Homotopy Type Theory for Sewn Quilts

Charlotte Clark
Worcester Polytechnic Institute
Computer Science Department
100 Institute Road
Worcester, MA 01609
ceclark@wpi.edu

Rose Bohrer
Worcester Polytechnic Institute
Computer Science Department
100 Institute Road
Worcester, MA 01609
rbohrer@wpi.edu

Abstract

This paper introduces PieceWork, an imperative programming language for the construction of designs for sewn quilts, whose semantics are inspired by Homotopy Type Theory. The goals of PieceWork include improving the diversity of sewn designs that can be represented in computational methods, demonstrating a creative application of Homotopy Type Theory, and demonstrating that the craft of quilting is a worthy object of study in programming language theory. We develop an operational semantics, provide a prototype implementation and examples, and provide initial theoretical results. Type system design is in-progress.

1. Introduction

Quilting is a popular craft across a variety of cultures worldwide, leading to a wide diversity of design and creative expression throughout the craft. Due to its popularity, quilting is well-studied as a medium for computational crafting, yet existing approaches to computational modeling of quilts do not reflect the full diversity of the craft. To overcome limitations of expressiveness, this work develops a flexible theoretical framework for modeling quilts as programs.

To achieve this, we take a mainly-topological approach, using higher equality structures from Homotopy Type Theory (HoTT) to represent pieces of fabric in a quilt. HoTT is a branch of type theory that supports higher-order equality structures [19] which we use as the foundations for our quilting programming language, PieceWork. This work is an extension of the first author's Master's thesis.

Our topological interpretation is based on the fundamental idea that *exact equality of points corresponds to points being located at the same place, isomorphism corresponds to points being connected by fabric, and sewing corresponds to unification*. Using topology (as opposed to, e.g., geometry) for the basis of our language allows us to represent complex quilting designs. Compared to prior geometric approaches, which are often limited to pre-set base components to ensure compatibility between neighboring pieces [11, 22], PieceWork allows for any shapes to be joined together to form pieces beyond simple square or rectangular quilt blocks.

PieceWork's flexibility is especially important for representing quilts from non-Western and non-White cultural backgrounds. As an example, Harriet Powers' Bible Quilt [18], depicted in Figure 1, has complex geometric properties such as nonconvex shapes, curves, and assymmetry. This quilt is difficult or impossible to represent using existing patch-based tools [11, 22]. In contrast, Western-style designs (e.g., in Figure 2) are easy to represent in existing tools.

PieceWork's operational semantics have been developed and implemented, but its type system is in-development. Nonetheless, PieceWork is a HoTT-based language insofar as its *values* correspond to types in HoTT; the eventual role of a types in PieceWork is to assure that the construction of types in HoTT succeeds. Future work could refine PieceWork into a valuable foundation for inclusive computational quilt drafting tools. We expect PieceWork to generalize to other crafts, but focus on quilts to limit scope.

Figure 1: Bible Quilt (1886) by Harriet Powers.

FARM'23 September 8, 2023, Seattle, Washington, USA.

ISBN [to be supplied]. . . $15.00

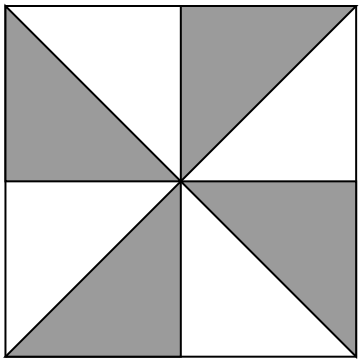

Figure 2: Design 1262a on page 172 of Barbara Brackman's Encyclopedia of Pieced Quilt Patterns [5] is a geometrically simple pinwheel block.

In Section 8, we outline what a potential type system would look like and explain the subtle challenges underlying it; readers are welcome to skim Section 8 before reading the semantics of PieceWork.

1.1 Motivations

Our motivations are multifaceted. Computational craft is important in its own right, and advancing the foundations of computational craft could have impacts ranging from novel tools for creative exploration to novel interface paradigms, like those explored in projects such as Threadsteading [1] and Loominary [20]. We expect PieceWork to serve as the foundation for a higher-level, user-facing graphical tool appropriate for quilters without programming experience to procedurally generate quilt patterns. Procedural generation of quilt patterns can help crafters seeking inspiration, or a use for spare scraps [14], while also exposing them to computing on their own terms, in a context they are familiar with [11]. Crafting has likewise been explored to encourage underrepresented groups to engage with computing [17]. LilyPad Arduino [6] combined circuits into sewing lessons to introduce 10 to 14 year old children to electronic textile design and fabrication. With this approach, they reached students who had not considered the subject before [6].

In contrast to prior work on computational craft that emphasized exposing crafters to computing [6, 11, 17], we conjecture that PieceWork's potential for cultural impact lies with self-identified computer scientists and mathematicians who practice crafts. By applying higher mathematics to the representation of a feminine-coded activity, PieceWork promotes the message that feminine-coded activities are deserving as first-class objects of study in mathematically-intensive fields. Complementary to prior works emphasizing *diversity* in computing through crafting, the underlying social message of PieceWork is one of *belonging*.

This message of belonging is ripe for exploration as the HoTT community has recently undertaken new community outreach efforts. In July and August of 2022, Western University in Canada ran the HoTTest Summer School seminar, with the goal of making "homotopy type theory accessible to, and inclusive of, everyone who is interested, regardless of cultural background, age, ability, formal education, ethnicity, gender identity, or expression" [12]. The target audience was students with background in abstract mathematics or theoretical computer science, and the seminar ran online to attract participants from any physical location. The message of this work is well-aligned with the target audience and goals of the summer school, providing a welcoming and concrete medium for mathematically-experienced students to engage with HoTT while expressing their whole selves.

2. Related Work

Creative tools are a critical part of computational crafting, and provide application to the algorithms created for procedural generation. Coarhen and Fiume [9] describe a tool to produce patterns for Bargello quilts from a simple sketch. These quilts are comprised of rectangular pieces arranged in columns, where each column contains pieces of the same height. Variations in the color of the pieces form large, sweeping curves across the quilt surface. These requirements make manually designing Bargello quilts challenging. To alleviate this, the program takes a curve as input, and generates the pattern needed to produce the curve in the Bargello style. Users can then select fabric colors and preview what the final design would look like. However, Bargello quilts are only one of many different quilt design techniques, and this type of program is not applicable to other styles of quilts.

Another creative tool called PatchProv [14] enables users to easily improvise quilt designs from pieces of fabric they already have. Users upload images of the pieces they wish to use, which then can be virtually arranged and sewn to test possible patterns. The program also tracks the steps used to create the virtual pattern, such that it can be recreated from the physical pieces. However, one challenge quilters encountered in their user study was the physical practicality of the virtual designs. Participants noted that PatchProv did not take all domain-specific constraints into account, such as material usage or whether it can feasibly be sewn. In this work, we will be taking a more abstracted approach to pattern generation, because we will be building from generic shapes instead of literal pieces in the user's possession. The long-term goal is to serve as a planning resource, rather than an improv-oriented tool like PatchProv.

