Equational Reasoning in Programming

Tetsuo Ida
Institute of Information Sciences and Electronics
University of Tsukuba
Tsukuba, 305-8573, Japan
email: ida@score.is.tsukuba.ac.jp

Abstract
Equality plays an important role in our life, and we practise equational reasoning everyday. We can take advantage of our ability of reasoning with equalities and make explicit the equational reasoning in programming and symbolic computation. Based on this observation we developed an equational programming system called CFLP (Constraint Functional Logic Programming system). We present various examples to show the importance of equations in programming.

1. Introduction

Modern life demands a certain level of mathematical maturity. One of the most important is the ability to reason with equalities. We do sophisticated reasoning with equalities in arithmetic in everyday life, although we are often unaware of it.

Let us take a concrete example. Suppose we buy an item priced at 975 yen. Since in Japan consumer tax is 5 %, we have to pay over 1000 yen. We calculate the exact amount of money that we have to pay, while at the same time we fumble in the pocket and try to find appropriate coins and notes so as to avoid filling up the pocket with many coins of change. We may decide to give a note of 5000 yen and three 10 yen coins. How do we decide?

The involved reasoning is complex; it is not mere simplification of numerical expressions. It involves equational reasoning. In the above example we need at least 6 steps of equational reasoning even if integer arithmetic is assumed. Most people can somehow perform this kind of mathematics. Indeed they master it at a fairly early age. They can comfortably handle reflexivity, symmetry and transitivity of equality relation defined over numbers.

In this paper we will show that reasoning with equality over various domains of objects is also important, and easy to practise if we are provided with appropriate tools for reasoning. One particular example that we are interested in here, and is relevant to mathematics education, is programming. All the programming examples including these texts are in *Mathematica* Notebook.

2. Equality in Programming

We will first show a very basic programming system, a subset of *Mathematica*, and extend it to a system for equational programming. We begin by defining terms with which we construct a program. Our vocabulary $\mathcal{S}$, called *signature*, is given as a set of symbols {0, s, $\oplus$, $\otimes$}. The symbols 0, s, $\oplus$ and $\otimes$ take 0, 1, 2 and 2 arguments, respectively. These numbers are called arity. Furthermore we will use unlimited number of variables, say *x*, *y*, *z*, ... Then the syntax of terms is as follows:

0 is a term.

x is a term if *x* is a variable.

$f[t_1, ..., t_n]$ is a term if $t_1, ..., t_n$ are terms where *n* is the arity of *f* and $f \in \mathcal{S}$.

Nothing other than those constructed in this way is a term.

At this point we only have syntactic equality, i. e. terms are equal only if they look the same. Really interesting things emerge when we introduce rewriting rules on the domain of the terms. We define rewrite rules for the symbols $\oplus$ and $\otimes$. For readability we use those symbols as infix operators. The rewrite rules are specified in the following way in *Mathematica*.

■ Rewrite rules

```
0⊕y_ := y;
s[x_]⊕ y_ := s[x⊕y] ;
0⊗y_ := 0;
s[x_]⊗y_ := (x⊗y)⊕y :
```

In order to distinguish variables from non-variable symbols, *Mathematica* has the convention that the variables on the left hand side of the rewrite rule are marked by _ (underscore).

These four lines of rewrite rules define a term rewrite system $\mathcal{R}$. $\mathcal{R}$ induces the reduction relation $\rightarrow_{\mathcal{R}}$.

This means that for any terms *s* and *t* in $s := t \in \mathcal{R}$ and any substitution θ, $s\,\theta \rightarrow_{\mathcal{R}} t\,\theta$ and for any context $C[\ ]$, $C[u] \rightarrow_{\mathcal{R}} C[v]$, if $u \rightarrow_{\mathcal{R}} v$.

Here, context $C[\ ]$ denotes a term that has a hole to be filled by some term.

The reflexive and transitive closure of $\rightarrow_{\mathcal{R}}$ is denoted by $\rightarrow^*_{\mathcal{R}}$. The reflexive, transitive and symmetric closure of $\rightarrow_{\mathcal{R}}$ is denoted by $\leftrightarrow^*_{\mathcal{R}}$. Finally we identify equality $==_{\mathcal{R}}$ with $\leftrightarrow^*_{\mathcal{R}}$.

