Polynomial-Time Program Equivalence for Machine Knitting

NATHAN HURTIG, University of Washington, USA
JENNY HAN LIN, University of Utah, USA
THOMAS S. PRICE, Carnegie Mellon University, USA
ADRIANA SCHULZ, University of Washington, USA
JAMES MCCANN, Carnegie Mellon University, USA
GILBERT LOUIS BERNSTEIN, University of Washington, USA

We present an algorithm that canonicalizes the algebraic representations of the topological semantics of machine knitting programs. Machine knitting is a staple technology of modern textile production where hundreds of mechanical needles are manipulated to form yarn into interlocking loop structures. Our semantics are defined using a variant of a monoidal category, and they closely correspond to string diagrams. We formulate our canonicalization as an Abstract Rewriting System (ARS) over words in our category, and prove that our algorithm is correct and runs in polynomial time.

CCS Concepts: • **Software and its engineering** → Domain specific languages; *Visual languages*; *Semantics*; • **Theory of computation** → **Categorical semantics**; **Program analysis**; • **Mathematics of computing** → Topology.

Additional Key Words and Phrases: machine knitting, string diagrams, program equivalence, canonicalization, normal forms, monoidal categories, fenced tangles, braids.

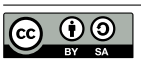
ACM Reference Format:
Nathan Hurtig, Jenny Han Lin, Thomas S. Price, Adriana Schulz, James McCann, and Gilbert Louis Bernstein. 2025. Polynomial-Time Program Equivalence for Machine Knitting. *Proc. ACM Program. Lang.* 9, ICFP, Article 248 (August 2025), 29 pages. https://doi.org/10.1145/3747517

1 Introduction

Machine knitting is a powerful textile fabrication technique. Computer-controlled knitting machines can form yarns into complex 3D textile shapes with diverse surface textures, thicknesses, colors, and mechanical properties – all under programmatic control. These machines are widely deployed and account for 20–40% of existing textile artifact production. This includes everyday garments like socks, gloves, and sweaters; medical devices like compression socks and respiration sensors; and more surprising applications like composite preforms for wind turbine blades, automobile seat covers, and formwork for cast concrete architecture. However, despite the widespread use of knitting machines, their full potential is not currently exploited because knitting machine programming pipelines are underdeveloped.

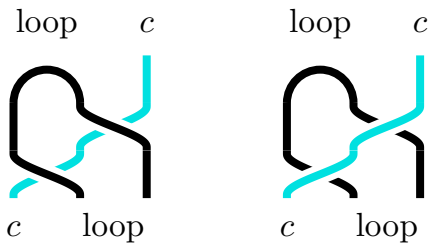
Knitting machines are controlled by low-level programs that specify machine actions. Knit programmers must explicitly translate their desired knitted object into a series of steps that the

Authors' Contact Information: Nathan Hurtig, University of Washington, Seattle, USA, hurtig@cs.washington.edu; Jenny Han Lin, University of Utah, Salt Lake City, USA, jenny.h.lin@utah.edu; Thomas S. Price, Carnegie Mellon University, Pittsburgh, USA, tomprice@andrew.cmu.edu; Adriana Schulz, University of Washington, Seattle, USA, adriana@cs.washington.edu; James McCann, Carnegie Mellon University, Pittsburgh, USA, jmccann@cs.cmu.edu; Gilbert Louis Bernstein, University of Washington, Seattle, USA, gilbo@cs.washington.edu.



ACM 2475-1421/2025/8-ART248
https://doi.org/10.1145/3747517

machine can perform to make it, and then express these steps in an appropriate language – often using proprietary, machine-specific design tools [14, 23, 25].[1] The knitout language [16] is a cross-machine low-level language used by some researchers and hobbyists which provides a more machine-neutral view of knit programming. Knitout does not include control flow operations, and each of its operations refers to some movement of a piece of the knitting machine: it is analogous to G-code in 3D printers, or assembly in traditional computing. However, all low-level knit programming – regardless of the tool – is error-prone, as it is difficult for programmers to correctly express the final topology and shape of their intended object using low-level machine code. The runtime efficiency of machine knitting and robustness to mechanical errors also suffer.



(a) Loop on f.2 (b) Loop on b.2

Fig. 1. The result of the knitout instruction `miss + f.2 c` depends on the machine state. If *c* crosses a loop, the resulting topology depends on whether that loop is held on a front needle (a) or back needle (b).

For instance, Figure 1 illustrates a key subtlety in writing low-level machine knitting code. The knitout operation `miss + f.2 c` instructs the machine to move yarn carrier *c* to a position just right of needle `f.2`, effectively leaving a trail of yarn between the carrier's current position and the new location. The yarn structure created by that operation depends on the state of loops of yarn held on the machine, which is in turn determined by earlier code. This example shows how machine knitting differs from traditional computing contexts: as strands and loops (analogous to values) move between needles (analogous to registers), *how* they move relative to other in-flight values is as important as *where* they arrive. This requires programmers to maintain a virtual model of the machine's state as they program, a task similar to requiring traditional programmers to maintain a virtual model of register assignments.

Recently, formal semantics were proposed for knitout [12] as a way to alleviate some of these concerns. Denotational semantics allow the communication and analysis of a knitting program's *result* rather than the specific *actions* the program specifies that a knitting machine should take to get there. When a creator designs a 3D printed object, they rarely write G-code to control the 3D printer directly. Instead, creators use Computer-Aided Design (CAD) tools to specify 3D objects in a fabrication-oblivious format. Then, Computer-Aided Manufacturing (CAM) tools, like slicers, compile the design to G-code. Machine knitting semantics enable analogous design tools. They also enable verification of compilers and optimization tools.

The semantics proposed for knitout by Lin et al. [12] are unique. In traditional programming languages, programs represent computations. In a simplified sense, they are functions that map inputs to outputs. In contrast, knitout programs represent knitted objects – topologically complex structures made from braiding (crossing strands over strands) and stitches (passing loops through loops). The proposed knitout semantics are therefore topological: they translate knitout code to tangle diagrams, considered up to equivalence by ambient isotopy[2]. Lin et al. [12] prove some basic rewrites of knitout programs preserve semantic equivalence. However, their described equations are not complete: they do not capture all topological equivalences implied by their semantics.

[1]These tools also offer high-level templates and wizards to create common shapes, but these templates support only a small portion of the expressiveness of the machines.
[2]Ambient isotopies are a technical formulation of continuous deformations of knots, without allowing one to break or cut strands, nor pass strands through other strands.

In this paper we answer affirmatively the question, "is it possible to efficiently decide the equivalence of two knitting machine programs?" by presenting a new *polynomial-time canonicalization algorithm* in a suitable semantic domain. This result is somewhat surprising, given known results in algorithmic knot theory. For instance, the unknotting problem (detecting whether a diagram is knotted or not) is known to be in both NP [7] and co-NP [11], but has no known polynomial-time algorithm. Thus, it is one notable candidate for an NP-intermediate problem. On the other hand, equivalence of braids (via canonicalization) can be computed in $O(b^2 n \log(n))$ [4] for braids with b crossings on n strands. We achieve a polynomial-time result by leveraging structure in machine knitted objects that is not present in general knot theory.

We present an algebraic form of Lin et al. [12]'s semantics for knitout using a suitable monoidal category, similar to Joyal and Street [10]. We then formalize our algorithm as an Abstract Rewriting System (ARS) over words (well-formed sequences of generating morphisms) in this category. We show that this canonicalization is both sound and complete. Because our algorithm is theoretically efficient and based on canonicalization, we anticipate that it will enable other efficient and complete analyses for knitting machine programs, inform the design of intermediate representations, and directly support equivalence checking for translation validation, debugging of compilers, and novel support in design tools.

Similar to efficient algorithms for braids, our algorithm exploits the fact that machine-knitted objects are topologically *progressive* (sometimes called *monotonic*): strands always flow from past stitches to future stitches, and do not turn back to braid with previous structures. This mirrors the observation of Lin et al. [13] that machine-knittable structures are always – in their terminology – "upward."

We identify two contributions of our paper:

- We develop a topologically-inspired algebraic semantics for machine knitting programs in Section 3. We show how to translate knitout code to those semantics in Section 8.
- We describe a polynomial-time canonicalization algorithm of those semantics in Sections 4, 5, and 6. We provide an overview of why the algorithm is correct in Section 7, leaving formal treatment to our supplementary material.

