval Comparison*).

Both red and green activation signals are destroyed when colliding with the signals of a comparison that close off a rectangle in the space time diagram.

Algorithm 34. *Interval Comparison (Fig. 48)*

We compare the lengths of two neighboring intervals x and y, where x is to the left of y, and want to have the larger interval to be at the right side afterwards. The marking m between x and y sends a signal a to the left and a signal b to the right. Both are reflected at the other boundaries of x, y. The collision point of the reflected signals describes the marking between x, y if they switched places. If $|x| = |y|$, m collides with both reflected signals at once. Since no permutation is necessary, red activation signals are sent to both sides. If $|x| < |y|$, the reflected signal a passes m. At the time of the collision, a red activation signal is sent to the left and when a and b collide, both are destroyed and a red activation signal is sent to the right. If $|x| > |y|$, signal b passes m. The marking m will be replaced by a proxy signal $\tilde{m}$, which may be necessary for a reset of the permutation (see algorithm 35 (*Bilateral Interval Sort*)). When a and b collide, a new marking m' is created that separates the interchanged intervals x, y. Additionally, green activation signals are sent in both directions. When the right one collides with the proxy signal $\tilde{m}$, then $\tilde{m}$ is destroyed.

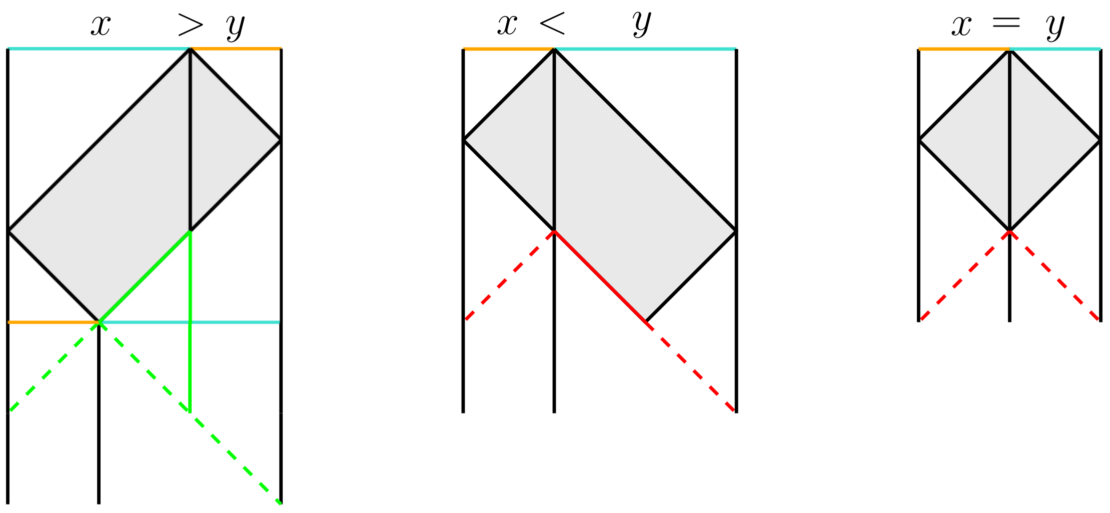


Figure 48: Interval Comparison

Algorithm 35. *Bilateral Interval Sort (Fig. 49, 50, 51, 52, 53, 54)*

The input is a sequence of unsorted intervals and we want to sort them by length in an ascending order. In the beginning, both outer boundaries send red activation signals towards the middle. Upon collision with inner boundaries, the adjacent intervals are compared using algorithm 34 (*Interval Comparison*) and are potentially interchanged. If no permutation is necessary, red activation signals are sent to both sides, so only neighboring intervals are compared that definitely have not been compared before. If a permutation is necessary, green activation signals are sent to both sides to compare the neighboring intervals. The algorithm ends when only stationary signals remain. Alternatively, an ending signal can be used like the one in algorithm 36 (*Hourglass Compression*).
Due to the parallelism of the comparisons, a number of conflicts can arise, which we now solve. Consider the case, that the rectangles in the space time diagram of two adjacent comparisons overlap. In figure 49, left side, the large rectangle overlaps the red one. In the example, the overlap begins at the upper red circle. The right comparison's signal a is reflected in the center red circle, leading to an error. The two left-most intervals are supposed to be interchanged but a's reflection occurs before the permutation finishes, thus creating intervals of wrong lengths. Additionally, a green activation signal collides with a proxy signal (lower red circle) which is not allowed to start a new comparison, as it does not describe the actual interval boundary. To avoid such problems, we give priority to the left comparison and reset the right one. Figure 49, right side, shows the same scenario, but when the comparison rectangles begin to overlap, the right comparison is interrupted and reset. The right comparison's first attempt is shown in blue. When the comparison signals collide, a reset signal (*orange*) is sent to the right that initiates a new comparison when reaching the marking between the two intervals on the right-hand side. Even though the rectangles are now disjoint, a problem remains, as the left comparison's proxy signal does not reach the corresponding green activation signal (red