Other works take a more algorithmic approach, using graphs to represent quilting patterns [13, 15] using techniques such as foundation paper piecing [13] and topstitching [15]. Topstitching is a technique used to attach the pieced quilt pattern to a solid sheet of backing fabric to make the quilt thicker and more durable. Many quilters will take this as another opportunity for creativity, and use a long-armed sewing machine to create intricate patterns in their topstitching. Li et al. [15] developed a method to procedurally generate designs to fill a bounded area. Similarly, Liu et al. [16]

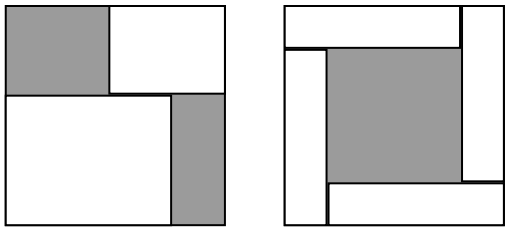

Figure 4: Mondrian and Rose blocks cannot be paper pieced.

create complex single-line topstitching patterns from photographs. Both works use spanning trees as a basis for the more detailed final design.

Procedural generation of piece arrangements is less explored. One example is Leake et al.'s [13] use of dual hypergraphs to represent straight-cut (foundation) paper-pieceable quilts. Foundation paper pieceable quilts tend to be easier to sew, and are popular with beginners. Each node of the hypergraph represents a piece of fabric within the design, with hyperedges representing seams. If two or more pieces share a seam, then there is a hyperedge connecting the corresponding nodes. Using this model, they developed an algorithm to determine whether a quilt design is paper pieceable, as well as how to physically construct paper pieceable designs. However, paper pieceable quilts are only a subset of the quilts that crafters may want to make. Paper pieceable quilts require that every added piece be added at a T-junction, and that only one piece can be added at a time. Paper pieceable designs are useful, but many popular quilting motifs, such as roses, cannot be created using this method.

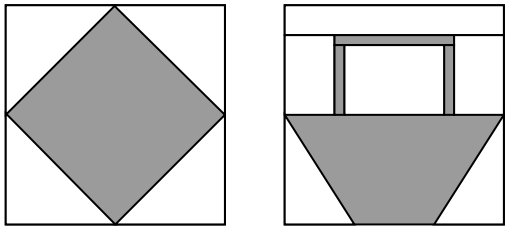

Figure 3: Examples of foundation paper-pieceable blocks.

Barbara [22] is a live-coding language for generating quilt designs. Barbara is a graphics programming language which allows users to input which shapes they would like to add to the canvas. Then, colors and animations can be applied to the shapes, as well as filters to distort the overall image. Designs are created by layering rectangles, triangles, and squares on top of each other. Commands can be changed live to update the image in real time, lending Barbara to live performances and demonstrations. The shapes that users can place using Barbara are predefined geometrically, which is useful for streamlining the live development process. However, this means that it is inherently inflexible, as creating new basic blocks requires language changes.

In this work, we are also developing a programming language for quilts, but with a different focus. We aim to develop a topological method for representing quilts and quilt construction. This contrasts from Barbara and the dual-hypergraph approach in that we want to have a powerful abstraction for shapes and sewing, even those which include highly irregular shapes or non-paper-pieceable motifs. Barbara is abstracted from sewing and is focused on the graphical output of the language, while the dual-hypergraph approach is limited in what it can represent. Our topological system allows us to represent a variety of shapes with fewer constraints than a geometric or graphical approach would entail, and our language design and operations are sewing-oriented. We hope this work will serve as a foundation for future work in generating quilt patterns in a user-facing tool. To this end, we use higher inductive types (HITs) from HoTT to represent pieces and how they are joined.

3. Homotopy Type Theory Background

Homotopy type theory (HoTT) is a way to describe relationships between types using homotopies. In traditional homotopy theory, points x in a space A are written as $x \in A$. However, in HoTT, we use the type-theoretic notation $x : A$, which means that the element x is of type A. For isomorphism between types, we retain the symbol $\simeq$ from homotopy theory. When we have two elements where a continuous deformation exists between them, we can write $x_1 \simeq_A x_2$ where A is the type at which we are comparing the two elements. We can notate higher equivalences similarly: $pq : p \simeq_{x_1 \simeq_A x_2} q$ indicates a homotopy between p and q, where each is a path between $x_1 : A$ and $x_2 : A$. This sort of equivalence is called propositional equivalence, and is the foundation of the core axiom of HoTT: Univalence.

Univalence Axiom. For any $A, B : U$, a function $(A \simeq B) \to (A =_U B)$ is an equivalence.
This gives us that $(A =_U B) \simeq (A \simeq B)$

Here, we use equivalence to mean two points which are congruent, but may be translated, rotated, or scaled differently while equality refers to two points which are exactly identical. The univalence axiom states that propositional equivalence is isomorphic to propositional equality. This means that we only have to reason to the level of propositional equivalence, and it is indistinguishable from equality within the theory of HoTT.

Nonetheless, extra-logical reasoning on exact (definitional) equality is often helpful, which helped motivate the development of two-level type theory [3]. The core underlying metaphor of PieceWork is best understood in two-layer terms: propositionally-equivalent points are connected by fabric and definitionally equal points are colocated.

We can express infinitely many "layers" of equivalence using HoTT. Paths can be found between elements, but they can also be found between paths, and we can even

draw paths-between-paths-between-paths. We can also define types inductively, using these higher equalities to generate constructors. These are known as Higher Inductive Types (HIT), which can be used to describe connectivity structures between elements [2] and are thus used in PieceWork to represent the *state of a quilt*. A typical example HIT models a circle (depicted in Figure 5), which has a path connected at both ends to a single point. This can be written as:

```
Circle: type,
base: Circle,
loop: base = base
```

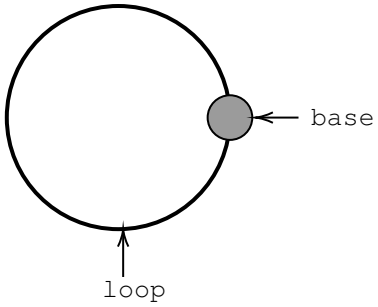

Figure 5: Visualization of the Circle type.

This definition introduces `base` as an element of type `Circle`, which can be visualized as a point in space. Then, `loop` is defined as a constructor for `base = base`, meaning there is a non-trivial path that starts and ends at the `base`. This creates the circle surrounding the `base` point. We hope to use a similar definition to describe the shapes we will use in our quilt patterns. Rather than only circles, we will use points and and paths between them to represent the topologies of quilt blocks. Though those blocks will typically be visualized as polygons, this topological abstraction applies equally well to blocks with organic curves.

4. PieceWork

PieceWork is a language to represent and manipulate quilt designs. We use two-dimensional embeddings of topological structures to represent pieces of fabric. These shapes can be subdivided into smaller pieces, or joined at the edges to form complex quilt designs featuring various shapes and colors.

