CDMTCS Research Report Series

Turing Completeness of Water Computing

Alec Henderson
Radu Nicolescu
Michael J. Dinneen
Department of Computer Science,
University of Auckland,
Auckland, New Zealand

TN Chan
Compucon New Zealand

Hendrik Happe
Thomas Hinze
Department of Bioinformatics,
Friedrich Schiller University of Jena,
Jena, Germany

CDMTCS-554
July 2021
Centre for Discrete Mathematics and
Theoretical Computer Science

Turing completeness of water computing

Alec Henderson, Radu Nicolescu, Michael J. Dinneen
School of Computer Science, University of Auckland, Auckland, New Zealand

TN Chan
Compucon New Zealand, Auckland, New Zealand

Hendrik Happe, and Thomas Hinze
Department of Bioinformatics, Friedrich Schiller University of Jena, Jena, Germany

Abstract

We further develop water computing as a variant of P systems. We propose an improved modular design, which duplicates the main water flows by associated control flows. We first solve the three open problems of the previous design, by demonstrating: how functions can be stacked without a combinatorial explosion of valves; how termination of the system can be detected; and how to reset the system. We then prove that the system is Turing complete by modelling the construction of μ-recursive functions. The new system is based on directed acyclic graphs, where tanks are nodes and pipes are arcs; there are no loops anymore, water falls strictly in a 'top down' direction. Finally, we demonstrate how our water tank system can be viewed as a restricted version of cP systems. We conclude with a list of further challenging problems.
keywords: ater-based computing Membrane systems μ recursion

1 Introduction

There are many different computational models from the standard Turing model to bio-inspired models such as P systems. Almost all of these systems are proven to be Turing equivalent. However, some of these computational models can compute things provably faster [1] or able to model computational problems easier than others.

Water has been used for information processing for over 2000 years [2]. Although water computers are not commonly used today, they have had many successful uses in the past. In 1901, water was used to calculate the nth root of a polynomial [3]. In the 1930s, water integrators were made to solve ordinary differential equations and not surpassed by digital computers until the 1980s [4]. In the late 1940s, the first Phillips machine was built [5]. The Phillips machine was used to model macroeconomic theory and was used in lectures for many years after the original prototype of the 1940s [6]. In the 1960s, water was used to implement logic gates such as AND, OR and NOT [7]. In the early 2000s, a fluid based bilateral system was proposed and used to solve the satisfiability

problem [8]. For a more detailed history see [9]. As our work is theoretical, we note physical implementations of water based machines are still being developed such as in [10].

In [11] a new model of water computing was proposed. The model worked by having a set of tanks interconnected using pipes. Each pipe would be controlled by a set of valves which allow water to pass through it if and only if all valves on the pipe are open. The system would terminate after an arbitrary given amount of time, making it undecidable to determine if the system had completed its computation. Further work was also required to reset the system for the next evaluation or to combine a set of functions into a directed acyclic graph without an exponential explosion of valves.

P systems are a parallel and distributed model of computing, first proposed by Gheorghe Păun in [12], without any central control, where each membrane applies applicable rules at each time step. These systems have been able to solve computationally hard problems [13] utilising a time-space trade off. Recent work includes simulating P systems on mainstream hardware [14], as well as research on the capabilities of restricted P system models [15].

In this work we present an alternative definition for the water tank system based on rules, which more closely aligns with cP systems [16] (but also other P system variants). Our definition removes the timed termination of the previous model by replacing it with a set of control tanks that tightly control the execution of the actual operation. Our model also removes pipes having different flows as we assume that each pipe will move one unit of water at each time step. Our system has two types of tanks: value tanks, which correspond to the tanks presented in [11]; and control tanks, which are essentially Boolean values. Each input and output value tank of the system has a corresponding control tank. If a control tank is full, then the corresponding value tank has been filled. This means that an operation has terminated if and only if all of its output control tanks are full.

Value tanks have capacities in $\mathbb{N}_+ \cup \{\infty\}$: infinite ($\infty$) for unbounded tanks, and finite ($\mathbb{N}_+$) for bounded tanks. We assume that all bounded tanks have overflow pipes, which cleanly drain any possible overflow down to an infinite sink; but, while still there, overflow pipes are not represented in our diagrams (to keep these simple). Most theoretical results, such as μ-recursivity, assume that all value tanks are unbounded; while, obviously, all practical implementations need bounded tanks.

In Section 2 we briefly describe the basics of water computing systems. In Section 3 we describe how a control tank can be constructed and how to reset a system. In Section 4 we define formally our new model. In Section 5 we prove that the system is Turing complete and that the tank system can be viewed as a directed acyclic graph. In Section 6 we describe how the valves are viewed as a set of membrane computing rules and establish the water tank system as a restricted version of cP systems.

2 Background

Saturation arithmetic. As discussed in [17], saturation arithmetic restricts operations to a fixed range. If an operation results in a number exceeding the upper bound, then the result is the upper bound. Conversely, if the operations result goes below 0, then the result is 0. If the upper and lower bounds are $+\infty$ and $-\infty$ respectively, then saturation arithmetic is standard arithmetic. For example, in the range [0,100], $5 \times 30 = 100$ and $20 - 30 = 0$. The tank system presented in [11], as well as our system, both use saturation arithmetic where the lower bound value is 0 (no water) and the upper bound is the capacity of the result tank (sometimes unbounded). Saturation arithmetic produces more 'natural' results than the usual modular arithmetic used by current computers We denote saturating addition as $\oplus$, and saturating subtraction as $\ominus$.

Water tank system. Tanks are displayed as open rectangles, the pipes as lines between the tanks and valves as lines crossing the pipe it belongs. We denote water going to the infinite sink by a black arrow at the end of the pipe. If the pipe starts with an empty rectangle, then the water comes from the infinite source.

Figure 1 and Figure 2 illustrate two tank systems that only contain value tanks (and no control tanks).

1. **Subtraction**($\ominus$): Basic saturated subtraction can be achieved using three tanks. The input tanks x and y, and the output tank z as shown in Figure 1. The system drains from both x and y until y is empty. Once y is empty, x corresponds to the result to be stored in z.
2. **Addition**($\oplus$): Basic saturated addition is achieved using three tanks: the two inputs and the output total as shown in Figure 2. The system does not contain valves and the values of x and y go directly to the result. Once x and y are empty, z contains the final result.

