Constraint Functional Logic Programming for Origami Construction

Tetsuo Ida[1], Mircea Marin[2]*, and Hidekazu Takahashi[3]

[1] Institute of Information Sciences and Electronics
University of Tsukuba, Tsukuba, Japan
`Tetsuo.Ida@acm.org`
[2] Johann Radon Institute for Computational and Applied Mathematics
Johannes Kepler University, Linz, Austria
`mircea.marin@oeaw.ac.at`
[3] Kawase High School, Hikone, Shiga, Japan
`hidezaku@pop.biwako.ne.jp`

Abstract. We describe origami programming methodology based on constraint functional logic programming. The basic operations of origami are reduced to solving systems of equations which describe the geometric properties of paper folds. We developed two software components: one that provides primitives to construct, manipulate and visualize paper folds and the other that solves the systems of equations. Using these components, we illustrate computer-supported origami construction and show the significance of the constraint functional logic programming paradigm in the program development.

1 Introduction

Origami is a Japanese traditional art of paper folding. Recently, origami received wide range of interest among mathematicians, math educators and computer scientists, as well as origami artists, as origami proves helpful in understanding essence of fundamental geometrical problem solving. To computer scientists and mathematicians the interest lies largely in the integration of various modes of computing during the origami construction.

The purpose of this paper is twofold: to show origami construction as an application of constraint functional logic programming, and to show the effectiveness of origami construction in the math and computer science education. To show the former is the main objective, and to attain this we will use a constraint functional logic programming system called Open CFLP [1, 2] and an origami construction environment [3]. The illustrative examples will serve as the concrete examples of pedagogical effectiveness of this approach.

In our scheme, computer origamists can perform an origami construction by invoking paper folding functions with parameters that determine the kind of the paper fold. Moreover, the final result, as well as the results of the intermediary

* Supported by the Austrian Academy of Sciences.

steps, of these operations can be visualized and manipulated. The process of the computer origami construction is driven by the computation of constraint solving, where the constraints are on geometrical objects in Euclidean space. The constraints are internally represented as equations. It is, therefore, a challenging theme to study origami construction as an application of constraint functional logic programming, which is a programming paradigm of, in essence, solving equations over various domains. In particular, we will explore the capabilities of Open CFLP that can solve equations over specified domains, such as the domains of terms, of polynomials, and of reals.

In origami construction, the problem specifications are clear and explicit, as we will see later. The specifications are then reduced to those for solving equations with respect to a theory provided as a set of conditional rewrite rules. Writing a theory is the task of programmers, and computer origamists will focus on finding appropriate folds.

In the sequel we will show how the origami construction can be programmed. Section 2 introduces the basics of origami geometry and of constraint functional logic programming. We will fully develop the origami construction in constraint functional logic programming in the following sections with instructive examples. In section 6 we draw some conclusions and directions of further research.

2 Preliminaries

In this section we summarize basic notions and notations that are necessary for understanding the principles of our computer origami construction. We give the representation of point, line, plane, (line) segment and polygon in the following, assuming that those notions are understood.

We rely on the notation of *Mathematica* [12] to describe our program constructs since the systems we will be using are written in *Mathematica*. Notably, we deviate from the common notation of mathematics in that functions are represented as $f[a_1, \ldots, a_n]$ rather than $f(a_1, \ldots, a_n)$. In *Mathematica* when $f[a_1, \ldots, a_n]$ can not be reduced further, the value of $f[a_1, \ldots, a_n]$ is itself. This property is taken advantage of to define a new structure. Namely, when f is not defined to be a function, f is regarded as a constructor symbol. In this case, $f[a_1, \ldots, a_n]$ is interpreted as a structure whose name is f and which contains the values of $a_1, \ldots, a_n$.

Origami. The word *origami* is a combined word coming from *ori* (fold) and *kami* (paper). Traditionally, we use origami to mean sometimes a folding paper, act of paper folding or art of origami. As such, we also use the word origami in a flexible way depending on the context.

An origami is constructed in a stepwise manner, where in each construction a fold is made along a line. An origami at its i-th construction step is represented by a structure `Origami`$[i, \mathit{planes}, \mathit{order}]$, where *planes* is a list of plane structures (see below) and *order* is the overlay relation between the planes that constitute the origami. Origamis are indexed by asserting $\texttt{Origami}[i] = \texttt{Origami}[i, \mathit{planes}, \mathit{order}]$ for quick access[1].

Plane. A plane is represented by a structure $\texttt{Plane}[\mathit{id}, \mathit{poly}, \mathit{nbrs}, \mathit{mpoints}]$, where *id* is an integer identifying the plane, $\mathit{poly} = \texttt{Polygon}[\{P_1, \ldots, P_k\}]$ represents a polygon made of vertices $P_1, \ldots, P_k$, *nbrs* is a list of neighboring planes $\{o_1, \ldots, o_k\}$, where o_j is the *id* of the plane that is adjacent to the edge $\overline{P_j P_{j+1}}$ (we let $P_{k+1} = P_1$), and *mpoints* is a list of mark points (see below for the definition).

Polygon. Given a sequence of vertices $P_1, \ldots, P_k$, ordered counter-clockwise, a polygon is represented by a structure $\texttt{Polygon}[\{P_1, \ldots, P_k\}]$.

Point and Mark point. A point, whose *x*-, *y*- and *z*- coordinates are *u, v* and *w* respectively, is represented by a structure $\texttt{Point}[u, v, w]$. The *z*- coordinate is omitted when we are working only in 2D on the plane $z = 0$. Internally, however, all points are represented in 3D. The translation from 2D to 3D is done automatically by the system. A mark point is a structure used to refer to a designated point on planes at some steps of the origami construction. It is represented by a pair of a point name and a point. For example, $\{\text{"M"}, \texttt{Point}[1, 2]\}$. For short, we call this mark point "M".