Notation. We briefly explain the notation for lists used throughout the work. We desire notation for extracting both sublists and elements of lists. In our notation, sub-lists are followed by semicolons, and singular elements are separated by commas. For example:

```
wholeList = [SubListA; elementA,
             SubListB; elementB]
```

In most cases lists and sub-lists are named with uppercase letters and vector notation, while elements are only given lowercase letters to further distinguish between the two.

$$
\begin{aligned}
e ::= \; & point(c,c) \mid edge(e,e) \\
& \mid material(c,c,c) \\
& \mid simpleshape(\overrightarrow{E},\ material) \\
& \mid complexshape(\overrightarrow{S},\ \sigma) \\
& \mid e.x \mid e.y \mid e.beg \mid e.end \\
& \mid e.edges \mid e.mat \\
& \mid e.shapes \mid e.subst \\
& \mid e.length \mid e[n] \\
& \mid f(e) \mid (e_1, \ldots, e_n) \\
& \mid x \mid n \mid p \\
& \mid \mathbf{mark(e,n)} \mid \mathbf{cut(e,e)} \mid \mathbf{sew(e,e)} \\
n,\ \tilde{n} ::= \; & c \mid n + \tilde{n} \mid n \times \tilde{n} \mid n - \tilde{n} \mid x \\
p ::= \; & n > \tilde{n} \mid n \geq \tilde{n} \mid n = \tilde{n}
\end{aligned}
$$

Figure 6: PieceWork Expressions

$$
\begin{aligned}
prog ::= \; & e \mid prog_1; prog_2 \mid lhs := prog \\
& \mid if(e)\ \{prog_1\}\ else\ \{prog_2\} \\
& \mid while(e)\ \{prog\} \mid return\ e \\
\\
lhs ::= \; & x \mid \_ \mid [\ lhs_1, ..., lhs_n\ ]
\end{aligned}
$$

Figure 7: PieceWork Programs

We present the syntax of PieceWork in Figure 6 (Expressions) and Figure 7 (Programs), then elaborate on each PieceWork feature individually. In Figure 6, c stands for any literal natural number, n for any numeric expression, p for any predicate, and e for any expression.

4.1 Expressions

To construct more complex expressions such as shapes, we must first define two critical building blocks: points and edges. A `Point` is the most fundamental element of PieceWork, consisting of an (x, y) coordinate pair representing a point in space. Here, x and y are real numbers ($\mathbb{R}$).

Edges allow us to specify which points are connected to each other. We define our `Edge` type as a path between two points. The `Edge` data structure specifies a beginning and endpoint as in directed edges, but the underlying isomorphism is understood in traditional undirected HoTT. If we were to write the definition in the style of the `Circle` example, it would look like this:

```
Point: type,
beg, end: Point,
```

```
edge: beg = end
```

We define a typing judgement $\Gamma \vdash e : \tau$ for expressions, meaning expression e has type τ under context Γ. We define typing rules for points and edges. Points are built from numeric literals; edges can be composed of any expression with the proper type.

$$\textsc{Point}\ \frac{*}{\Gamma \vdash (c_x, c_y) : Point}$$

$$\textsc{Edge}\ \frac{\Gamma \vdash beg : Point \qquad \Gamma \vdash end : Point}{\Gamma \vdash (beg, end) : Edge}$$

Now, we have the tools needed to discuss quilts. Three or more edges together can form a closed shape (as defined in Algorithm 1), under the condition that the endpoint of each edge is equal to the starting point of the next. We define a simple shape as a list of edges which form a closed loop, and a value called material which describes the color of the shape. Materials are defined as a triple of three numbers between 0 and 1, representing an RGB color. RGB is an additive color model where the first number represents the proportion of red light, the second number the proportion of green light, and the last color the proportion of blue light in the overall color. Materials could be expanded upon in the future to improve generality, but for the purposes of this work, RGB colors are sufficient. The subexpressions of a material are required to be numeric literals.

$$\textsc{Material}\ \frac{*}{\Gamma \vdash material(c_r, c_g, c_b){:}Material}\ c_{r,g,b} \in [0, 1]$$

$$\textsc{SimpleShape}\ \frac{\Gamma \vdash \overrightarrow{E} : (Edge, \cdots, Edge) \qquad \Gamma \vdash \mu : Material}{\Gamma \vdash (\overrightarrow{E}, \mu) : SimpleShape}\ \overrightarrow{E}\ \textsc{is closed}$$

Algorithm 1 A list of edges $\overrightarrow{E}$ is closed if:

$|\overrightarrow{E}| \geq 3$
`and`
`for each` i `in [1 ...` $|\overrightarrow{E}|$`]`
E_i`.end =` E_{next}`.beg`
`where if` i `=` $|\overrightarrow{E}|$
E_{next} `=` E_1
`else`
E_{next} `=` E_{i+1}

Complex shapes represent quilts, or portions of quilts, consisting of multiple pieces of fabric sewn together. Similar to how a `SimpleShape` is a collection of edges, the `ComplexShape` type consists of a collection of simple shapes, which can be connected together with seams. The seams within a `ComplexShape`, if any, are represented via a list of edge substitutions, one edge substitution for each seam joining two edges. In HoTT terms, a seam induces a definitional equality between the edges, making them not only connected but identified.

It is important to note that not all shapes within the `ComplexShape` need to be seamed together. There could be shapes which are completely detached from the rest of the `ComplexShape`. This is because `ComplexShapes` are also used to represent the current quilt state in the program, even the initial state where no sewing has been performed yet, or states where only some of the pieces are seamed.

$$\textsc{ComplexShape}\ \frac{\Gamma \vdash \overrightarrow{S} : (Shape, \cdots, Shape) \qquad \Gamma \vdash \sigma : Var \rightharpoonup Var}{\Gamma \vdash (\overrightarrow{S}, \sigma) : ComplexShape}$$

The edge substitution is specifically a renaming substitution. A renaming substitution σ is a partial function of type $Var \rightharpoonup Var$ such that $Range(\sigma) \cap Dom(\sigma) = \emptyset$. The σ property of a `ComplexShape` is a renaming substitution which unifies the edges of the sub-shapes in $\overrightarrow{S}$ into the final `ComplexShape`. Thinking physically, we can only fully seam two edges if we can move the pieces of fabric so that the endpoints align. For our renaming substitutions, we will be checking that the first edge can be obtained by substituting its endpoints for the second edge's endpoints.

These four types are the core data of PieceWork which will be explored throughout the rest of this work. The values and other expressions of PieceWork are summarized in Figure 6. Along with the fabric-oriented values, we include numeric terms n, which feature basic mathematical operations $(+, -, \times)$. In this work, n refers to any natural number or natural number expression.

Expressions include Boolean propositions p; for simplicity, we limit propositions to comparison operations over numeric terms $(>, \geq, =)$. Expressions using the point, edge, simple shape, and complex shape types are included for accessing their components. To access the x and y values of a point, we can use the notations `point.x` and `point.y`. To access the beginning and end points of an edge, we can use `edge.beg` and `edge.end`. To access the list of edges which comprise a shape, we use `simpleshape.edges`. To access the material of a shape, we use `simpleshape.mat`. Finally, to access the list of simple shapes making up a complex shape, we can use `complexshape.shapes`, and the list of substitutions can be accessed with `complexshape.subst`.

