DESIGN

OF A

GENERIC LINEAR EQUATION SOLVER

AND ITS

IMPLEMENTATION

HASSAN AÏT-KACI

School of Computing Science
Simon Fraser University
Burnaby, British Columbia
V5A 1S6, Canada

+1 (604) 291-5589
hak@cs.sfu.ca
http://www.isg.sfu.ca/~hak

January 17, 1999

(INCOMPLETE DRAFT)

Abstract This work exploits the convenience of object-orientation—as supported by, *e.g.*, C++ (*viz.*, multiple inheritance, template classes and functions, and operator overloading)—for designing a minimal set of generic classes implementing linear-equation solvers for a large variety of specific semi-ring structures. This illustrates using a simple relativistic paradigm to obtain, with a minimal set-up, a large collection of algorithms which can all be obtained as derived classes and instance objects of a single very abstract scheme. The resulting system is a truly generic solver which can single-handedly and efficiently solve left or right linear equational systems for optimization problems in number structures, but also in graphs and networks.

Copyright © Hassan AÏT-KACI

Contents

1 Purpose of this work

This work means to illustrate how the we can exploit the convenience of object-orientation—as supported by, *e.g.*, C++ (*viz.*, multiple inheritance, template classes and functions, and operator overloading)—for designing a minimal set of generic classes implementing linear-equation solvers for a large variety of specific semi-ring structures. This will also illustrate a relativistic interpretation of object orientation[1] yields, with a minimal set-up, a large collection of algorithms which can all be obtained as derived classes and instance objects of a single very abstract scheme.

Because algebraic structures were invented in mathematics for the precise same purpose and use as those of object-orientation in programming, it comes as no surprise that the two paradigms match quite harmoniously. The design specified in this paper is a proof of this in the domain of linear equation solving in a variety of algebraic structures.

This document is a specification of an Application Program Interface (API) in the form of a few generic classes for linear-equation solving in an abstract *semi-ring* structure. This specification is detailed below, along with all the mathematical background that is needed to understand it.

If implemented correctly, this API can solve a variety of linear equation-solving problems ranging from familar numerical equations, to regular expression equations, to graph path problems, including network flow optimization problems [2], Abtract Interpretation of programs [3, 1], and Program flow analysis [5].

2 One Equation and One Unknown

2.1 Inverses and quasi-inverses

The left linear *fixed-point* equation:

$$x = ax + b \tag{1}$$

is easily solved in a *ring* structure by:[2]

$$\begin{aligned} x &= ax + b \\ x - ax &= b \\ (1-a)x &= b \\ x &= (1-a)^{-1}b \end{aligned} \tag{2}$$

The right linear version of Equation (1) is:

$$x = xa + b \tag{3}$$

which is, too, solved by:

$$\begin{aligned} x &= xa + b \\ x - xa &= b \\ x(1-a) &= b \\ x &= b(1-a)^{-1} \end{aligned} \tag{4}$$

If the ring is a *commutative* ring—*i.e.*, $*$ is commutative as well—then both Equations (1), and (3) "collapse" into one, and so do solutions (2), and (4) "collapse" into:

$$x = \frac{b}{1-a}. \tag{5}$$

as the solution of Equation (1). This is the most familar case for most readers of the commutative ring structure $\langle \mathbb{Q}, +, 0, *, 1 \rangle$ that most of us know for having learned it early in school. Note that for this to be a ring, the *complete* rationals must be considered; *i.e.*, $\mathbb{Q}$ must also contains a multiplicative inverse for 0 which is noted $0^{-1} \stackrel{\text{def}}{=} \infty$.

Let us define x^*, the *quasi-inverse* of x, as the infinite sum:

$$x^* \stackrel{\text{def}}{=} \sum_{n \geq 0} x^n. \tag{6}$$

This sum is well known as the simplest of all Taylor series expansion:

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + \cdots = \sum_{n=0}^{\infty} x^n = x^*. \tag{7}$$

It is then possible to rewrite the solution in (5) as either:

$$x = a^*b. \tag{8}$$

or:

$$x = ba^*. \tag{9}$$

Both can also be verified to be indeed *bona fide* solutions of Equations (1) and (3), respectively, by direct substitution:

$$\begin{aligned} a(a^*b) + b &= (aa^*)b + b \\ &= (a^* - 1)b + b \\ &= a^*b. \end{aligned}$$

and

$$\begin{aligned} (ba^*)a + b &= b(aa^*) + b \\ &= b(a^* - 1) + b \\ &= ba^*. \end{aligned}$$

The forms $x = a^*b$ and $x = ba^*$ of the solutions of Equations (1) and (3), are more general than the forms (2) and (4) in the sense

[1] See Section D.
[2] Please refer to Section B.9.

that they involve only the additive operation $+$ and the multiplicative operation $\times$, whereas the forms (2) and (4) involve also both an additive and multiplicative *inverse* operations. This is a more general property because, after all, Equations (1) and (3) use only $+$ and $\times$, no inverses. Therefore, the forms (8) and (9) may be used to compute a solution to Equations (1) and (3) for different interpretations of $+$ and $\times$, when the sets wherein a, b, and x take their values do not possess sufficient algebraic structure for $+$ and $\times$ to provide all elements with inverses. The only requirement is that the quasi-inverse's *infinite* "Taylor" expansion (6) *converge* to a *limit*; *i.e.*, it must denote a finitely expressible element, or a finitely approximable element.

Indeed, for well-known structures with different interpretations of $+$ and $\times$, such as *multiplicative semi-lattice*[3] (also known as *path algebras* [2]),[4] these operations are also idempotent and therefore quasi-inverses exist. Then, using the solution's form (8) or (9) allows to solve systems of linear equations in a wider variety of algebraic structures, including *graphs*, *regular sets*, *distributive lattices*, as well as the familiar ring structures where the form (5) happens to be more easily expressible, as well as all the multi-dimensional variations of all these structures using matrix algebra.

In *all* (!) these structures, a simple generic elimination algorithm such as, *e.g.*, the standard Gaussian elimination procedure, may be used to solve systems of linear fixed-point equations. Equation solving may be made more efficient in specific structures using the particular algebraic properties local to the specific structures. For example, the `Ring` class has both additive and multiplicative inverse methods; if it has as well exact precision, then an algorithm based on Equation (5), rather than on quasi-inverses, can be used.

2.2 Examples

2.2.1 $\langle \mathbb{Q}, +, 0, *, 1 \rangle$

This is the most familiar setting: the usual rational numbers arithmetic.[5]

This is how the structure $\langle \mathbb{Q}, +, 0, *, 1 \rangle$ is interpreted:

- it is an *Abelian ring* on the set $\mathbb{Q}$ of rational numbers;
- the *additive* operation $+$ is the addition of rationals;
- the additive unit (or *zero*) is $0 \in \mathbb{Q}$;
- the additive inverse of a rational r is its *negative* $-r$;
- the *multiplicative* operation is the multiplication of rationals;
- the multiplicative unit (or *one*) is $1 \in \mathbb{Q}$;
- the multiplicative inverse of a rational r is its *reciprocal* $\frac{1}{r}$.

2.2.2 $\langle \mathbb{R}, +, 0, *, 1 \rangle$

2.2.3 $\langle \mathfrak{RE}_\Sigma, +, \emptyset, \cdot, \epsilon \rangle$

The set $\mathfrak{RE}_\Sigma$ is the set of all *regular sets* of finite strings of symbols of an alphabet Σ (*e.g.*, as denoted by *regular expressions* on Σ).