As discussed in [11], although we can intuitively model simple stand-alone gates – such as basic addition and subtraction - several essential open problems remain:

- Termination detection: a tank system that self-determines when it has completed, with a control tank becoming full when the system has finished.
- System reset: a way to evaluate the system again, possibly on other data.

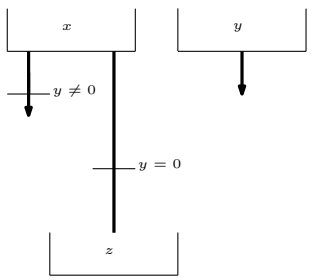


Figure 1: A diagram representing basic subtraction $z = x \ominus y$.

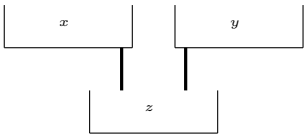


Figure 2: A diagram representing basic addition $z = x \oplus y$.

- Simplifying the control valves, to avoid a combinatorial explosion in number of tanks – essential for building more complex systems.
- Limiting the system structure to be a directed acyclic graph – the previous paper [11] used loops for implementing more complex arithmetic, such as multiplication and division. Where a loop requires a 'pump' to move the water against the natural flow (gravitational gradiant).

This paper solves all these problems, by extending the basic approach with a parallel support network of control tanks. Also, we prove that this extended water system is Turing complete (by way of μ-recursive functions).

3 Modularisation and control tanks

In this section we describe the use of control tanks for termination detection. We also discuss how these allow for modularisation and composition of functions.

Constructing complex expression such as $h(x, y, u, v) = (x \oplus y) \ominus (u \ominus v)$, from the functions in Figures 1 and 2 requires additional valves to be added. If one combines these without the addition of valves, then the result may be unexpected, because of the synchronisation requirements of the subtraction operator. To overcome this issue we use modularisation to allow for easier composition of operations.

Thus, the following restrictions are placed on all functions. A function $f(x_1, ..., x_n) = y_1, ..., y_m$ has accompanying controls on the inputs $x'_1, ..., x'_n$ and outputs $y'_1, ..., y'_m$. The function executes when all input control tanks are filled. Each output of the function has finished being computed, when its corresponding control tank has been filled. For example, a function f with n inputs and m outputs is presented in Figure 3. We note that inner control tanks initially are empty for example, q' in Figure 4.

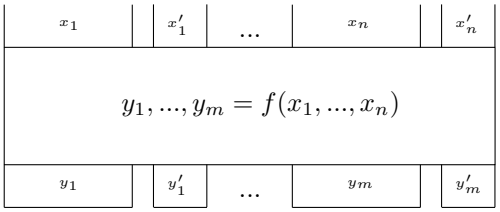


Figure 3: A diagram representing a function f following the conventions $y_1, ..., y_m = f(x_1, ..., x_n)$.

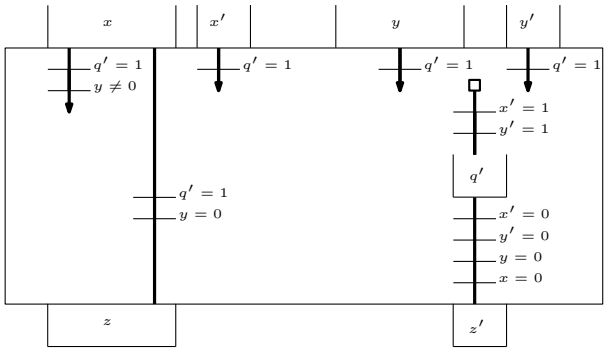


Figure 4: A diagram representing the controlled saturating subtraction operator $z = x \ominus y$.

Saturating subtraction following these conventions becomes the system presented in Figure 4. Although the system looks more complicated, it is much easier to *combine* the system into a complex system. Any function relying on the results from another function only needs a relationship with its result and control tanks. Thus a function can be viewed as a black box, cf. Figure 3, where the inputs are passed to the function and the outputs are a control and result tank. The inner workings can be ignored as before, and after execution they are the same. The Appendix presents an example trace of the controlled subtraction via images as well as, cell contents of a cP representation.

Complex expressions can be built using a high level design. For example, if we have the addition function $f(x, y) = x \oplus y$ as seen in Figure 5, then we can create the complex expression $h(x, y, u, v) = (x \oplus y) \ominus (u \ominus v)$ by simply using the outputs and output controls as the inputs to the subtraction, as seen in Figure 6.

4 New model

With the addition of control tanks we formalise our model based on the work presented in [11]. Our water based system:

$$\Pi = (T, T', F, E, R, L, C, V, S, P)$$

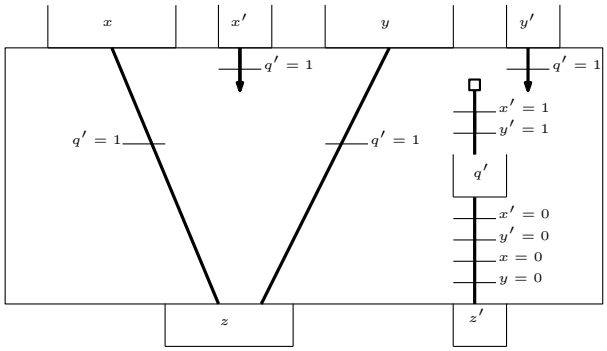


Figure 5: A diagram representing the controlled saturating addition operator $z = x \oplus y$.

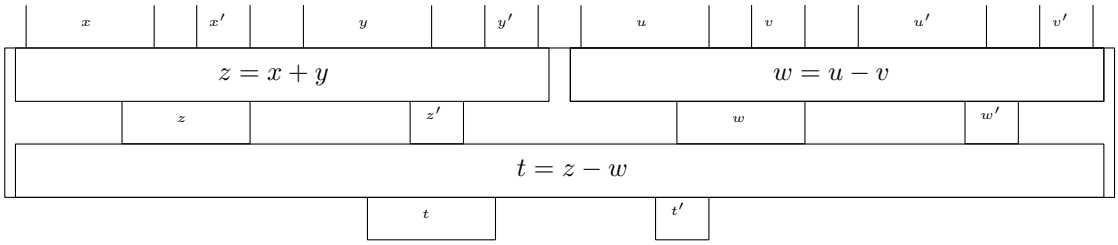


Figure 6: A diagram showing how complex expressions can be created.
$t = (x \oplus y) \ominus (u \ominus v)$.