circle). If the left comparison has to be reset after that point, the proxy signal would be necessary to recreate the original intervals. It follows, that the right comparison has to wait until the left comparison finishes its computation and destroys its proxy signal if one was created. Let us now have a look at the proxy signal's purpose. In figure 50, left side, we see the reset from above that did not require a proxy signal to work properly. This is the case when the length v of the right interval is larger or equal to the distance u between the interval boundary and the collision point of the rectangles. However, if $u > v$ and we do not use a proxy signal, the error shown in figure 50, middle, occurs. If the right comparison leads to a permutation of its intervals and we just remove the old boundary, the reset signal cannot find it anymore (red circle). If we create the proxy signal, though, the reset signal can find the original interval boundary location, replace the proxy signal with the boundary signal and attempt the comparison again (see fig. 50, right side). Let us now consider the scenario, where the right comparison collides with the proxy signal of the left comparison (see fig. 51, right side, red circle). Since we cannot be certain that the left comparison will not be interrupted, the "right yields for left" rule also applies to the proxy signal, that is, the right comparison resets upon colliding with the left comparison's proxy signal. If the left comparison does not lead to a permutation of its intervals, there is no issue with the rectangles having direct contact (see fig. 51, left side). An exemplary input that leads to a collision with a proxy signal can be seen in figure 52.
Figures 53, 54 show a complete execution of the sorting algorithm.

Theorem 16. *The run time of algorithm* 35 *(Bilateral Interval Sort) for an input of length n consisting of k intervals is $\mathcal{O}((2k-1)n)$.*

Proof. Consider two intervals a, b that are to be compared and which have the lengths $|a|, |b|$. If $|a| \leq |b|$, the intervals are not interchanged and the comparison ends after the time $t \leq \frac{3}{2}(|a| + |b|)$ (equality when $|a| = |b|$), when the red activation signals reach the neighboring boundaries. If $|a| > |b|$, the intervals need to be interchanged. Due to the use of proxy signals the comparison takes longer and ends after the time $t < 2(|a|+|b|)$ (closer to the maximum for $|b| \approx 0$). A worst case input of length n with k intervals consists of an interval n_1 of length $|n_1| \approx n$ at the very left and $k-1$ intervals $n_2, ..., n_k, |n_i| > 0$, to the right of it. The algorithm has to compare and interchange n_1 with the other $k-1$ intervals, which takes $\mathcal{O}(2n)$ each. At the end, n_1 is compared with its neighboring interval once more in the time n, but the two are not interchanged. The run time is therefore $\mathcal{O}((k-1)2n + n) = \mathcal{O}((2k-1)n)$ in total. □