Line and Segment. A segment $\overline{PQ}$ between points P and Q is represented by $\texttt{Segment}[P, Q]$. A line $a\ x + b\ y + c = 0$ on the plane $z = 0$ is represented by $\texttt{Line}[a, b, c]$. A line is said to extend the segment $\overline{PQ}$ if it passes through the points P and Q.

Constraint functional logic program. A constraint functional logic program consists of a goal and a theory. The latter is given by a set of conditional rewrite rules of the form $f[\mathit{args}] \rightarrow r$ or $f[\mathit{args}] \rightarrow r \Leftarrow C$ where C is a list of boolean expressions which denotes their logical conjunction. The boolean expressions are equations over some domains. Note that any boolean expression can be turned to an equation by equating the expression to boolean value True, if necessary. $f[\mathit{args}]$ and r are called the left-hand and right-hand side of the rewrite rule, respectively. C is called the conditional part of the rewrite rule. Extra variables (*i.e.*, variables without occurrences on the left-hand side of the rewrite rule) are allowed in the conditional part, where they are supposed to be existentially quantified. A rewrite rule $f[\mathit{args}] \rightarrow r \Leftarrow C$ partially defines function f. It is used to reduce an instance $f[\mathit{args}]\theta$ to $r\theta$ if $C\theta$ holds, where θ is an arbitrary substitution. A function symbol which appears at the outermost position of the left-hand side of a rewrite rule is called *defined symbol.* The other function symbols are either *constructors* or *constraint operators.*

[1] Note that indexing this way is the feature of *Mathematica.*

Equations are expressions of the form $s \approx t$ or $s \rhd t$. Their meaning is: $s \approx t$ if s and t are reduced to the value, and $s \rhd t$ if s is reduced to t.

A goal is a list of boolean expressions. Solving a goal G with respect to a theory T is a computation of the constraint functional logic program. It yields a substitution θ such that $G\theta$ holds in the theory T. Logically speaking, solving a goal is proving an existential formula $\exists x_1 ... x_m.G$, where our concern is not only about the validity of $\exists x_1 ... x_m.G$, but about the substitutions θ over x_1, ..., x_m that make $G\theta$ True.

3 Principles of Origami Construction

An origami is folded along a line called *crease*. The crease can be determined by the points it passes through or by the points (and/or lines) it brings together. The fold determined in the former is called *through* action (abbreviated to Th), and the latter *bring* action (abbreviated to Br). We can combine the two actions, and we have the following six basic fold operations formally stated as Origami Axioms [4, 5] of Huzita. It is shown that by the six basic operations commanded by Origami Axioms we can construct 2D geometrical objects that can be constructed by a ruler and a compass. Actually, origami is more powerful than the ruler and compass method since origami can construct objects that are not possible by the ruler and compass method [6]. One of them is trisecting an angle, which we will show later.

3.1 Origami axioms

The origami axioms consist of the following assertions. The action involved in each axiom is indicated by the words within the square brackets.

(O1) [Th] Given two points, we can make a fold along the crease passing through them.
(O2) [Br] Given two points, we can make a fold to bring one of the points onto the other.
(O3) [Br] Given two lines, we can make a fold to bring one of the lines onto the other.
(O4) [Th] Given a point P and a line m, we can make a fold along the crease that is perpendicular to m and passes through P.
(O5) [BrTh] Given two points P and Q and a line m, we can make a fold along the crease that passes through Q, such that the fold brings P onto m.
(O6) [BrBr] Given two points P and Q and two lines m and n, we can make a fold that brings P and Q onto m and n, respectively.

Algorithmically, these axioms purport to two operations: finding a crease(s) and folding the origami along the crease. Let us first focus on the issue of finding the crease and examine the implementation of these axioms.

Axiom (O1) is specified by a rewrite rule expressing an elementary formula of high-school geometry.

$$\texttt{Thru}[\texttt{Point}[\texttt{a1},\texttt{b1}],\texttt{Point}[\texttt{a2},\texttt{b2}]] \rightarrow \texttt{Line}[\texttt{b2}-\texttt{b1},\texttt{a1}-\texttt{a2},\texttt{a2 b1}-\texttt{a1 b2}]$$

Axiom (O2) is to find the perpendicular bisector of the segment connecting the two points. Axiom (O3) is to find the bisector of the angle formed by the two lines. Axioms (O3)∼(O6) can also be concisely specified in functional logic programs, and will be the subject of fuller discussion in section 4, after we see the origami construction of trisecting an angle.

3.2 Trisecting an angle by origami

We explain the principles of origami construction by the example of trisecting a given angle by origami. This example makes non-trivial use of Axiom (O6) and shows the power of origami. The method is due to H. Abe as described in [7, 8].

First, we define a square origami paper, whose corners are designated by the mark points "A","B","C" and "D". The size may be arbitrary, but for our presentation, let us fix it to 4×4. The function call

```
DefOrigami[4, 4, MarkPoints → {"A", "B", "C", "D"},
  Context → "TrisectAngle`"];
```

creates the desired origami. This call generates, under the context of TrisectAngle, the variables A, B, C and D that hold the coordinates of the mark points "A", "B", "C" and "D", respectively. By this support, we can safely use wording *point* A instead of mark point "A".

Generally, the operations are performed by function calls of the form $Op[args, opt_1 \rightarrow val_1, \ldots, opt_n \rightarrow val_n]$, where *args* is a sequence of parameters, opt_1, …, opt_n are keywords for optional parameters, and val_1, …, val_n are the corresponding values of these options.