With its components:

- T finite set of tank identifiers.
- $T' \subset T$ finite set of control tank identifiers.
- $F \subset T'$ set of control tanks that when full indicate termination of the system.
- $E \in T \setminus T'$ the unique infinite sink of the system.
- $R \in T \setminus T'$ the unique infinite source in the system.
- $L : T \to \mathbb{N}_+$ The level which the tanks are built, the lower the number the conceptually higher the tank. Water can only flow from a tank with a lower number to one with a higher number. r is at 0 and s at ∞.
- $C : T \to \mathbb{N}_+ \cup \{\infty\}$ capacity of the tanks. Where we assume that value tanks are able to be unbounded. Of course for practical cases all value tanks will need to have finite capacity. Control tanks all have capacity 1 (they act as Boolean values).
- V finite set of valve identifiers.
- $S : V \to (T = \mathbb{N}_+ \cup T \neq \mathbb{N}_+)$ An expression from a valve identifier to check whether or not a tank has a certain volume.
- $P \subset T \times T \times \mathcal{P}(V)$ ($\mathcal{P}(V)$ denotes the power set over V) finite set of pipes where water flows from the first element to the second. A pipe (i, j, v) must have $L(i) < L(j)$, meaning water only flows in one direction ('down').

To clarify the model we present controlled subtraction:

- $T = \{x, y, x', y', q', z, z', r, s\}$
- $T' = \{x', y', z'\}$
- $F = \{z'\}$
- $E = s$
- $R = r$
- $L = (x, 1), (y, 1), (x', 1), (y', 1), (q', 2), (z, 3), (z', 3), (r, 0), (s, \infty)$

- $C = (x, \infty), (y, \infty), (x', 1), (y', 1), (q', 1), (z, \infty), (z', 1), (r, \infty), (s, \infty)$
- $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$
- $S = (1, q = 1), (2, y \neq 0), (3, y = 0), (4, x = 0), (5, y' = 1), (6, y' = 0), (7, x' = 0), (8, x' = 1)$
- $P = (x, s, \{1, 2\}), (x, z, \{1, 3\}), (y, s, \{1\}), (x', s, \{1\}), (y', s, \{1\}),$
  $(r, q', \{5, 8\}), (q', z', \{3, 4, 5, 7\})$

As with the majority of models of computation the formal definition is not used in the majority of cases but rather more intuitive descriptions which can be transformed into the formal definition with a small amount of work. In this work we shall present where appropriate a pictorial view of our functions as well as a set of equations which both describe the system. Both of these can be transformed back into the formal definition.

5 Turing completeness

We first prove that our system can construct all unary primitive recursive functions. We assume that our value tanks are unbounded for this proof. In [18, 19, 20, 21] it was shown that these base functions and closure operators are sufficient to construct the unary primitive recursive functions:

- **Successor function**: $S(x) = x + 1$ (cf. Figure 10)
- **Subtraction function**: $B(x, y) = x - y$ (cf. Figure 4)
- **Composition operator**: $C(h, g)(x) = h(g(x))$ (cf. Figure 11)
- **Difference operator**: $D(f, g)(x) = g(x) - f(x)$ (cf. Figure 12)
- **Primitive recursion operator**: $P(f) = p, p(0) = 0, p(x+1) = f(p(x))$ (cf. Figure 13)

To construct these functions we make two copy functions: an inplace copy and a destructive copy. Inplace copy as seen in Figure 7 takes input x and outputs x_1, whilst x receives a copy of its original content. The destructive copy as seen in Figure 8 takes an input x and outputs two copy's of x_1 and x_2, emptying x. in Figure 6.

Successor function $S(x) = x + 1$: The successor function $S(x)$ adds one to the given input x. Using similar ideas to the addition displayed in Figure 5 we arrive at the function displayed in Figure 10, where instead of draining from y we drain from the input control (the control would always contain a unit amount of water).

Composition operator $C(h, g)(x) = h(g(x))$: The composition operator as seen in Figure 11 is started by filling the control tank x'. Once the composition is started it runs the function $g(x)$, where $g(x)$'s output y is the input for $f(y)$, and $f(y)$'s output z is the output for the composition.

Difference operator $D(f, g)(x) = g(x) - f(x)$: The difference operator works similar to that of the composition, where the input to the difference is the input to the destructive

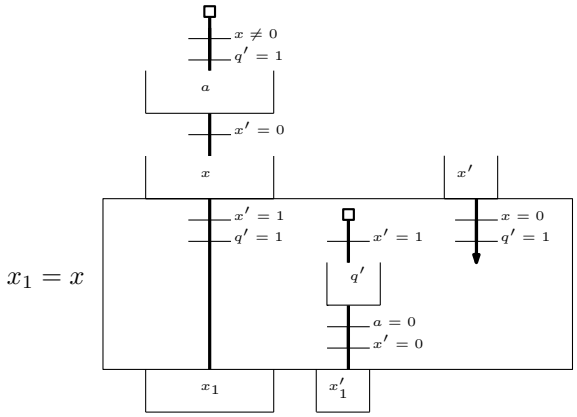


Figure 7: A diagram representing the inplace copy function $i(x) = x$.

Table 1: Equations for inplace copy.

$a(t+1) =$ **if** $x(t) \neq 0$ & $q'(t) = 1$ **then** $a(t) \oplus 1$
else if $x'(t) = 0$ **then** $a(t) \ominus 1$
else $a(t)$

$x(t+1) =$ **if** $x'(t) = 0$ & $a(t) > 0$ **then** $x(t) \oplus 1$
else if $x'(t) = 1$ & $q'(t) = 1$ **then** $x(t) \ominus 1$
else $x(t)$

$x'(t+1) =$ **if** $x(t) = 0$ & $q'(t) = 1$ **then** $x(t) \ominus 1$ **else** $x(t)$

$q'(t+1) =$ **if** $x'(t) = 1$ **then** $q'(t) + 1$
else if $a(t) = 0$ & $x'(t) = 0$ **then** $q'(t) \ominus 1$
else $q'(t)$

$x_1(t+1) =$ **if** $x'(t) = 1$ & $q'(t) = 1$ & $x(t) > 0$ **then** $x_1(t) \oplus 1$ **else** $x_1(t)$

$x'_1(t+1) =$ **if** $a(t) = 0$ & $x'(t) = 0$ & $q'(t) > 0$ **then** $x'_1(t) \oplus 1$ **else** $x'_1(t)$

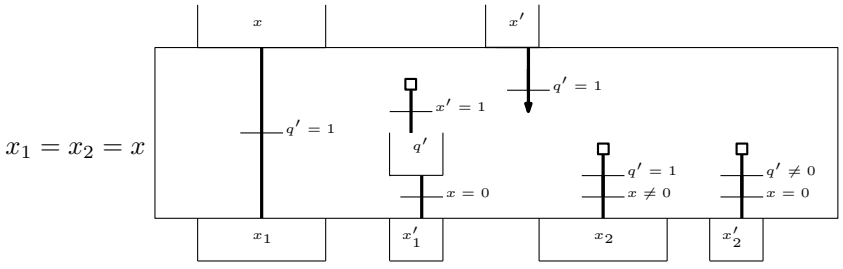


Figure 8: A diagram representing the destructive copy function $c(x) = x, x$.

