Embedded Languages for Origami-Based Geometry

Gaetano Caruana and Gordon J. Pace

Department of Computer Science, University of Malta

Abstract. Embedded languages have been used to support compositional descriptions for various domains. In this paper, we look at the domain of paper folding, or Origami-based geometry, in which sequences of paper folding are used to describe points and lines on the plane. Based on seven basic origami axioms, we design and develop an embedded domain specific language for the descriptions of such constructions in Haskell. We argue that the embedded language approach, that is composing a model using the basic constructors in the domain specific language, gives a compositional and concise way to describe Origami models. We look into analysis, manipulation and generation of origami models using this approach, including textual explanations of models, analysis of models to discover inherent preconditions (or constraints) in a description and basic animation of the folding of a model. Finally, we look into the tagging of blocks within a construction, enabling different evaluations at various levels of abstraction according to the user's knowledge of Origami.

1 Introduction

Language design has been an important area of research in computer science right since its inception. It is generally accepted that for designing a specialised, domain-specific language can aid and simplify the design and specification of algorithms in that domain. Various such domain specific languages have been developed for various different areas. Rather than design a full domain specific language (DSL) from scratch, with operators to features generally found in most languages to support features such as iteration and algorithm reuse, an increasingly used approach is to embed the basic domain specific language inside another programming language. The language in which the DSL is embedded, called the host language, acts like a container for the embedded language, and is used to build functions that describe, analyse and manipulate programs written in the embedded language. This approach guarantees that the features of the host language are automatically inherited by the embedded language [Hud96].

By inheriting features from the host language, the domain-specific embedded languages designers are relieved from having to reinvent the wheel to support commonly found features in general-purpose programming languages. Furthermore, building a new embedded language does not require the implementation of new development tools by sharing the host language's compilers and interpreters,

providing only interpretation mechanisms for the domain-specific features, making the building of a DSEL much simpler than designing and supporting a DSL from scratch. Apart from making use of existing infrastructure of the host language, one can combine different DSELs embedded in the same host language, thus enabling easy combination of languages, making them more extendable than DSLs [Hud96,LM99].

In this paper, we look at design of an domain-specific embedded language for Origami (paper-folding) based geometrical constructions. Ever since Euclid gave his axioms of planar geometry, expressing how one can construct and reason about points, lines and circles on the plane, different techniques have been explored into the expressiveness and relation between different 'tools' which limit what constructions can be derived from other ones. It is well known, for instance, that trisecting an angle using straight-edge and collapsing-compass is, in general, impossible. Various alternative tools have been explored, one of which is based on Origami, the Japanese art of paper folding. Using this geometrical tools, one only construct new points and lines by folding the plane (and unfolding it back) using already known points and lines for reference. We embed a small domain-specific language to express such constructions in Haskell [Jon03]. Based on these descriptions, we build a library of functions to manipulate, analyse and visualise such constructions. Since large models can become rather repetitive to explain to a user, and one should be able to simply state the name of commonly found sequences of constructions to advanced users, we explore different techniques to tag blocks in constructions to enable us to give more compact descriptions.

2 Related Work

Various embedded domain-specific languages have been developed in the literature for domains as diverse as financial contracts [Jon01], hardware-description languages [CSS01,DLC99] and stage-lighting [Spe01]. The common trait in these applications is that objects in these domains can be complex and difficult to handle as a single entity, but can be expressed into the composition of smaller objects. Such decomposition, especially if regular, enables simple descriptions of complex compound objects. A library of basic objects, and combinators for their composition thus make up the domain-specific language. These descriptions can then be manipulated and evaluated using different techniques without having to change their description. For example, in the case of financial contracts, one can analyse the expected value of a contract, or simulate it with projected future data on which it may depend (such as exchange rates), or even produce a natural-language rendition of the contract. In the case of the embedded hardware-description language Lava [CSS01] one can not only simulate circuits, but also produce VHDL descriptions and verify properties of the circuits by using external model checking tools.