We then introduce an arbitrary point, say Point[3, 4], marked by "E" by the following function call:

```
PutPoint[{"E", Point[3, 4]}];
```

Our problem is to trisect the angle ∠EAB.

We want to let F, G, H and I be the midpoints of points A and D, points B and C, points A and F, and points B and G, respectively. We mark those points by "F","G","H" and "I", respectively, and put them on the current origami.

```
PutPoint[{{"F", Midpoint[A, D]}, {"G", Midpoint[B, C]},
    {"H", Midpoint[A, F]}, {"I", Midpoint[B, G]}}]
```

Note that we can obtain the midpoints by applying Axiom (O2). We skip this step, as it is straightforward.

At this point, our origami can be displayed by calling ShowOrigami[. . .]: (right one, left one immediately after marking E)

```
ShowOrigami[More → Graphics3D[{Hue[0], GraphicsLine[E, A],
    GraphicsLine[H, I]}]];
```

The keyword `More` instructs the system to display more graphical objects using the built-in *Mathematica* functions. In this case, lines in different hues are displayed.

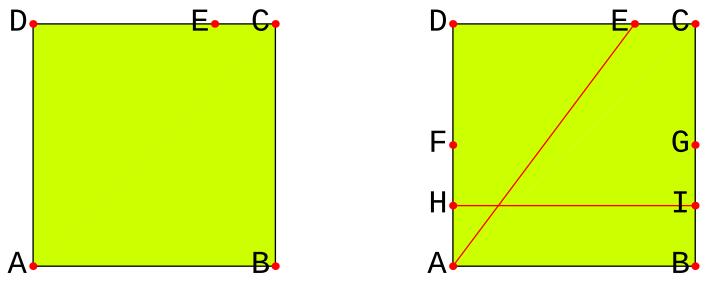

Fig. 1. Origami displays.

We then make a fold along the line ensured by Axiom (O6), such that point `F` is brought onto line $\overline{\mathtt{AE}}$ and point `A` is brought onto line $\overline{\mathtt{HI}}$. The line is computed by `BrBr[F, Segment[A, E], A, Segment[H, I]]`, where `BrBr` is the implementation of Axiom (O6). The interface to the computer origamist is `FoldBrBr[...]` which does folding after finding creases by `BrBr[...]`.

```
FoldBrBr[1, F, Segment[A, E], A, Segment[H, I]]
```

`Which line(1, 2, 3)?`

$$\begin{aligned}&\{\mathtt{Line}\left[\tfrac{3}{4}+\tfrac{5}{2}\ \cos\left[\tfrac{1}{3}\ \arctan\left[\tfrac{4}{3}\right]\right],\right.\\&\quad\left.1,-\tfrac{75}{32}-\tfrac{15}{8}\ \cos\left[\tfrac{1}{3}\ \arctan\left[\tfrac{4}{3}\right]\right]-\tfrac{25}{16}\ \cos\left[\tfrac{2}{3}\ \arctan\left[\tfrac{4}{3}\right]\right]\right],\\&\ \mathtt{Line}\left[\tfrac{3}{4}-\tfrac{5}{4}\ \cos\left[\tfrac{1}{3}\ \arctan\left[\tfrac{4}{3}\right]\right]-\tfrac{5}{4}\ \sqrt{3}\ \sin\left[\tfrac{1}{3}\ \arctan\left[\tfrac{4}{3}\right]\right],\right.\\&\quad 1,-\tfrac{75}{32}+\tfrac{15}{16}\ \cos\left[\tfrac{1}{3}\ \arctan\left[\tfrac{4}{3}\right]\right]+\\&\quad\quad\tfrac{25}{32}\ \cos\left[\tfrac{2}{3}\ \arctan\left[\tfrac{4}{3}\right]\right]+\tfrac{15}{16}\ \sqrt{3}\ \sin\left[\tfrac{1}{3}\ \arctan\left[\tfrac{4}{3}\right]\right]-\\&\quad\quad\left.\tfrac{25}{32}\ \sqrt{3}\ \sin\left[\tfrac{2}{3}\ \arctan\left[\tfrac{4}{3}\right]\right]\right],\\&\ \mathtt{Line}\left[\tfrac{3}{4}-\tfrac{5}{4}\ \cos\left[\tfrac{1}{3}\ \arctan\left[\tfrac{4}{3}\right]\right]+\tfrac{5}{4}\ \sqrt{3}\ \sin\left[\tfrac{1}{3}\ \arctan\left[\tfrac{4}{3}\right]\right],\right.\\&\quad\ 1,-\tfrac{75}{32}+\tfrac{15}{16}\ \cos\left[\tfrac{1}{3}\ \arctan\left[\tfrac{4}{3}\right]\right]+\\&\quad\quad\tfrac{25}{32}\ \cos\left[\tfrac{2}{3}\ \arctan\left[\tfrac{4}{3}\right]\right]-\tfrac{15}{16}\ \sqrt{3}\ \sin\left[\tfrac{1}{3}\ \arctan\left[\tfrac{4}{3}\right]\right]+\\&\quad\quad\left.\tfrac{25}{32}\ \sqrt{3}\ \sin\left[\tfrac{2}{3}\ \arctan\left[\tfrac{4}{3}\right]\right]\right]\}\end{aligned}$$

In this case there are three lines that make this fold possible. The system responds by asking us along which line we want to make a fold. Let us answer to this question by giving 1 (the 6th argument of `FoldBrBr`) to the system, i.e. choosing the first line.

```
FoldBrBr[1, F, Segment[A, E], A, Segment[H, I], 1,  MarkCrease → True];
```

Although `FoldBrBr` is called twice with the same 5 arguments, the computation is not entirely duplicated in our system since the result of the most expensive sub-computation is memoized.