2 Overview

The goal of this paper is to efficiently check whether two machine knitting programs will create the same knitted object. We accomplish this by mapping knitting programs to diagrams like the one shown in Figure 3d, and then checking whether the diagrams are equivalent. More specifically, we present a canonicalization algorithm that rewrites two diagrams into the same form if and only if they are equivalent. Prior work by Joyal and Street [10] implies the connection between our diagrams, category theory, and the topology of yarns. While the details of machine knitting are necessary to explain our mapping, the core canonicalization algorithm can be appreciated with minimal knitting knowledge. We leave a more detailed definition of the knitout language to Section 8 and instead begin by

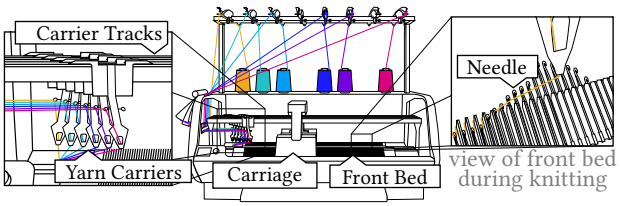
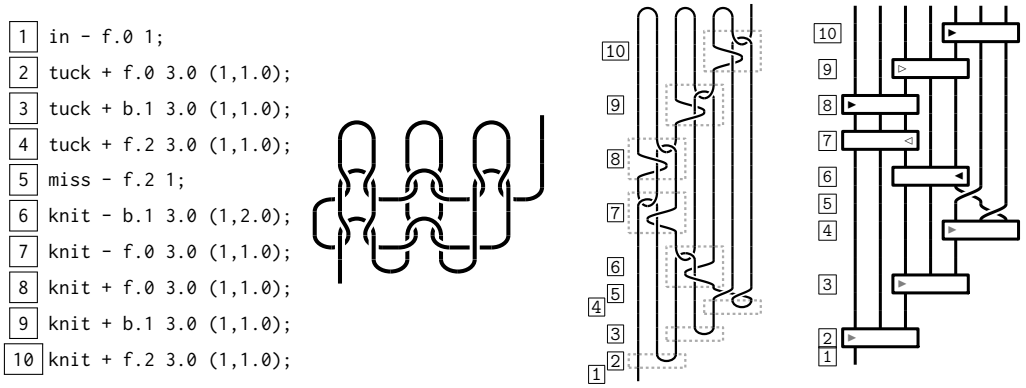


Fig. 2. A knitting machine forms a fabric from loops of yarn. The loops are formed and held by needles arranged in two parallel beds. Yarn is brought to the needles by yarn carriers. Figure from Lin et al. 2023 [12], based on Sanchez et al. 2023 [21], used with permission.

providing the high-level intuition required to understand the link between machine knitting and algebraic string diagrams. For a more in-depth explanation of machine knitting and its connection to mathematics, see Lin et al. [12].



(a) Knitout code (b) Output knit object (c) Sequential orientation (d) Algebraic diagram

Fig. 3. From left to right: a piece of formal knitout code, the knitted object it represents, that same object organized by execution order, and a diagram representing our algebraic semantics of the code. Figures 3c, 3d are labeled with the line numbers from Figure 3a.

Knitout is a machine knitting language that describes the actions performed by a v-bed knitting machine (Figure 2). These machines use hook-shaped *needles* arranged in rows called *beds* to manipulate yarn delivered by *carriers* into interlocking loops. In this paper we use formal knitout, a variant of knitout.[3] The formal knitout operations are described in Table 1.

To give a more in-depth intuition for formal knitout, we describe the execution of the snippet of formal knitout code presented in Figure 3a. It begins by bringing `in` yarn carrier `1` to the left of needle `0` in the front bed (i.e. `- f.0`). From there, three `tuck` operations use carrier `1` to drape loops over needles `f.0`, `b.1`, and `f.2`. After repositioning the carrier with a `miss` operation, five `knit` operations are used to pull yarn `1` through existing loops on corresponding needles. Executing this snippet on an empty machine produces the object shown in Figure 3b.

While the result of this relatively simple program was drawn manually, the topology for larger programs quickly grows in complexity. Prior work by Lin et al. [12] provides a denotational semantics mapping knitout to topological diagrams (Figure 3c). In addition to the yarn topology, these diagrams introduce fenced regions (drawn as dashed boxes) that surround loop entanglements and prevent uncontrolled unraveling. This means that salient transformations and equalities between machine knitting programs happen *outside* the dashed boxes. The yarn topology in Figure 3c is a vertically stretched version of Figure 3b, where the diagram can be partitioned into slices with at most a single box per slice. Each of the eight dashed boxes from bottom to top in the diagram correspond exactly with the eight `tuck` and `knit` operations from top to bottom in the program.[4]

The contents of each box come from a countable set of topologies determined by the corresponding operation; for example, see how the internals of the first three boxes (labeled 2, 3, 4) corresponding to the `tuck` operations are identical, and how the fourth box (6) can be rotated

[3]Formal knitout is easier to reason about and manipulate, but more syntactically verbose than knitout. There is a translator from knitout to formal knitout [12], so our results extend to knitout.

[4]The crossings between the third and fourth dashed boxes from the bottom correspond to the `miss` operation.

Table 1. The nine formal knitout operations, excluding nop. Parameters l and s control metric properties (e.g., length) and are irrelevant to topology.

Operation	Parameters	Description
`in`	*dir* $n.x$ y	Activates yarn carrier y and moves it to the *dir* side of needle $n.x$.
`out`	*dir* $n.x$ y	Deactivates yarn carrier y by cutting the yarn at the *dir* side of needle $n.x$.
`miss`	*dir* $n.x$ y	Moves yarn carrier y to the *dir* side of needle $n.x$.
`tuck`	*dir* $n.x$ l (y, s)	Drapes a loop formed in direction *dir* over needle $n.x$ using the yarn from carrier y.
`knit`	*dir* $n.x$ l *yarns*	Forms new loops in direction *dir* by pulling the yarns from carrier set *yarns* through the existing loops on needle $n.x$. The existing loops are dropped off the needle.
`split`	*dir* $n.x$ $n'.x'$ l *yarns*	Forms new loops in direction *dir* by pulling the yarns from carrier set *yarns* through the existing loops on needle $n.x$. The existing loops are moved to needle $n'.x'$.
`drop`	$n.x$	Drops all loops on $n.x$.
`rack`	r	Aligns front and back bed such that `f`$.x$ is across from `b`$.(x - r)$.
`xfer`	$n.x$ $n'.x'$	Moves loops from needle $n.x$ to needle $n'.x'$.

to produce the sixth box (8). We further simplify these diagrams by abstracting away the exact topology inside boxes to produce Figure 3d, again with eight boxes corresponding to the `tuck` and `knit` operations in order of execution. We preserve topological information of the internals of boxes with annotations. We use triangles such as ▶ in our diagrams, where the direction of the triangle encodes the direction of carrier yarn movement, and the fill categorizes its symmetry: gray triangles ▶ are used for rotationally symmetric (antichiral) `tuck` boxes, while filled ▶ and unfilled ▷ triangles represent boxes of different chiralities from `knit` and `split`.

Using the link between algebraic string diagrams and category theory, we present an algorithm that canonicalizes our knit diagrams in polynomial time. This in turn means we can decide equivalence for knitout programs in polynomial time. Figure 4 shows an example of our algorithm in action.[5] It flips boxes over, drags yarns around them, and swaps the vertical order of boxes to achieve a canonical form. Our algorithm runs in three stages:

(1) Layer (Section 5) runs from top to bottom, rotating and horizontally shifting each box to cancel out specific yarn crossings above the box.
(2) Swap (Section 6) swaps the boxes' vertical orders to achieve a canonical order. This is analogous to reordering operations in the knitout program.
(3) Braid (Section 4) simplifies the yarn crossings between boxes.

3 Semantic Domain

In this section, we define the categorical semantics that represent knitout. Our semantics are inspired by Joyal and Street [10], who formalized the connections between the topology of diagrams and certain monoidal categories that we use. Readers interested in a more complete description of the connections between monoidal categories and string diagrams should see Selinger [22].

[5]Machine knitting can use multiple colors of yarn in an object to achieve colored designs. In this paper, we color yarns solely as a visual aid; the colors of yarns in diagrams do not correspond with the true colors in physical objects.

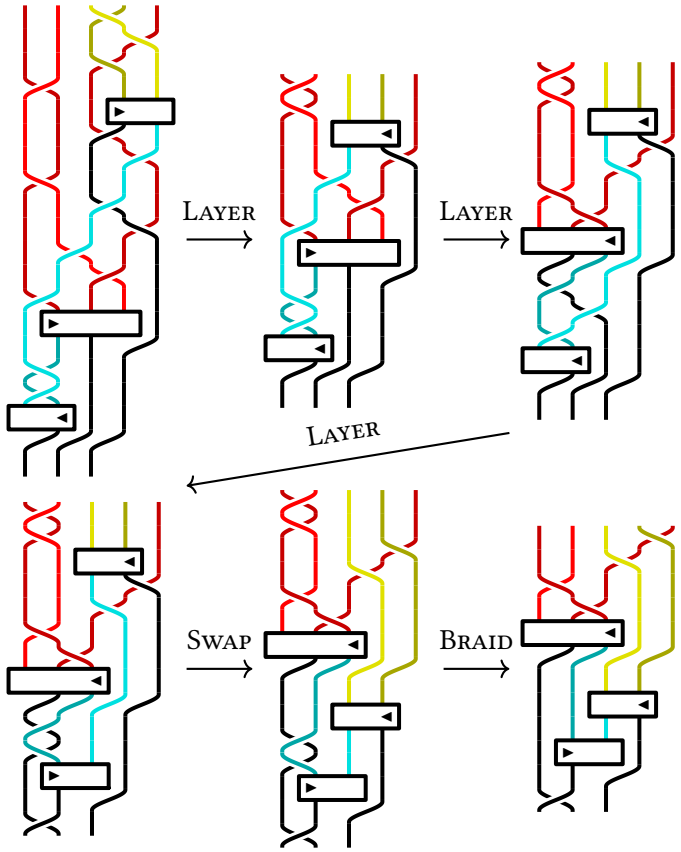
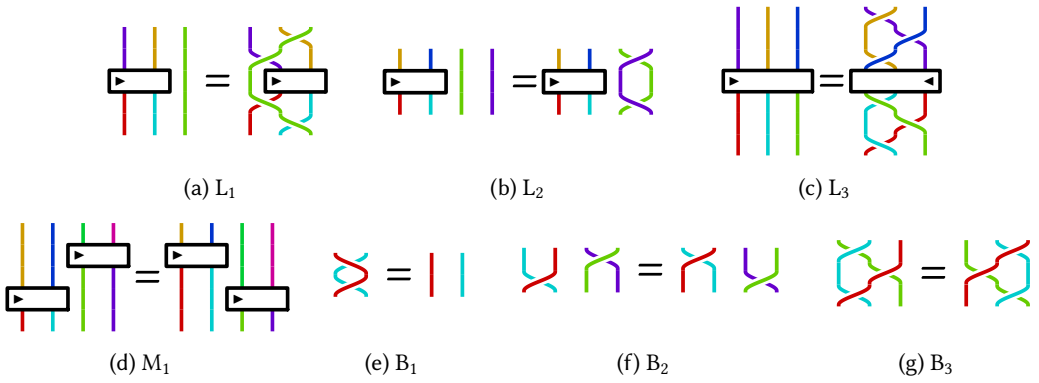