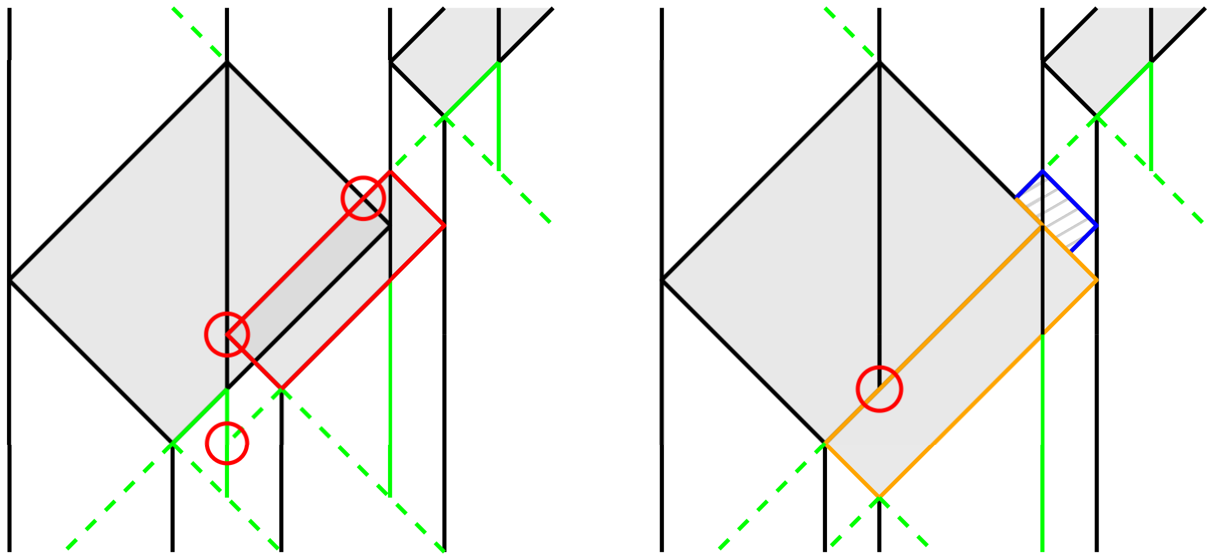


Figure 49: Comparison Conflict (1)

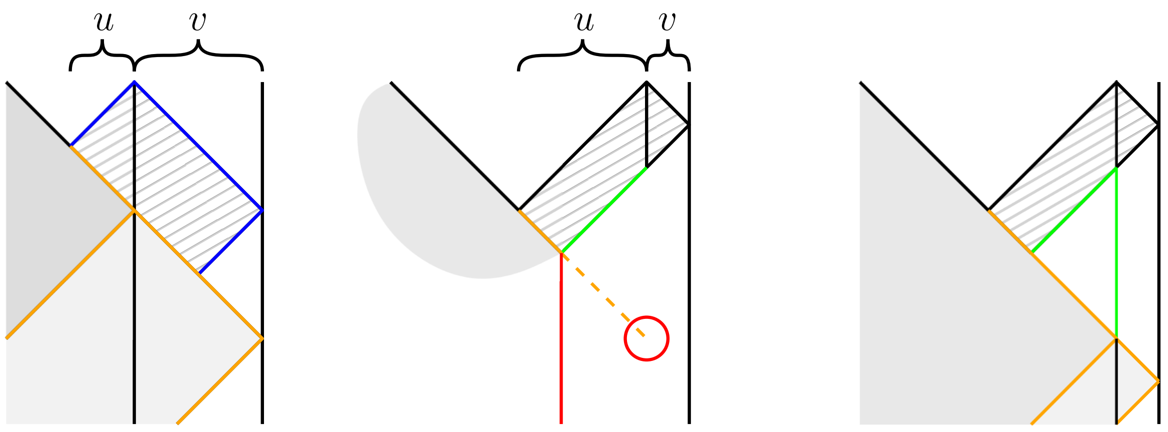


Figure 50: Comparison Conflict (2)