Points `J` and `K` are the end points of the crease generated by the above call since option `MarkCrease` → `True` is specified. Let `A1`, `H1` and `F1` be the points `A`, `H` and `F` after the fold. Then we revert to the origami at step 1.

```
pts = PutPoint[{{"A1", A}, {"H1", H}, {"F1", F}}];
ToStep[1, pts];
```

We now see that lines $\overline{\mathtt{AH1}}$ and $\overline{\mathtt{AA1}}$ trisect ∠`EAB` in Fig. 2. The origami to the left is the one after the evaluation of `FoldBrBr[...]` above, and the origami to the right is displayed by the following call:

```
ShowOrigami[More →
    Graphics3D[GraphicsLine[J, K],
        Hue[0.2], GraphicsLine[A, F1], GraphicsLine[A, H1],
          GraphicsLine[A, A1]},
        Hue[0], GraphicsLine[A, E], GraphicsLine[H, I]}}]];
```

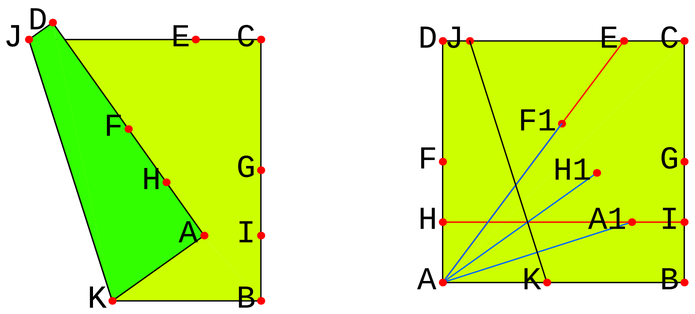

Fig. 2. Fold that trisects $\angle EAB$.

We can prove that $\overline{\mathtt{AH1}}$ and $\overline{\mathtt{AA1}}$ trisect ∠`EAB` both symbolically and geometrically. A geometrical proof is more elegant and a good exercise for high school students. The proof is found in [7]. A proof based on symbolic computation is rather laborious. An easy way would be to use the trigonometric identity:

$$\tan[3\ \theta] = \frac{3\tan[\theta] - \ \tan[\theta]^3}{1 - 3\ \tan[\theta]^2}.$$

The coordinates of the points `A1`, `F1` and `H1` are obtained simply by evaluating the variables `A1`, `F1` and `H1`. The rest of the work can be done symbolically. This leads to an interesting (albeit well-known) observation: `BrBr` amounts to solving a cubic equation. Indeed `BrTh` is related to solving quadratic equations, and `BrBr` to solving cubic equations. We will come to this point in section 5.

4 Definitions of Basic Operations in Constraint Functional Logic Programming

In this section we define the theories that are at work behind the scene of the origami construction described in section 3. We explain them in the language of Open CFLP. We begin by giving type declarations using a type discipline similar to that of Haskell, but in a *Mathematica* syntax. We can declare our own type constructors (`TLine` for lines, `TPoint` for points, etc.) by `Instance`[. . .] calls, and the associated data constructors (`Line` for `TLine` type, `Point` for `TPoint` type, etc.) by `DataConstructors`[. . .] calls. `Eq` is the class of types with equality, $\mathbb{R}$ is the built-in type of real numbers, and $\mathbb{B}$ is the built-in type boolean.

```
Instance[TLine : Eq]; Instance[TPoint : Eq];
Instance[TSegment : Eq]; Instance[TPolyPair : Eq];
Instance[TSPPair : Eq];

DataConstructors[{
    Line : ℝ ⟶ ℝ ⟶ ℝ ⟶ TLine,
    Point : ℝ ⟶ ℝ ⟶ TPoint,
    SPPair : TLine ⟶ TPoint ⟶ TSPPair,
    PolyPair : list[TPoint] ⟶ list[TPoint] ⟶ TPolyPair,
    Segment : TPoint ⟶ TPoint ⟶ TSegment}];
```

Similarly, defined (function) symbols are declared by `DefinedSymbols`[. . .] calls. We first declare the defined symbols with types as follows.

```
DefinedSymbols[{
    DoubleVerticalBar : TLine ⟶ TLine ⟶ 𝔹,
    UpTee : TLine ⟶ TLine ⟶ 𝔹,
    DownRightVector : TPoint ⟶ TLine ⟶ 𝔹,
    SegmentToLine : TSegment ⟶ TLine,
    Midpoint : TPoint ⟶ TPoint ⟶ TPoint,
    PerBisector : TPoint ⟶ TPoint ⟶ TLine,
    SymmetricPoint : TPoint ⟶ TLine ⟶ TPoint,
    BrTh : TPoint ⟶ TSegment ⟶ TPoint ⟶ TSPPair,
    BrBr : TPoint ⟶ TSegment ⟶ TPoint ⟶ TSegment ⟶ TLine}];
```