Although purely-functional languages have been shown to be very appropriate vehicles for embedding languages, one recurring problem in such descriptions has always been that of identifying blocks induced by the description visible at

the host language level, but which are lost at the domain specific level. An area in which this restriction has been recurringly encountered is that of embedded hardware description languages. Although various regular, yet complex circuits can be described using a recursive algorithm which induces the gate connections [CSS01,DLC99], one loses the 'virtual' blocks one automatically goes through upon every function call recursive or otherwise. Some different techniques have appeared recently [ACS05,Pac07] looking at component and block oriented view of circuits. In Wired [ACS05], circuits are seen as relational blocks which can be composed together using a set of combinators, allowing various non-functional features of such circuits to be analysed. In [Pac07], another component-based approach is taken, moving away from the purely functional descriptions in Lava and combinator-based approach in Wired, looking more at the connection descriptions with explicit wire parameters. Our approach to describing blocks in our language differs from these, by providing an explicit tagging construct which can be used to tag the boundaries of a sub-construction. Although the use of explicit tagging can be tedious and prone to error, we only require tagging in few places for our domain. Furthermore, our descriptions still look functional, and less component-oriented than the other approaches we have mentioned, more in keeping with the host language.

3 Origami

The ancient Japanese art of paper folding, Origami, may seem deceptively simple and straightforward. Origami models can be constructed after a few days of practice, but advanced techniques and models require years of experience and knowledge to execute well, and the design of new models requires a high degree of creativity. Origami is based on the repeated folding of a sheet of paper, to form models, usually resembling real-life objects.

Although variants of Origami allowing the use multiple sheets of paper (see, for example, [Mit77]), or glueing or even the cutting of the sheet of paper exist, the traditional approach is to use one sheet of paper and allow only creases with respect to other known creases or points (corners of the sheet of paper or intersections of creases) are allowed.

3.1 The Geometry of Origami

Usually Origami is viewed simply as the art of building models, through the performance of a sequence of folds, in some cases folding the sheet of paper and unfolding it back (to mark a crease on the paper), in others keeping the paper folded. However, the mathematical analysis of these constructions yielded an interesting geometry. Most people are familiar with straight-edge and collapsing compass geometry in which lines can be constructed through the use of an unmarked straight-edge (to draw a line between two known points), circles constructed using a collapsing compass (which allows the drawing of a circle centred on a known point, passing through another known point) and points

which can be constructed only through finding the intersection of two known shapes. It is well-known that using only such construction techniques, one cannot deduce certain positions on the plane. Most famously, one cannot trisect an arbitrary angle, square a circle or double a cube using such techniques. Origami constructions are made up of only folds (straight lines) and points. New lines can be deduced by folding the paper (in a limited number of ways) and points through finding the intersection of two lines. Folds are temporary, used only to induce a line, with the sheet of paper then being unfolded. Interestingly, certain points and lines can be constructed using the Origami operations but not using straight-edge and collapsing compass (and vice-versa).
In our Origami DSEL (OriDSEL), we consider this type of geometric interpretation of Origami:

- the paper is considered to be an idealized mathematical plane;
- a fold line is considered to be an infinitely extended line on the plane;
- a reference point is an idealized point on the plane.

3.2 The Axioms of Origami Geometry

As in the case of straight-edge and collapsing compass geometry, in Origami-based geometry, one can only construct lines and points in a set of well-defined ways. Origami constructions have been reduced to seven underlying axioms[1]:

Axiom 1: Given two points p_1 and p_2, one can construct a line that passes through both of them.

Axiom 2: Given two points p_1 and p_2, one can construct a line, folding along which, places p_1 onto p_2.

Axiom 3: Given two lines l_1 and l_2, one can construct a line, folding along which, places l_1 onto l_2.

Axiom 4: Given a point p_1 and a line l_1, one can construct a line folding along which places l_1 onto itself (in other words, is perpendicular to l_1) and that passes through point p_1.

Axiom 5: Given two points p_1 and p_2 and a line l_1, one can construct a line passing through p_2, and folding along which, places p_1 onto l_1.

Axiom 6: Given two points p_1 and p_2 and two lines l_1 and l_2, one can construct a line, folding along which, places p_1 onto l_1, and p_2 onto l_2.

Axiom 7: Given a point p_1 and two lines l_1 and l_2, one can construct a line, folding along which, places p_1 onto l_1 and l_2 onto itself (in other words, is perpendicular to l_2).