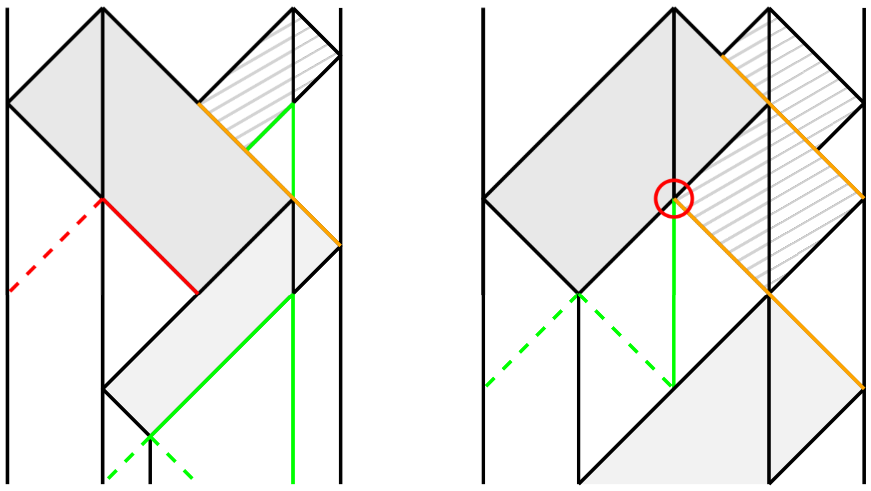


Figure 51: Comparison Conflict (3)

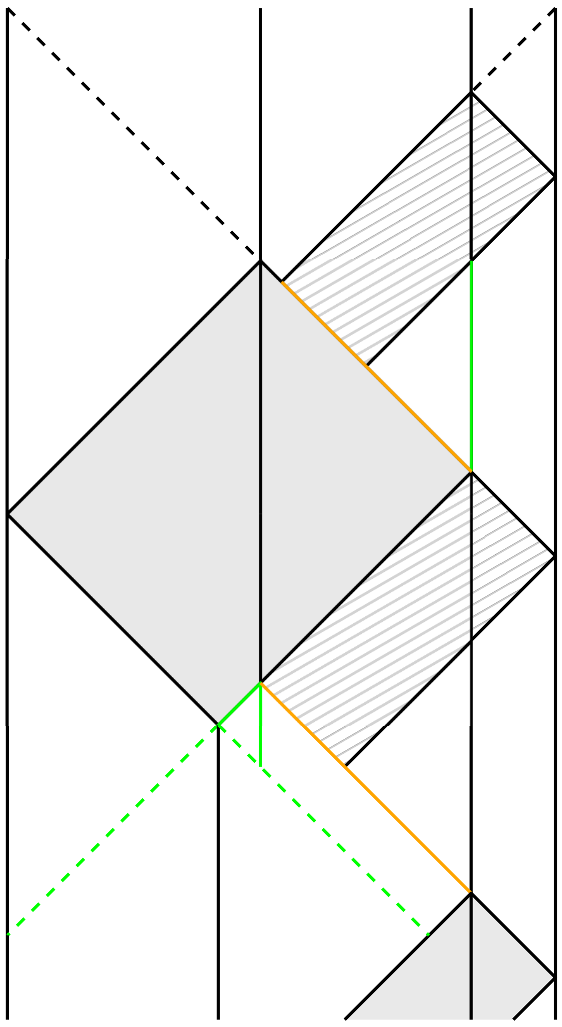


Figure 52: Comparison Conflict (4)