Fig. 4. An example of our canonicalization algorithm. It runs in three stages: Layer, Swap, and Braid.

Figure 5 summarizes the diagram rewrites that our algorithm canonicalizes.[6] All of these diagram rewrites represent physical continuous deformations in 3D space. We read and write each of our diagrams from bottom to top. Each of these diagram rewrites corresponds to some axiom or property of our final algebraic representation. Rules B_1, B_2, and B_3 are analogous to the presentation of the braid group. Rules L_1, L_3, B_1, and B_3 are analogous to the four rewrites presented in Lin et al. [12]. Because we impose a grid-like ordering on boxes and strands, we include rules L_2, M_1 and B_2 beyond the four of Lin et al. to capture unrelated subdiagrams shifting past each other.

We organize this section starting with the basics of categories, building up algebraic assumptions and structure. With each new assumption, we list the diagram rewrites that the assumption implies, and define how we draw diagrams for the new structure. Subsection 3.1 reviews the theory and definitions from Joyal and Street [10], which is sufficient for all the diagram rewrites in Figure 5 save L_3. Subsection 3.2 defines the twisted involutive monoidal category [5], giving us L_3. Finally, Subsection 3.3 details the specific category $\mathcal{K}$ whose words we canonicalize.

[6]Some readers may find our presentation of L_1, L_2, and L_3 awkward, especially L_2. Indeed, instead of braids above and below the box canceling in L_2, we could have defined L_2 to be a braid moving past an unrelated box, similar to B_2 moving unrelated braids past each other. We have chosen to define L_1, L_2, and L_3 as introducing "opposite" crossings above and below a box to align with our algorithm and proof.



(a) L_1 (b) L_2 (c) L_3

(d) M_1 (e) B_1 (f) B_2 (g) B_3

Fig. 5. Diagrammatic equivalence rules. The L_1, L_2, and L_3 moves correspond to our first step Layer, M_1 corresponds to Swap, and B_1, B_2, B_3 correspond to Braid. Our canonicalization algorithm rewrites two words X_1, X_2 to a shared canonical form if and only if they can be rewritten into each other with these seven moves.

3.1 Prior Definitions

Categories. A category C has a collection of objects C_0 (which we denote with uppercase letters) and a collection of morphisms C_1 (denoted with lowercase letters and Greek letters). Every morphism $x \in C_1$ has a domain $A \in C_0$ and codomain $B \in C_0$, and is denoted $x : A \to B$. In our diagram algebra, we draw arbitrary morphisms as boxes, and objects as strands connecting them, with the domain object flowing in from below the box and the codomain flowing out above the box. For every object in $A \in C_0$, there is an identity morphism $\mathtt{id}_A : A \to A$ with the expected properties of an identity; we draw identity morphisms as straight strands. Morphisms are composed with the ; operator to form new morphisms, but their objects must align: for any $x : A \to B$ and $y : B \to C$, $x; y : A \to C$ is a morphism in C_1 as well. Note that the composition order of ; is the opposite of the composition operator $\circ$.[7] In our diagram algebra, we interpret $x; y$ as vertical concatenation of x below y, connecting x's output strands to y's inputs. When composing many morphisms with product notation, we also use ;'s diagrammatic order: $\prod_{i=1}^{3} f_i := f_1; f_2; f_3$.

Categories additionally have commutative diagrams: statements of equivalence between specific sequences of morphisms. In this paper, we encode the commutative diagrams of our category as the diagrammatic equivalences in Figure 5. Our main contribution is a solution to the word problem: given two sequences of well-formed morphisms called words (represented by diagrams), can we rewrite one word into the other using the commutative diagrams?

For categories with only one object, composition is defined between all morphisms. These one-object categories are called *monoids*. Morphisms are not necessarily invertible; when they are, we call them isomorphisms. A *group* is a monoid in which all morphisms are isomorphisms. We can generalize this notion to a *groupoid*, relaxing the single-object requirement for groups and allowing any number of objects in the category. Equivalently, a groupoid could be defined as a category with the restriction that all morphisms are isomorphisms. A *functor* is a structure-preserving map between categories that maps objects to objects, morphisms to morphisms, and respects composition and commutative diagrams. Functors are analogous to group homomorphisms.

[7] We use ; to align with the order of composition in the Artin braid group [18] and programming languages such as knitout.

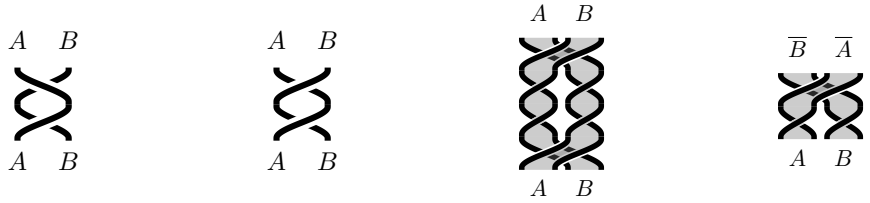
Monoidal Categories. A strict[8] *monoidal category* C is a category equipped with an operator $\otimes : C \times C \to C$, which we interpret as horizontal concatenation of diagrams. For objects $A, B \in C_0$, $A \otimes B$ is drawn as the objects A, B next to each other and is itself an object in C. The objects of a strict monoidal category form a monoid, from which the monoidal category draws its name. Similarly, for morphisms $x : A \to B$ and $y : C \to D$, $x \otimes y : A \otimes C \to B \otimes D$ is a morphism in C, drawn as the diagram for x to the left of the diagram for y.

In every monoidal category, there is a commutative diagram for morphisms a, b, c, d such that

$$(a \otimes b); (c \otimes d) = (a; c) \otimes (b; d)$$

whenever compositions are well-defined.[9] Together with the properties of `id` morphisms, this allows unrelated morphisms to shift past each other as in diagrams L_2, M_1, and B_2 in Figure 5.

Braided Monoidal Categories. A strict *braided monoidal category* C is a strict monoidal category with isomorphisms $\sigma_{A,B} : A \otimes B \to B \otimes A$ for every pair of objects. We draw $\sigma_{A,B}$ as A crossing over B from the left. Because $\sigma_{A,B}$ is an isomorphism, it is invertible, and we denote its inverse as $\sigma^{-1}_{A,B}$, and draw it as B crossing over A from the right – see Figure 6a. Note that unlike the case of symmetric monoidal categories, $\sigma_{B,A}$ is not necessarily the inverse of $\sigma_{A,B}$, as their composition represents a full rotation of A around B – even though the objects end up in the same spots they started, the crossings do not cancel out, as in Figure 6b. There are additional conditions imposed on $\sigma_{A,B}$, but we omit them for brevity. In summary, it must obey the diagrams B_3 and L_1 in Figure 5; B_1 is supplied by $\sigma_{A,B}$'s invertibility.



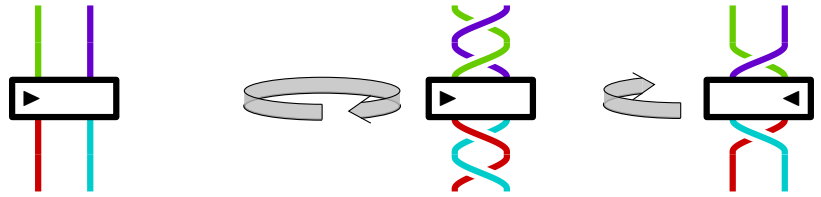
(a) $\sigma_{A,B}; \sigma^{-1}_{A,B} = \mathtt{id}_{A \otimes B}$ (b) $\sigma_{A,B}; \sigma_{B,A}$ (c) $\theta_{A \otimes B}$ (d) $\Delta_{A \otimes B}$

Fig. 6. Various morphisms in braided (6a, 6b), balanced (6c), and twisted involutive monoidal categories (6d).