A purely functional language is based on the notion of rewriting terms by $\rightarrow_{\mathcal{R}}$. The term which is no longer related by $\rightarrow_{\mathcal{R}}$ to any term is called a ($\rightarrow_{\mathcal{R}}$) normal form. For example, we see the term `s[s[0]]⊗s[s[s[0]]]` is reduced to its normal form.

```
s[s[0]]⊗s[s[s[0]]]

s[s[s[s[s[0]]]]]
```

When $\mathcal{R}$ is understood from the context, we will drop subscript $\mathcal{R}$ in the above relations.

Handling equality is more difficult in general than reduction. To see if $s =\!=_{\mathcal{R}} t$, we have to show that s and t are related by $\rightarrow_{\mathcal{R}}$ via several terms. We do not know in advance in what way we should rewrite s and t so that the rewrites eventually lead to the same term. This seems a formidable problem, unless we know certain properties of the term rewrite system $\mathcal{R}$.

Equality in programming is discussed in a more challenging context. Instead of proving $\forall x_1,\ldots,x_n \,.\, s =\!=_{\mathcal{R}} t$, we want to prove $\exists x_1,\ldots,x_n \,.\, s =\!=_{\mathcal{R}} t$, where we are interested not only in the truth of the statement, but a substitution θ that makes $s\theta =\!=_{\mathcal{R}} t\theta$. The computation to find the substitution is called *solving* in this paper.

In *Mathematica* we have a special function called `Solve` which computes such substitutions.

- **Solving equations: Cranes and Tortoise (cats and birds) problem**

There are 32 legs and 10 heads of tortoises and cranes. The number of tortoises and cranes is found by applying `Solve` to equations:

```
Solve[{cranes + tortoises == 10, 2 cranes + 4 tortoises == 32},
 {cranes, tortoises}]
```

```
{{cranes → 4, tortoises → 6}}
```

We obtain the substitution {cranes→4, tortoises→6} as the answer.

- **Solving equations on the domain of terms**

Then, we will try to solve a similar problem over the domain of terms that we have defined.

```
Solve[{x⊕y == s[s[s[0]]]}, {x, y}]
```

```
Solve::dinv :
 The expression x⊕y involves unknowns in more than one
   argument, so inverse functions cannot be used.
```

```
Solve[{x⊕y == s[s[s[0]]]}, {x, y}]
```

Mathematica does not give solutions that we would expect. You are invited to find the reason why it does not give the solutions.

The important observation that has to be made is that the symbols that we are using are uninterpreted, i.e. they are used as mere symbols. The symbol 0, for instance, is not an integer zero in this setting. We do not assume any mathematical properties on the symbols except that we define rewrite relations and equality induced by $\rightarrow_{\mathcal{R}}$. The situation is very different from the case of solving linear equations.

3. Narrowing

Fortunately, we already have a method called *narrowing* for solving equations over the domain of terms. Narrowing is a procedure to prove existentially quantified equations by presenting values that bind the existential variables Formally, given a rewrite system $\mathcal{R}$ and a sequence of equations $s_1 == t_1, \ldots, s_n == t_n$, it computes a substitution θ such that $s_k\theta ==_{\mathcal{R}} t_k\theta$ for $k = 1, \ldots, n$. The basic idea is similar to rewriting; rewrite the terms of both sides of an equation repeatedly until they become the same term using the rewrite rules and the substitutions. In narrowing, the substitutions are used not only to substitute terms for the variables in rewrite rules, but also in equations to be solved.

This method can be formalized as a calculus, which we call *lazy narrowing calculus*. The calculus is called *lazy* because we incorporate in the calculus an algorithm for systematically identify and rewrite a certain preferred parts of equations. The lazy narrowing calculus is the interpreter of the programming language, which we discuss in the next section.

4. Language for solving equations

We now define the language for our equational programming. The signature consists of two disjoint sets of function symbols; the set of constructors and the set of defined function symbols. In our previous example, s is a constructor symbol, and $\oplus$ and $\otimes$ are defined function symbols. The defined function symbols are associated with rewrite rules, whereas the former is not. The syntax of the terms is as given before except that they are (sometimes implicitly) typed. With that syntax we will represent a simply typed λ-terms, e.g. $\lambda[\{x, y\}, \text{plus}[x, y]]$: int $\rightarrow$ int $\rightarrow$ int.

The equations to be solved is often called a *goal*. To avoid confusion of a *Mathematica*'s built-in equation $s == t$, we hereafter denote our equation by $s \approx t$.