Then we give theories using conditional rewrite rules associated with the defined symbols. The program `Prog` shown in Fig. 3 declares the basic functions for origami construction.

```
Prog = 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[x1, y1], Point[x2, y2]] →
          Point[(x1 + x2)/2, (y1 + y2)/2],
        PerBisector[Point[a1, b1], Point[a2, b2]] →
          Line[2(a2 − a1), 2(b2 − b1), a1^2 − a2^2 + b1^2 − b2^2],
        SymmetricPoint[Point[x, y], Line[a, b, c]] →
          Point[((b^2 − a^2)x − 2 a b y − 2 a c)/(a^2 + b^2),
              ((a^2 − b^2)y − 2 a b x − 2 b c)/(a^2 + b^2)],
        BrTh[P, seg, Q] → SPPair[n, R] ⇐
          {PerBisector[P, R] ▷ n, Q ⇁ n, R ⇁ SegmentToLine[seg]},
        BrBr[P, s1, Q, s2] → m ⇐
          {m ≈ Line[a, 1, c], P2 ≈ SymmetricPoint[P, m],
            Q2 ≈ SymmetricPoint[Q, m], SegmentToLine[s1] ▷ n1,
            P2 ⇁ n1, SegmentToLine[s2] ▷ n2, Q2 ⇁ n2},
        BrBr[P, s1, Q, s2] → m ⇐
          {m ≈ Line[1, 0, c], P2 ≈ SymmetricPoint[P, m],
            Q2 ≈ SymmetricPoint[Q, m], SegmentToLine[s1] ▷ n1,
            P2 ⇁ n1, SegmentToLine[s2] ▷ n2, Q2 ⇁ n2}}];
```

Fig. 3. Basic origami theory in Open CFLP.

`DoubleVerticalBar` (its infix operator name is ∥) and `UpTee` ($\perp$) test whether the given lines are parallel or perpendicular respectively. Evaluation of $P \rightharpoondown m$ yields True iff point P is on the line m. `Midpoint`[P, Q] yields the midpoint of points P and Q, and `SegmentToLine`[*seg*] transforms segment *seg* to the line that extends the segment. Function `Thru`, the implementation of Axiom (O1) is omitted here since in practice `SegmentToLine` can do the same job. `PerBisector`(perpendicular bisector) and `SymmetricPoint` deserve more explanation in the context of constraint functional logic programming. We initially had a definition of `PerBisector` below.

```
PerBisector[P, Q] →
  ToLine[x, y] ⇐ {Segment[P, Q] ⊥ Segment[S, R],
    R ≈ Point[x, y], S ≈ Midpoint[P, Q]}
```

This means that there exists a point $R(x, y)$ such that $\overline{PQ} \perp \overline{SR}$, where S is the midpoint of $\overline{PQ}$. This is the declarative meaning of a perpendicular bisector. A slight complication occurs in our program that a set of points has to be converted to a line by `ToLine`[x, y].

It is possible, for instance, to solve the equation PerBisector[Point[1, 0], Point[0, 1]] $\triangleright x$. The solution is $x \rightarrow$ Line[−1, 1, 0], and can be solved easily by our system. However, even for symbolic values a1, b1, a2 and b2, the constraint PerBisector[Point[a1, b1], Point[a2, b2]] $\triangleright x$ is solvable. It will return the solution $x \rightarrow$ Line$\left[2(\mathtt{a2}-\mathtt{a1}), 2(\mathtt{b2}-\mathtt{b1}), \mathtt{a1}^2-\mathtt{a2}^2+\mathtt{b1}^2-\mathtt{b2}^2\right]$. Therefore, we refine the original rewrite rule by the above result. This gives the following rewrite rule:

$$\begin{aligned}&\mathtt{PerBisector}[\mathtt{Point}[\mathtt{a1},\mathtt{b1}],\mathtt{Point}[\mathtt{a2},\mathtt{b2}]] \rightarrow\\ &\quad \mathtt{Line}[2(\mathtt{a2}-\mathtt{a1}), 2(\mathtt{b2}-\mathtt{b1}), \mathtt{a1}^2-\mathtt{a2}^2+\mathtt{b1}^2-\mathtt{b2}^2]\end{aligned}$$

Likewise we can define Bisector, the partial implementation of Axiom (O3), starting with the definition, using the auxiliary function Distance[P, m], which computes the distance between point P and line m.

$$\begin{aligned}&\mathtt{Bisector}[m,n] \rightarrow \mathtt{ToLine}[x,y] \Leftarrow\\ &\quad \{\mathtt{Distance}[P,m] \approx \mathtt{Distance}[P,n], P \approx \mathtt{Point}[x,y]\}\end{aligned}$$

As in PerBisector, Bisector[Line[a1, b1, c1], Line[a2, b2, c2]] $\triangleright x$ is solvable symbolically. Since the solution is complicated, we did not include the definition of Bisector in Fig. 3. Our system, however, runs the more efficient code for given parameter Line[a1, b1, c1] and Line[a2, b2, c2] as shown below. The elimination of the duplicated common subexpressions by local variables (using With construct of *Mathematica*) is done manually.

```
With[{u = √(a1²+b1²)/√(a2²+b2²)},
  With[{v1 = a1 + u a2, w1 = b1 + u b2, v2 = a1 − u a2,
      w2 = b1 − u b2},
    If[(v1 == 0 ∧ w1 == 0),
      If[(v2 == 0 ∧ w2 == 0), {}, {Line[v2, w2, c1 − u c2]}],
      If[(v2 == 0 ∧ w2 == 0), {Line[v1, w1, c1 + u c2]},
        {Line[v1, w1, c1 + u c2], Line[v2, w2, c1 − u c2]}]]]]
```

The definition of SymmetricPoint is similarly derived.

The application of the above refined rewrite rules involves no solving of equations, and hence it is more efficient. This kind of program manipulation without resort to other frame of reasoning about the program is possible in our systems.

We can further make the following more general remark on constraint (functional logic) programming. Constraint solving is in many cases slow; slower than other possible methods that do not involve solving, due to the very nature of problem solving. Namely, constraints are statements closer to problem specification rather than those for execution. However, as our examples show, when constraint solvers deliver solutions in the form of executable symbolic programs, we can partially overcome the problem of inefficiency. This is an instance of partial evaluation in the context of constraint programming.