Table 2: Equations for destructive copy.

$$
\begin{aligned}
x(t+1) = \quad & \textbf{if } q'(t) = 1 \textbf{ then } x(t) \ominus 1 \textbf{ else } x(t) \\
x'(t+1) = \quad & \textbf{if } q'(t) = 1 \textbf{ then } x'(t) \ominus 1 \textbf{ else } x'(t) \\
q'(t+1) = \quad & \textbf{if } x'(t) = 1 \textbf{ then } q'(t) \oplus 1 \\
& \textbf{else if } x(t) = 0 \textbf{ then } q'(t) \ominus 1 \\
& \textbf{else } q'(t) \\
x_1(t+1) = \quad & \textbf{if } q'(t) = 1 \ \& \ x(t) > 0 \textbf{ then } x_1(t) \oplus 1 \textbf{ else } x_1(t) \\
x_2(t+1) = \quad & \textbf{if } q'(t) = 1 \ \& \ x(t) \neq 0 \textbf{ then } x_2(t) \oplus 1 \textbf{ else } x_2(t) \\
x'_1(t+1) = \quad & \textbf{if } x(t) = 0 \ \ \& \ \ q'(t) > 0 \textbf{ then } x'_1(t) \oplus 1 \textbf{ else } x'_1(t) \\
x'_2(t+1) = \quad & \textbf{if } x(t) = 0 \ \ \& \ \ q'(t) \neq 0 \textbf{ then } x'_2(t) \oplus 1 \textbf{ else } x'_2(t)
\end{aligned}
$$

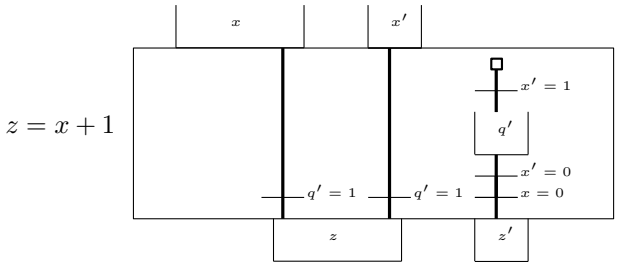
copy function (Figure 8). The output of the destructive copy is then passed to f and g. The output of f and g is then passed to the subtraction function (Figure 4). Finally the output of the subtraction is the output of the difference function D.

Primitive recursion operator $P(f) = p, p(0) = 0$**,** $p(x+1) = f(p(x))$: The primitive recursion operator P as seen in Figure 13 can be explained by the program in Figure 9. If the counter is 0, then the result is 0. Otherwise, function f is repeatedly executed while decrementing x, until x becomes 0. Once x is zero, the function returns the result and fills the control tank . □

Theorem 1. *Our water system can construct all unary primitive recursive functions.*

```
1  if x' = 1 then
2      if x = 0 then
3          y' ← 1; y ← 0;
4          return
5      else
6          u ← 0;
7          loop:
8              v ← f(u);
9              if x = 0 then
10                 y ← v; y' ← 1;
11                 return
12             else
13                 x ← x − 1;
14                 u ← v;
15                 goto loop
```

Figure 9: Code to describe the execution of the primitive recursion operator $p = P(f), p(0) = 0$, $p(x+1) = f(p(x))$.



$z = x + 1$

operator.pdf

Figure 10: A diagram representing the successor function $S(x) = x + 1$.

Table 3: Equations for the successor function.

$$
\begin{aligned}
x(t+1) = \quad & \textbf{if } q'(t) = 1 \textbf{ then } x(t) \ominus 1 \textbf{ else } x(t) \\
x'(t+1) = \quad & \textbf{if } q'(t) = 1 \textbf{ then } x'(t) \ominus 1 \textbf{ else } x'(t) \\
q'(t+1) = \quad & \textbf{if } x'(t) = 1 \textbf{ then } q'(t) \oplus 1 \\
& \textbf{else if } x(t) = 0 \textbf{ then } q'(t) \ominus 1 \\
& \textbf{else } q'(t) \\
z(t+1) = \quad & \textbf{if } q'(t) = 1 \;\&\; x(t) > 0 \;\&\; x'(t) > 0 \textbf{ then } z(t) \oplus 2 \\
& \textbf{else if } q' = 1 \;\&\; (x(t) > 0 \textbf{ or } x'(t) > 0) \textbf{ then } z(t) \oplus 1 \\
& \textbf{else } z(t) \\
z'(t+1) = \quad & \textbf{if } x'(t) = 0 \;\&\; x(t) = 0 \;\&\; q'(t) > 0 \textbf{ then } z'(t) \oplus 1 \textbf{ else } z'(t)
\end{aligned}
$$

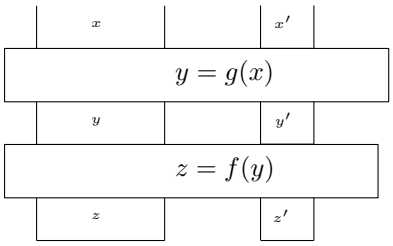


Figure 11: A diagram representing the composition of functions $C(x) = f(g(x))$.

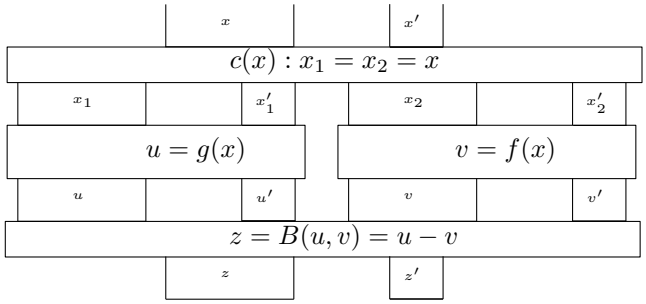


Figure 12: A diagram representing the difference of functions $D(x) = g(x) - f(x)$.