The program is given by a higher-order rewrite system called pattern rewrite system. With higher-order rewrite system we can treat functions systematically. Moreover, we can find higher-order solutions, i.e. functions, wherever possible.

A pattern rewrite system is a set of unconditional rewrite rules

$$f[t_1, \ \ldots, \ t_n] \rightarrow t,$$

or conditional rewrite rules

$$f[t_1, \ldots, t_n] \rightarrow t \Leftarrow E.$$

We have certain restrictions on the syntactic structure and the types of the terms that can be used to form a pattern rewrite system (See [1] for technical details).

5. Solving equations over the domain of terms

■ Solving first-order equations

We return to the problem of solving the equation in Section 2 with our system [2]. The system is called Constraint Functional Logic Programming system (CFLP for short). CFLP is implemented as a package of *Mathematica*. After loading CFLP package, we start a CFLP session by declaring the signature. $\mathbb{Z}$ denotes some first-order basic type, in this case type integer.

```
<< FrontendCFLP.m
```

```
DataConstructor[s : ℤ → ℤ]
```

`CirclePlus` and `CircleTimes` are actual function names of $\oplus$ and $\otimes$, respectively.

```
(* We clear previous definitions
  about CirclePlus and CircleTimes *)
Clear[CirclePlus, CircleTimes];
```

```
DefinedSymbol[CirclePlus : ℤ → ℤ → ℤ, CircleTimes : ℤ → ℤ → ℤ]
```

The following is our program assigned to a variable **R.** All the rewrite rules in this example turn out to be unconditional. In CFLP we have to declare all function symbols. Hence the other symbols are automatically identified as variables. So we do not have to mark variables with _.

```
R = FLPProgram[{
    0⊕y → y, s[x]⊕y → s[x⊕y],
    0⊗y → 0, s[x]⊗y → (x⊗y)⊕y}];
```

This is the goal to be solved.

```
G1 = {exists[{X, Y}, X⊕Y ≈ s[s[s[0]]]]}
```

```
{∃{X:ℤ,Y:ℤ} X⊕Y ≈ s[s[s[0]]]}
```

We specify the solver to solve the above goal. Our system is designed to be general, in that we can also specify solvers for the problem and the strategy to apply various solvers. We omit the explanation of the following two lines of program, as we discuss the solver collaboration in later sections. The reader can see that we use Lazy Narrowing solver (LNSolver) for solving our problem. Actually, the lazy narrowing solver consists of several narrowing calculi, and we will use the one called LCNCd tailored to the first-order solving.

```
(* solver *)
flp = MkLocalSolver["LNSolver`"];
```

```
ConfigSolver[flp, {Program → R1, Calculus -> "LCNCd"}]
```

Computation, i.e., solving the equation, starts when we apply the solver LNSolver to the goal.

```
ApplyCollaborative[flp, G1]
```

Then LNSolver yields the following solution:

```
LNSolver` yields
```

```
{{X → 0, Y → s[s[s[0]]]}, {X → s[0], Y → s[s[0]]},
 {X → s[s[0]], Y → s[0]}, {X → s[s[s[0]]], Y → 0}}
```

We can ask ourselves the following questions. Are these correct? Are these all the solutions? The former is concerned with so-called soundness, and the latter with completeness. There are several ways that you can convince yourselves of the affirmative of these questions. Concerning the soundness, we apply the substitutions to the goal, and rewrite both sides of the equation. In our case the right hand side is already a normal form. So what you have to do is to reduce the left hand side of the equation. Another way of convincing yourselves is to use a model. Interpret 0 as the natural number zero, s as a function of increment by one over the natural numbers. You will see that $\oplus$ and $\otimes$ are addition and multiplication operators on natural numbers. Hence the goal is actually $x + y = 3$, and we solve for x and y in the domain of natural numbers. It is easy to see (by simple enumeration) that we have 4 solutions, i.e., $(x, y) = (0,3), (2,1), (1,2), (3,0)$. These correspond to the solutions that we obtained.

Proving soundness and completeness of the calculus in general setting requires deeper investigation [3].