2.2.4 $\langle \{0, 1\}, \vee, 0, \wedge, 1 \rangle$

2.2.5 $\langle \mathbb{R}, \max, -\infty, \min, \infty \rangle$

3 Many Equations and Many Unknowns

A *system* of $m > 1$ left linear equations with $n > 1$ unknowns in fix-point form is shown in Figure 1. Luckily, this case can be reduced to the previous single equation and single unknown case.

3.1 Reduction to one equation and one unknown

There are two (equivalent) ways in which this reduction can be done. The first one is based on *Dynamic Programming*, and the second one is based on *Matrix Algebra*.

3.1.1 Dynamic programming

The system of Figure 1 is expressed more concisely as:

$$S_0 = \left\{ x_i = \sum_{j=0}^{n-1} a_{ij} x_j + b_i \right\}_{i=0}^{m-1} \tag{10}$$

Expression (10) can be rewritten as:

$$S_0 = \{ x_0 = \alpha_0 x_0 + \beta_0 \} \cup S_1 \tag{11}$$

where:

$$\alpha_0 = a_{00}, \tag{12}$$

$$\beta_0 = b_0 + \sum_{j=1}^{n-1} a_{ij} x_j \tag{13}$$

[3] Please refer to Section B.6.

[4] Please refer to Section B.8.

[5] Strictly speaking, the C++ types `float` and `double` are rational numbers because they use only a finite representation. The fact that *real* numbers can be approximated by finite rational number representations is the reason why such types are also used for computing with real numbers. The only important difference to keep in mind for the latter is that finite-representation types do rounding and/or truncating beyond the precision imposed by the finite representation. Such errors propagate and therefore, comparisons among `float`s and `double`s must be done up to that precision. That is, rather than $x == y$, it is better to use $x - y < \varepsilon$, where ε is a small number (*e.g.*, $\varepsilon = 2^{-p}$, where p is any non-negative number of precision bits allowed by the representation). The lesser the *precision* p is, the slacker the approximation will be, but the faster will the convergence.

$$\begin{array}{rcccccccccccc}
x_0 & = & a_{00}\,x_0 & + & \cdots & + & a_{0j}\,x_j & + & \cdots & + & a_{0(n-1)}\,x_{n-1} & + & b_0 \\
 & \vdots & & \vdots & & \vdots & & \vdots & & \vdots & & \vdots & \\
x_i & = & a_{i0}\,x_0 & + & \cdots & + & a_{ij}\,x_j & + & \cdots & + & a_{i(n-1)}\,x_{n-1} & + & b_i \\
 & \vdots & & \vdots & & \vdots & & \vdots & & \vdots & & \vdots & \\
x_{m-1} & = & a_{(m-1)0}\,x_0 & + & \cdots & + & a_{(m-1)j}\,x_j & + & \cdots & + & a_{(m-1)(n-1)}\,x_{n-1} & + & b_{m-1}
\end{array}$$

Figure 1: System of m left linear fix-point equations with n unknowns.

and

$$S_1 = \left\{ x_i = \sum_{j=0}^{n-1} a_{ij}x_j + b_i \right\}_{i=1}^{m-1}. \tag{14}$$

Since β_0 is independent of x_0, the equation:

$$x_0 = \alpha_0 x_0 + \beta_0 \tag{15}$$

in Expression (11) is solved by:

$$x_0 = \alpha_0^* \beta_0. \tag{16}$$

Expression (16) gives x_0 only as a *parametric* solution in terms of the $n-1$ remaining *parametric* variables $x_1, \ldots, x_{n-1}$.

Substituting the value of x_0 given by Expression (16) in Expression (14), we get:

$$S_1 = \left\{ x_i = \sum_{j=1}^{n-1} a_{ij}^1 x_j + b_i^1 \right\}_{i=1}^{m-1} \tag{17}$$

where, for all $i = 1, \ldots, m-1$ and all $j = 1, \ldots, n-1$:

$$a_{ij}^1 = a_{ij} + a_{i0}a_{00}^* a_{0j}, \tag{18}$$

and for all $i = 1, \ldots, m-1$:

$$b_i^1 = b_i + a_{i0}a_{00}^* b_0. \tag{19}$$

Having proceeded thus, the new system obtained as Expression (17) is a system of $m-1$ equations and $n-1$ variables $(x_1, \ldots, x_{n-1})$. In other words, the system (17) contains one less variable (x_0 has been eliminated) and one less equation ($x_0 = \sum_{j=0}^{n-1} a_{ij}x_j$ has been eliminated).

Repeating this elimination process, it is straightforward to generalize the foregoing scheme by induction as follows. We start with the base case ($k = 0$): for all $i = 0, \ldots, m-1$ and all $j = 0, \ldots, n-1$,

$$a_{ij}^0 = a_{ij} \tag{20}$$

and, for all $i = 0, \ldots, m-1$,

$$b_i^0 = b_i. \tag{21}$$

For all k, $k = 0, \ldots, m-1$, we have,

$$S_k = \left\{ x_i = \sum_{j=k}^{n-1} a_{ij}^k x_j + b_i^k \right\}_{i=k}^{m-1}. \tag{22}$$

Expression (22) can be rewritten as:

$$S_k = \{ x_k = \alpha_k x_k + \beta_k \} \cup S_{k+1} \tag{23}$$

where, for $k = 0, \ldots, m-1$:

$$\alpha_k = a_{kk}^k, \tag{24}$$

$$\beta_k = b_k^k + \sum_{j=k+1}^{n-1} a_{ij}^k x_j. \tag{25}$$

such that, for all $i = k, \ldots, m-1$ and all $j = k+1, \ldots, n-1$:

$$a_{ij}^k = \begin{cases} a_{ij} & \text{if } k = 0, \\ a_{ij}^{k-1} + a_{i(k-1)}^{k-1}\alpha_{k-1}^* a_{(k-1)j}^{k-1} & \text{if } 0 < k < m; \end{cases} \tag{26}$$

and for all $i = 1, \ldots, m-1$:

$$b_i^k = \begin{cases} b_i & \text{if } k = 0, \\ b_i^{k-1} + a_{i(k-1)}^{k-1}\alpha_{k-1}^* b_{k-1}^{k-1} & \text{if } 0 < k < m. \end{cases} \tag{27}$$

Again, since β_k is independent of $x_0, \ldots, x_k$, the equation:

$$x_k = \alpha_k x_k + \beta_k \tag{28}$$

in Expression (23) is solved by:

$$x_k = \alpha_k^* \beta_k. \tag{29}$$

Thus, Expression (29) gives x_k as a *parametric* solution in terms of the $n-k-1$ remaining *parametric* variables $x_{k+1}, \ldots, x_{n-1}$.

Clearly, after *at most* m *steps*, this iterated parametric solving process halts. Indeed, substituting m for k in Expression (22), we obtain:

$$S_m = \left\{ x_i = \textstyle\sum_{j=m}^{n-1} a_{ij}^m x_j + b_i^m \right\}_{i=m}^{m-1} = \emptyset. \tag{30}$$

Therefore, the previous step's equational system S_{m-1} is independent of variables $x_0, \ldots, x_{m-1}$:

$$S_{m-1} = \{x_{m-1} = \alpha_{m-1} x_{m-1} + \beta_{m-1}\}. \tag{31}$$

where,

$$\alpha_{m-1} = a^{m-1}_{(m-1)(m-1)}, \tag{32}$$

$$\beta_{m-1} = b^{m-1}_{m-1} + \sum_{j=m}^{n-1} a^{m-1}_{ij} x_j. \tag{33}$$

Since β_{m-1} is independent of variables $x_0, \ldots, x_{m-1}$, the equation:

$$x_{m-1} = \alpha_{m-1} x_{m-1} + \beta_{m-1} \tag{34}$$

in Expression (31) is solved by:

$$x_{m-1} = \alpha^*_{m-1} \beta_{m-1}. \tag{35}$$

There are three situations to consider:

1. $m < n$: more variables than equations;
2. $m = n$: as many variables as equations;
3. $m > n$: more equations than variables.