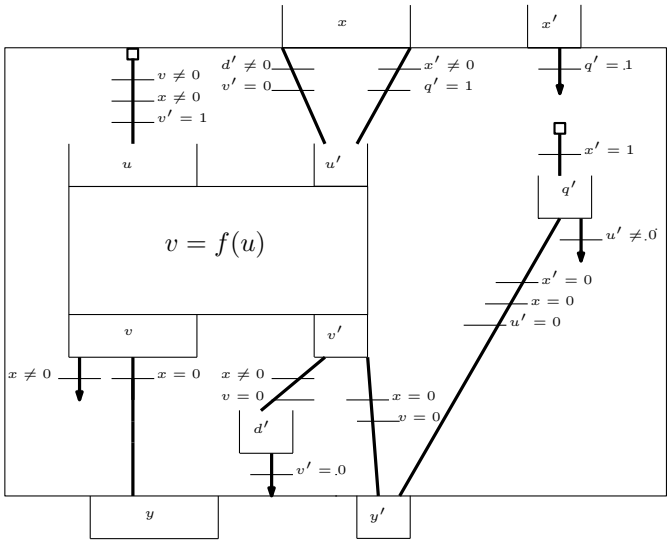


Figure 13: A diagram representing the primitive recursion operator $P(f) = p, p(0) = 0, p(x + 1) = f(p(x))$.

To prove Turing completeness, we require that our system can construct the unary primitive recursive functions as well as [19]:

- **Addition function**: $A(x, y) = x + y$ (see Figure 5)
- **μ operator**: $\mu_y(f)(x) = \min_y\{f(x, y) = 0\}$

We note that the μ operator currently takes two arguments and all other functions take one (except addition and subtraction). To overcome this issue we apply the Cantor pairing function [19]:

$$\Pi(x, y) = \frac{(x + y)(x + y + 1)}{2} + y = \frac{((x + y)^2 + 3y + x)}{2}$$

As squaring, multiplying by a constant, and division of one variable are primitive recursive, the only two variable function required to construct the Cantor pairing function is addition (see Figure 5). μ operator $\mu_y(f)(x) = \min_y\{f(x, y) = 0\}$: The μ operator as seen in Figure 14 first transfers the data from x to x'. It then utilises the inplace copy function (Figure 7) i on both x' and y (y is initially 0). It uses these copies to run $f(x, y)$. If $f(x, y)$ results in 0, then y is copied into the result z and the control tank is filled. If the result was not 0, then y is incremented and starts the copying process again until the result becomes 0 (which may or may not occur).

Theorem 2. *Our water system is Turing complete.*

Theorem 3. *Our water system can be viewed as a directed acyclic graph.*

Our construction of μ recursive functions (all base functions and the operators) required the pipes to only go in one direction ('down'). Hence, if the pipes are viewed as arcs and tanks as nodes, then the digraph contains no cycles. This is in contrast to the system presented in [11], where loops were used to achieve operations such as multiplication and division.

6 High Level Rules

6.1 P systems

There are many P system variants such as: the original [12], spiking neural [22], tissue [23] and cP systems [16]. Our water tank system could be described in a variety of ways, however, we find that a cP system model fits easily.

Using a BNF-like notation, Tables 4 and 5 describe the basic structure of cP systems. The grammar presented in Table 4 describes the contents for top-cells and sub-cells, i.e. how data is stored in cP multisets. The grammar presented in Table 5 describes the high-level rewriting rules for cP systems.

Before a rule can be applied, it must match, by way of unification, all conditions specified by its left-hand-side, and promoter constraints. vterm arguments enclosed in round parentheses '()' require complete match. weak priority order – i.e., rules are sequentially