[1] The original six can be found in [Huz92], while the seventh, so-called Hatori's axiom, was added later — see `http://origami.ousaan.com/library/conste.html` for more details

4 OriDSEL — Embedding Origami Axioms

The axiomatisation of Origami constructions provides a straightforward way in which we choose to embed the language of Origami constructions. In OriDSEL, we choose to use a deep embedding, to have access to the structure of a construction, enabling us to provide different interpretations of a single model. Internally, two basic types are used to model points and lines (or folds):

```
data Point =
    Intersection Line Line
  | ...

data Line =
    Axiom1 Point Point
  | Axiom2 Point Point
  | Axiom3 Line  Line
  | ...
```

In keeping with the host language, we choose to show constructions to the user as functions from a structure of points and folds to a structure of points and folds. The axioms themselves are visible to the user as functions, with names indicating their behaviour:

```
intersect :: (Line, Line) -> Point
intersect = uncurry Intersection

foldThroughPoints :: (Point, Point) -> Line
foldThroughPoints = uncurry Axiom1

foldPointOntoPoint :: (Point, Point) -> Line
foldPointOntoPoint = uncurry Axiom2

foldLineOntoLine :: (Line, Line) -> Line
foldLineOntoLine = uncurry Axiom3

...
```

Thus, for example, given a rectangular sheet of paper (as four points), we can give a construction to select the four edges (lines) of the sheet of paper:

```
fourSidedPaper ::
  (Point, Point, Point, Point) -> (Line, Line, Line, Line)
fourSidedPaper (nw,ne,se,sw) = (n,s,e,w)
  where
    n = foldThroughPoints (nw, ne)
    s = foldThroughPoints (sw, se)
    e = foldThroughPoints (se, ne)
    w = foldThroughPoints (sw, nw)
```

We can generalise this, such that, given a polygonal sheet of paper (as a sequence of points), we can give a construction to select the edges (lines) of the sheet of paper. This can be used to give an alternative definition of `fourSidedPaper`:

```
edgesOfPolygon :: [Point] -> [Line]
edgesOfPolygon vertices@(v:_) =
  [ foldThroughPoints (p,p') | (p:p':_) <- tails (vertices ++ [v]) ]

fourSidedPaper' (nw,ne,se,sw) = edgesOfPolygon [nw,ne,se,sw]
```

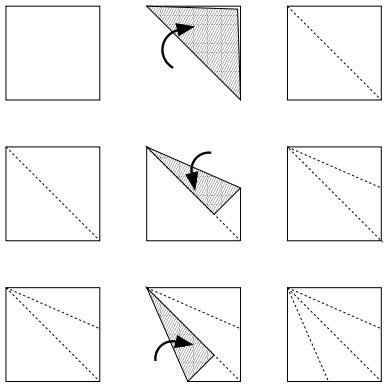

Fig. 1. Folding a kite

Using an abstract datatype to describe Origami models, constructions correspond to a tree. In practice, one can have sharing of points or lines as in the following example of a standard construction (see figure 1):

```
kite (nw,ne,se,sw) = (nw, point_e, se, point_s)
  where
    diagonal     = foldThroughPoints (nw, se)
    (n, s, e, w) = fourSidedPaper (nw,ne,se,sw)

    l1           = foldLineOntoLine (n, diagonal)
    l2           = foldLineOntoLine (w, diagonal)

    point_e      = intersect (l1, e)
    point_s      = intersect (l2, s)
```

In this example, the diagonal is used twice in the construction. However, due to referential transparency, there is no way we can differentiate the construction from a similar one, but which replaces the two references to `diagonal` by `foldThroughPoints (nw, se)`. In practice, when traversing the data structure

produced, we would want to identify the common parts. This will avoid, for instance, describing how to obtain `diagonal` twice, when explaining how to construct the kite. Various solutions have been proposed in the literature, including explicit naming of structures [O'D92] and the use of the state monad to thread references to definitions [CSS01]. Finally, we opted for the use of observable sharing [CS99], which enables a cleaner description, albeit breaking the functional purity of the language. In practice, the use of references is hidden within the basic datatypes and the axiom definitions, thus enabling the user to write the programs we have shown with no knowledge of the underlying machinery.

The advantages of the use of an embedded language, with the host language as a meta-language of the domain specific language become more apparent when we produce regular repetitive constructions as in the following example:

```
cross (nw, ne, se, sw) = (ns,ew)
  where
    ns = foldPointOntoPoint (nw, ne)
    ew = foldPointOntoPoint (ne, se)

subsquare corners =
  (intersect (ns,n), intersect (ew, e),
   intersect (ns,s), intersect (ew, w))
  where
    (ns, ew) = cross corners
    (n, s, e, w) = fourSidedPaper corners

repeatedSubsquare 0 square = square
repeatedSubsquare n square =
  subsquare (repeatedSubsquare (n-1) square)
```

4.1 Manipulation of Origami Constructions

The availability of a meta-language to the domain-specific language enables us to provide a library for the analysis and manipulation of programs in the embedded language. We provide a number of such functions to enable the user to explore and study the constructions described.