This is what happens in each case:

1. $m < n$—*Underdefined system:* in this case, Equation (35) gives an expression of x_{m-1} in terms of the $n - m$ remaining variables $x_m, \ldots, x_{n-1}$. Therefore, since x_j, for $j = 0, \ldots, m-1$, depends on the $n - j - 1$ variables $x_{j+1}, \ldots, x_{n-1}$, all m variables $x_0, \ldots, x_{m-1}$ are expressed in terms of the $n - m$ remaining *parametric variables* $x_m, \ldots, x_{n-1}$.
2. $m = n$—*Well-defined system:* in this case, Equation (33) becomes $\beta_{m-1} = b^{m-1}_{m-1}$, and hence Equation (35) gives an expression of x_{m-1} independently of any variable. Therefore, since x_j, for $j = 0, \ldots, m-2$, depends on the $m - j - 1$ variables $x_{j+1}, \ldots, x_{m-1}$, all m variables $x_0, \ldots, x_{m-1}$ are expressed independently of any parametric variable. In this case the system is fully solved, and solutions are obtained by the propagation of values *from* x_{m-1} *back* to x_0.
3. $m > n$—*Overdefined system:* in this case, when we have a solution for $x_0, \ldots, x_{m-1}$ by back propagation of eliminated variables, there are still additional equations outstanding in the system. The only way the outstanding $m - n$ equations may be satisfied is if they are redundant with the m first equations; that is, if the solution $x_0, \ldots, x_{m-1}$ verifies the $m - n$ remaining equations.

If the structure $\mathfrak{R}$ happens to be a *ring* $\langle D, +, \emptyset, *, \mathbf{1}\rangle$, then the expressions solving the system in Figures 1 become, for all $k = 0, \ldots, m-1$, for all $i = k, \ldots, m-1$ and all $j = k+1, \ldots, n-1$:

$$a^k_{ij} = \begin{cases} a_{ij} \ \text{ if } k = 0, \\ a^{k-1}_{ij} + a^{k-1}_{i(k-1)} * \left(\mathbf{1} + (-\alpha_{k-1})\right)^{-1} * a^{k-1}_{(k-1)j} \\ \quad \text{if } 0 < k < m; \end{cases} \tag{36}$$

and, for all $i = 1, \ldots, m-1$:

$$b^k_i = \begin{cases} b_i \ \text{ if } k = 0, \\ b^{k-1}_i + a^{k-1}_{i(k-1)} * \left(\mathbf{1} + (-\alpha_{k-1})\right)^{-1} * b^{k-1}_{k-1} \\ \quad \text{if } 0 < k < m. \end{cases} \tag{37}$$

The equation (28) is solved by:

$$x_k = \left(\mathbf{1} + (-\alpha_k)\right)^{-1} * \beta_k \tag{38}$$

and the equation (34) is solved by:

$$x_{m-1} = \left(\mathbf{1} + (-\alpha_{m-1})\right)^{-1} * \beta_{m-1}. \tag{39}$$

We leave expressions of the right version of the ring solutions as an exercise to the reader.

3.1.2 Matrix algebra

In the case where $m = n$, the systems of Figures 1 and 2 can be respectively rewritten, using matrix notation, as:[6]

$$X = AX + B \tag{40}$$

where $X \in D^{n1}$, $A \in D^{nn}$ and $B \in D^{n1}$, and

$$X = XA + B \tag{41}$$

where $X \in D^{1n}$, $A \in D^{nn}$ and $B \in D^{1n}$.

Therefore, by Theorem B.1,[7] it comes that the solutions of Equations (40) and (41) are, respectively:

$$X = A^* B \tag{42}$$

and

$$X = BA^*. \tag{43}$$

[6] See Section B.13.
[7] See Page 8.

3.2 Examples

3.2.1 $\langle \mathbb{Q}, +, 0, *, 1 \rangle$

This structure is a *commutative* ring. Therefore, the two systems in Figures 1 and 2 are identical, and the left and right solutions collapse into one solution. Namely, for all $k = 0, \ldots, m-1$, for all $i = k, \ldots, m-1$ and all $j = k+1, \ldots, n-1$:

$$a_{ij}^k = {a'}_{ij}^k = \begin{cases} a_{ij} & \text{if } k = 0, \\ a_{ij}^{k-1} + \dfrac{a_{i(k-1)}^{k-1} a_{(k-1)j}^{k-1}}{1 - \alpha_{k-1}} & \text{if } 0 < k < m; \end{cases} \quad (44)$$

and for all $i = 1, \ldots, m-1$:

$$b_i^k = {b'}_i^k = \begin{cases} b_i & \text{if } k = 0, \\ b_i^{k-1} + \dfrac{a_{i(k-1)}^{k-1} b_{k-1}^{k-1}}{1 - \alpha_{k-1}} & \text{if } 0 < k < m. \end{cases} \quad (45)$$

Finally, Equation (28) is solved in $\langle \mathbb{Q}, +, 0, *, 1 \rangle$ by:

$$x_k = \frac{\beta_k}{1 - \alpha_k} \quad (46)$$

and Equation (34) is solved by:

$$x_{m-1} = \frac{\beta_{m-1}}{1 - \alpha_{m-1}} \quad (47)$$

where, for $k = 0, \ldots, m-1$:

$$\alpha_k = \alpha'_k = a_{k\,k}^k, \quad (48)$$

$$\beta_k = \beta'_k = b_k^k + \sum_{j=k+1}^{n-1} a_{ij}^k x_j. \quad (49)$$

3.2.2 $\langle \mathbb{R}, +, 0, *, 1 \rangle$

3.2.3 $\langle \mathfrak{RE}_\Sigma, +, \emptyset, \cdot, \epsilon \rangle$

3.2.4 $\langle \{0, 1\}, \vee, 0, \wedge, 1 \rangle$

3.2.5 $\langle \mathbb{R}, \max, -\infty, \min, \infty \rangle$

4 Three Solving Schemes

4.1 Structures with inverses and exact precision

4.1.1 $\langle \mathbb{Q}, +, 0, *, 1 \rangle$

4.1.2 $\langle \mathbb{R}, \max, -\infty, \min, \infty \rangle$

4.2 Structures without inverses, but with stationary points

4.2.1 $\langle \mathfrak{RE}_\Sigma, +, \emptyset, \cdot, \epsilon \rangle$

4.2.2 $\langle \{0, 1\}, \vee, 0, \wedge, 1 \rangle$

4.3 Structures with inverses, but no exact precision

4.3.1 $\langle \mathbb{R}, +, 0, *, 1 \rangle$

Appendix

A Right Linear Equations

A system of m right linear equations with n unknowns in fix-point form is shown in Figure 2.