Dot notation in PieceWork accesses fields similarly to dot notation in C-like languages. Our typing rule for the dot operation is as follows.

$$\textsc{Dot}\ \frac{\Gamma \vdash \tau_1 \qquad \Sigma(f) : \tau_1 \to \tau_2}{\Gamma \vdash e.f : \tau_2}$$

where the signature Σ specifies the types of all fields as follows:

$$\begin{aligned}
\Sigma = \{\ & x \mapsto (\texttt{Point} \to \mathbb{R}),\\
& y \mapsto (\texttt{Point} \to \mathbb{R}),\\
& \mathit{beg} \mapsto (\texttt{Edge} \to \texttt{Point}),\\
& \mathit{end} \mapsto (\texttt{Edge} \to \texttt{Point}),\\
& \mathit{edges} \mapsto (\texttt{SimpleShape} \to (\texttt{Edge}, \cdots, \texttt{Edge})),\\
& \mathit{mat} \mapsto (\texttt{SimpleShape} \to \texttt{Material}),\\
& \mathit{shapes} \mapsto (\texttt{ComplexShape} \to\\
& \qquad (\texttt{SimpleShape}, \cdots, \texttt{SimpleShape})),\\
& \mathit{subst} \mapsto (\texttt{ComplexShape} \to \mathit{Var} \rightharpoonup \mathit{Var}),\\
& \mathit{length} \mapsto ((\tau_1, \cdots, \tau_n) \to \mathbb{R})\}
\end{aligned}$$

The dot operator returns the value associated with that field within the given instance. The below table matches each dot operator with its specific return value.

Element	Syntax	Field Returned
Point	x	x
Point	y	y
Edge	beg	beg
Edge	end	end
SimpleShape	edges	$\overrightarrow{E}$
SimpleShape	mat	μ
ComplexShape	shapes	$\overrightarrow{S}$
ComplexShape	susbt	σ

Tuples are included in PieceWork, denoted $(e_1, \ldots, e_n)$ or $(v_1, \ldots, v_n)$ for a value. The projection operation $e[n]$ extracts the n'th element of tuple e, which is zero-indexed, or returns a runtime error if that element is undefined. Such errors are typically avoided by checking the length of a tuple ($e.\mathit{length}$). Elements can also be accessed using pattern-matching in assignments (Figure 7). Functions calls are denoted $f(e)$. Function definitions are not included as part of the *expression* syntax because all function definitions in PieceWork occur at the top level, using the syntax `f(argSpec){` prog `}` where `argSpec` is a comma-separated list of specifications (type names followed by variable names). The types of expressions are discussed in Section 4.3

We are now ready to present the core novel operations of PieceWork language, specialized to constructing new sewn designs out of pieces. The operations `mark(e, n)`, `cut(e, e)`, and `sew(e, e)` are the computational counterparts of core quilting tools: chalk for marking fabric, scissors for cutting, and a sewing machine for stitching. To represent our chalk, we have the mark operation, which allows users to specify a point along an edge using ratios. Cutting with scissors is defined between two points on a simple shape, and splits one contiguous piece of fabric into two. Sewing joins two pieces of fabric into one contiguous piece, given two edges which can be aligned together.

We can string these operations together to form PieceWork programs which construct quilts.

4.2 Program Statements

Program statements prog include single expressions e, such as cut, sew, or mark operations. Compound programs include sequential compositions $(\mathit{prog}_1; \mathit{prog}_2)$, assignments $\mathit{lhs} := \mathit{prog}$, conditionals $\mathit{if}(e)\ \{\mathit{prog}_1\}\ \ \mathit{else}\ \{\mathit{prog}_2\}$, loops $\mathit{while}(e)\ \{\mathit{prog}\}$, and function returns $\mathit{return}\ e$. Variables do not need to be declared before assignment; the assignment statement creates a new variable in the current scope if needed, or else updates its value. The left-hand sides of assignments perform pattern-matches consisting of variable patterns x which bind a value to a single variable, wildcard patterns _ which discard a value, and tuple patterns $[\ \mathit{lhs}_1, ..., \mathit{lhs}_n\ ]$ which bind each component of a tuple according to its corresponding pattern.

Explicit function returns are included in the syntax for the sake of simplifying the implementation; we assume that each path through a function ends in a single `return` statement.

4.2.1 Definitions

In PieceWork, all definitions occur at the top level and the only definition construct is a function definition:

$$\mathit{decl} ::= \texttt{f(argSpec)\{prog\}}$$

where `argSpec` is a comma-separated list of specifications in the form `type x`. In the present prototype of PieceWork (Section 6), types are not enforced, but we include them in the syntax because we expect to check them in the future. Functions in PieceWork are first-order (i.e., second-class) but potentially recursive, including mutual recursion. The type of a function is determined by its parameter(s) and output. We present rules for the single-argument case, but the implementation described in (Section 6) supports any fixed number of arguments. The typing rule for function definitions follows, using the judgement $\Gamma \vdash d : \Delta$ to indicate that a definition d checked in context Γ generates the new definition(s) Δ.

$$\textsc{DefFunc}\ \frac{\Gamma, (x : \tau_1), (f : \tau_1 \to \tau_2) \vdash e : \tau_2}{\Gamma \vdash f(\tau_1\ x)\{e\} : (f : \tau_1 \to \tau_2)}$$

4.3 Types

The types τ of PieceWork include as base types the real `numbers`, `shapes`, `points`, `edges`, and `Booleans`. In the written presentation, the `shape` type only includes simple shapes, but it is generalized in the implementation described

in (Section 6) to include both simple and complex shapes. The composite types are tuple types and function types. Tuple types $(\tau_1, \ldots, \tau_n)$ are fixed-length and heterogeneous. Function types $(\tau_1, \ldots, \tau_n) \rightarrow \tau$ have any fixed number of heterogeneous arguments and a single result type.

4.4 Operational Semantics

We present the operational semantics. In order to define the operational semantics, we need a formal representation of runtime states s. We define states by $s \equiv (E, C)$ where C is a `ComplexShape` and $E : \mathit{Var} \rightharpoonup \mathit{Value}$ is an *environment* which maps the names of (bound) variables to their current values. The complex shape s represents the entirely of the quilt project state, whose pieces may be assigned many different names within E. The main judgements of the operational semantics are $(s, e) \mapsto (s', v)$ to mean that executing expression e in state s produces state s' and value v as well as $(s, prog) \mapsto s'$ to mean that executing program statement $prog$ in state s produces state s'. We present semantics for core sewing operations: cutting, sewing, and marking. We also describe the operational semantics of critical language features such as variable assignment and function definition. However, we do not go into detail for other basic features common to other programming languages such as numbers and tuples in the interest of maintaining focus on the unique capabilities of PieceWork.

4.4.1 Marking

Marking is the most basic operation of our language, as it gives us the infrastructure to talk about specific locations within our fabrics. Intuitively, this can be interpreted as using chalk to mark a measured distance on a piece of fabric. In the easy case of marking a piece of fabric along an edge, we specify which edge we want to mark, and then provide the relative position where we want to place our mark. For example, if we want to be able to seam half way along an edge, which is required in many non-paper-pieceable quilt designs, we can use mark to indicate a stopping point for the seam like so: `mark(e, 0.5)`

When we use `mark()`, we are creating a new exterior edge and point on our shape. This gives us the ability to reference these new edges and points directly in other operations, and allows for more complex quilt designs. Figure 8 illustrates the result of marking an edge.