`BrTh` and `BrBr` are the implementations of Axioms (O5) and (O6), respectively. `BrTh`[*P, seg, Q*] returns a pair `SPPair`[*n,R*] consisting of the crease n passing through point Q and point R, where R is the point on the line extending segment *seg*, to which point P is brought by the fold along n. `BrBr`[*P, s1, Q, s2*] returns the crease m such that the fold along m brings P onto the line extending segment *s1*, and Q onto the line extending *s2*. It is very easy to read off this declarative meaning from the rewrite rules.

5 Examples

In this section we show two examples of origami constructions: one for illustrating mathematical origami construction and the other for illustrating art origami construction. The first one is an origami construction for solving a quadratic equation. We solve this to show the functionality of Open CFLP.

Problem 1. Solve the quadratic equation $x^2 + p\ x + q = 0$ by origami.

The following method is due to [7]. We consider a sheet of paper whose corners are

$$\mathtt{O} = \mathtt{Point}[0,0];\mathtt{P} = \mathtt{Point}[-\mathtt{p},0];\mathtt{X} = \mathtt{Point}[-\mathtt{p},\mathtt{q}];\mathtt{Q} = \mathtt{Point}[0,\mathtt{q}];$$

To make our description concrete, we assume $p = 4$ and $q = 3$. We consider the points $\mathtt{U} = \mathtt{Point}[0,1], \mathtt{M} = \mathtt{Midpoint}[\mathtt{X},\mathtt{U}]$ and the segment `seg` determined by `P` and `O`

$$\mathtt{p} = 4; \mathtt{q} = 3; \mathtt{U} = \mathtt{Point}[0,1]; \mathtt{M} = \mathtt{Midpoint}[\mathtt{X},\mathtt{U}];$$
$$\mathtt{seg} = \mathtt{Segment}[\mathtt{P},\mathtt{O}];$$

It can be shown that the solution X of the equation `SPPair`[n, `Point`[X, 0]] $\approx$ `BrTh`[`U`, `seg`, `M`] in variables n and X is a solution of $x^2 + p\ x + q = 0$. With Open CFLP, we pose the problem of finding X as follows:

$$\mathtt{Prob} = \mathtt{exists}[\{\mathtt{X}\}, \mathtt{SPPair}[\mathtt{n}, \mathtt{Point}[\mathtt{X},0]] \approx \mathtt{BrTh}[\mathtt{U},\mathtt{seg},\mathtt{M}]];$$

Finally, we inform the system of what solvers to employ and how to combine their solving capabilities. For solving `Prob` we need only 2 solvers: a solver which can process user defined functions, and a solver which can solve polynomial equations over $\mathbb{R}$. Open CFLP provides default implementations for both solvers: LNSolver (Lazy Narrowing Solver) for equational reasoning with user defined functions, and PolynSolver for systems of polynomial equations.

```
polyn = MkLocalSolver["PolynSolver`"];
flp = MkLocalSolver["LNSolver`"];
```

Some solvers must be configured in order to work properly. To solve the goal `Prob`, we configure `flp` so that it can reason with the theory `Prog` and employ the calculus LCNCd (deterministic lazy conditional narrowing calculus). We first apply `flp` to the goal `Prob`. The solver `flp` reduces `Prob` to an equivalent problem without user defined function symbols, and then the solver `polyn` computes the bindings for n and X by solving the intermediate problem. The sequential application of these solvers is specified by the combinator `seq`.

```
ConfigSolver[flp, {Program → Prog, Calculus → "LCNCd"}];
ApplyCollaborative[seq[{flp, polyn}], {Prob}]
```

yields the following solution.

$$\{\{\mathtt{X} \to -3\}, \{\mathtt{X} \to -1\}\}$$

Figure 4 shows the origami folds that give the solutions: the value of the x-coordinate of point `U` is the solution.

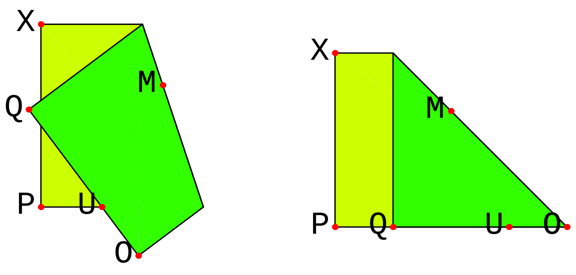

Fig. 4. Origami folds giving solutions of a quadratic equation.

Problem 2. Construct an origami of a crane.

This example shows another feature of our origami environment. Here we are more interested in producing an art piece, and the system can perform approximate numeric computation for the sake of efficiency. A computer origamist can change the mode of computation by setting the global variable `$exact` = False.

We developed the following folding algorithm for manipulating origamis. Recall that one-step fold of an origami consists of finding a crease and making a fold along the crease. The fold obeys the following algorithm.

Given `Origami`$[i]$ = `Origami`$[i, \mathit{planes}, \mathit{order}]$, and a plane plane = `Plane`$[\mathit{id}, \mathit{poly}, \mathit{neighbors}, \mathit{mpoints}]$ in *planes*, the algorithm performs the following:

1. Divide the polygon *poly* into two polygons *poly1* and *poly2,* where *poly1* is to be rotated by the fold and *poly2* is not rotated. We take *poly1* = *empty* and *poly2* = *poly* if *poly* is not divided.
2. Collect all the planes that are to be rotated together with *poly1* using the neighborhood relation *neighbors*. Let *moved* be the list of collected planes.
3. Rotate all the planes in *moved* and mark points in *mpoints* if they are on the moved planes. Let *newplanes* be the list of all the planes after the rotation.
4. Compute the overlay relation *neworder* of all the planes in *newplanes*.
5. Set `Origami`$[i + 1]$ = `Origami`$[i + 1, \mathit{newplanes}, \mathit{neworder}]$.