The right version of all that was done for the *left* system in Figure 1 is of course valid for the *right* system in Figure 2. Namely, the base case ($k = 0$): for all $i = 0, \ldots, m-1$ and all $j = 0, \ldots, n-1$,

$${a'}_{ij}^0 = a_{ij} \quad (50)$$

and, for all $i = 0, \ldots, m-1$,

$${b'}_i^0 = b_i. \quad (51)$$

For all k, $k = 0, \ldots, m-1$, we have,

$$S'_k = \left\{ x_i = \sum_{j=k}^{n-1} x_j a_{ij}^k + b_i^k \right\}_{i=k}^{m-1}. \quad (52)$$

Expression (52) can be rewritten as:

$$S'_k = \{ x_k = x_k \alpha'_k + \beta'_k \} \cup S'_{k+1} \quad (53)$$

$$\begin{array}{rcccccccccccc}
x_0 & = & x_0 a_{00} & + & \cdots & + & x_j a_{0j} & + & \cdots & + & x_{n-1} a_{0(n-1)} & + & b_0 \\
 & \vdots & & \vdots & & \vdots & & \vdots & & \vdots & & \vdots & \\
x_i & = & x_0 a_{i0} & + & \cdots & + & x_j a_{ij} & + & \cdots & + & x_{n-1} a_{i(n-1)} & + & b_i \\
 & \vdots & & \vdots & & \vdots & & \vdots & & \vdots & & \vdots & \\
x_{m-1} & = & x_0 a_{(m-1)0} & + & \cdots & + & x_j a_{(m-1)j} & + & \cdots & + & x_{n-1} a_{(m-1)(n-1)} & + & b_{m-1}
\end{array}$$

Figure 2: System of m right linear fix-point equations with n unknowns.

where, for $k = 0, \ldots, m-1$:

$$\alpha'_k = a'^{k}_{kk}, \tag{54}$$

$$\beta'_k = b'^{k}_{k} + \sum_{j=k+1}^{n-1} x_j a'^{k}_{ij}. \tag{55}$$

such that, for all $i = k, \ldots, m-1$ and all $j = k+1, \ldots, n-1$:

$$a'^{k}_{ij} = \begin{cases} a'_{ij} & \text{if } k = 0, \\ a'^{k-1}_{ij} + a'^{k-1}_{i(k-1)} a'^{k-1}_{(k-1)j} \alpha'^{*}_{k-1} & \text{if } 0 < k < m; \end{cases} \tag{56}$$

and for all $i = 1, \ldots, m-1$:

$$b'^{k}_{i} = \begin{cases} b_i & \text{if } k = 0, \\ b'^{k-1}_{i} + a'^{k-1}_{i(k-1)} b'^{k-1}_{k-1} \alpha'^{*}_{k-1} & \text{if } 0 < k < m. \end{cases} \tag{57}$$

Hence, the equation:

$$x_k = x_k \alpha'_k + \beta'_k \tag{58}$$

in Expression (53) is solved by:

$$x_k = \beta'_k \alpha'^{*}_{k}. \tag{59}$$

After $m-1$ steps, we obtain:

$$S'_{m-1} = \{x_{m-1} = x_{m-1} \alpha'_{m-1} + \beta'_{m-1}\}. \tag{60}$$

where,

$$\alpha'_{m-1} = a'^{m-1}_{(m-1)(m-1)}, \tag{61}$$

$$\beta'_{m-1} = b'^{m-1}_{m-1} + \sum_{j=m}^{n-1} x_j a'^{m-1}_{ij}. \tag{62}$$

The equation:

$$x_{m-1} = x_{m-1} \alpha'_{m-1} + \beta'_{m-1} \tag{63}$$

in Expression (60) is solved by:

$$x_{m-1} = \beta'_{m-1} \alpha'^{*}_{m-1}. \tag{64}$$

B A Definition Hierarchy of Algebraic Structures

B.1 Primal structure

This is just the case of a domain D of elements—*i.e.*, a set—with an *internal* binary operation

$$\star : D \times D \to D. \tag{65}$$

B.2 Semi-group

This is the case of a primal structure with domain D whose operation $\star$ (65) is *associative*. That is, for all $x, y, z \in D$:

$$x \star (y \star z) = (x \star y) \star z. \tag{66}$$

B.3 Monoid

This is the case of a semi-group with a special element $\mathtt{1} \in D$ called a *unit* such that, for all $x \in D$:

$$x \star \mathtt{1} = \mathtt{1} \star x = x. \tag{67}$$

B.4 Group

This is the case of a monoid such that any element x has an *inverse*. That is, for any $x \in D$, there exists a unique $x^{-1} \in D$ such that:

$$x \star x^{-1} = x^{-1} \star x = \mathtt{1}. \tag{68}$$

B.5 Abelian structure

This is the case of any of the foregoing structures whose operation $\star$ (65) is *commutative*. That is, for all $x, y \in D$:

$$x \star y = y \star x. \tag{69}$$

Thus we speak of *Abelian* operation, *Abelian* semi-group, *Abelian* monoid, *Abelian* group.[8] Very often we say more suggestively "commutative" rather than "Abelian." Thus, *commutative* operation, *commutative* semi-group, *commutative* monoid, *commutative* group.

B.6 Semi-lattice

A *semi-lattice* $\langle D, +, \emptyset \rangle$ is a special case of a commutative semi-group such that $+$ is *idempotent*; *i.e.,* for all $x \in D$:

$$x + x = x, \tag{70}$$

and such that, for all $x, y, z \in D$:

$$\text{if } \; y \leq_+ x \; \text{ and } \; z \leq_+ x \; \text{ then } \; y + z \leq_+ x. \tag{71}$$

where the relation $\leq_+$ is defined by:

$$\forall x, y \in D, \; x \leq_+ y \; \text{ iff } \; x + y = y. \tag{72}$$

Note that semi-lattice is also a *partially ordered set* thanks to the relation $\leq_+$. Indeed, $\leq_+$ is a relation on D which is reflexive (by idempotence of $+$), anti-symmetric (by commutativity of $+$) and transitive (by associativity of $+$).

Note that when a semi-lattice is also a monoid, Equation (72) entails that $\emptyset$ is necessarily the *least* element of D for $\leq_+$.

B.7 Semi-ring

A *semi-ring* is a dual (additive and multiplicative) structure on a single set $\langle D, +, \emptyset, *, \mathtt{1} \rangle$ such that:

1. $\langle D, +, \emptyset \rangle$ is a commutative monoid;
2. $\langle D, *, \mathtt{1} \rangle$ is a monoid;
3. the multiplicative operation $*$ is *distributive* over the additive operation $+$; that is, for all $x, y, z \in D$:

$$x * (y + z) = (x * y) + (x * z) \tag{73}$$

and

$$(x + y) * z = (x * z) + (y * z). \tag{74}$$

To distinguish between the two operations's unit elements in a semi-ring, the additive unit $\emptyset$ is referred to as the *zero* element, and the multiplicative unit $\mathtt{1}$ as the *unit* element.

A semi-ring is a *commutative* or *Abelian* semi-ring if its multiplicative operation $*$ is *commutative* (*i.e.,* if $\langle D, *, \mathtt{1} \rangle$ is a commutative monoid).

[8] This is after the French mathematician Abel who worked on group theory (also known as Galois Theory, after the French mathematician E. Galois who had time to invent Group Theory before he died stupidly in a duel at a very early age (early twenties)—the night before he died, he hurriedly wrote up notes that completed a draft of his thesis work that laid the foundations of what became known as Group Theory...).