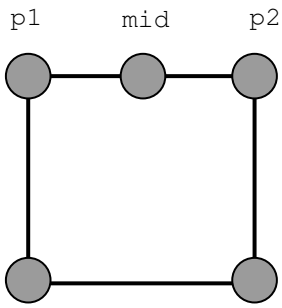

Figure 8: Visual result of `mark((p1, p2), 0.5)`

PieceWork places no restriction on which edges can be sewn together, but sewing a longer edge to a shorter edge in real life can create effects like ruffles or pleats where one edge is folded to be the same length as the other. This usually is not the desired effect in quilting, so the `mark` operation was introduced to allow seaming only part of the way along an existing edge, eliminating ruffle concerns and allowing for partial seaming techniques used in quilts that are not foundation paper pieceable [13].

$$\textsc{Mark}\ \frac{\begin{array}{c}((E, (\overrightarrow{S}, \sigma)), e) \mapsto^* ((E', C'), v_e) \\ ((E', C'), n) \mapsto^* ((E'', (\overrightarrow{S''}, \sigma'')), v_n)\end{array}}{\begin{array}{c}((E, ([sh, \overrightarrow{S}], \sigma)), mark(e, n)) \mapsto \\ ((E'', ([mrk(sh, v_e, v_n), \overrightarrow{S''}], \sigma'')), \\ v_e.beg + (v_n \times d))\end{array}}$$

where 1) $v_n \in [0, 1]$, 2) sh is some shape containing edge v_e, 3) $d = v_e.end - v_e.beg$, and 4) $v_e.beg + (v_n \times d)$ stands for a literal point value, not an addition term. The Mark rule relies on a helper algorithm for marking, depicted in Algorithm 2. Variable names within the algorithm are used to reference values within the algorithm, and are not added to the program state.

Algorithm 2 Creation of new edges using mark

```
mrk(([E1⃗; e, E2⃗], μ), e, r): Shape
    let d = ((e.end.x - e.beg.x)^2
             + (e.end.y - e.beg.y)^2)^1/2
    let mid = e.beg + (r * d)
    let e' = (e.beg, mid)
    let e'' = (mid, e.end)

    return ([E1⃗; e', e'', E2⃗], μ)
```

4.4.2 Sewing

In our language, sewing is represented by recording a renaming substitution of the two edges being sewn together. A visual depiction of the semantics of sewing is provided

in Figure 9. In order for a sew to be completed, we must provide the points in question, from which edges are identified and a substitution introduced that makes them equal. Then, we can create a `ComplexShape(`$\overrightarrow{S}, \sigma$`)`, containing the two shapes which were sewn and the substitution between the two edges. We consider sewn edges to be definitionally equal because they become one unified seam, and the reflexive path connects the two edges.

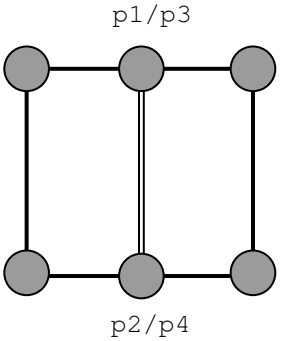

Figure 9: Visual result of `sew((p1, p2), (p3, p4))`

$$\textsc{Sew}\ \frac{\begin{array}{c}((E,(\overrightarrow{S},\sigma)),p_1)\mapsto(s_1,v_1)\\ (s_1,p_2)\mapsto(s_2,v_2)\qquad (s_2,p_3)\mapsto(s_3,v_3)\\ (s_3,p_4)\mapsto((E_4,(\overrightarrow{S}_4,\sigma_4)),v_4)\end{array}}{\begin{array}{c}((E,(\overrightarrow{S},\sigma)),sew((p_1,p_2),(p_3,p_4)))\mapsto\\ ((E_4,(\overrightarrow{S}_4,\sigma_{sew}\circ\sigma_4)),\\ ComplexShape([sh_1,sh_2],\sigma_{sew}))\end{array}}$$

where sh_1, sh_2 are the S_i and S_j that respectively contain the first and second sewn edge, where $\sigma_{sew} \equiv (E^{-1}(v_1) \mapsto E^{-1}(v_3), E^{-1}(v_2) \mapsto E^{-1}(v_4), E^{-1}(edge(v_1, v_2)) \mapsto E^{-1}(edge(v_3, v_4)))$, where E^{-1} is the inverse map of E and where $\circ$ is composition. The Sew rule assumes that the sewn points and edges all have unique names in the environment E, which are looked up using E^{-1} in order to construct a unifying substitution.

4.4.3 Cutting

The most complex operation in our language is cutting. Physically, cutting allows a quilter to take one shape, and divide into two along a specified path. Cutting also produces two new matching edges: one for each resulting shape.

We needed to create a mechanism to divide the edges of the original shape and create the necessary new edges, while ensuring that a cut is truly a single cut—cutting through empty space should not occur. However, these challenges also highlight the strengths of the topological approach.

First, a requirement of all cuts is that the points specified for the start and end of the cut must be propositionally equal. This means that we must be able to find a path from point A to point B through our shape, for all points specified in the usage. Due to our topological approach, we do not have to consider non-convex polygons when cutting, so our only restriction is that the two points are within the same `SimpleShape`.

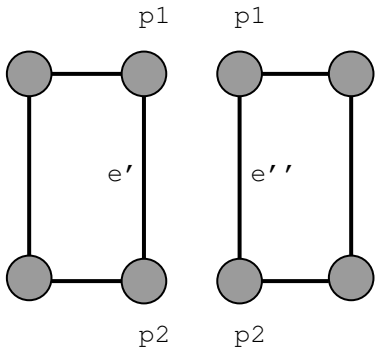

Figure 10: Visual result of `cut(p1, p2)`

Our design of cut in this work only applies to simple shapes. When cutting through complex shapes, smaller cuts through the sub-shapes can be chained together to create the desired effect. This is an intentional design decision, as cutting between two points on a complex shape directly could result in many different possible topologies, depending on which sub-shapes the cut does or does not pass through. Since we do not specify exact edge trajectories in PieceWork, restricting cutting to simple shapes ensures that the results are consistent and predictable for future end users. The cut rule relies on a helping algorithm for cutting, depicted in Algorithm 3.

$$\textsc{Cut}\ \frac{\begin{array}{c}((E,(\overrightarrow{S},\sigma)),e_1)\mapsto^*((E',(\overrightarrow{S'},\sigma')),v_1)\\ ((E',(\overrightarrow{S'},\sigma')),e_2)\mapsto^*((E'',(\overrightarrow{S''},\sigma'')),v_2)\end{array}}{\begin{array}{c}((E,([s,\overrightarrow{S}],\sigma)),cut(p_1,p_2))\mapsto\\ ((E'',([ct(sh,v_1,v_2),\overrightarrow{S''}],\sigma'')),ct(sh,v_1,v_2))\end{array}}$$

where sh is the shape S_i containing both v_1 and v_2.