■ Solving higher-order equations

The next example is more difficult. It involves higher-order, i.e. function, variables. Even in high school mathematics, we often encounter higher-order mathematical objects, and equations involving higher-order objects. However, mostly problems are restricted to obtain first-order quantities. In the following example, we want to solve for higher-order variables, i.e. we want to obtain a function as a solution. As in the previous example, *Mathematica* is able to give solutions of certain kind of equations for certain domains. For example, *Mathematica*'s DSolve gives solutions to differential equations.

```
DSolve[ f'[x] == x, {f}, {x}]
```

$$\left\{\left\{f \to \text{Function}\left[\{x\}, \frac{x^2}{2} + C[1]\right]\right\}\right\}$$

In the domain of terms, however, this is not the case. Solving for higher-order variables is non-trivial task. For certain class of programs, we do have a method for solving equations over the domain of terms. The method is generically called *higher-order narrowing.* As in the first-order case, we can formalize the higher-order narrowing as a narrowing calculus. So in this case let us use higher-order lazy narrowing calculus HOLN. Our problem is as follows:

```
G2 = {exists[{F, Y : ℤ → ℤ},
   λ[{x}, F[x, Y[x]]] ≈ λ[{x}, x⊕s[s[s[0]]]]]}
```

$$\{\exists_{\{F:\mathbb{Z}\to\mathbb{Z}\to\mathbb{Z},\,Y:\mathbb{Z}\to\mathbb{Z}\}}\ \lambda[\{x:\mathbb{Z}\},\ F[x,\ Y[x]]] \approx \lambda[\{x:\mathbb{Z}\},\ x\oplus s[s[s[0]]]]\}$$

We want to solve this equation for higher-order variables. For those who are not familiar with the lambda calculus, read λ as Function.

```
holn = MkLocalSolver["LNSolver`"];
ConfigSolver[holn, {Program → R, Calculus -> "HOLN"}]
```

```
ApplyCollaborative[holn, G2]
```

Then, CFLP returns the following solutions.

```
LNSolver` yields
```

```
{{F→λ[{x1 : ℤ, x2 : ℤ}, x1⊕s[s[s[0]]]]},
 {F→λ[{x1 : ℤ, x2 : ℤ}, x1⊕s[s[s[x2]]]], Y→λ[{x57 : ℤ}, 0]},
 {F→λ[{x1 : ℤ, x2 : ℤ}, x1⊕s[s[x2]]], Y→λ[{x45 : ℤ}, s[0]]},
 {F→λ[{x1 : ℤ, x2 : ℤ}, x1⊕s[x2]], Y→λ[{x31 : ℤ}, s[s[0]]]},
 {F→λ[{x1 : ℤ, x2 : ℤ}, x1⊕x2], Y→λ[{x19 : ℤ}, s[s[s[0]]]]},
 {F→λ[{x1 : ℤ, x2 : ℤ}, x2⊕s[s[s[0]]]], Y→λ[{x9 : ℤ}, x9]},
 {F→λ[{x3 : ℤ, x4 : ℤ}, x4], Y→λ[{x7 : ℤ}, x7⊕s[s[s[0]]]]}}
```

Let us take a closer look at the second solution: `{F→λ[{x1:ℤ,x2:ℤ}, x1⊕s[s[s[x2]]]], Y→λ[{x57:ℤ},0]}`. In order to solve the higher-order equation, the system needs type information. The solution is attached with types by the system. Let us ignore the types temporarily and describe the answer in the mathematical representation familiar to college students.

$$F[x, y] = x \oplus s[s[s[y]]]$$

$$Y[x] = 0$$

It is easy to see that those are indeed solutions.

6. Solving equations over various domains

Many scientific problems are modeled as a set of equations. Specialized algorithms have been developed for solving various equations over various domains. The domain of terms that we discussed in the previous sections is very important since it is in this domain that equational programs are interpreted. Equations and rewrite rules are regarded as programs. These programs are different from those of procedural programming languages such as JAVA and C.

The next challenge is whether we can combine solvers to make a single framework in which specific solvers are called for specific problems automatically. CFLP is actually designed to work in this way. CFLP is at present equipped with four solvers including HOLN. The system coordinates those solvers to work on a given goal. A programmer can either use a default combination of solvers or can program the collaboration of solvers using a simple coordination language.