B.8 Path algebra

A *path algebra* $\langle D, +, \emptyset, *, \mathtt{1} \rangle$ is a semi-ring such that:

1. $+$ is *idempotent*; *i.e.,* Equation (70) holds for all $x \in D$.
2. $*$ is *idempotent*; *i.e.,* Equation (75) holds for all $x \in D$.

$$x * x = x; \tag{75}$$

3. $\emptyset$ is *absorptive* for $*$; *i.e.,* for all $x \in D$:

$$x * \emptyset = \emptyset * x = \emptyset; \tag{76}$$

In fact a path algebra can also be described as a multiplicative semi-lattice.

B.9 Ring

A *ring* is a special case of a semi-ring. In fact, a ring structure is to a group what a semi-ring structure is to a monoid. Indeed, a ring is a dual (additive and multiplicative) structure on a single set $\langle D, +, \emptyset, *, \mathtt{1} \rangle$ such that:

1. $\langle D, +, \emptyset \rangle$ is a commutative *group*;
2. $\langle D, *, \mathtt{1} \rangle$ is a *group*;
3. the multiplicative operation $*$ is *distributive* over the additive operation $+$; that is, Equations (73) and (74) hold for all $x, y, z \in D$.

A ring is a *commutative* or *Abelian* ring if its multiplicative operation $*$ is *commutative* (*i.e.,* if $\langle D, *, \mathtt{1} \rangle$ is a commutative group).

B.10 Lattice

A *lattice* $\langle D, +, \emptyset, *, \mathtt{1} \rangle$ is a dual structure such that:

1. $\langle D, +, \emptyset \rangle$ is a semi-lattice (also called *additive* semi-lattice);
2. $\langle D, *, \mathtt{1} \rangle$ is a semi-lattice (also called *multiplicative* semi-lattice).

Note that in a lattice, it is necessarily true that:

- $+$ is *absorptive* for $*$; *i.e.,,* for all $x, y \in D$:

$$x + (x * y) = x = (x * y) + x; \tag{77}$$

and,

- $*$ is *absorptive* for $+$; *i.e.,* for all $x, y \in D$:

$$x * (x + y) = x = (x + y) * x. \tag{78}$$

Note that a lattice is neither a case of, nor is it more general than, a semi-ring. In fact, a structure that is both a lattice and a semi-ring is necessarily a distributive lattice, which is also a *commutative* semi-ring as well as a path algebra.

Note also that a lattice is also a partially ordered set both as an additive semi-lattice and as a mutiplicative semi-lattice. By duality, it comes as a consequence that the two partial orders are mutually dual. That is,

$$\leq_+ \; = \; \leq_*^{-1}, \tag{79}$$

and

$$\leq_* \; = \; \leq_+^{-1} . \tag{80}$$

Note that if a lattice is also an additive *monoid*, then $\emptyset$ is the *least element* for $\leq_+$ as well as the *greatest element* for $\leq_*$. Moreover, it is then also necessarily true that $\emptyset$ is *absorptive* for $*$; *i.e.,* Equation (76) holds for all $x \in D$.

Dually, if a lattice is also a multiplicative *monoid* then $\mathtt{1}$ is the *least element* for $\leq_*$ as well as the *greatest element* for $\leq_+$. Moreover, it is then also necessarily true that $\mathtt{1}$ is *absorptive* for $+$; *i.e.,* for all $x \in D$:

$$x + \mathtt{1} = \mathtt{1} + x = \mathtt{1}. \tag{81}$$

A *distributive lattice* is a lattice which is also a path algebra. That is, the multiplicative operation $*$ is distributive over the additive operation $+$; that is, Equations (73) and (74) hold for all $x, y, z \in D$;[9]

Note that a distributive lattice is also necessarily a *commutative semi-ring* as well as a path algebra. In fact, a distributive lattice is simultaneously *two* mutually dual commutative semi-rings; it is simultaneously *two* mutually dual path algebras as well:

- the multiplicative operation $*$ is distributive over the additive operation $+$; that is, Equations (73) and (74) hold for all $x, y, z \in D$; and,
- the additive operation $*$ is also distributive over the multiplicative operation $+$; that is Equations (82) and (83) hold for all $x, y, z \in D$.

[9] This is equivalent, by duality, to the additive operation $+$ being also distributive over the multiplicative operation $*$; that is:

$$x + (y * z) = (x + y) * (x + z) \tag{82}$$

and

$$(x * y) + z = (x + z) * (y + z). \tag{83}$$

B.11 Boolean ring

A *boolean ring* is a ring in which any element admits a unique *complement* with respect to the additive and multiplicative operations. That is, for any $x \in D$, there exists a unique $\bar{x} \in D$ such that:

$$x + \bar{x} = \bar{x} + x = \mathtt{1}, \tag{84}$$

and

$$x * \bar{x} = \bar{x} * x = \emptyset. \tag{85}$$

B.12 Boolean lattice

A *boolean lattice* is a lattice which is also a boolean ring; *i.e.,* it is a distributive complemented lattice.

B.13 Matrix liftings

Given a semi-ring structure $\mathfrak{R} = \langle D, +, \emptyset, *, \mathtt{1} \rangle$ and two positive natural numbers m and n, we can construct its $m \times n$ *matrix lifting*:

$$\mathfrak{M}^{mn}(\mathfrak{R}) \; = \; \langle D^{mn}, +^{mn}, \emptyset^{mn}, *^{mn}, \mathtt{1}^{mn} \rangle. \tag{86}$$

as shown in Equations (87)–(91).

- Domain of $m \times n$ matrices over $\mathfrak{R}$:

$$D^{mn} \stackrel{\text{def}}{=} \{ \; d \in D^{m \times n} \mid d = \{d_{ij} \in D\}_{i=0,\; j=0}^{m-1, n-1} \; \} \tag{87}$$

- Addition of $m \times n$ matrices over $\mathfrak{R}$: if $a \in D^{mn}$ and $b \in D^{mn}$ then $\forall i, j, \; 0 \leq i \leq m-1, 0 \leq j \leq p-1$,

$$(a +^{mn} b)_{ij} \stackrel{\text{def}}{=} a_{ij} + b_{ij}; \tag{88}$$

- Zero $m \times n$ matrix over $\mathfrak{R}$: $\forall i, j, \; 0 \leq i \leq m-1, 0 \leq j \leq n-1$,

$$\emptyset_{ij}^{mn} \stackrel{\text{def}}{=} \emptyset \in D; \tag{89}$$

- Multiplication of $m \times n$ and $n \times p$ matrices over $\mathfrak{R}$: if $a \in D^{mn}, b \in D^{np}$, then $\forall i, j, \; 0 \leq i \leq m-1, 0 \leq j \leq p-1$,

$$(a *^{mp} b)_{ij} \stackrel{\text{def}}{=} \sum_{k=0}^{n-1} a_{ik} \times b_{kj}; \tag{90}$$

- One $m \times n$ matrix over $\mathfrak{R}$: $\forall i, j, \; 0 \leq i \leq m-1, 0 \leq j \leq n-1$,

$$\mathtt{1}_{ij}^{mn} = \begin{cases} \mathtt{1} \in D & \text{if } i = j, \\ \emptyset \in D & \text{otherwise.} \end{cases} \tag{91}$$

We can simplify the $\_^{mn}$ notation in Equations (88)–(91) by dropping the dimension superscripts, with the dimension constraints implicit. Hence, Equations (88)–(91) become Equations (92)–(95):

$$(a+b)_{ij} \stackrel{\text{def}}{=} a_{ij} + b_{ij}; \tag{92}$$

$$\emptyset_{ij} \stackrel{\text{def}}{=} \emptyset \in D; \tag{93}$$

$$(a * b)_{ij} \stackrel{\text{def}}{=} \sum_{k=0}^{n-1} a_{ik} \times b_{kj}; \tag{94}$$

$$\mathbf{1}_{ij} = \begin{cases} \mathbf{1} \in D & \text{if } i = j, \\ \emptyset \in D & \text{otherwise.} \end{cases} \tag{95}$$