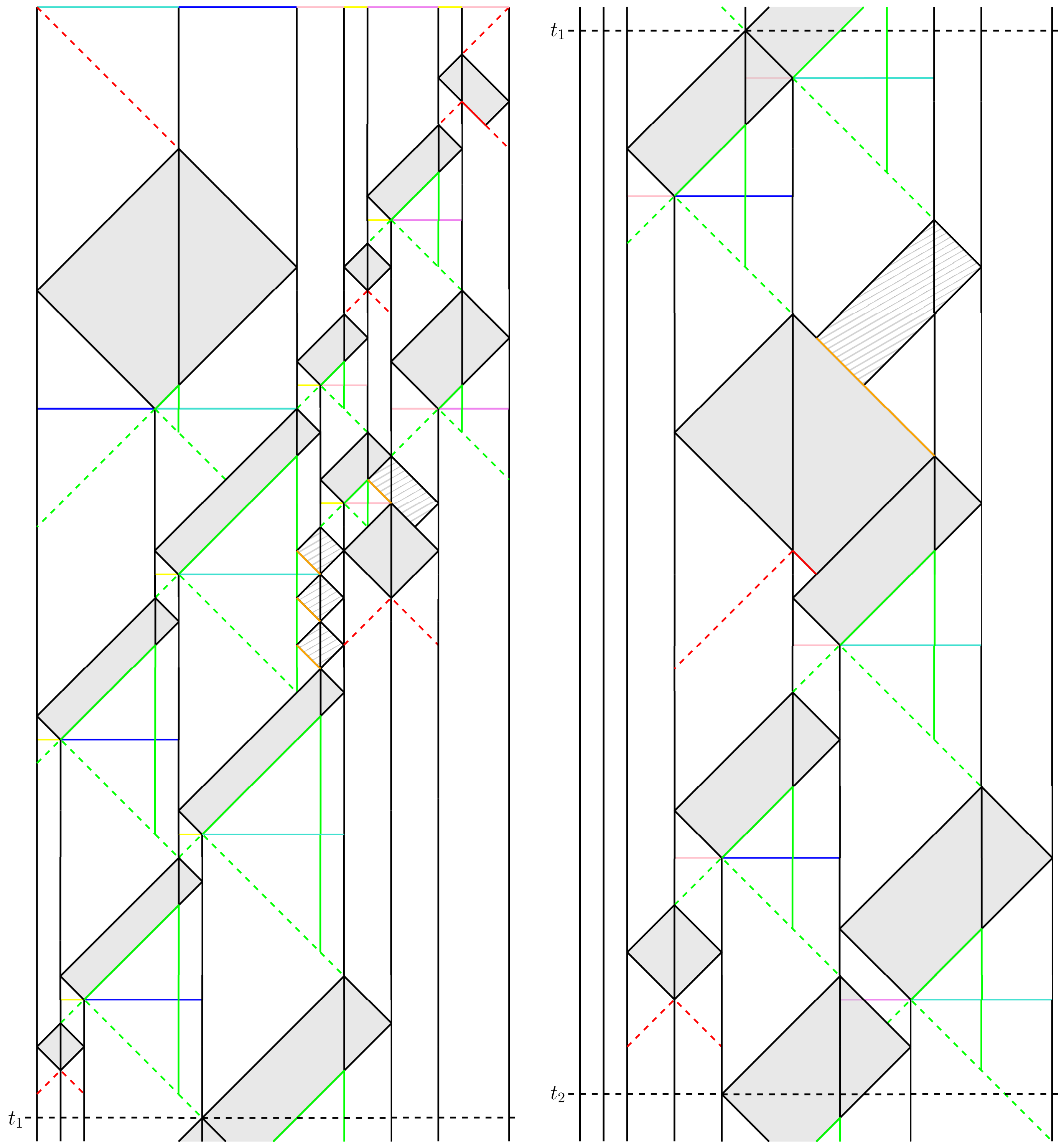


Figure 53: Bilateral Interval Sort (1)