Balanced Monoidal Categories. A strict *balanced monoidal category* C is a strict braided monoidal category equipped with some balance isomorphism $\theta_A : A \to A$ that performs a 360° rotation on some object A – see Figure 6c. Typically, balanced monoidal categories are drawn with framed strands (ribbons) so rotations can be seen. In knitting, rotating a yarn has no discernible effect in a knitted object, so we do not draw strands as framed in our later diagrams. The balance must respect the braiding σ so that the balance on some composite object $A \otimes B$ is the same as braiding A over B, the balance on both A and B independently, and then braiding B over A. Equationally, $\theta_{A \otimes B} = \sigma_{A,B}; (\theta_B \otimes \theta_A); \sigma_{B,A}$. Additionally, the balance must be a natural isomorphism. This means it can "commute with" morphisms, so for any $x : A \to B$, we have $\theta^{-1}_A; x; \theta_B = x$. This property implies that the diagrams in Figures 7a and 7b are equivalent: one can be rewritten into the other via commutative diagrams. However, a balance is not enough for L_3 – a balance is a 360° rotation, and L_3 encodes rotations of 180°. Our next subsection introduces a *twist* morphism, visualized in Figure 7c.

[8]In this paper, we only use *strict* categories for ease in definitions. Non-strict variants have additional structure, but coherence theorems state that every non-strict category is equivalent to a strict one.

[9]Note that in our diagram algebra, we draw both terms in $(a \otimes b); (c \otimes d) = (a; c) \otimes (b; d)$ identically – the algebraic representations describe the same diagram in row-major order and column-major order, respectively.



(a) Identity (b) Balance (c) Twist

Fig. 7. Various rotations on a box. Figure 7a shows a box that has not been rotated, Figure 7b shows the same box after a balance, and Figure 7c shows the original box after a twist. Arrows denote the direction and extent of rotation for each twist. Joyal and Street [10]'s balanced monoidal categories provide a 360° rotation, but not 180°.

3.2 Twisted Involutive Monoidal Categories

The axioms of braided monoidal categories imply all of the rewrites in Figure 5 save L_3. Balanced monoidal categories only allow 360° rotations, so their axioms cannot imply L_3. One reason why the axioms of balanced categories cannot guarantee half twists is because if $x : A \otimes B \to C \otimes D$ is a morphism, there is no guarantee there is some $x' : B \otimes A \to D \otimes C$ that represents its "flipped" counterpart.

This idea of flipping can be represented by an involutive functor $\overline{(\cdot)} : C \to C$ that reverses the order of object composition: $\overline{A \otimes B} = \overline{B} \otimes \overline{A}$. The functor $\overline{(\cdot)}$ is called involutive because it is its own inverse, on both objects and morphisms: $\overline{\overline{A}} = A$ and $\overline{\overline{x : A \to B}} = x : A \to B$. In our diagram algebra, we represent $\overline{(\cdot)}$ as rotating a diagram 180°.

A strict *twisted involutive monoidal category* [5] is a monoidal category equipped with some involutive functor $\overline{(\cdot)}$ and a natural isomorphism $\Delta_A : A \to \overline{A}$, called a twist. This twist is drawn as a 180° rotation of its input object around itself; see Figure 6d for how we denote a twist on framed strands diagrammatically. Additionally, the twist Δ_A must satisfy certain conditions on its interaction with $\overline{(\cdot)}$. These conditions are satisfied in our diagrammatic language; see Egger 2011 [5] for details.

Twisted involutive monoidal categories also satisfy the requirements of balanced monoidal categories, with $\sigma_{A,B} := \Delta_{A\otimes B}; (\Delta_B^{-1} \otimes \Delta_A^{-1})$ and $\theta_A := \Delta_A; \Delta_{\overline{A}}$. This second equality agrees with diagrammatic intuition: θ_A, a 360° rotation, is composed of two Δ_A morphisms, both 180° rotations.

Because Δ_A is natural, it also "commutes" with morphisms: for any $x : A \to B$, we have $\Delta_A^{-1}; x; \Delta_B = \overline{x}$. This naturality implies L_3, and $\overline{(\cdot)}$ allows us to express the flipped-over version of a stitch (in knitting terminology, a knit versus a purl).

3.3 Our Categorical Semantics

We specify the twisted involutive monoidal category on which we define our canonicalization procedure.

Definition 3.1 (Knit semantics category, $\mathcal{K}$). $\mathcal{K}$ is a twisted involutive monoidal category whose objects are the free monoid with generators Σ. This means that its objects are lists of letters in Σ. Every letter labels a strand, and so objects are lists of strands: $A := [a_1, a_2, \ldots, a_n]$. The identity object I is the empty list: $I := []$. The monoidal product $\otimes$ is the concatenation of lists: $[a_1, a_2, \ldots, a_n] \otimes [b_1, b_2, \ldots, b_m] := [a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_m]$. The involution $\overline{(\cdot)}$ reverses a list: $\overline{[a_1, a_2, \ldots, a_n]} := [a_n, \ldots, a_2, a_1]$.

For every object A, there is an identity morphism $\mathtt{id}_A : A \to A$. For any two objects, there is a braiding morphism $\sigma_{A,B} : A \otimes B \to B \otimes A$. We define Δ_A to be a specific morphism composed of σ braids:[10]

$$\Delta_{[a]\otimes A'} := \sigma_{[a],A'}; (\Delta_{A'} \otimes \mathtt{id}_{[a]})$$
$$\Delta_{[]} := \mathtt{id}_{[]}.$$

Note that $\sigma_{A,B} = \Delta_{A\otimes B}; (\Delta_B^{-1} \otimes \Delta_A^{-1})$, as [5] suggested: for any lists A, B,

$$\Delta_{A\otimes B} = \sigma_{A,B}; (\Delta_B \otimes \Delta_A).$$

See Figure 8 for a visual justification of this equality; the left diagram is a canonical visual representation of Δ.[11]

In addition to the braid isomorphism $\sigma_{A,B}$ and the half twist Δ_A, $\mathcal{K}$ has some collection of morphisms $\mathsf{box} : A \to B$ for $|A| \geq 1$ and $|B| \geq 2$. These are the boxes in our diagrams. We delay the formal definition of the box morphisms to Section 8, as their definition relies on a procedure we call slurping that is specific to our translation of knitout to $\mathcal{K}$. The box morphisms act freely within the axioms of twisted involutive monoidal categories, meaning the only manipulations involving box morphisms are those required to exist by twisted involutive monoidal categories – that is, L_1, L_2, L_3, and M_1. The words in $\mathcal{K}$ are therefore composed of only $\sigma_{A,B}$ crossings and box morphisms, as Δ_A can be written using $\sigma_{A,B}$ crossings.

On top of the categorical structure of $\mathcal{K}$, we impose some additional constraints for the sake of our canonicalization. We first assume there is some given equivalence relation on the box morphisms of $\mathcal{K}$ – even though a knit and a purl may have the same counts of input and output strands, we do not consider them identical. We describe the collection of box morphisms and their equivalence relation in full detail in Subsection 8.1.

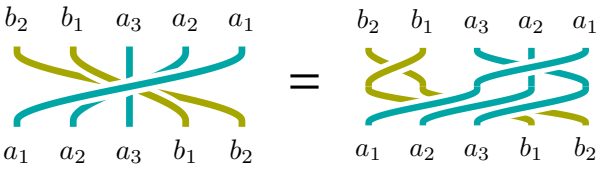


Fig. 8. Visual justification that $\Delta_{A\otimes B} = \sigma_{A,B}; (\Delta_B \otimes \Delta_A)$ for $A = [a_1, a_2, a_3]$ and $B = [b_1, b_2]$. All crossings in Δ are positive (left over right).

For the algorithm, we also impose that every box morphism's input and output strands have some total ordering. If a box morphism is denoted S, we use $S := \{s_1, s_2, \ldots\}$ to represent its ordered output strands (drawn above the box) and $S' := \{s'_1, s'_2, \ldots\}$ for its ordered inputs (drawn below the box). We also restrict that $\overline{(\cdot)}$ reverses total orderings. For example, if the leftmost output strand is foremost in the order of some box morphism S (hence it is denoted s_1), then the rightmost input strand in $\overline{S}$ will be foremost (still denoted s_1). When translating from formal knitout, we always assign the total order of strands as either left to right or right to left, depending on the direction of knitting.

4 Getting Started with Braids

We first detail the most primitive step of our canonicalization algorithm – how can we canonicalize a sequence of braids σ, that is, words without box morphisms? Equivalently, how can we canonicalize

[10] Readers familiar with the braid group may recognize this definition of the half twist as *the* Δ braid in the braid group; this is why we use the symbol Δ. The naturality of $\theta_A = \Delta_A; \Delta_{\overline{A}}$ corresponds to Δ^2 generating the center of the braid group.
[11] In our later diagrams, we choose to draw braids so only one crossing happens per horizontal row: for Δ_A with $|A| = n$, we draw the $\binom{n}{2}$ crossings in a staggered diagram instead of all at once.

words in $\mathcal{K}$ up to equivalence using only the diagram moves B_1, B_2, and B_3? Such braids closely correspond with the *braid group*, the crossings of n progressive (monotonic) strands over each other. There is a known polynomial-time canonicalization of the braid group [4] called the *greedy normal form*, which we adapt to our setting.