Below we explicitly declare solvers and put the references to solvers in variables `holn`, `elim`, `deriv` and `polyn`. The latter three variables hold the references to the solver for a system of linear equations which implements Gaussian elimination method, the solver for partial and differential equations, and the solver for general polynomial equations which implements Gröbner basis algorithm, respectively.

```
(* solvers *)
holn = MkLocalSolver["LNSolver`"];
elim = MkLocalSolver["ElimSolver`"];
deriv = MkLocalSolver["DerivSolver`"];
polyn = MkLocalSolver["PolynSolver`"];
```

Using these elementary solvers we can define a new solver that combines these solvers. The following defines a new solver which applies solvers HOLN, ElimSolver, DerivSolver and PolynSolver sequentially (seq) in this order. This application is repeated until the goal becomes fixed point (which means the goal is solved).

```
newSolver = repeat[seq[{holn, elim, deriv, polyn}]];
```

Finally we apply the new solver to the goal

```
ApplyCollaborative[newSolver, G2];
```

Of course, newSolver returns the same solution in this case since solvers other than HOLN does not change the goal. We will see in the next section that the collaboration of solvers can solve more sophisticated problems.

7. Examples from geometry

Our final examples are taken from elementary geometry. We give these examples since many geometrical properties are stated declaratively, i.e., without resort to describing a concrete method to realize those propertied. If declarative statements are given in equations, it is easy to make them run on the computer. Those statements can be regarded programs.

Consider parallelism of lines. Let us first represent a line $a\,x + b\,y + c = 0$ by `line`[a, b, c] using constructor `line`. The following one taught in a high school is easy to see.

```
line[a1,b1,c1] ∥ line[a2,b2,c2] → True ⇐ a1 b2-a2 b1 ≈ 0
```

It says that two lines are parallel if the coefficients of the equations of each line satisfies `a1 b2 - a2 b1` ≈ 0. In our language we can omit "→ `True`", and write simply

```
line[a1,b1,c1] ∥ line[a2,b2,c2] ⇐ a1 b2 - a2 b1 ≈ 0
```

We will give the definitions of several geometric functions below. Our language requires function declarations with types. In this paper we omit the explanation, but the declaration is necessary for our examples to run.

```
Constructor[TyLine[α] = line[α, α, α]];
Constructor[TyPoint[α] = point[α, α]];
Constructor[TySegment[α] = segment[TyPoint[α], TyPoint[α]]]
```

```
Instance[{α : Reals} ⇒ TyLine[α] : Eq];
Instance[{α : Reals} ⇒ TyPoint[α] : Eq];
Instance[{α : Reals} ⇒ TySegment[α] : Eq]
```

```
DefinedSymbol[
 DoubleVerticalBar : TyLine[ℝ] → TyLine[ℝ] → 𝔹,
 UpTee : TyLine[ℝ] → TyLine[ℝ] → 𝔹,
 DownRightVector : TyPoint[ℝ] → TyLine[ℝ] → 𝔹,
 SegmentToLine : TySegment[ℝ] → TyLine[ℝ],
 PerpendicularBisector :
  TyPoint[ℝ] → TyPoint[ℝ] → TyLine[ℝ],
 MidPoint : TyPoint[ℝ] → TyPoint[ℝ] → TyPoint[ℝ],
 BrTh : TyPoint[ℝ] → TySegment[ℝ] → TyPoint[ℝ] → TyLine[ℝ]
]
```

Then we have the following function definitions.

```
R2 = FLPProgram[{
    line[a1, b1, c1] ∥ line[a2, b2, c2] ⇐ a1 b2 - a2 b1 ≈ 0,
    line[a1, b1, c1] ⊥ line[a2, b2, c2] ⇐ a1 a2 + b1 b2 ≈ 0,
    (point[x, y] ⇁ line[a, b, c]) ⇐ a x + b y + c ≈ 0,
    SegmentToLine[segment[point[x1, y1], point[x2, y2]]] →
     line[y2 - y1, x1 - x2, x2 y1 - x1 y2],
    MidPoint[point[px, py], point[qx, qy]] →
     point[(px + qx) / 2, (py + qy) / 2],
    PerpendicularBisector[P, Q] → m ⇐
     {m ⊥ SegmentToLine[segment[P, Q]], MidPoint[P, Q] ⇁ m},
    BrTh[P, seg, Q] → n ⇐ {PerpendicularBisector[P, R] ≈ n,
      Q ⇁ n, R ⇁ SegmentToLine[seg]}}
  ];
```

```
ConfigSolver[flp, {Program → R2, Calculus → "LCNCd"}];
```

The functions ⊥, ⇁ , MidPoint, SegmentToLine, PerpendicularBisector are defined as auxiliary functions to function BrTh. Similar to the parallelism of lines, the condition of the perpendicularity of two lines are asserted by :

liine[a1,b1,c1]⊥ line[a2,b2,c2] ⟸ a1 a2 + b1 b2 ≈ 0

We next represent a point whose *x*- and *y*-coordinates by `Point[x, y]`. The fact that point *P* is on line *m* is expressed by the notation $P \rightharpoonup m$.