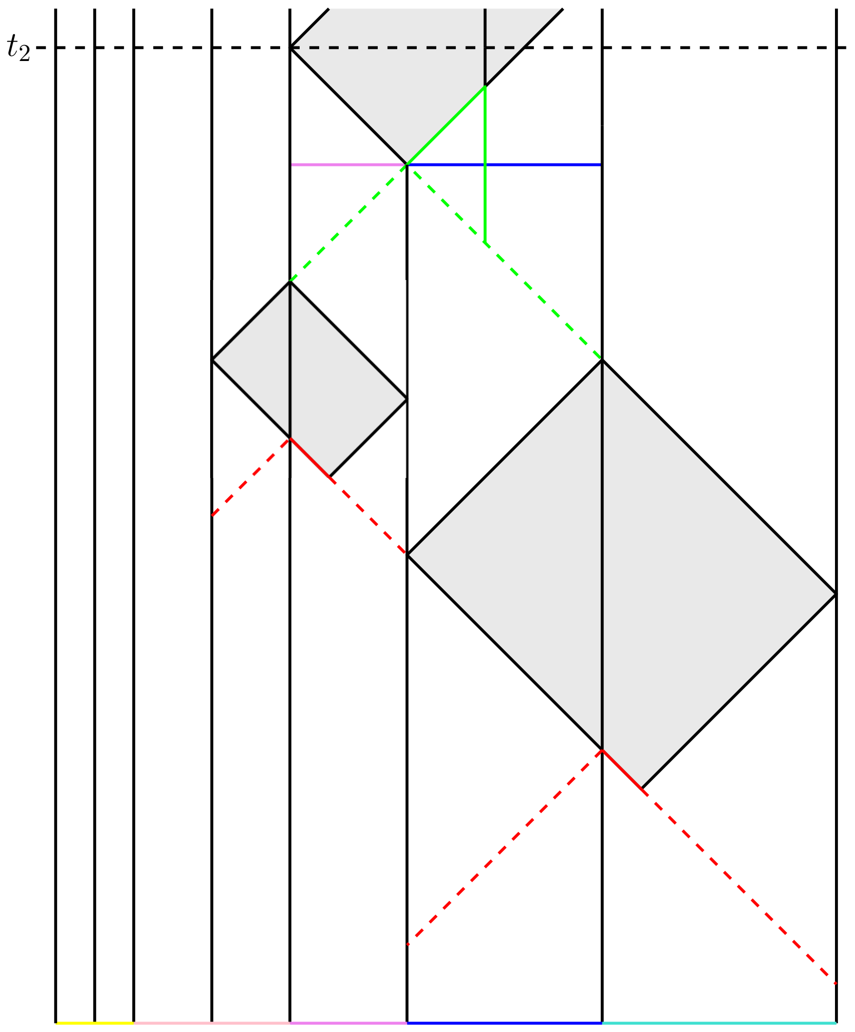


Figure 54: Bilateral Interval Sort (2)

Algorithm 36. *Hourglass Compression (Fig. 55)*

We want to exploit the properties of real numbers to solve problems more efficiently. Problems for which a signal machine requires at most an exponential amount of time, proportional to the number of intervals of the input, can be solved in linear time, proportional to the input length, through compression. We will show this on the example of the sorting problem for intervals, but it can also be applied to other problems. At first we will compress the input to an arbitrarily small size in time $2n$, where n is the total length of the input. We repeatedly use algorithm 2 (*Stretch and Compress*) to compress by a factor of 2, while alternating the direction of the compression. In figure 55, left side, we see that the resulting funnel can shrink down to a single point if the compression is repeated an infinite amount of times. Since such a singularity destroys all of the encoded information, we only allow a finite amount of compressions. In the figure, the number of compressions is linear in the number m of intervals. To count to m, a signal moving between the funnel's boundaries marks one unmarked inner interval boundary per trip. If all inner boundaries are already marked, the actual algorithm can start (*gray area*). To compress exponentially, use algorithm 3 (*Binary Counter*) to count the compressions.
In figure 55, right side, the sorting algorithm running on the compressed input is shown. To determine the end of the algorithm, an ending signal (*blue*), emerging from the side where the compression ended last, tries to move to the other side. It is halted when it collides with proxy signals or interval boundaries involved in a comparison. Green activation signals push the ending signal back towards the side it originated from. When the ending signal reaches the other side, it is reflected and moves back to its original side without any obstruction, and initiates the decompression there. The decompression works analogously to the compression. Instead of a compression, a stretch occurs. The counting mechanism is unchanged. The compressing and stretching requires linear time proportionate to the total length of the input. The actual algorithm operating on the compressed input only requires $\mathcal{O}(1)$ time with sufficient compression. One can also find counting mechanisms that allow for more than an exponential amount of compressions, making it possible to solve even harder problems in linear time.