An element $d = \{d_{ij} \in D\}_{i=0,\ j=0}^{m-1,n-1}$ of D^{mn} is written:

$$\begin{bmatrix} d_{00} & \cdots & d_{0j} & \cdots & d_{0(n-1)} \\ \vdots & \ddots & \vdots & & \vdots \\ d_{i0} & \cdots & d_{ij} & \cdots & d_{i(n-1)} \\ \vdots & & \vdots & \ddots & \vdots \\ d_{(m-1)0} & \cdots & d_{(m-1)j} & \cdots & d_{(m-1)(n-1)} \end{bmatrix} \tag{96}$$

Given a *matrix* $d^{mn} \in D^{mn}$, the ordered pair mn is called the *dimension* of the matrix: m is called the *row dimension* and n is called the *column dimension*. Note that the multiplicative matrix operation is *not* an internal function, but can only be applied if the first matrix' column dimension is equal to the second matrix' row dimension. However,

THEOREM B.1 *Given a semi-ring $\mathfrak{R}$, its matrix lifting $\mathfrak{M}^{n^2}(\mathfrak{R})$ for n fixed, is also a semi-ring.*

C A Class Hierarchy of Algebraic Structures

C.1 Primal structure class

This is the most abstract class from which all others are derived. It consists of a *template* class, `Primal_Structure`, with parameterized type `Domain`. It has just one method: `op`, which takes a reference to another `Primal_Structure<Domain>` object and implements the binary operation of the structure. It also has a one-argument constructor which takes a pointer to an operation with which to initialize the method `op`. It also allows the *friend* function `op`.

C.2 Semi-group class

C.3 Monoid class

C.4 Group class

C.5 Abelian structure class

C.6 Semi-lattice class

C.7 Semi-ring class

C.8 Path algebra class

C.9 Ring class

C.10 Lattice class

C.11 Boolean ring class

C.12 Boolean lattice class

D A Relativistic View of Object Orientation

The essence of object-orientation coincides with that of Einstein's Special and General Relativity theories [4].

Einstein's Special Relativity Theory (SRT) is all based on the observation that there is a mathematical duality between being at rest on one hand, and being in motion on the other hand: all motion is relative to a set of reference. Hence, it is mathematically irrelevant whether I sit in a train moving along with it at some speed with respect to the scenery, or whether I sit in a motionless train while the scenery moves by in the opposite direction at the same speed.

Similarly, Einstein's General Relativity Theory (GRT) is all based on the observation that there is a mathematical duality between free falling frictionless in a straight line on one hand, and the texture of space being warped by massive bodies on the other hand: the curvature of all trajectory of motion is relative to space's own curvature. Hence, it is mathematically irrelevant whether the Earth is orbiting the Sun elliptically is a closed curve, or whether it free-falls frictionless indefinitely in a straight line, while space in which it moves is itself curved by the same opposite factor into the (hyper) elliptical (hyper) "eddy" created by the Sun's gravity.[10] Thus is GRT the key to explaining the mystery of "action at a distance" of gravity.

Similarly as well, object-orientation (OO) is based on the observa-

[10] "Hyper" because space is at least 3-dimensional . . .

tion that there is a mathematical duality between an object being acted upon by a function on one hand, and a function being acted upon by an object on the other hand: the *orientation* of $f(x)$ is relative to the structure of interpretation of the object or the function. Hence, it is mathematically irrelevant whether the function f is applied to the object x, or whether the object x is *sent* the *message* f. In the first case (the conventional view), the function f *knows* what to do with an object of the type of x and performs it on x; in the second case (the *object-oriented view*), the object x *knows* what to do when it is asked to respond to the message sent to it as f, and performs it. Thus is OO the key to a new *decentralizing* view of computation which allows *distributed* computation and code modularity: whereas the conventional view's *centralizing* computation in functions made them huge, inefficient, and quickly impractical to maintain, the (mathematically equivalent) OO view now delegates computation to objects by making them react to messages sent to them by using methods specified for them by their class definitions.

Thus, object-orientation may simply be construed as exploiting a mathematical relativity principle. This relativistic view can be used as a systematic object-oriented software design methodology.

To be precise, the change of perspective, when orienting computation with *reference* to an object rather than a function, is expressed mathematically by the set isomorphism:

$$A \to (B \to C) \simeq B \to (A \to C). \tag{97}$$

This equation essentially captures the dual *relativity* of computation alluded to above.

This article is an example of the general case that can be expressed as follows:

$$\begin{aligned} &\mathit{method} : \mathit{Context} \to (\mathit{Object} \to \mathit{Object}) \\ &\qquad\qquad \simeq \\ &\mathit{method} : \mathit{Object} \to (\mathit{Context} \to \mathit{Object}). \end{aligned} \tag{98}$$

Therefore, we can define two class structures, `Object` and `Context`, which *always* respectively declare a method (here called `method`) as shown in Figures 3 and 4.[11] Some examples are given in Figure 5.

E Implementation

A simple solver over an algebraic dual structure (the parameter class `Structure`) should provide:

- a class to substitute for `Structure`, the type of elements in the structure's domain. This is the type of the coefficients `a`, and `b`, and that of the unknown `x` as well. This algebraic structure class will have:
  - a *private* member `rep` whose type is an adequate representation of the structure's domain elements.
  - a *public* `friend` method `operator+` that takes two arguments of type `const &Structure` and returns a result of type `&Structure`;[12]
  - a *public* `friend` method `operator-` that takes *one* argument of type `const &Structure` and returns a result of type `&Structure`;
  - a *public* `friend` method `operator-` that takes *two* arguments of type `const &Structure` and returns a result of type `&Structure`;
  - a *public* `const Structure zero;`
  - a *public* `friend` method `operator*` that takes two arguments of type `const &Structure` and returns a result of type `&Structure`;
  - a *public* `friend` method `operator/` that takes two arguments of type `const &Structure` and returns a result of type `&Structure`;
  - a *public* `const Structure one;`
  - a *public* `friend` method `operator==` that takes two arguments of type `const &Structure` and returns a result of type `bool`;
- a class `Equation` representing a linear fix-point equation on the `Structure`s; this class must have a *private* member `structure` of type `*Structure`;

We must also provide the *methods* `solve` for both `Structure` and `Equation<Structure>` classes following the design scheme of Section D.

For example, solving over rational numbers should provide:

- a class `Rational` representing a rational number; this can be represented by pairs of integers, or decimal doubles, or whatever other equivalent representation of a rational number one may decide;[13]
  - a *private* member `rep` of, say, type `double`;
  - a *public* `friend` method `operator+` that takes two arguments of type `const &Rational` and returns a result of type `&Rational`;

[11] Using C++ syntax:.

[12] The & return type may appear odd; however, keep in mind that the *generic* design eventually will actually allow the overloading of operators on *very big* structures such as, *e.g.*, matrices (*i.e.*, multidimensional arrays), and therefore saving the return copy space/time is worth saving. Be that as it may, we are free to choose to return a `Rational` instead of `&Rational` if we so wish. The & return type version has the advantage of generic uniformity for inheritance if doing the complete generic API.