Theorem 4.1 (Dehornoy [4], Section 2). *There exists a canonicalization of the braid group called the greedy normal form that runs in $O(b^2 n \log(n))$ time on a braid with b crossings and n strands.*

We use this canonicalization of the braid group as the Braid step of our canonicalization of $\mathcal{K}$. The middle two braids in Figure 10 show an example of the greedy normal form on the braid group.

We define the *braid groupoid*, $\mathcal{B}$. It is closely related to the braid group, except strands have labels. This corresponds exactly to $\mathcal{K}$ without box morphisms. Each σ is an isomorphism, so $\mathcal{B}$ is a groupoid.

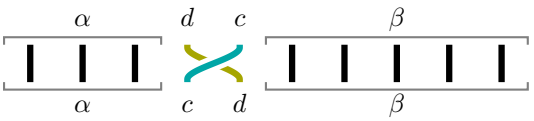


Fig. 9. Diagrammatic representation of $\sigma_{c,d} : \alpha cd\beta \to \alpha dc\beta$ for $|\alpha| = 3$, $|\beta| = 5$. The strands in α, β are not moved, and only c, d cross.

Definition 4.2 (Braid groupoid, $\mathcal{B}$). $\mathcal{B}$ is the *braid groupoid*. Informally, it represents braids from the braid group where each strand is labeled; two braids can compose only when their labels line up.

Its objects are sequences of *distinct* letters, where each letter labels a strand. This is a slight modification of $\mathcal{K}$'s objects, which allow duplicate letters. Because $\mathcal{K}$'s object monoid is free, we can easily rename strands to make a distinct sequence to represent some braid from $\mathcal{K}$ in $\mathcal{B}$. We often label strands by which box they connect to and index them by their total order from that box.

$\mathcal{B}$'s morphisms are generated by crossings of two strands. For any α, β that are possibly empty sequences of distinct letters not containing the letters c, d, there is a generator $\sigma_{c,d} : \alpha cd\beta \to \alpha dc\beta$ that crosses strand c over strand d. $\sigma_{c,d} : p \to -$ is defined for any sequence p where c is directly to the left of d. See Figure 9 for an example of $\sigma_{c,d} : \alpha cd\beta \to \alpha dc\beta$. In the notation of $\mathcal{K}$, we would write $\sigma_{c,d} : \alpha cd\beta \to \alpha dc\beta$ as $\mathtt{id}_\alpha \otimes \sigma_{[c],[d]} \otimes \mathtt{id}_\beta$.

Note that the domain and the crossed strands c, d uniquely determine the codomain. Hence for some sequence of strands p that contains $[c, d]$, we can describe a unique generator $\sigma_{c,d} : p \to -$. When context allows, we omit the object p and refer to some class of generators $\sigma_{c,d}$, which all cross c over d but differ in the strands to the left and right of c, d. The identity braid (no crossings) for any sequence object p is denoted as $\mathtt{id}_p : p \to p$. Similarly, we denote an identity morphism, regardless of object, with $\mathtt{id}$.

$\mathcal{B}$ has commutative diagrams (the category-theoretic analog of the equations in the presentation of a group) for the shift-past commuting relation

$$\sigma_{a,b}; \sigma_{c,d} = \sigma_{c,d}; \sigma_{a,b}$$

and the Yang-Baxter equation

$$\sigma_{a,b}; \sigma_{a,c}; \sigma_{b,c} = \sigma_{b,c}; \sigma_{a,c}; \sigma_{a,b},$$

where a, b, c, d are distinct in both. These relations correspond to diagram moves B_2 and B_3 respectively, and B_1 arises from every braid being invertible.

The greedy normal form also canonicalizes the braid groupoid; see Figure 10 for an example.

Corollary 4.3. *The greedy normal form canonicalizes words in $w \in \mathcal{B}$ in $O(b^2 n \log(n))$ time, where w can be written with b generators $\sigma_{x,y}$ and there are $n = |dom(w)| = |cod(w)|$ strands.*

Proof. $w \in \mathcal{B}$ can be mapped to a braid in the braid *group* by forgetting the labels of its strands. After performing the greedy normal form canonicalization, restore the labels. □

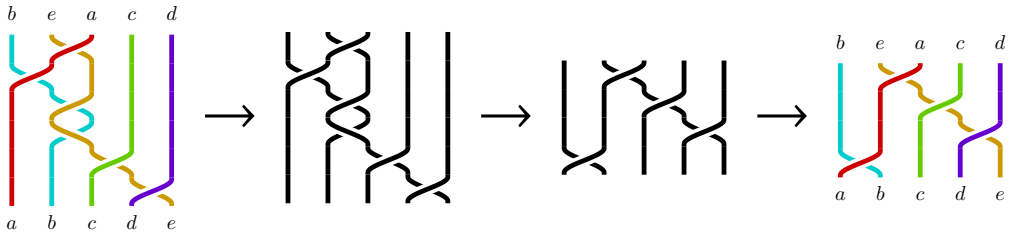


Fig. 10. The canonicalization of the braid groupoid $\mathcal{B}$ described in Corollary 4.3. A braid groupoid word in $\mathcal{B}$ loses its labels to become part of the braid group, is converted to its greedy normal form, and has its labels replaced.

5 Introducing Boxes

In this section, we present our first nontrivial result: a procedure for canonicalizing words in $\mathcal{K}$ up to equivalence by diagram moves B_1, B_2, B_3 and L_1, L_2, L_3. We reserve handling move M_1 until the next section. Because M_1 is the only move that changes the order of boxes, this section solves program equivalence if some canonical order on boxes is guaranteed. For instance, when knitting with only one yarn carrier, that yarn touches every box morphism, enforcing a total order on stitches. For such programs, this section's solution suffices.

We begin with some diagrammatic intuition. Figure 11 shows a word in $X \in \mathcal{K}$ on the left. Applying L_1, L_2, and L_3 in that order follows the solid black arrow to yield a new word $X' \in \mathcal{K}$ that is equivalent to the original X. Consider a naive attempt to canonicalize X, X' to a shared normal form that applies Braid (the braid group's greedy normal form) to the portions of the word on the top and bottom of the box. This approach is insufficient: the L_1 move has changed the horizontal position of the box, and the L_3 move has changed its orientation.

Instead, we focus our attention on the circled crossings in Figure 11. We index the ordered strands coming out (above) the box as s_i, strands coming into (below) the box as s'_i, and strands not touching the box (in an arbitrary order) as r_i. We circle crossings above the box that involve (1) s_1 and any r_i; (2) any two r_i; or (3) s_1 and s_2. These cases correspond to L_1, L_2, and L_3 respectively: each diagram move always introduces one and only one circled crossing above the box. For ease in identifying these crossings, we draw the s_1 strand with a slightly lighter color than the other s_i. Given any word with one box morphism in it, we circle crossings above the box and perform the corresponding L_1, L_2, and L_3 moves to *invert* those circled crossings in a canonicalization step we call Layer. The circling of crossings can be recognized as an algebraic projection via a functor, and Layer ensures that the projection is the identity. Later, we define our canonical form in part to be words where all projections above boxes are the identity.

The red dashed arrows in Figure 11 show the result of Layer on both words X, X'. The L_1, L_2, L_3 rewrite sequence from X to X' is undone when applying Layer on X', and each crossing circled in gray is inverted by a crossing circled in dashed red. If one were to apply Braid on the top and bottom braids of both Layer(X) and Layer(X'), the results would be syntactically identical. Figure 12 shows an example of Layer in full on a more typical example.

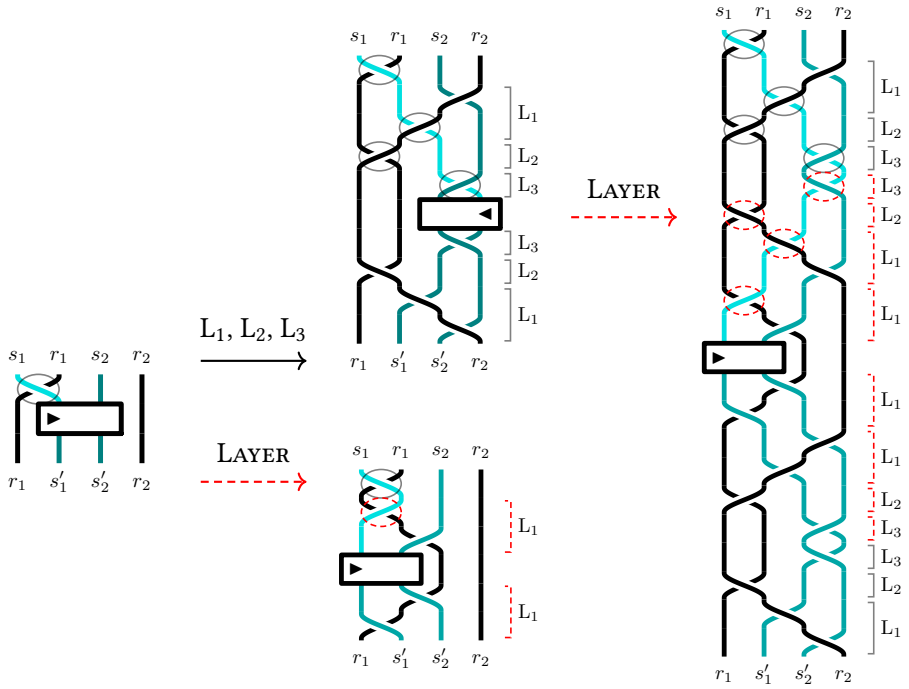


Fig. 11. The word $X \in \mathcal{K}$ on the left can be rewritten to a new word X' via the black solid arrow by applying L_1, L_2, then L_3. The new crossings are labeled with which diagram rule created them. Later applications of diagram rules push out the earlier crossings away from the box. The crossings used by our LAYER step are circled in solid gray. The crossings introduced by LAYER are circled and bracketed in dashed red. After applying LAYER to both X, X', they are equivalent up to B_1, B_2, and B_3, which can be canonicalized by BRAID.