Theorem 17. *The run time of algorithm 36 (Hourglass Compression) for an input of length n is $\mathcal{O}(4n)$.*

Proof. The compression ends at the latest when the outer boundaries collide (in that case a singularity occurs). Consider the signal with speed 1 that is reflected at these boundaries. The traveled distance between two reflections is n at first, then $\frac{n}{2}$, then $\frac{n}{4}$ and so on. Thus, the compression ends at the latest at time $2n$. Since the expansion matches the compression (just in reverse), the algorithm requires $O(4n)$ time in total. □

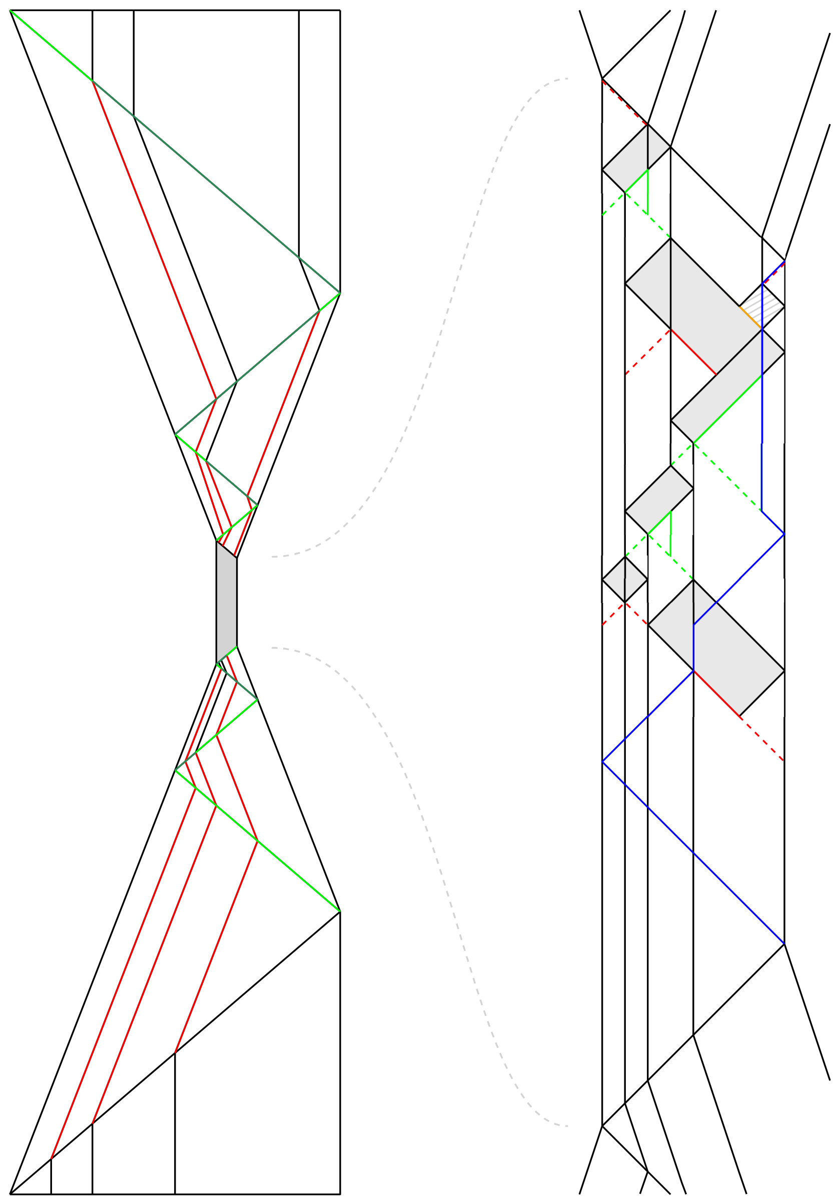


Figure 55: Hourglass Compression

9 Conclusion

We presented a range of efficient algorithms which form a basis for operating signal machines and allow to shift the focus on more complex computations. Besides the manipulation and sorting of intervals and arithmetic from basic operators up to logarithms, we introduced a range of number representations between which one can switch to fit the requirements. We generalized the base of the positional notation system from integers to real numbers. The special properties of real numbers, that elevate them from the discrete domain of integers, allow algorithms to solve difficult problems faster. By using the hourglass compression, even problems that cellular automata require an exponential amount of time for can be solved in linear time. Additionally, the signal storage allows the sorted storage of an arbitrarily large finite amount of signals in an arbitrarily small constant amount of space. However, the real numbers also bring along new challenges like singularities, which do not occur in the discrete realm and which make some computations impossible. We presented some techniques to avoid certain catastrophic singularities, which force the signal machine to halt, by using on demand systems for the multiplication and division algorithms. Also, we carried out generalizations in other domains like formal languages.