Explaining a given construction: We provide functions which explain an Origami construction and produce a textual explanation of how it be achieved by traversing the directed acyclic graph describing the construction. We provide both plain text and HTML descriptions, explaining the model in a step-by-step manner. HTML descriptions are richer, giving links between the different parts of the construction when referring to previously described folds or reference points. It is important for us to identify sharing in the given model, to avoid repeated descriptions of the same construction.
Below is the text description of a kite fold given earlier:

```
Fold the paper along points NW and point NE, calling it line 1.
Fold the paper along points SW and point NW, calling it line 2.
```

```
Fold the paper along points SW and point SE, calling it line 3.
Fold the paper along points SE and point NE, calling it line 4.

Fold the paper along points NW and point SE, calling it line 5.

Fold the paper by putting line 1 over line 5, calling it line 6.
Fold the paper by putting line 2 over line 5, calling it line 7.

Find the intersection of line 4 and line 6, calling it point 1.
Find the intersection of line 3 and line 7, calling it point 2.

The result is (NW, point 1, SE, point 2)
```

Animation: Textual descriptions can be useful for small constructions, but tend to become too long and complex for larger ones. Having to keep track of previously identified creases and reference points can quickly get out of hand. OriDSEL provides a link to an external tool we have build to show an animation showing, step-by-step, how the construction is achieved.

Constraint Checking: The Huzita Hatori axioms are partial functions in that they are only defined for some inputs. For example, axiom 5 (given two points and a line, draw a line going through one of the points and folding along which places the other point on the line) cannot be applied to if the points lie on opposite sides of the line. Because of this, constructions are also partial, in that for certain inputs, the model will fail. We provide functions to calculate the constraints that are to be satisfied in order for a construction to be well defined. These constraints can then be checked for concrete values of the vertices.

5 Partitioning of Models

As the number of folds in an Origami model increases, so does its complexity. In most of the Origami literature, complex Origami models are not described in terms of the basic folds, but rather in terms of so called base folds — each of which being an often used sequence of basic folds. Using Haskell, the user descriptions of a model in OriDSEL can be written to resemble the ones in the Origami literature. Starting off with a library of base folds, one can describe complex models in terms of these library functions. However, the internal description of these models obviously contains no information about which parts of the model were generated by which functions. We would like to add sufficient information in the internal structure to enable concise output descriptions, using compound constructions (such as base folds) in the description. Since such base folds vary in difficulty, we would like to enable tagging of blocks not only with a name (for reference), but also with a difficulty level. The user can then request textual descriptions taking his or her expertise level into account.

Various techniques have recently been proposed in the literature to resolve this issue of named blocks in embedded languages. For instance, [ACS05,Pac07]

take a component, or block, oriented view of circuits, composed together using a set of combinators rather than simply functional composition. Other approaches, such as [MO06] look at the use of meta-programming features to access this information. In our case, the problem is simpler — we only want to name and tag the difficulty of a few blocks, and their access is based entirely on difficulty level which ranges over few possible values.

One possible approach is to tag all nodes in a structure wite a name and difficulty level. The main problem with this solution is that without additional internal machinery, there is no way of differentiating between two blocks with the same name connected together, and one large block with that name. Furthermore, one can make do with much less information in the model, to enable outputting descriptions in terms of named blocks.

The approach we take is to label output boundaries of a block with all relevant information about the block (name, difficulty level, input and output nodes) enabling structured output descriptions by referring to blocks of the appropriate level as a whole using the name stored on the boundaries.

```
data SkillLevel = Beginner | Intermediete | Expert
data Boundary = Boundary String SkillLevel [Ref] [Ref]

data Point =
    OutputBoundaryP Boundary Point
  | Intersection Line Line
  | ...

data Line =
    OutputBoundaryL Boundary Line
  | Axiom1 Point Point
  | Axiom2 Point Point
  | ...
```

As with the underlying axioms, the above data structures are abstracted away from the user:

```
block skill name construction ins =
  markAsOutput skill name (structToRefs ins, structToRefs outs) outs
  where
    outs = construction ins

beginnner    = block Beginner
intermediete = block Intermediate
expert       = block Expert
```

The function `markAsOutput` marks a structure of output lines and points with the appropriate constructor and `structToRefs` transforms a structure of lines and points into a list of references to the data.