Algorithm 3 Creation of new shapes using cut

```
ct(([E_a;e_1, E_b;e_2, E_c],μ), e_1
.end,e_2.end):(SimpleShape, SimpleShape)

    let e' = (e_1.end, e_2.end)
    let e'' = (e_2.end, e_1.end)
    let s_1 = ([E_a; e_1, e', E_c], μ)
    let s_2 = ([E_b; e_2, e''], μ)

    return [s_1, s_2]
```

We use the `end` of edges for the cutting algorithm. The beginning of edges could also have been used, as long as it is internally consistent.

4.5 The Left-Hand Side

The left hand side of assignments in PieceWork can either be a variable in the typical sense, or a wildcard denoted with an underscore. When a value is assigned to a wildcard, that value is not stored and the environment is not updated. For variable assignments, the environment is updated to include the new value of the variable.

$$\textsc{DefVar}\ \frac{*}{(s, x := v) \mapsto (s, x \mapsto v)}$$

$$\textsc{DefWildcard}\ \frac{*}{(s, \_ := v) \mapsto s}$$

We also support tuple assignment, which proceeds recursively. Despite the recursive definition, we consider the assignment to be a single small step of execution, because an entire assignment is a conceptualized as one step.

$$\textsc{DefTuple}\ \frac{\begin{array}{c}(lhs_1 := v_1, s) \mapsto s_1 \\ \vdots \\ (lhs_n := v_n, s_{n-1}) \mapsto s_n\end{array}}{(s, [lhs_1, ..., lhs_n] := (v_1, ..., v_n)) \mapsto s_n}$$

4.6 Functions

PieceWork's functions use a call-by-value evaluation strategy, by evaluating the arguments to values and then supplying those values to the body in an environment. Some other HoTT-related works use call-by-name evaluation, but it is not a requirement.

The basic semantics are depicted in the following rules; the implementation described in Section 6 generalizes these semantics to support recursion by maintaining nested scopes.

$$\textsc{OpFuncS}\ \frac{(s, e) \mapsto (s', e')}{(s, f(e)) \mapsto (s', f(e'))}$$

$$\textsc{OpFuncBeta}\ \frac{*}{((E, C), f(v)) \mapsto (((x \mapsto v), C), e)}$$

where, in OpFuncBeta, $f(x)$ is a function defined with a body e.

5. Theory

We now present theoretical results about PieceWork. Our main result characterizes the relation between sewing and cutting, proving them to be quasi-inverses to one another.

Fundamentally, when we sew two or more pieces of fabric together, our result is one larger piece that resembles a combination of those original pieces—this is the basis of almost everything we have done so far. Similarly, when we cut a piece of fabric into two or more pieces, those resulting pieces each resemble a portion of that original piece.

With our definitions of sewing and cutting in hand, we can consider what would happen if we were to cut a piece of fabric into two, and then attempt to sew them back together. This should result in a shape very similar to the original, albeit with a new internal seam.

Firstly, and perhaps somewhat obviously, we must be sewing the same edges that we cut. One way to think about cutting is that it essentially creates two new edges, one per new shape. When we reverse the cut using sew, we want to be sure we are sewing those two edges together. If we were to sew two other edges together instead, then those two new edges would still exist and the final shape may be totally different.

If the other edges of the shape are altered between the cut and sew operations, this will also result in the final shape being different from the initial shape. For instance: if we perform our cut, mark one of the other edges, and then sew together our cut edges, the resulting shape will have an extra edge produced by the mark operation. In order for sew to undo a cut, the other edges of the shape when the sew is performed must be the same as when the cut is performed.

Functions which invert each other under the following conditions are known as quasi-inverses.

Definition 5.1 (Quasi-Inverse)**.** For a function $f : A \to B$, a quasi-inverse of f is a triple (g, α, β) consisting of a function $g : B \to A$, and homotopies $\alpha : f \circ g \simeq id_B$ and $\beta : g \circ f \simeq id_A$ where $\circ$ is composition and id_A is the identity at type A.

In our case, we have f as cut and g as sew. Now, we have to find the specific homotopies α and β that satisfy the definition above as well as our constraints.

Theorem 5.1 (sew $\circ$ cut)**.** *There exists a homotopy such that sew $\circ$ cut $\simeq id_{Shape}$*

Theorem 5.2 (cut $\circ$ sew)**.** *There exists a homotopy such that cut $\circ$ sew $\simeq id_{Shape}$*

For each of these theorems, we start with an initial `ComplexShape` context Q. For *sew* $\circ$ *cut*, we cut one of the subshapes in Q, and sew between the newly created edges from the cut. Then, we apply substitute and find that we have the same original quilt state of Q.

Similarly, for *cut* $\circ$ *sew*, we start by sewing together two edges in subshapes of Q. Then, we apply substitute to create a `SimpleShape` which we can cut. Finally, we cut between the beginning and endpoints of the two edges that were sewn originally, which results in the original state of Q.

Full proofs of these theorems can be found in the first author's Master's thesis [8].

6. Implementation

We implemented a prototype of PieceWork, available at `https://github.com/rbohrer/PieceWork`. The prototype is implemented in Scala and consists of ≈850 lines of implementation and ≈500 lines of tests, of which the main tests are Sierpinksi triangles (Section 7.1) and Chaikin's corner-cutting algorithm. Scala was chosen due to the authors' familiarity with it; any strongly-typed functional language would be suitable.

In order to provide usable diagrams with minimal implementation cost, we implemented the prototype as a transpiler from PieceWork to Penrose [21], a domain-specific language for mathematical diagrams. A major benefit of Penrose is its builtin optimization engine which provides completely automatic layout. Penrose enforces a separation between style and substance; we reuse a builtin style for triangular meshes and translate each PieceWork file to a substance file which can then be rendered in Penrose. As a limitation of the builtin style file, our visualizations currently do not propagate color information from PieceWork, and use a solid color instead.

The translation from PieceWork to Penrose is as follows: i) the PieceWork program is parsed into an abstract syntax tree, ii) the program is interpreted according to a slight generalization of the operational semantics, the output of which is a state, iii) a heuristic selects which shapes from the state should be visualized, and iv) a Penrose program visualizing those shapes is emitted. The operational semantics is generalized in the following ways: i) Penrose requires every value to have a name, so we automatically assign names to unnamed values, ii) visualized values often include intermediate values which have gone out of scope, so we maintain a list of such values with auto-generated names, and iii) substitutions are applied to states eagerly since Penrose expects pre-substituted values. The choice of which shapes to visualize is driven by the following heuristic: if the state contains complex shapes, all complex shapes are shown; if not, all simple shapes are shown. This prevents duplicating shapes that have been stitched together, but leaves out some shapes in the case that some but not all pieces have been stitched together. Better selection of shapes for visualization is a potential topic for future work.

7. Examples

We present two examples: a Sierpinski triangle design and the block structure of the Bible quilt.