[13] Recall that a rational number $r \in \mathbb{Q}$ is a pair of integers written $r = \frac{n}{d}$, where $n \in \mathbb{N}$ is the *numerator* and $d \in \mathbb{N}$ is the *denominator*, or equivalently as a number in decimal "dot" notation written $r = i.d$, where $i \in \mathbb{N}$ is the *integer part* and $d \in \mathbb{N}$ is the *decimal part*.

```
class Object
  {
    virtual Object *method (Context *context);
  }
```

Figure 3: Object class skeleton

```
class Context
  {
    Object *method (Object *object) { return object.method(this); }
  }
```

Figure 4: Context class skeleton

  - a *public* `friend` method `operator-` that takes *one* argument of type `const &Rational` and returns a result of type `&Rational`;
  - a *public* `friend` method `operator-` that takes *two* arguments of type `const &Rational` and returns a result of type `&Rational`;
  - a *public* `const Rational zero(0.0);`
  - a *public* `friend` method `operator*` that takes two arguments of type `const &Rational` and returns a result of type `&Rational`;
  - a *public* `friend` method `operator/` that takes two arguments of type `const &Rational` and returns a result of type `&Rational`;
  - a *public* `const Rational one(1.0);`
  - a *public* `friend` method `operator==` that takes two arguments of type `const &Rational` and returns a result of type `bool`;

- a class `Equation` representing a linear fix-point equation on the `Rationals`; this class must have a *private* member `structure` of type `*Rational`;

We must also provide the *methods* `solve` for both `Rational` and `Equation<Rational>` classes.

F Discussion

F.1 Purpose of `structure`

The class `Equation` representing a linear fix-point equation on the `Rationals` actually does not need to have the `structure` member of type `*Rational`. In fact, this member comes in handy only when carrying out the implementation for arbitrary semi-rings.

If one does not wish carry out the implementation for arbitrary semi-rings, this member should be inherited by the instance `Equation<Rational>` from a generic class `Equation<Semi_Ring>`. In the latter, the `structure` member is of type `Semi_Ring`, the type parameter.

The hierarchy of algebraic structures of semi-rings, or special cases of semi-rings, can easily be encoded as a class hierarchy deriving from a base class `Dual_Structure`:[14]

Figures 6, 6, and 6 shows the inheritance relation for the dual algebraic structures that were defined in Sections B.7–B.12. Each of these classes is parameterized by the type variable `Domain`, of its (private) representation. The class `Rational` is therefore a subclass of `Abelian_Ring<double>`.[15]

The figures at the end of this paper show a skeleton for a C++ implementation of a solver using dynamic programming.[16]

F.2 Purpose of `Rational::solve(Equation)`

We would not need to worry about invoking `Rational::solve(Equation)` unless the system also means to allow the scheduling of the simultaneous resolution of several systems from the context of a given semi-ring structure. Only then is this method needed.

F.3 Testing the design

Since $\mathbb{Q}$ is not quite a ring (because 0 has no multiplicative inverse), we must test whether `a[0][0]` is equal to `structure.one`. If so, the equation $x = x + b$ is solvable only if the structure is an additive semi-lattice—that is, has an idempotent plus (*i.e.*, such that

[14] We do not have to include all these classes, of course, unless we actually want to implement a complete API library...

[15] Assuming that we use a `double` to represent a rational number in $\mathbb{Q}$.

[16] Please note the "informal" C++ syntax... This is just a program skeleton, not a complete solution.

Context	Object	method
`Name_Value_Environment`	`Expression`	`evaluate`
`Name_Type_Environment`	`Expression`	`typecheck`
`Run_Time_Environment`	`Instruction`	`execute`
`Algebraic_Structure`	`Equation`	`solve`
`Logical_Theory`	`Theorem`	`prove`
`Constraint_Structure`	`Constraint`	`resolve`
`WinMain()`/window function		

Figure 5: Some instances using the context/object relativity principle

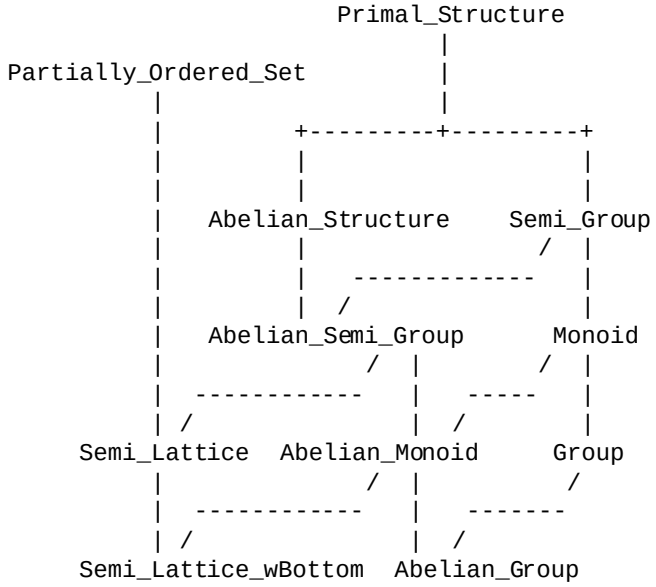

Figure 6: A Hierarchy of Primal Algebraic Structures

$x + x = x$). In this case, any x such that $b \leq_+ x$ is a solution. The alternatives are:

- abort solving;
- if underdefined, give a parameterized solution;
- if overdefined:
  - solve for the square subsystem, and check whether or not the partial solution satisfies the outstanding equations;
  - solve for the *least-squares*.[17]

Finally, let us note that the skeleton given above can solve only for *well-defined* systems, and aborts otherwise. One should use exceptions for a more graceful control.

[17] The *least-square* approximant of a system in canonical form $Ax + b = 0$ that is overdefined is the well-defined system $A^t Ax + A^t b = 0$. This system is always square and can therefore be solved. Its solution x_{lq} is such that its "distance" from any solution x of the overdefined system $Ax + b = 0$—i.e., the inner product $(x - x_{lq})^t(x - x_{lq})$—is minimal; that is, $\forall x, \emptyset \leq_+ x - x_{lq}$.