When we deal with 2D origamis that are performed by Origami Axioms, it suffices to consider two modes of folds, i.e. mountain fold and valley fold, which rotate the planes by angle π and $-\pi$, respectively. However, for art origami we need more sophisticated folds. We furthermore implemented inside reverse fold (which is called *naka wari ori* in Japanese, meaning "fold by splitting in the middle") and outside reverse fold (*kabuse ori* in Japanese, meaning overlay fold). The inside reverse fold is used in constructing an origami of a crane. The following sequence of calls of the fold functions will construct the crane as shown in Fig. 6.

$\texttt{SetView}[\text{"3D"}];\texttt{DefOrigami}[4,4];$
$\texttt{ValleyFold}[\texttt{DefCrease}[\{4,4\},\{0,0\}],\pi];$
$\texttt{InsideReverseFold}[\{2,3\},\texttt{DefCrease}[\{4,2\},\{2,2\}]];$
$\texttt{InsideReverseFold}[\{4,6\},\texttt{DefCrease}[\{2,2\},\{2,0\}]];$
$\texttt{InsideReverseFold}\left[\{13,12\},\texttt{DefCrease}\left[\left\{2,2\sqrt{2}-2\right\},\{4,0\}\right]\right];$
$\texttt{InsideReverseFold}\left[\{24,7\},\texttt{DefCrease}\left[\{4,0\},\left\{6-2\sqrt{2},2\right\}\right]\right];$
$\texttt{InsideReverseFold}\left[\{8,9\},\texttt{DefCrease}\left[\left\{2,2\sqrt{2}-2\right\},\{4,0\}\right]\right];$
$\texttt{InsideReverseFold}\left[\{5,16\},\texttt{DefCrease}\left[\{4,0\},\left\{6-2\sqrt{2},2\right\}\right]\right];$
$\texttt{ValleyFold}\left[\texttt{DefCrease}\left[\left\{2,2\sqrt{2}-2\right\},\left\{6-2\sqrt{2},2\right\}\right],\pi\right];$
$\texttt{ValleyFold}\left[\texttt{DefCrease}\left[\left\{2,2\sqrt{2}-2\right\},\left\{6-2\sqrt{2},2\right\}\right],-\pi\right];$
$\texttt{ValleyFold}\left[\texttt{DefCrease}\left[\left\{2,2-2\sqrt{2}+2\sqrt{4-2\sqrt{2}}\right\},\{4,0\}\right],\pi\right];$
$\texttt{ValleyFold}\left[\texttt{DefCrease}\left[\{4,0\},\left\{2+2\sqrt{2}-2\sqrt{4-2\sqrt{2}},2\right\}\right],\pi\right];$
$\texttt{ValleyFold}\left[\texttt{DefCrease}\left[\left\{2,2-2\sqrt{2}+2\sqrt{4-2\sqrt{2}}\right\},\{4,0\}\right],-\pi\right];$
$\texttt{ValleyFold}\left[\texttt{DefCrease}\left[\{4,0\},\left\{2+2\sqrt{2}-2\sqrt{4-2\sqrt{2}},2\right\}\right],-\pi\right];$
$\texttt{InsideReverseFold}\left[\{20,28\},\right.$
$\quad\left.\texttt{DefCrease}\left[\left\{4-\sqrt{2},\sqrt{2}\right\},\left\{2\sqrt{2}+2\sqrt{10-7\sqrt{2}},2\right\}\right]\right];$
$\texttt{InsideReverseFold}\left[\{36,52\},\right.$
$\left.\texttt{DefCrease}\left[\left\{2,4-2\sqrt{2}-2\sqrt{10-7\sqrt{2}}\right\},\left\{4-\sqrt{2},\sqrt{2}\right\}\right]\right];$
$\texttt{InsideReverseFold}\left[\{41,57\},\texttt{DefCrease}\left[\{3,4\},\left\{2,2\sqrt{2}\right\}\right]\right];$
$\texttt{ValleyFold}\left[\texttt{DefCrease}\left[\left\{2+2\sqrt{2}-2\sqrt{4-2\sqrt{2}},2\right\},\right.\right.$
$\qquad\left.\left.\left\{2,2-2\sqrt{2}+2\sqrt{4-2\sqrt{2}}\right\}\right],\pi/2\right];$
$\texttt{ValleyFold}\left[C\left[\left\{2+2\sqrt{2}-2\sqrt{4-2\sqrt{2}},2\right\},\left\{2,2-2\sqrt{2}+2\sqrt{4-2\sqrt{2}}\right\}\right],\right.$

Fig. 5. Program for constructing Origami crane.

Above, `DefCrease`[{*x1, y1*}, {*x2, y2*}] defines the crease from `Point`[*x1, y1*] to `Point`[*x2, y2*]. The first parameter of `InsideReverseFold`[...] is the pair of plane id's, which is determined interactively.

Finally, we view the crane in a different angle by evaluating the following:

```
ShowOrigami[ShowId → False, ViewPoint → {−0.723, 1.114, −3.112}];
```

The image produced by this call is shown in Fig. 7.