7.1 Sierpinski Triangles

PieceWork can be used to programmatically generate quilt designs. Here, we define a recursive function to generate the Sierpinski triangle, a well-known fractal.

```
divideTriangle(shape t, number i) {
    if (i > 0) {
        [te1, te2, te3] := t.edges;
        new1 := mark(te1, 0.5);
        new2 := mark(te2, 0.5);
        new3 := mark(te3, 0.5);

        [_, t1] := cut(new1, new2);
        [_, t2] := cut(new3, new2);
        [t4, t3] := cut(new3, new1);

        div1 := divideTrianlge(t1, i-1);
        div2 := divideTriangle(t2, i-1);
        div3 := divideTriangle(t3, i-1);
        div4 := divideTriangle(t4, i-1);

        return complexShape([div1, div2,
                             div3, div4], [])
    } else {
        return t;
    }
}

main() {
    p1 := point(0, 0);
    p2 := point(2, 0);
    p3 := point(1, 1.732);

    e1 := edge(p1, p2);
    e2 := edge(p2, p3);
    e3 := edge(p3, p1);

    red := material(1, 0, 0);
    tri := shape([e1, e2, e3], red);
    stri := divideTriangle(tri, 1);
}
```

The above program generates a once-divided Sierpinski triangle. Figure 11 is an example rendering of the resulting `ComplexShape` generated using the PieceWork prototype. While Sierpinski triangles are equilateral, the rendering has unequal side lengths due to the topological nature of PieceWork. Any geometric data in PieceWork are erased completely before being sent to Penrose for rendering.

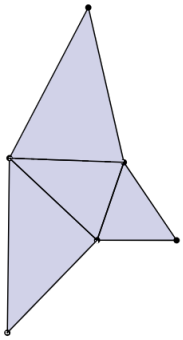

Figure 11: Sierpinski triangle generated using PieceWork.

7.2 Bible Quilt

Harriet Powers' Bible Quilt 1 is an example of a quilt which would be difficult to represent with existing computational models, as it is not foundation paper pieceable and has patches of different size and shape. In this section, we will model the patch structure of Bible Quilt using PieceWork.

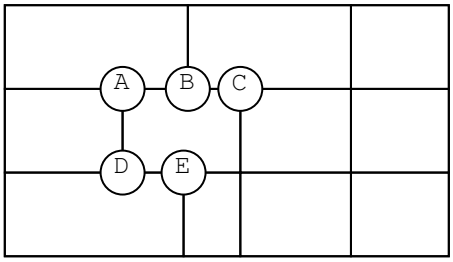

Figure 12: The patch structure of Bible Quilt, with five key intersections highlighted

To reduce the amount of boilerplate needed for this example, we will assume we have a helper function `quad()` to generate the eleven initial rectangular pieces. The variable names for each rectangle indicates its position in the final quilt: for example, `r1b1` is the leftmost piece in the top row, while `r3b4` is the rightmost piece in the bottom row.

In Figure 12, the labeled intersections are points where edges of blocks are only partially sewn. In our program, we mark those points to enable sewing to multiple edges. This capability is what sets PieceWork apart from other works, such as the dual-hypergraph model which cannot support Mondrian-style arrangements [13].

```
main() {
    [r1b1, r1b2, r1b3,
     r2b1, r2b2, r2b3, r2b4,
     r3b1, r3b2, r3b3, r3b4] := quad(11);

    mark(r1b1.edges[3], 0.6); // A
    mark(r2b2.edges[2], 0.5); // B
    mark(r1b2.edges[3], 0.3); // C
    mark(r3b1.edges[2], 0.6); // D
    mark(r2b2.edges[4], 0.5); // E

    // sew blocks into rows
    row1 := sew(r1b1.edges[2],
                r1b2.edges[0]);
    row1 := sew(row1.shapes[1].edges[2],
                r1b3.edges[0]);

    row2 := sew(r2b1.edges[2],
                r2b2.edges[0]);
    row2 := sew(row2.shapes[1].edges[2],
                r2b3.edges[0]);
    row2 := sew(row2.shapes[2].edges[2],
            r2b4.edges[0]);

    row3 := sew(r3b1.edges[2],
                r3b2.edges[0]);
    row3 := sew(row3.shapes[1].edges[2],
                r3b3.edges[0]);
    row3 := sew(row3.shapes[2].edges[2],
                r2b4.edges[0]);

    // sew rows together to form quilt

    // row 1 to row 2
    sew(row1.shapes[0].edges[4],
        row2.shapes[0].edges[1]);
    sew(row1.shapes[0].edges[3],
        row2.shapes[1].edges[1]);
    sew(row1.shapes[1].edges[4],
        row2.shapes[1].edges[2]);
    sew(row1.shapes[1].edges[3],
        row2.shapes[2].edges[1]);
    sew(row1.shapes[2].edges[4],
        row2.shapes[3].edges[1]);

    // row 2 to row 3
    sew(row2.shapes[0].edges[3],
        row3.shapes[0].edges[1]);
    sew(row2.shapes[1].edges[4],
        row3.shapes[0].edges[2]);
    sew(row2.shapes[1].edges[3],
        row3.shapes[1].edges[1]);
    sew(row2.shapes[2].edges[3],
        row3.shapes[2].edges[1]);
    sew(row2.shapes[3].edges[3],
        row3.shapes[3].edges[1]);
}
```

8. Conclusion and Future Work

This paper introduced PieceWork, a language for quilting based on homotopy type theory, under the interpretation that definitionally equal points are in contact, propositionally equivalent points are connected by fabric, and sewing is unification. Our homotopical foundations are motivated by the promise of flexibility, allowing us to represent diverse styles of quilts so long as we can represent their topological structure. We developed syntax and semantics for PieceWork, implemented a prototype, and rendered example programs such as a Sierpinski triangle.

The development of typing rules for the mark, sew, and cut operations remains future work. To demonstrate the fundamental challenge of developing these rules, we present initial ideas in Figure 13; these ideas should not be construed as sound typing rules. In this exploration, $\Gamma[x_1, y_1/x_2, y_2]$ is the simultaneous substitution of x_1 and y_1 for y_1 and y_2 respectively. Likewise $\Gamma[e_1 \backslash x_1; x_2][e_2 \backslash y_1; y_2]$ is an imagined notation for the effect of cutting fabric on the context, which

separates e_1 into two points x_1 and x_2, likewise for e_2. In these rules, we write propositional equality as $\simeq$ and definitional equality as $=$.

$$\textsc{Mark?}\ \frac{\Gamma \vdash e : x \simeq y \qquad \Gamma \vdash n : \mathbb{R}}{\Gamma, z : Point \vdash mark(e, n) : (x \simeq z, z \simeq y)}$$

$$\textsc{Sew?}\ \frac{\Gamma \vdash p : x_1 \simeq y_1 \qquad \Gamma \vdash q : x_2 \simeq y_2}{\Gamma[x_1, y_1/x_2, y_2], pq : p = q \vdash sew(p, q) : x_1 \simeq y_1}$$

$$\textsc{Cut?}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : Point \\ \Gamma \vdash e_2 : Point \qquad \Gamma \vdash e_3 : e_1 \simeq e_2 \text{ (some } e_3\text{)}\end{array}}{\Gamma[e_1 \backslash x_1; x_2][e_2 \backslash y_1; y_2] \vdash cut(e_1, e_2) : (x_1 \simeq y_1, x_2 \simeq y_2)}$$

Figure 13: Initial ideas for core typing rules

At first, we seem successful in describing the meaning of each operation. Marking takes two points connected by a path and introduces a new point z through which the connection travels. Sewing takes two paths p, q and induces equalities between them by substitution. Cutting takes points e_1, e_2, infers they are connected by finding a path e_3, then separates the points.