We often represent a word in $X \in \mathcal{K}$ as $X = x_0; S_1; x_1; S_2; \cdots; x_{m-1}; S_m; x_m$, where the x_i are braids in $\mathcal{B}$ and the S_i[12] are box morphisms. Note that there is only one way to divide a word X into a sequence of braids and box morphisms. As a reminder, we read words left to right, and draw them bottom to top to agree with machine knitting.

To canonicalize a word in $\mathcal{K}$ up to equivalence by the six diagram moves not including M_1, we first cancel out circled crossings in x_m by executing LAYER around S_m, then cancel circled crossings in x_{m-1} by executing LAYER around S_{m-1}, and so on until all circled crossings from x_m down through x_1 are canceled. Note that there is no box below x_0, so we do not execute LAYER to cancel out crossings in it. After

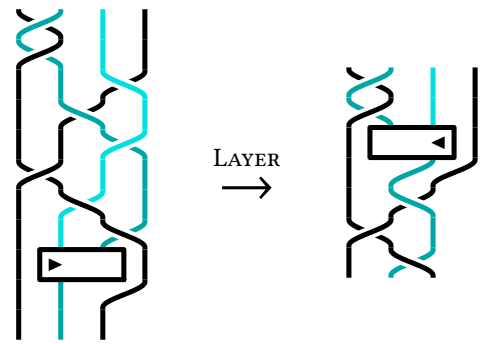


Fig. 12. Example application of LAYER. We have applied some braid simplifications to the resulting word to illustrate how crossings have moved.

[12]We use the letter S because box morphisms represent stitches in knitting.

repeatedly applying Layer, we have canonicalized the L_1, L_2, and L_3 moves. We can then apply Braid on every braid in the result to canonicalize the B_1, B_2, and B_3 moves.

5.1 Formalizing Layer

This subsection is meant to give the reader a feel for how we formalize concepts in Layer like circled crossings. We introduce some notation that will be used later, but reserve formal definitions until Appendix A of our supplementary material.

Let S be some box morphism. As mentioned in Section 3, we also use S to refer to the ordered set of strands coming out of (above) the box S, so the set $S := \{s_1, s_2, \ldots\}$. Similarly, S' is the ordered set of strands coming into (below) the box S, so $S' := \{s'_1, s'_2, \ldots\}$.

We encode circled crossings with two functors (i.e., structure-preserving transformations), γ_S and δ_S. They both map braids in $\mathcal{B}$ containing the s_i strands (so these braids must be above the box S) to braids on less strands by ignoring certain strands, leaving only circled crossings behind. The γ_S functor records circled crossings from L_1 and L_2, while δ_S records crossings from L_3. The crossings recorded by L_3 always commute with those from L_1 and L_2: the Δ twist from L_3 slides through L_1's crossings via B_3, and past L_2's crossing via B_2.

Then, we use a bifunctor ϕ_S that acts as a pseudo-inverse to γ_S and δ_S. It reconstructs braids above the S box representing the L_1, L_2, and L_3 moves using the braids output by γ_S and δ_S. This is guaranteed to reconstruct the circled crossings exactly, up to L_3's crossings commuting with L_1's and L_2's. To simplify our notation, we let ψ_S be shorthand for this operation on braids: $\psi_S(x) := \phi_S(\gamma_S(x), \delta_S(x))$.

Finally, there is a bifunctor $\phi_{S'}$, similar to ϕ_S, that takes the circled crossings from γ_S and δ_S and reconstructs braids *below* the S box representing the diagram moves. We use the abbreviation $\tau_S(x)$ for reconstructing circled crossings below the box: $\tau_S(x) := \phi_{S'}(\gamma_S(x), \delta_S(x))$. Note that τ_S takes a braid above the box S, but returns a braid below the box S. Figure 13 contains examples of these functors applied to a word $x \in \mathcal{B}$.

Our Layer move takes some portion of a word $x_i; S_i; x_{i+1}$, and calculates both $\gamma_{S_i}(x_{i+1})$ and $\delta_{S_i}(x_{i+1})$. It uses those results to calculate

$$\psi_{S_i}(x_{i+1}) = \phi_{S_i}(\gamma_{S_i}(x_{i+1}), \delta_{S_i}(x_{i+1})) \text{ and } \tau_{S_i}(x_{i+1}) = \phi_{S'_i}(\gamma_{S_i}(x_{i+1}), \delta_{S_i}(x_{i+1})),$$

and rewrites

$$\begin{aligned} x_i; S_i; x_{i+1} &= x_i; \tau_{S_i}(x_{i+1}); S_i; \psi_{S_i}(x_{i+1})^{-1}; x_{i+1} \\ &= x_i; \phi_{S'_i}(\gamma_{S_i}(x_{i+1}), \delta_{S_i}(x_{i+1})); S_i; \phi_{S_i}(\gamma_{S_i}(x_{i+1}), \delta_{S_i}(x_{i+1}))^{-1}; x_{i+1}, \end{aligned}$$

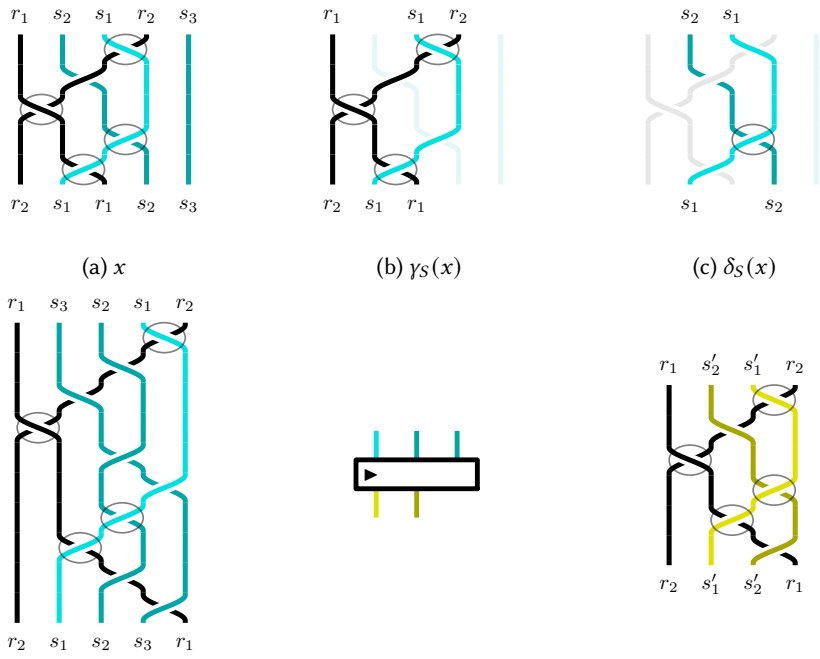
effectively conjugating the S_i box with braids on either side to make the braid above S_i the identity in the projections of both $\gamma_{S_i}(x_{i+1})$ and $\delta_{S_i}(x_{i+1})$.

We prove later that Layer's rewrite can be recreated using only L_1, L_2, and L_3, so Layer preserves word equivalence. A key algebraic insight is that after applying Layer around some box morphism S_i, the braid on top of S_i will always have circled crossings that cancel out:

$$\gamma_{S_i}(\psi_{S_i}(x_{i+1})^{-1}; x_{i+1}) = \mathtt{id} \text{ and } \delta_{S_i}(\psi_{S_i}(x_{i+1})^{-1}; x_{i+1}) = \mathtt{id},$$

so $\psi_{S_i}(\psi_{S_i}(x_{i+1})^{-1}; x_{i+1}) = \mathtt{id}$. We refer to this property as the braid being *simplified* by Layer.

Repeatedly applying Layer to move complexity down the word canonicalizes the L_1, L_2, and L_3 moves. After that, Braid on each braid canonicalizes B_1, B_2, and B_3. All that remains is to canonicalize M_1 by achieving a canonical order of box morphisms; our next section details that process.