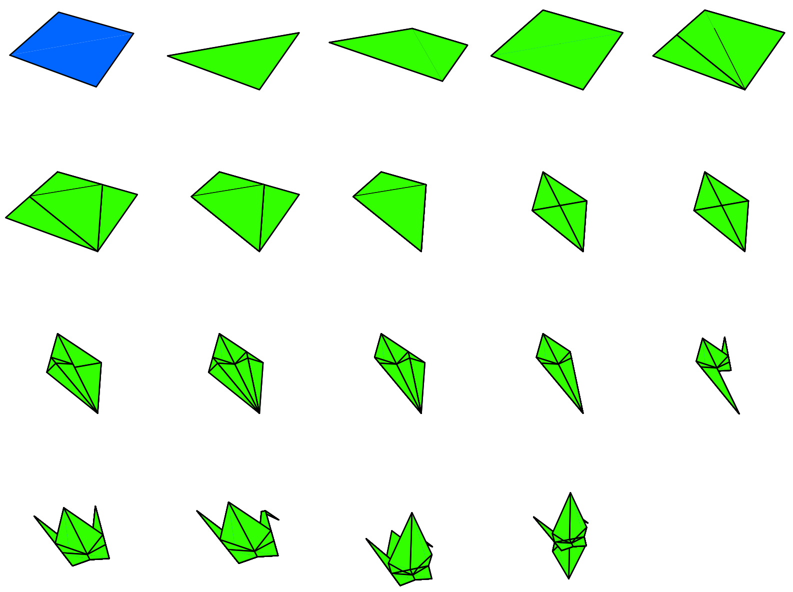

Fig. 6. Stepwise construction of Origami crane.

6 Conclusion

We have illustrated computer-supported origami construction and have shown the significant role played by the constraint functional logic programming paradigm in program development. Our examples have shown that origami construction has an aspect of geometrical problem solving, which requires constraint solving on geometrical objects. Origami bridges geometrical construction and numeric and symbolic computations. Implementing origami construction has revealed many interesting research themes: integration of constraint, functional and logic programming paradigms and combination of modes of computation such as symbolic, numeric and graphics computation. The implementation of an origami environment based on our previous work on Open CFLP and the experiences with the symbolic algebra system *Mathematica* brought insights to new programming paradigms and pedagogical implications to mathematics and computer science.

Concerning the related works, we draw ideas from the research community of constraint functional logic programming and mathematical and art origami communities. As far as we know the application of constraint functional logic programming to origami is new, although some researchers recognize the im-

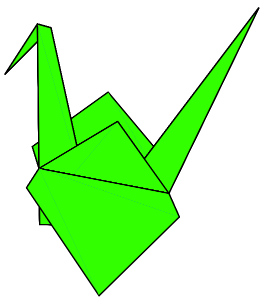

Fig. 7. Origami crane.

portance of integrating constraint solvers in the functional logic programming systems such as Curry [10] and Toy [11].

The research on origami has by far longer history and has enjoyed wide range of supports and enthusiasm from artists, educators, mathematicians and computer scientists. We can find many interesting web sites devoted to origami. Nevertheless, our computer origami as described in this paper is original in that we can successfully integrate geometrical constraint solving, symbolic and numeric computing and graphics processing in computerizing origami construction. We are aware of research on origami simulations, for example [9], which also has references to some earlier works. Their work focusses on simulating the operation of paper folding on the computer screen, not on the computational aspect of origami. Currently, we are working on the full integration of Open CFLP with an origami simulation engine and a web page interface to support the activities of computer origamists. By providing the internet origami construction service, we hope to open a new research area of computer origami.

References

[1] N. Kobayashi, M. Marin, and T. Ida. Collaborative Constraint Functional Logic Programming System in an Open Environment. *IEICE Transactions on Information and Systems*, E86-D(1), pp. 63–70, January 2003.

[2] M. Marin., T. Ida, and W. Schreiner. CFLP: A *Mathematica* Implementation of a Distributed Constraint Solving System. In *The Mathematica Journal,* 8(2), pp. 287–300, 2001.

[3] H. Takahashi and T. Ida. Origami Programming Environment. In *Challenging the Boundaries of Symbolic Computation, Proceedings of 5th International Mathematica Symposium (IMS'2003)*, P. Mitic, P. Ramsden, and J. Carne, editors, Imperial College Press. pp. 413 – 420, 2003.

[4] H. Huzita. Axiomatic Development of Origami Geometry. In *Proceedings of the First International Meeting of Origami Science and Technology*, pp. 143–158, 1989.

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

[6] Tzer-lin Chen. Proof of the impossibility of trisecting an angle with Euclidean tools, *Math. Mag. 39,* pp. 239-241, 1966.

[7] R. Geretschläger. *Geometric Constructions in Origami* (in Japanese, translation by H. Fukagawa), Morikita Publishing Co., 2002.

[8] K. Fushimi. *Science of Origami,* a supplement to Saiensu, p.8, Oct. 1980.

[9] S. Miyazaki, T. Yasuda, S. Yokoi and J. Toriwaki. An Origami Playing Simulator in the Virtual Space, The Journal of Visualization and Computer Animation, Vol.7, No. 1, pp.25-42, 1996.

[10] M. Hanus (eds.) Curry: A Truly Integrated Functional Logic Language, `http://www.informatik.uni-kiel.de/~curry`, 2002.

[11] J. C. Gonzales-Moreno, T. Hortala-Gonzalez, F.J. Lopez-Fraguas, and M. Rodriguez-Artalejo, An Approach to Declarative Programming Based on a Rewrite Logic, Journal of Logic Programming, Vol. 40, No. 1, pp. 47 –87, 1999.

[12] S. Wolfram, The Mathematica Book, 3rd edition, Wolfram Media and Cambridge University Press, 1996.