On closer inspection, however, one fails to interpret the proposed rules as an algorithm to typecheck a PieceWork program in a given context Γ. In each rule, the context of the conclusion is more complex than the premise, sometimes substantially and uninvertibly so. This complexity makes it more natural to read the context as an *output* rather than *input*: one can compute a context in which a PieceWork program is well-typed, and this computation is roughly as complex as execution.

This insight suggests that the most promising path for realizing the proposed rules is to develop a logic programming language along the lines of PieceWork, in which execution is naturally conceived as solving for a context in which a proof term is well-typed. Building toward a logic programming language would clarify other potential surprises in PieceWork, namely the treatment of mutation and the threading of the state through the semantics. These features align with formalized virtual machines for logic programming languages [4], where terms exhibit sharing with limited mutation via unification.

A move toward logic programming would not invalidate the development of PieceWork. On the contrary, any logic programming successor would likely compile to PieceWork as an implementation-level target language. The semantics of PieceWork presented here could be used to prove the correctness of such a translation. Such a language would potentially build on recent approaches for HoTT-friendly unification [10].

In such future work, we also hope to generalize the sewing operation so that it works when paths between given points are not unique or when values are not given names. A stretch goal for a type system is to use topological compatibility ("sewability") as a typing abstraction by building on related work in parametricity for HoTT [7].

In the long term, we are interested in exploring the use of PieceWork or some variant thereof for fabrication and procedural generation topics beyond quilting, including three-dimensional objects. As the topological interpretation of computational craft matures, we hope to incorporate this interpretation into community outreach activities to support sense of belonging among underrepresented students who craft in computer science.

Acknowledgements. Special thanks go to Gillian Smith for the discussions that inspired this project and Karen Royer for sharing her quilting expertise in discussions.

This material is based upon work supported by the National Science Foundation under Grant No. 2244839.

References

[1] ALBAUGH, L., GROW, A., LIU, C., MCCANN, J., SMITH, G., AND MANKOFF, J. Threadsteading: Playful interaction for textile fabrication devices. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems* (New York, NY, USA, 2016), CHI EA '16, Association for Computing Machinery, p. 285–288.

[2] ANGIULI, C., MOREHOUSE, E., LICATA, D. R., AND HARPER, R. Homotopical patch theory. *ACM SIGPLAN Notices 49*, 9 (Aug 2014), 243–256.

[3] ANNENKOV, D., CAPRIOTTI, P., AND KRAUS, N. Two-level type theory and applications. *CoRR abs/1705.03307* (2017).

[4] BOHRER, R., AND CRARY, K. TWAM: A certifying abstract machine for logic programs. In *Verified Software. Theories, Tools, and Experiments - 10th International Conference, VSTTE 2018, Oxford, UK, July 18-19, 2018, Revised Selected Papers* (2018), R. Piskac and P. Rümmer, Eds., vol. 11294 of *Lecture Notes in Computer Science*, Springer, pp. 112–134.

[5] BRACKMAN, B. *Encyclopedia of pieced quilt patterns*. American Quilter's Society, 1993.

[6] BUECHLEY, L., EISENBERG, M., CATCHEN, J., AND CROCKETT, A. The LilyPad Arduino: Using computational textiles to investigate engagement, aesthetics, and diversity in computer science education. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2008), CHI '08, Association for Computing Machinery, p. 423–432.

[7] CAVALLO, E. *Higher inductive types and internal parametricity for cubical type theory*. PhD thesis, Carnegie Mellon University, 2021.

[8] CLARK, C. Homotopy type theory for sewn quilts. Tech. rep., Worcester Polytechnic Institute, 2023. Master's Thesis (In Press).

[9] COAHRAN, M., AND FIUME, E. Sketch-Based Design for Bargello Quilts. In *Eurographics Workshop on Sketch-Based Interfaces and Modeling* (2005), J. A. P. Jorge and T. Igarashi, Eds., The Eurographics Association.

[10] COCKX, J., AND DEVRIESE, D. Proof-relevant unification: Dependent pattern matching with only the axioms of your type theory. *Journal of Functional Programming 28* (2018).

[11] GRAVES, J., ROYER, K., SMITH, G., AND SULLIVAN, A. Procedural patchwork: Community-focused generative design for quilting. *Creativity and Cognition* (Jun 2021).

[12] KAPULKIN, C. The HoTTest summer school. `https://www.uwo.ca/math/faculty/kapulkin/seminars/hottest_summer_school_2022.html`, 2022. Accessed: April 18, 2023.

[13] LEAKE, M., BERNSTEIN, G., DAVIS, A., AND AGRAWALA, M. A mathematical foundation for foundation paper pieceable quilts. *ACM Transactions on Graphics 40*, 4 (2021), 1–14.

[14] LEAKE, M., LAI, F., GROSSMAN, T., WIGDOR, D., AND LAFRENIERE, B. PatchProv: Supporting improvisational design practices for modern quilting. *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (2021).

[15] LI, Y., BREEN, D. E., MCCANN, J., AND HODGINS, J. Algorithmic quilting pattern generation for pieced quilts. In *Proceedings of Graphics Interface 2019* (2019), GI 2019, Canadian Information Processing Society.

[16] LIU, C., HODGINS, J., AND MCCANN, J. Whole-cloth quilting patterns from photographs. *Proceedings of the Symposium on Non-Photorealistic Animation and Rendering - NPAR '17* (Jul 2017).

[17] MIRECKI, V., SPITAELS, J., ROYER, K., GRAVES, J., SULLIVAN, A., AND SMITH, G. "my brain does not function that way": Comparing quilters' perceptions and motivations towards computing and quilting. *Designing Interactive Systems Conference* (Jun 2022).

[18] POWERS, H. Bible quilt. `https://www.mfa.org/exhibition/fabric-of-a-nation`, 2021. Accessed: April 18, 2023.

[19] PROGRAM, U. F. *Homotopy type theory: Univalent foundations of mathematics*. The Univalent Foundations Program, Institute for Advanced Study, 2013.

[20] SULLIVAN, A., MCCOY, J. A., HENDRICKS, S., AND WILLIAMS, B. Loominary: Crafting tangible artifacts from player narrative. In *Proceedings of the Twelfth International Conference on Tangible, Embedded, and Embodied Interaction* (New York, NY, USA, 2018), TEI '18, Association for Computing Machinery, p. 443–450.

[21] YE, K., NI, W., KRIEGER, M., MA'AYAN, D., WISE, J., ALDRICH, J., SUNSHINE, J., AND CRANE, K. Penrose: from mathematical notation to beautiful diagrams. *ACM Trans. Graph. 39*, 4 (2020), 144.

[22] ZELLERBACH, K., AND ROBERTS, C. Barbara: live coding language for quilters - about. `https://www.barbara.graphics/about`, 2021. Accessed: April 18, 2023.