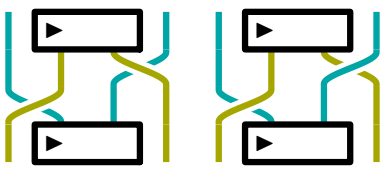
(a) x (b) $\gamma_S(x)$ (c) $\delta_S(x)$

(d) $\psi_S(x) := \phi_S(\gamma_S(x), \delta_S(x))$ (e) box : $S' \to S$ (f) $\tau_S(x) := \phi_{S'}(\gamma_S(x), \delta_S(x))$

Fig. 13. Example of functors applied to a word $x \in \mathcal{B}$. Note that all circled crossings in x are in one of $\gamma_S(x), \delta_S(x)$ but never both. The circled crossings in x are the same as in $\psi_S(x)$ and $\tau_S(x)$. In this example, $|S| = 3$ and $|S'| = 2$, as illustrated by Figure 13e.

6 Finishing the Canonicalization

We present the final piece of our canonicalization algorithm, called Swap, which recognizes equalities by the M_1 diagram rule. This means that we achieve a canonical vertical order of all the box morphisms in a word. We first apply Layer repeatedly, moving down the word as before. Next, we use canonical properties of the input to find a canonical total order on the boxes. We start with the total order of the word's input strands. A straightforward depth-first traversal starting with the word's input strands in order and following the output strands of every box in order will yield a canonical total order. The boxes are then renamed $S_1 \ldots S_m$ according to this total order (note the indices likely do not match the vertical order of boxes). We refer to this order as the *ideal* order – in the ideal world, we could rearrange boxes to achieve this order, and this step would be complete!



(a) Can move past (b) Cannot move past

Fig. 14. Examples of boxes that can and cannot move past each other, respectively.

Even though we have a total order on boxes, it is likely not realizable: if two boxes are connected via a strand, then they cannot move past each other. Unfortunately, this is not the only reason two boxes cannot trade places: Figure 14 shows two simple words, one where disconnected boxes can move past each other and the other where they cannot due to how strands cross each other.

To disambiguate the situations in Figure 14, we introduce the move-past decision procedure. It takes as input some subword $B; x; T$[13] where $\psi_B(x) = \texttt{id}$ (guaranteed after applying Layer) and decides whether it is possible for B and T to move past each other. If they are, it returns the L_1, L_2, L_3, and M_1 moves to cancel out the x in the middle, swap, and then resimplify the word with respect to Layer. We later verify the decision procedure's completeness and prove that if B and T cannot move past each other in a $B; x; T$ subword, then no matter what rewrites and reorderings are made on the word containing that subword, B and T can never move past each other.

6.1 Move-past Procedure

Given a subword $B; x; T$ with $\psi_B(x) = \texttt{id}$, we execute a rewrite similar to Layer, but instead of moving complexity down past B, we move it up past T. To describe this operation, we introduce new functors that are similar to those from Layer. In Layer, the functors ψ_B and τ_B take a braid above the box B, circle some key crossings, and return braids representing L_1, L_2, and L_3 above and below B respectively. Similarly, we use functors ψ'_T and τ'_T that take a braid below T (still between B and T) circle potentially *different* key crossings, and return braids representing L_1, L_2, and L_3 *below and above* T respectively. We use the $'$ marking to denote that these functors circle crossings *below* the box T, while ψ_T and τ_T's domains would be braids above the box T.

The difference in circled crossings is because ψ'_T and τ'_T record L_1, L_2, and L_3 moves around the box T, while ψ_B and τ_B record moves around the box B. Let $T' = \{t'_1, t'_2, \ldots\}$ be the input set of strands below the box T. The crossings circled by $\gamma_{T'}, \delta_{T'}$ are (1) t'_1 and any non-t'_i ($\gamma_{T'}$); (2) any two non-t'_i ($\gamma_{T'}$); and (3) t'_1 and t'_2 ($\delta_{T'}$).[14] Each circled crossing corresponds to L_1, L_2, or L_3 around T as before, and the functors $\psi'_T(x) := \phi_{T'}(\gamma_{T'}(x), \delta_{T'}(x))$ and $\tau'_T(x) := \phi_T(\gamma_{T'}(x), \delta_{T'}(x))$ map those circled crossings to braids below and above the box T respectively. Let

$$f(x) := \textsc{Braid}(x; \psi'_T(x)^{-1}).$$

We later prove that $f(x) = \texttt{id}$ if and only if B can move past T.

When $f(x) = \texttt{id}$, our move-past procedure rewrites

$$B; x; T; y; S \longrightarrow T; \tau_B(y); B; \psi_B(y)^{-1}; \tau'_T(x); y; S.$$

See Figure 15 for an example of a successful box swap. We denote such a move as Swap. We show later this rewrite is composed of only L_1, L_2, L_3, and M_1, so it preserves word equivalence. We also prove that the simplification property guaranteed by Layer (certain ψ projections are id) is preserved by this rewrite.

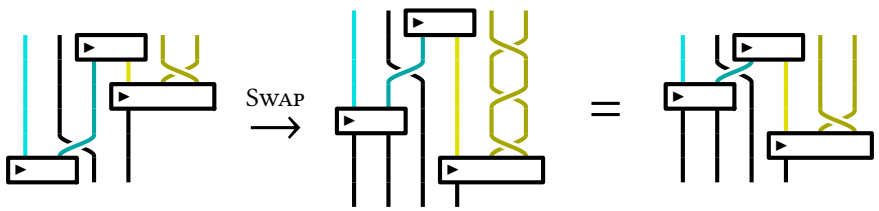


Fig. 15. Application of Swap on a small example.

[13] We use B for bottom and T for top.

[14] An astute reader will notice the possibility of $|T'| = 1$, which was not present describing Layer: every box morphism has 2 or more output strands, but only requires at least 1 input. Definition A.9 of our supplementary material details all the ramifications of this degenerate case; in that case, τ'_T is overloaded, ambiguously referring to one of two similar functors! We omit those details in this section, as they would muddy an already notationally involved algorithm.

6.2 Achievable Canonical Order

We use the ideal (but possibly not achievable) order of boxes and our move-past decision procedure to achieve a canonical order of boxes. The possibly unachievable order of boxes gives each box S_i some canonical index i. We repeatedly find the highest index i such that S_i can be moved to the top of the word. Some box must be found, since there is always some box already at the top of the word. We provide results in Subsection 7.1 that show this procedure's chosen box is canonical. We execute the swaps to move that target box to the top of the word, and lock its position. We repeat for the rest of the boxes, finding the highest index i such that S_i is unlocked and can be moved above all other unlocked boxes.

This step canonicalizes equalities by M_1. The earlier sequence of repeated Layer applications canonicalized L_1, L_2, and L_3. Applying Braid to every braid x_i canonicalizes B_1, B_2, and B_3, completing our canonicalization.

7 Outline of Algorithm's Correctness

We provide an outline of the proof of our canonicalization's correctness, omitting the more involved lemmas. We first present our full canonicalization algorithm in Algorithm 1.

Algorithm 1: Canonicalization algorithm

```
Data: Word X ∈ 𝒦
Result: Canonical form of X
for i = m ... 1 do
    Execute Layer around S_i ;                          /* Step 1: Layer */
end
S_i ← ideal order of boxes ;                            /* reindex S_i */
while some boxes have not been locked do
    for every unlocked box S in decreasing ideal order do
        if S can be moved above all unlocked boxes then
            Move S above all unlocked boxes using Swap ;   /* Step 2: Swap */
            Lock S;
        end
    end
end
for each braid x_i do
    Execute Braid on x_i ;                              /* Step 3: Braid */
end
```

7.1 Move-past Procedure

Theorem 7.1 establishes that the move-past procedure always returns the correct result up to rewrites of the $B; x; T$ subword. We state and prove a more general result than our algorithm uses – in our algorithm, $\psi_B(x) = \mathtt{id}$ always so the move-past check is simpler.

Theorem 7.1. *For all subwords $B; x; T$, let*

$$f(x) := \textsc{Braid}(\psi_B(x)^{-1}; x; \psi'_T(\psi_B(x)^{-1}; x)^{-1}).$$

Then

$$f(x) = \mathtt{id} \iff B, T \text{ can move past each other in the subword } B; x; T.$$

Theorem 7.2 extends Theorem 7.1 by showing that the move-past procedure always returns the same result for a pair of adjacent boxes B, T, regardless of interference by other boxes in a larger word:

Theorem 7.2. *For any word X containing the subword $B; x; T$,*

$$\exists X' \text{ a rewrite of } X : T \text{ is below } B \iff B, T \text{ can move past each other in the subword } B; x; T.$$

We provide proofs of both theorems in Appendix C of our supplementary material. Taken together, they imply that the move-past procedure always returns a correct and consistent result for any two boxes. When using the move-past procedure to query whether some box can move to the top of the word, if the move-past procedure fails to move that box above another, the box cannot move to the top of the word.

7.2 Algorithm Canonicalizes

We formalize our algorithm as an Abstract Rewriting System (ARS) over words in $\mathcal{K}$. An ARS is some binary relation $\to$ on words in $\mathcal{K}$, with $X \to X'$ meaning that the word X can be rewritten to X'. An important feature of rewriting systems is whether they are *canonicalizing*. Given any word X, a canonicalizing ARS will always rewrite X to some unique term X' that cannot be rewritten, called a *normal form*. This happens regardless of $\to$ choices made along the way when some term rewrites to multiple terms. A sufficient condition for an ARS to be canonicalizing is when it is both terminating (there are no infinite chains of rewrites) and weakly confluent (if $X \to X_1$ and $X \to X_2$, there exists some X' where $X_1 \overset{*}{\to} X'$ and $X_2 \overset{*}{\to} X'$, where $\overset{*}{\to}$ is the reflexive transitive closure of $\to$). We prove our ARS is canonicalizing by showing it is both terminating and weakly confluent.