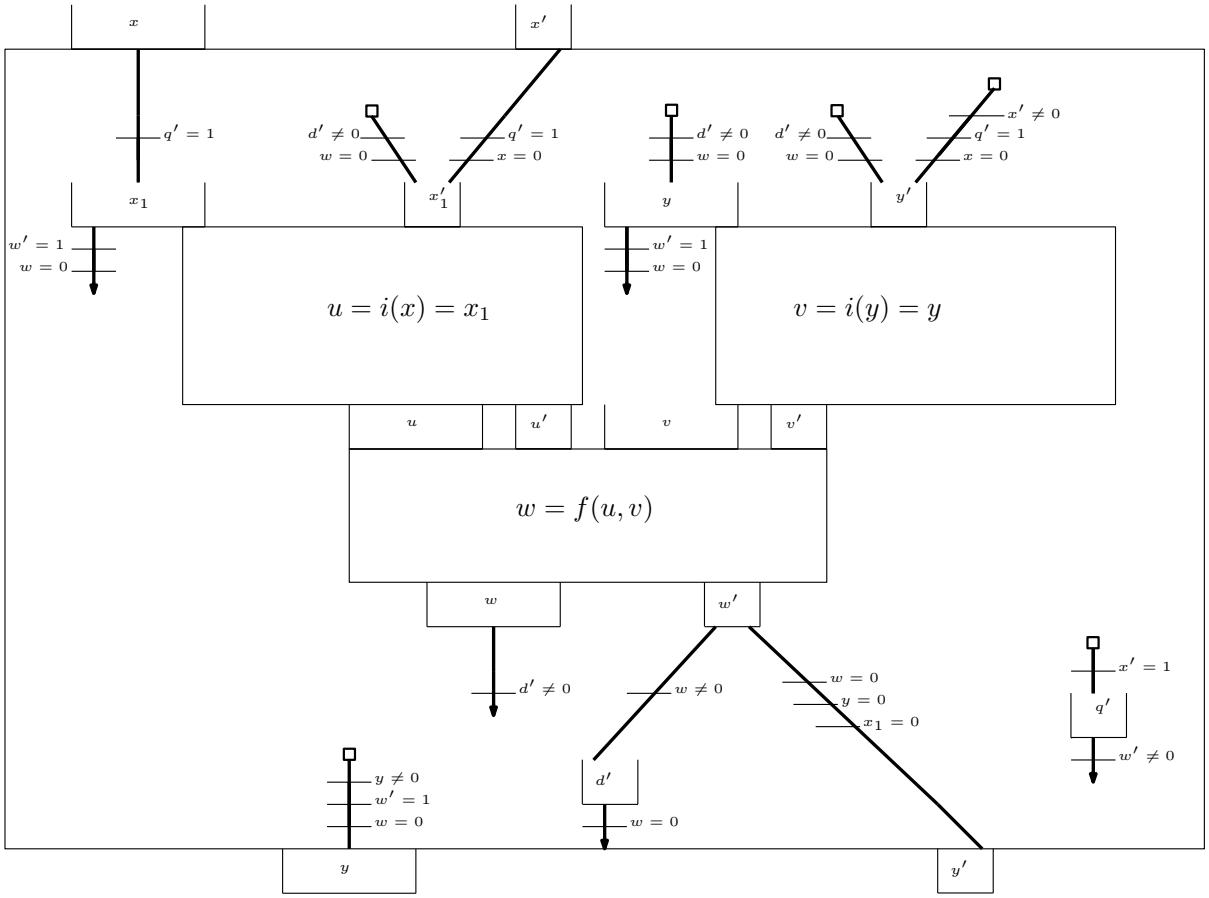


Figure 14: A diagram representing the μ operator $\mu_y(f)(x) = min_y\{f(x, y) = 0\}$.

considered in the top-down order. The first applied rule commits the target state, any subsequent rule that indicates a different target state is then disabled. This way, the weak priority order can be used to simulate if-then-else structures of traditional programming.

The system described in [11] contains a set of tanks which contain at any time step a fixed volume of water. Each tank is a data storage device working similarly to that of a cP system sub-cell. cP systems, however, are able to create or delete sub-cells during execution, whereas water tanks must be created before and cannot be removed. A rule in a cP system may cause a change to the number of sub-cells or the sub-cells content. Similarly the pipes and valves of a water tank change the volume of water contained in the tanks. These similarities allow us to define the tank system as a restricted version of cP systems. Our tank system rules can be defined by the grammar in Table 5. Of course this grammar makes no reference to the maximum value which can be contained in a tank. The maximum content in a cell can be easily described by a rule which subtracts from the tank any water which goes above that value. For example, if we have a tank v_x which has a maximum value of 2, then we add the rule in Table 6. This rule removes any water above the maximum and ensures the tank returns to its maximum value. We ignore this technical detail when describing our rule sets as the maximum value is implicitly defined or an implementation detail defining the maximum in the saturating arithmetic.

6.2 Examples

We now present a few examples of rules for some of the gates presented earlier(ignoring the overflow for brevity):

1. **Addition**: Figure 2 shows a tank system for the saturated addition $y_1 = x_1 \oplus x_2$. This tank system can be faithfully emulated by the following cP ruleset:
2. **Subtraction**: Figure 1 shows a tank system for the saturated subtraction $y_1 = x_1 \ominus x_2$. This tank system can be faithfully emulated by the following cP ruleset:
3. **Controlled subtraction:** The controlled subtraction presented in Figure 4 has the rules:

with a trace of the ruleset and tank system shown in the appendix.

We note that utilising the full power of cP systems we can simplify the rule sets further.

Table 4: BNF grammar for cP top-cells.

```
<top-cell> ::= <state> <term> ...
<state> ::= <atom>
<term> ::= <atom> | <sub-cell>
<sub-cell> ::= <compound-term> ...
<compound-term> ::= <functor> <args> ...
<functor> ::= <atom>
<args> ::= '(' <term> ... ')'
```

Table 5: BNF grammar for rules, here omitting inter-cell messaging.

```
<rule> ::= <lhs> →a <rhs> <promoters>
<lhs> ::= <state> (<vterm>)...
<rhs> ::= <state> (<vterm>)...
<promoters> ::= ('|' <vterm>)...
<vterm> ::= <variable> | <atom> | <compound-vterm>
<compound-vterm> ::= <functor> <vargs> ...
<vargs> ::=  '(' <vterm> ... ' )'
```

Table 6: Rule to ensure tank v_x does not exceed 2 units of water.

$s_1\ v_x(111\_)$	$\rightarrow$	$s_2\ v_x(11)$	(1)

Table 7: Rules to describe $y_1 = x_1 \oplus x_2$.

$s_1\ x_1(1X)\ x_2(1X)\ y_1(Y)$	$\rightarrow$	$s_1\ x_1(X)\ x_2(X)\ y_1(Y11)$	(1)
$s_1\ x_1(1X)\ y_1(Y)$	$\rightarrow$	$s_1\ x_1(X)\ y_1(Y1)$	(2)
$s_1\ x_2(1X)\ y_1(Y)$	$\rightarrow$	$s_1\ x_2(X)\ y_1(Y1)$	(3)

Table 8: Rules to describe $y_1 = x_1 \ominus x_2$.

$s_1\ x_1(X1)$	$\rightarrow$	$s_1\ x_1(X)$	|	$x_2(\_1)$	(1)
$s_1\ x_2(Y1)$	$\rightarrow$	$s_1\ x_2(Y)$			(2)
$s_1\ x_1(X1)\ y_1(Y)$	$\rightarrow$	$s_1\ x_1(X)\ y_1(Y1)$	|	$x_2()$	(3)

Table 9: A ruleset for the controlled saturating subtraction operator $z = x \ominus y$.

$s_1\ q()$	$\rightarrow$	$s_2\ q(1)$	|	$c_x(1)\ c_y(1)$	(1)
$s_2\ c_x(1)$	$\rightarrow$	$s_2\ c_x()$	|	$q(1)$	(2)
$s_2\ c_y(1)$	$\rightarrow$	$s_2\ c_y()$	|	$q(1)$	(3)
$s_2\ v_x(X1)$	$\rightarrow$	$s_2\ v_x(X)$	|	$q(1)\ v_y(\_1)$	(4)
$s_2\ v_y(Y1)$	$\rightarrow$	$s_2\ v_y(Y)$	|	$q(1)$	(5)
$s_2\ v_x(X1)\ v_z(Z)$	$\rightarrow$	$s_2\ v_x(X)\ v_z(Z1)$	|	$q(1)\ v_y()$	(6)
$s_2\ q(1)\ c_z()$	$\rightarrow$	$s_3\ q()\ c_z(1)$	|	$c_x()\ c_y()$	(7)

For example, a traditional cP solution to subtraction ($x \ominus y$) is achieved using the rules in Table 10. Noting that the full cP system solution takes only 1 step where as, the water based system will take a linear number of steps.