Users may now name blocks and label them with their difficulty level. For instance, the `subSquare` construction given earlier may be tagged as an intermediate level base fold with the name ‘subsquare-construction’ in the following manner:

```
subsquareBlock = intermediate "subsquare-construction" subsquare
```

The use of `subsquareBlock` is now identical to that of `subsquare`, enabling us to redefine the repeated subsqure construction given earlier using the base fold:

```
repeatedSubsquare 0 square = square
repeatedSubsquare n square =
  subsquareBlock (repeatedSubsquare (n-1) square)
```

Now, by calling `explainIntermediate (repeatedSubsquare 10)`, we get an explanation appropriate for the intermediate user. Rather than explaining all the constructions from scratch, the system will assume that the description is aimed at someone having an intermediate skill level, who thus knows what a `subsquare-construction` is, and will give just a ten line explanation, one for each application of the subsquare construction. On the other hand, `explainBeginner` will give a full description going down to the underlying axioms.

6 Conclusions

In this paper, we have explored the embedding of Origami geometric constructions in Haskell. The axiomatization of the problem domain has provided us with the necessary building blocks upon which to build the language. Using standard techniques from embedded languages, we have built a deeply embedding of such constructions and a number of functions to analyse and manipulate constructions. The axioms of Origami geometry have been explored well in the literature. However, looking into actual model creation with Origami introduces further challenges — the folding primitives remain unchanged, but extra information needs to be added to the folds to specify whether the fold is unfolded, the direction of folding and for more advanced models, the angle of the fold.

The main challenge in this domain is the abstract description of models, enabling the user to hide information away for an expert, for whom a more abstract description of a sequence of folds would be sufficient. The solution we have developed enables explicit tagging of such blocks, which can be impractical in certain other contexts and a potential source of errors in others (through the reuse of tags). In our case, the use of a small number of tags (corresponding to the difficulty levels and names of basic folds), enables a simpler solution, which is used to effectively enable different descriptions of the same model. It would be interested to explore this technique in other contexts, such as textual explanations of proofs in theorem proving.

References

[ACS05] Emil Axelsson, Koen Claessen, and Mary Sheeran. Wired: Wire-aware circuit design. In *Proceedings of Conference on Correct Hardware Design and*

Verification Methods (CHARME), volume 3725 of *Lecture Notes in Computer Science*. Springer-Verlag, October 2005.

[CS99] Koen Claessen and David Sands. Observable sharing for functional circuit description. In *Proceedings of Asian Computer Science Conference (ASIAN)*, Lecture Notes in Computer Science. Springer Verlag, 1999.

[CSS01] Koen Claessen, Mary Sheeran, and Satnam Singh. The design and verification of a sorter core. In *CHARME*. Springer, 2001.

[DLC99] Nancy A. Day, Jeffrey R. Lewis, and Byron Cook. Symbolic simulation of microprocessor models using type classes in Haskell. In *CHARME'99 Poster Session*, 1999.

[Hud96] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196, 1996.

[Huz92] H. Huzita. Understanding geometry through origami axioms. In J. Smith, editor, *Proc. of the First International Conference on Origami in Education and Therapy (COET91)*, pages 37–70. British Origami Society, 1992.

[Jon01] Simon L. Peyton Jones. Composing contracts: An adventure in financial engineering. In José Nuno Oliveira and Pamela Zave, editors, *FME*, volume 2021 of *Lecture Notes in Computer Science*, page 435. Springer, 2001.

[Jon03] Simon Peyton Jones. *Haskell 98 language and libraries: the Revised Report*. Cambridge University Press, 2003.

[LM99] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *DSL*, pages 109–122, 1999.

[Mit77] David Mitchell. *Mathematical Origami*. Tarquin Publications, 1977.

[MO06] Tom Melham and John O'Leary. A functional HDL in reFLect. In Mary Sheeran and Tom Melham, editors, *Sixth International Workshop on Designing Correct Circuits: Vienna, 25–26 March 2006: Participants' Proceedings*. ETAPS 2006, 2006.

[O'D92] John T. O'Donnell. Generating netlists from executable circuit specifications. In John Launchbury and Patrick M. Sansom, editors, *Functional Programming*, Workshops in Computing, pages 178–194. Springer, 1992.

[Pac07] Gordon J. Pace. A strongly typed, component-based embedded hardware description language. In *Proceedings of the Computer Science Annual Workshop 2007, University of Malta*, 2007.

[Spe01] M. Sperber. Developing a stage lighting system from scratch. In *Proceedings of the sixth ACM SIGPLAN International Conference on Functional programming*, 2001.