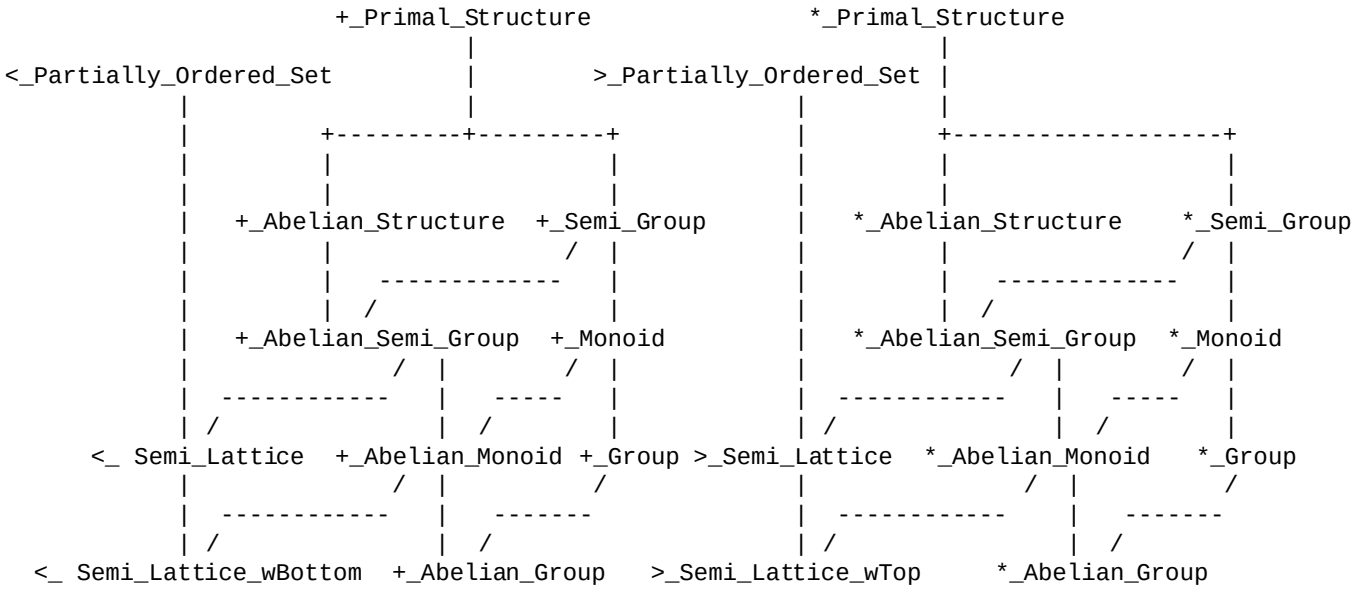

Figure 7: A Hierarchy of Additive and Multiplicative Primal Algebraic Structures

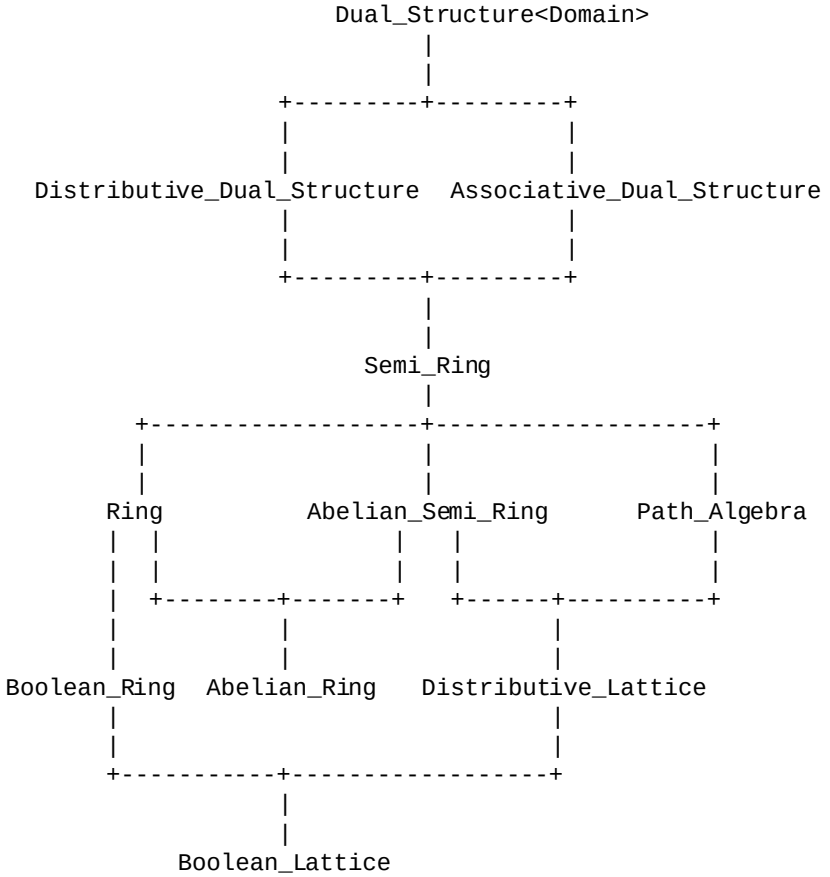

Figure 8: A Hierarchy of Dual Algebraic Structures

```
//  FILE. . . . /usr/home/hak/technical/papers/lineq/lineq.h
//  EDIT BY . . . Hassan Ait-Kaci
//  ON MACHINE. . Muenster
//  STARTED ON. . Fri Dec 11 09:43:33 1998

//  Last modified on Fri Dec 11 11:20:57 1998 by Hak

#ifndef LINEQ_H
#define LINEQ_H

template <class Structure>
class System;

template <class Structure>
class Equation
{
  Structure *a;
  Structure *b,
  Structure *x;

  bool left;

public:

  Structure *a () { return a; }
  Structure *b () { return b; }
  Structure *x () { return x; }

  bool isLeft () { return left; }

  Equation (Structure &a, Structure &b, bool left=true)
    : a      (a)
    , b      (b)
    , left (left)
    {
      solve();
    }

  Equation (System<Structure> s, bool left=true)
    {
      this = s.solve(left);
    }

  Equation *solve ()
    {
      x = isLeft() ? a().quasi_inverse() * b()
                   : b() * a().quasi_inverse();

      return this;
    }
};
```

Figure 9: Header File - part I

```
const int DefaultNumberOfEquations = 2;
const int DefaultNumberOfUnknowns  = 2;

template <class Structure>
class System
{
  int m = DefaultNumberOfEquations;
  int n = DefaultNumberOfUnknowns;

  Structure* a[m][n];
  Structure* b[m];
  Structure* x[n];

  bool left = true;

public:

  int numberOfEquations () { return m; }
  int numberOfUnknowns  () { return n; }

  Structure* a[][] () { return a; }
  Structure* b[]   () { return b; }
  Structure* x     () { return x; }

  bool isLeft () { return left; }

  System (Structure* a[], Structure* b[], bool left)
    : m      (sizeof(b))
    , n      (sizeof(a)/m)
    , a      (a)
    , b      (b)
    , left (left)
    {
      if (m == 0 | m != n) exit(1);
    }

  Equation<Structure> *solve ();

};

#endif
```

Figure 10: Header File - part II

```
#include "lineq.h"

template <class Structure>
Equation<Structure> *System<Structure>::solve ()
{
  Structure* newa[][], newb[];
  Equation<Structure> *eq;
  int i,j;

  if (m == 1) return new Equation<Structure>(a[0][0],b[0],left);

  newa = new Structure[m-1][n-1];
  newb = new Structure[m-1];

  Structure qi = a[0][0].quasi_inverse();

  if (isLeft())
    for (i=1;i<=m-1;i++)
      {
        for (j=1;j<=n-1;j++) newa[i-1][j-1] = a[i][j] + a[i][0] * qi * a[0][j];
        newb[i-1] = b[i] + a[i][0] * qi * b[0];
      }
  else
    for (i=1;i<=m-1;i++)
      {
        for (j=1;j<=n-1;j++) newa[i-1][j-1] = a[i][j] + a[i][0] * a[0][j] * qi;
        newb[i-1] = b[i] + a[i][0] * b[0] * qi;
      }

  eq = new Equation<Structure>(new System<Structure>(newa,newb,left),left);

  return new Equation<Structure>(a[0][0],
                                 b[0] + (left ? a[0][1] * eq->x
                                              : eq->x * a[0][1]),
                                 left);
}
```

Figure 11: Implementation File

References

[1] Fran¸ois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 1–15, 1986.

[2] Bernard Carré. *Graphs and Networks.* Oxford University Press, 1979.

[3] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Pogramming*, 13(2-3):103–179, 1992.

[4] Albert Einstein. *Relativity—The Special and the General Theory.* Crown Publishers, Inc., New York, NY, 1961.

[5] Steven S. Muchnick and Neil D. Jones. *Program Flow Analysis: Theory and Application*. Prentice Hall, 1981.