7 Conclusion and future work

We have proven that our water tank system based on the system proposed in [11] is Turing complete, via construction of μ recursive functions. We have demonstrated how termination can be detected, and how to combine different functions without an exponential explosion of the number of valves. Furthermore, we have shown that the water tank system only requires water to flow in one direction (no loops are required). Future work includes exploring asynchronous circuits which do not require all inputs to be filled before processing starts. For example, a logical 'or' gate could proceed as soon as one of the inputs is full. Although we have shown Turing completeness, more practical examples need to be explored. For example: equation solvers, non-numerical problems and models of biological systems.

We have shown how to create μ recursive functions, but practical constructions (without just relying on our μ constructions) of RAM and PRAM machines could also be useful. Constructing a programmed universal water machine, which takes a program as data, may allow the system to be more practical. Direct translation between water and cP systems would simplify the design of the water based system, and allow a practical implementation of a restricted version of cP systems.

8 Declaration Note

On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

[1] W. Maass, G. Schnitger, and E. Szemerédi, "Two tapes are better than one for off-line Turing machines," in *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pp. 94–100, 1987.

[2] K. Mahatantila, R. Chandrajith, H. Jayasena, and K. Ranawana, "Spatial and temporal changes of hydrogeochemistry in ancient tank cascade systems in Sri Lanka:

Table 10: Rules to describe $z = x \ominus y$.

$$s_1\ x(YZ)\ y(Y) \quad \rightarrow \quad s_2\ z(Z) \tag{1}$$
$$s_1\ x(\_)\ y(\_) \quad \rightarrow \quad s_2\ z() \tag{2}$$

evidence for a constructed wetland," *Water and Environment Journal*, vol. 22, no. 1, pp. 17–24, 2008.

[3] A. Emch, "Two Hydraulic Methods to Extract the nth Root of any Number," *The American Mathematical Monthly*, vol. 8, no. 1, pp. 10–12, 1901.

[4] G. Trogemann, A. Y. Nitussov, and W. Ernst, *Computing in Russia: the history of computer devices and information technology revealed*. Vieweg Braunschweig, 2001.

[5] A. W. Phillips, *Mechanical models in economic dynamics*. London School of Economics and Political Science, 1950.

[6] W. H. Ryder, "A System Dynamics View of the Phillips Machine," in *27th International Conference of the System Dynamics Society*, 2009.

[7] M. J. Moylan, *Fluid logic in simple terms*. Transatlantic Arts, 1968.

[8] J. J. Arulanandham, C. S. Calude, and M. J. Dinneen, "Solving SAT with Bilateral Computing," *Romanian Journal of Information Science and Technology*, vol. 6, no. 1-2, pp. 9–18, 2003.

[9] A. Adamatzky, "A brief history of liquid computers," *Philosophical Transactions of the Royal Society B*, vol. 374, no. 1774, 2019.

[10] N. Taberlet, Q. Marsal, J. Ferrand, and N. Plihon, "Hydraulic logic gates: building a digital water computer," *European Journal of Physics*, vol. 39, no. 2, p. 025801, 2018.

[11] T. Hinze, H. Happe, A. Henderson, and R. Nicolescu, "Membrane computing with water," *Journal of Membrane Computing*, vol. 2, no. 2, pp. 121–136, 2020.

[12] G. Păun, "Computing with membranes," *Journal of Computer and System Sciences*, vol. 61, no. 1, pp. 108–143, 2000.

[13] P. Sosík, "P systems attacking hard problems beyond NP: a survey," *Journal of Membrane Computing*, vol. 1, no. 3, pp. 198–208, 2019.

[14] L. Valencia-Cabrera, I. Pérez-Hurtado, and M. Á. Martínez-del Amor, "Simulation challenges in membrane computing," *Journal of Membrane Computing*, pp. 1–11, 2020.

[15] A. Alhazov, R. Freund, and S. Ivanov, "P systems with limited number of objects," *Journal of Membrane Computing*, vol. 3, no. 1, pp. 1–9, 2021.

[16] R. Nicolescu and A. Henderson, "An introduction to cP Systems," in *Enjoying Natural Computing: Essays Dedicated to Mario de Jesús Pérez-Jiménez on the Occasion of His 70th Birthday* (C. Graciani, A. Riscos-Núñez, G. Păun, G. Rozenberg, and A. Salomaa, eds.), vol. 11270 of *Lecture Notes in Computer Science*, pp. 204–227, Springer, 2018.

[17] F. Zappa and S. Esculapio, *Microcontrollers. Hardware and Firmware for 8-bit and 32-bit devices*. LIGHTNING SOURCE Incorporated, 2017.

[18] C. Calude and L. Sântean, "On a theorem of Günter Asser," *Mathematical Logic Quarterly*, vol. 36, no. 2, pp. 143–147, 1990.

[19] J. Robinson, "General recursive functions," *Proceedings of the american mathematical society*, vol. 1, no. 6, pp. 703–718, 1950.

[20] D. E. Severin *et al.*, "Unary primitive recursive functions," *Journal of Symbolic Logic*, vol. 73, no. 4, pp. 1122–1138, 2008.

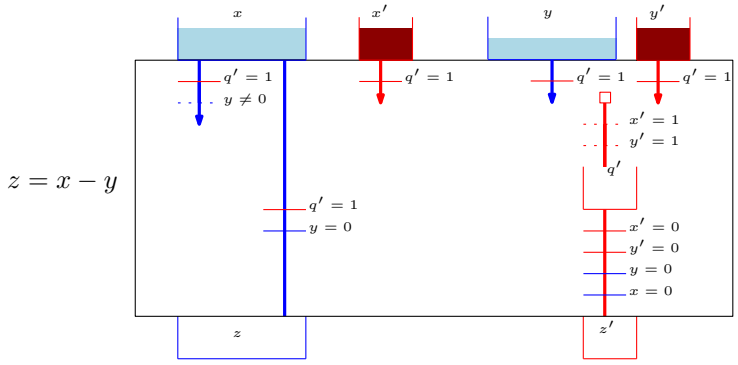
[21] C. Calude, *Theories of computational complexity*. Elsevier, 2011.

[22] M. Ionescu, G. Păun, and T. Yokomori, "Spiking neural P systems," *Fundamenta informaticae*, vol. 71, no. 2, 3, pp. 279–308, 2006.

[23] C. Martín-Vide, G. Păun, J. Pazos, and A. Rodríguez-Patón, "Tissue P systems," *Theoretical Computer Science*, vol. 296, no. 2, pp. 295–326, 2003.