9.1 Future Work

One question we already raised is whether all cellular automata can be simulated within a constant amount of space. For this, new strategies need to be developed to distribute the creation of signals and to avoid singularities. The work of Modanese et al. [8] presents the model of shrinking and expanding cellular automata, which allows a distributed creation of cells. Maybe a similar strategy can be found for signal machines. It is possible to generate an infinite amount of signals in a finite amount of space, without creating a singularity in any specific point, for example, by repeatedly bisecting an interval and all subdivided intervals or by marking all points of the Cantor set. However, this leads to an entirely different kind of singularity that cannot be restricted to certain positions. Apart from the simulation there are also many other open questions. Since we mostly focused on the ground work, many problems that have been solved for cellular automata, are unsolved for signal machines. An example is the firing squad synchronization problem, which exists in a number of variants. Considering signal machines entirely new challenges emerge. One may want to synchronize a set of stationary signals, but it is also a possibility to synchronize a set of moving signals, which increases the problem's difficulty because of the additional dynamics. Another interesting thought is the generalization of signal machines to the two-dimensional plane, which allows the representation of complex numbers. On the contrary to the one-dimensional case, collisions of signals with a size of zero is very unlikely, so the use of signals with a certain expanse may be necessary. Some rudiments are the *Billiard Ball Model of Computation* by Margolus [7] or the *Collision-based Computing* by Adamatzky et al. [1]. The last remaining topic we already touched on are generalized formal languages. Even a useful definition of a grammar for such languages proves to be challenging, let alone recognizing languages. The generalization to real-valued lengths of characters destroys some properties of words, but may also add interesting other properties.

References

[1] Adamatzky, Andrew and Jérôme Durand-Lose: *Collision-based computing.* In *Handbook of Natural Computing*, pages 1949–1978. Springer, 2012.

[2] Codd, Edgar F: *Cellular automata.* Academic Press, 2014.

[3] Durand-Lose, Jérôme: *Abstract geometrical computation for black hole computation.* In *International Conference on Machines, Computations, and Universality*, pages 176–187. Springer, 2004.

[4] Durand-Lose, Jérôme: *Abstract geometrical computation: Turing-computing ability and undecidability.* In *Conference on Computability in Europe*, pages 106–116. Springer, 2005.

[5] Durand-Lose, Jérôme: *The signal point of view: from cellular automata to signal machines.* In *JAC 2008*, pages 238–249. Izdatelstvo MCNMO, 2008.

[6] Ermentrout, G Bard and Leah Edelstein-Keshet: *Cellular automata approaches to biological modeling.* Journal of theoretical Biology, 160(1):97–133, 1993.

[7] Margolus, Norman: *Physics-like models of computation.* Physica D: Nonlinear Phenomena, 10(1-2):81–95, 1984.

[8] Modanese, Augusto and Thomas Worsch: *Shrinking and Expanding Cellular Automata.* In *International Workshop on Cellular Automata and Discrete Complex Systems*, pages 159–169. Springer, 2016.

[9] Nagel, Kai and Michael Schreckenberg: *A cellular automaton model for freeway traffic.* Journal de physique I, 2(12):2221–2229, 1992.

[10] Sipper, Moshe: *Evolution of parallel cellular machines*, volume 4. Springer Heidelberg, 1997.

[11] Smith III, Alvy Ray: *Simple computation-universal cellular spaces.* Journal of the ACM (JACM), 18(3):339–353, 1971.

[12] Wacker, Simon: *Signal Machine And Cellular Automaton Time-Optimal Quasi-Solutions Of The Firing Squad/Mob Synchronisation Problem On Connected Graphs.* arXiv preprint arXiv:1706.05893, 2017.

[13] Worsch, Thomas: *Parallel turing machines with one-head control units and cellular automata.* Theoretical computer science, 217(1):3–30, 1999.