We represent a segment whose end points are `Point[`*x1, y1*`]` and `Point[`*x2, y2*`]` by `Segment[Point[`*x1, y1*`]` , `Point[`*x2, y2*`]`]. Then the program `SegmentToLine` which transforms a segment to a line is given as above.

`PerpendicularBisector[`*P, Q*`]` returns a line that bisects and is perpendicular to `Segment[`*P, Q*`]`.

After the definition of the program, we can issue a question like:

```
G3 = exists[{l}, {line[2, 3, 5] ⊥ l}];
```

```
ApplyCollaborative[seq[{flp, elim}], {G3}]
```

Then the system returns the following answer.

```
ApplyCollaborative[seq[{flp, elim}], {G3}]

LNSolver` yields
```

$$\{\exists_{\{a2\$2268:\mathbb{R},b2\$2268:\mathbb{R}\}} \{l \to line[a2\$2268, b2\$2268, c2\$2268]\} \&\& \\ 2\,a2\$2268 + 3\,b2\$2268 == 0\}$$

```
ElimSolver` yields
```

$$\left\{\left\{l \to line\left[-\frac{3\,b2\$2268}{2}, b2\$2268, c2\$2268\right]\right\}\right\}$$

Note that `a2$2268`, `b2$2268` and `c2$2268` are internally generated variables.

With these preparations we can give a program of one of six basic Origami folds, known as Huzita's axioms [4, 5]. The axiom (O5) says that given two points *P* and *Q*, and a line *m* , we can make a fold that places *P* onto *m* and passes through *Q*.

`BrTh`[*P*, *s*, *Q*] computes a line *n* that passes through *Q*, such that the fold along the line *n* brings *P* onto *s*. `BrTh` is read as "bring *P* onto *s* along the line through *Q*".

We will try to solve the goal defined below:

```
G4 = exists[{l}, l ≈ BrTh[point[3 / 2, 1 / 2],
    segment[point[0, 0], point[2, 2]], point[1, -2]]]
```

```
ApplyCollaborative[seq[{flp, polyn}], {G4}]
```

We finally obtain the solutions of the goal.

$$\left\{\{l \to line[0, 0, 0]\}, \{l \to line[a1\$2625, 0, -a1\$2625]\}, \right. \\ \left.\left\{l \to line\left[\frac{3\,b1\$2625}{2}, b1\$2625, \frac{b1\$2625}{2}\right]\right\}\right\}$$

In this example, we have three solutions.

8. Conclusion

In this paper we have shown the following:

- Equality plays an important role in programming.
- Equality can be defined in many domains, but the equality defined over the domain terms is essential one in programming.
- Many properties defined in terms of equations naturally turn into algorithms when solvers for equations are developed.
- Solvers can be combined to make a more versatile and powerful solver. Collaboration of solvers are important for solving real-life problems.
- In summary, equations are (one of) bridges between mathematics and computer science, especially programming.

References

[1] Tetsuo Ida, Mircea Marin and Taro Suzuki, Higher-order Lazy Narrowing Calculus: a Solver for Higher-order Equations, Conference on Computer Aided Systems (EUROCAST 2001)}, Lecture Notes in Computer Science 2178, Las Palmas de Gran Canaria, Spain, pp. 478-493, 2001

[2] Tetsuo Ida, Mircea Marin and Norio Kobayashi, An Open Environment for Cooperative Scientific Problem Solving, Proceedings of the fourth International Mathematica Symposium (IMS 2001), Chiba Japan, pp. 71--78, 2001

[3] Mircea Marin, Taro Suzuki and Tetsuo Ida, Higher-Order Lazy Narrowing Calculi for Pattern Rewrite Systems, Technical Report, ISE-TR-01-180, Institute of Information Sciences and Electronics, University of Tsukuba, 2001

[4] Humiaki Huzita, Axiomatic Development of Origami Geometry, Proceedings of the First International Meeting of Origami Science and Technology, pp. 143--158, 1989

[5] Thomas Hull, Origami and Geometric Constructions, http://web.merrimack.edu/~thull/geoconst.thml, 1997