I Example trace of controlled subtraction

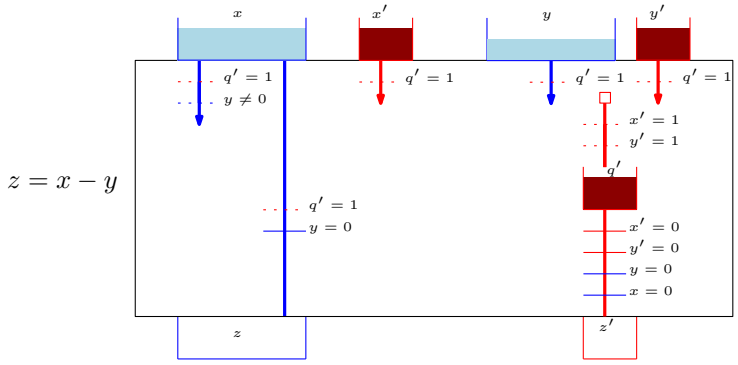
Here we present a trace of saturation subtraction, for both the tank system presented in Figure 4, and cP rules in Table 3.



s_1	$q()$	$c_x(1)$	$c_y(1)$	$c_z()$	$v_x(3)$	$v_y(2)$	$v_z()$

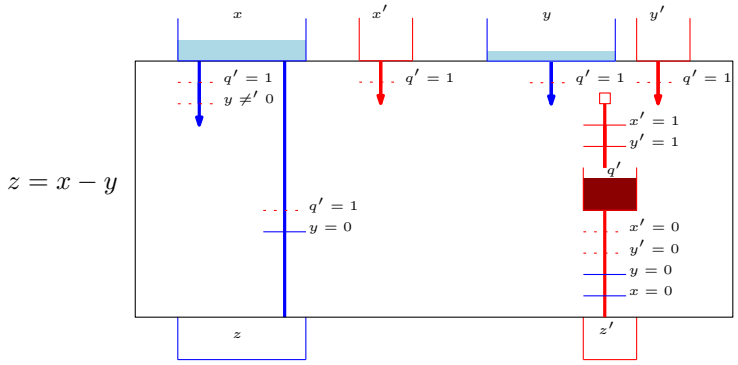
Figure 15: The initial state of controlled subtraction of $3 \ominus 2$.

$+$



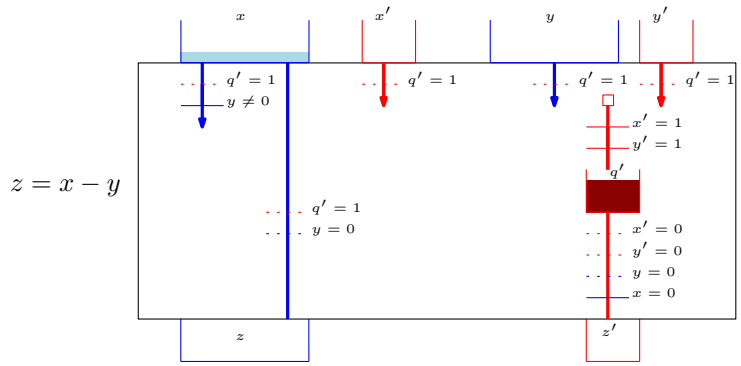
s_2	$q(1)$	$c_x(1)$	$c_y(1)$	$c_z()$	$v_x(3)$	$v_y(2)$	$v_z()$	rule (1)

Figure 16: The first step of controlled subtraction of $3 - 2$.



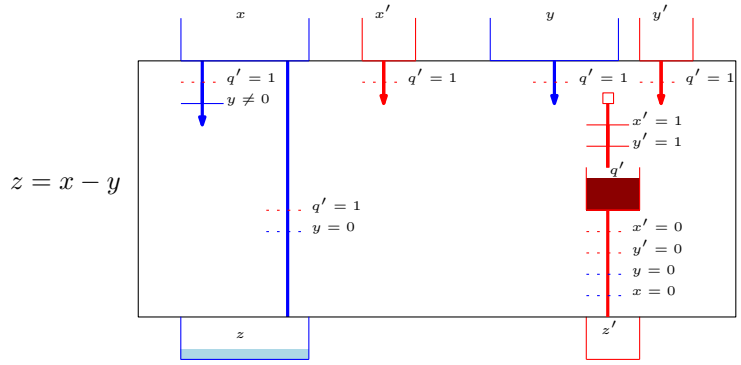
s_2	$q(1)$	$c_x()$	$c_y()$	$c_z()$	$v_x(2)$	$v_y(1)$	$v_z()$	rules (2, 3, 4, 5)

Figure 17: The second step of controlled subtraction of $3 \ominus 2$.



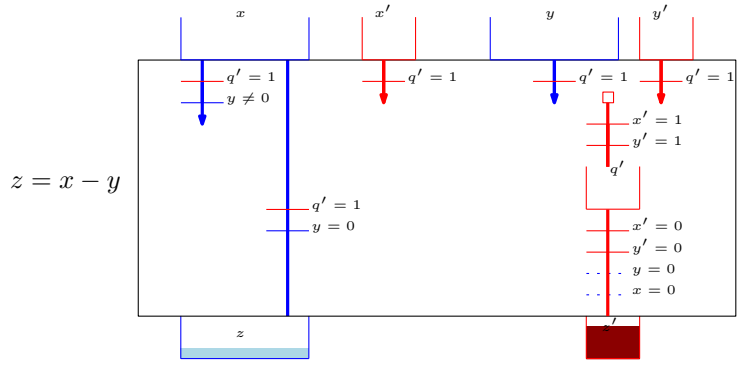
s_2	$q(1)$	$c_x()$	$c_y()$	$c_z()$	$v_x(1)$	$v_y()$	$v_z()$	rules (4, 5)

Figure 18: The third step of controlled subtraction of $3 \ominus 2$.



s_2	$q(1)$	$c_x()$	$c_y()$	$c_z()$	$v_x()$	$v_y()$	$v_z(1)$	rule (6)

Figure 19: The fourth step of controlled subtraction of $3 \ominus 2$.



s_3	$q()$	$c_x()$	$c_y()$	$c_z(1)$	$v_x()$	$v_y()$	$v_z(1)$	rule (7)

Figure 20: The last step of controlled subtraction of $3 \ominus 2$.