Definition 7.3. Let $\mathcal{X}$ be the set of words representing morphisms in our category $\mathcal{K}$, and let $\equiv_d \subseteq \mathcal{X} \times \mathcal{X}$ be the reflexive transitive symmetric closure of the diagram rules, so two words $x, y \in \mathcal{X}$ are equivalent in $\equiv_d$ when they can be rewritten to each other.

We describe the *energy* of a word, where the lower a word's energy is, the closer it is to its canonical form. We use energy to direct our rewrites, only allowing rewrites that strictly lower energy.

Definition 7.4. The energy of a word

$$X = x_0; S_1; x_1; \cdots; S_m; x_m$$

is a tuple in $\mathbb{N}^3$, and energies are ordered in dictionary order (the first coordinate is the most important).

Its first entry is

$$\sum_{i=1}^{m} \begin{cases} i & \psi_i(x_i) \neq \texttt{id} \\ 0 & \text{otherwise,} \end{cases}$$

which encodes the first step of the algorithm, where braids should be simplified from the top down.

For some box S_i, we define

$$\text{goal}(S_i) = \text{the vertical position of } S_i \text{ after the Swap step of the algorithm completes,}$$

so the box with the highest $\text{goal}(S_i)$ is the highest box in the (possibly unachievable) canonical order that can move to the top of the word. After moving that box to the top, the second-highest $\text{goal}(S_i)$ is defined similarly, ignoring the already-assigned box at the very top. The energy's second entry is the inversion number

$$\sum_{i=1}^{m} \sum_{j=i+1}^{m} \begin{cases} 1 & \text{goal}(S_i) > \text{goal}(S_j) \\ 0 & \text{otherwise.} \end{cases}$$

This entry encodes the second step of the algorithm, where boxes are vertically ordered.

Its third entry is

$$\sum_{i=0}^{m} \begin{cases} 1 & x_i \text{ not in canonical braid form} \\ 0 & \text{otherwise,} \end{cases}$$

which encodes the third step of the algorithm, where braids should be put into canonical form.

We describe an ARS with a new set of diagram rewrites, with 3 classes of rewrites instead of 7. The rewrites closely correspond to the stages of our algorithm. We restrict them to always strictly decrease the energy of a word, so these rewrites are directed. Because $\mathbb{N}$ and its powers are well-founded, this makes our rewriting system terminating. We first prove that if two words are equivalent by $\equiv_d$, then they can be rewritten into some shared form by our ARS. This shows that our new rewrites capture the correct equalities. We then prove our desired result: our rewriting system is canonicalizing.

Definition 7.5. We form an ARS $(\rightarrow) \subseteq \mathcal{X} \times \mathcal{X}$ from the following rewrite rules, included in $\rightarrow$ whenever they strictly reduce energy:

- Layer: Rewrites some subword $B; x; T$ to $\tau_B(x); B; \psi_B(x)^{-1}; x; T$, like in the first step of the algorithm.
- Swap: Rewrites some subword $B; x; T; y; S$ where $\psi_B(x) = \mathtt{id}, \psi_T(y) = \mathtt{id}$, and where B, T can move past each other to

  $$T; \tau_B(y); B; \psi_B(y)^{-1}; \tau'_T(x); y; S,$$

  like in the second step of the algorithm.
- Braid: Rewrites some subword $B; x; T$ to $B; \mathrm{canon}(x); T$ like in the third step of the algorithm.

Swap always preserves the simplification from Layer, so it does not increase Layer's higher-importance energy value:

Lemma 7.6. *For disjoint B, T, with both $|B|, |T| > 1$, if $\psi_B(x) = \mathtt{id}$ and $\psi_T(y) = \mathtt{id}$ then*

$$\psi_T(\tau_B(y)) = \mathtt{id} \text{ and } \psi_B(\psi_B(y)^{-1}; \tau'_T(x); y) = \mathtt{id}.$$

Proof. See Appendix D of our supplementary material. □

We prove these new rewrites express the same equalities as the old rewrites in Appendix D. Here, we provide an informal proof.

Lemma 7.7. *$\equiv_d$ (the reflexive transitive symmetric closure of the diagram rules) is the same equivalence relation as the reflexive transitive symmetric closure of $\rightarrow$.*

Proof. For the forward direction: if two words are connected by B_1, B_2, B_3, then applying Braid on both makes them the same. Similarly, L_1, L_2, and L_3 are undone by Layer. The M_1 case is difficult, as M_1 does not require words to be simplified, but Swap does. In the formal proof, we simplify the word, apply Swap, and show we arrive at the same result and reduced energy with each rewrite.

For the backwards direction, we recognize Layer, Swap, and Braid as compositions of diagram rewrites. □

After establishing that Layer, Swap, and Braid represent the desired equivalences, we show that $\rightarrow$ is weakly confluent. We again supply an informal proof; our full proof is in Appendix D.

Theorem 7.8. *The rewriting system $\rightarrow$ on words in $\mathcal{K}$ formed by Layer, Swap, and Braid is weakly confluent.*

Proof. Our proof handles the 6 cases of each of the 3 rules being weakly confluent with itself or another rule. Most of our cases are trivial. Layer with Layer and Swap with Swap are our most involved, and we provide sketches for each.

Let $B; x; T; y; S$ be a subword of X. Our Layer case shows that simplifying the braid at y then x results in the same word as simplifying x then y then x, showing that energy is reduced by each rewrite.

Our Swap case shows that for some initial order of boxes $A; B; C$, swapping A, B then A, C then B, C (reversing the boxes' order) results in the same word as swapping B, C then A, C then A, B, again showing each rewrite reduces energy. □

Corollary 7.9. *$\longrightarrow$ is canonicalizing.*

Proof. Because $\mathbb{N}$ and its powers are well-founded, $\longrightarrow$ is terminating. By Theorem 7.8, it is weakly confluent. Because $\longrightarrow$ is both terminating and weakly confluent, it is canonicalizing. □

7.3 Polynomial-Time Execution

We claim the algorithm presented in Section 6 calculates the normal form of our ARS. We provide full proofs of these statements in Appendix E of our supplementary material.

Lemma 7.10. *Given any word X, the algorithm in Algorithm 1 calculates the normal form of X in $\longrightarrow$.*

Proof. We show in Appendix E that the word returned by Algorithm 1 always has zero energy, so it must be irreducible. □

Finally, we show that Algorithm 1 runs in polynomial time. It should be noted that we do not believe our bound is tight, as the goal of this paper is to establish a polynomial result. Future work in efficient algebraic representations or algorithmic tricks specific to machine knitting may produce much faster results.

Theorem 7.11. *Algorithm 1 runs in $O(n^5 m^7 b^2 \log(n))$ time on words in $\mathcal{K}$ with n strands in total, m `box` morphisms, and braid words between pairs of `box` morphisms with $O(b)$ crossings each.*

8 Translating Knitout

In this section, we informally describe how to map knitout into our categorical semantics. The main contribution of this paper is the canonicalization of words in $\mathcal{K}$, and this section motivates our canonicalization by connecting knitout programs to words in $\mathcal{K}$. We provide an intuitive description of our translation here, and leave formal definitions to Appendix F of our supplementary material.

The formal knitout language consists of a list of statements that each describe a mechanical action to be performed by a knitting machine. Its syntax is as follows:

$$
\begin{aligned}
ks ::= {} & ks_1 ; ks_2 \mid \texttt{tuck}\ dir\ n.x\ l\ (y,s) \mid \texttt{knit}\ dir\ n.x\ l\ yarns \\
& \mid \texttt{split}\ dir\ n.x\ n'.x'\ l\ yarns \mid \texttt{miss}\ dir\ n.x\ y \mid \texttt{in}\ dir\ n.x\ y \\
& \mid \texttt{out}\ dir\ n.x\ y \mid \texttt{drop}\ n.x \mid \texttt{xfer}\ n.x\ n'.x' \mid \texttt{rack}\ r \mid \texttt{nop} \\
n, n' ::= {} & \texttt{f} \mid \texttt{b} \qquad yarns ::= (y,s)^+ \text{ (without repetition of } y \text{ values)} \\
dir \in {} & \{+,-\} \qquad s, l \in \mathbb{R} \qquad r, x, x' \in \mathbb{Z} \qquad y \in \mathbb{N}
\end{aligned}
$$

Each operation moves some part of the machine, potentially introducing crossings between strands and moving loops. Recall that the real-valued parameters $s, l \in \mathbb{R}$ describe metric parameters for the length of yarn used in an operation, so they are not used in our translation.

Lin et al. [12] formally describe how each formal knitout operation affects the topology of a knitted object. As mentioned in the introduction, the effect of a knitout statement depends on the program state: what strands and loops are attached to which needles. While some knitout operations (`miss`, `xfer`, `rack`) can be directly mapped to $\mathcal{K}$ using their topological semantics, other operations (`in`, `out`, `drop`) do not correspond to morphisms in $\mathcal{K}$. Thus, we first preprocess knitout using rewrite rules into a form amenable to mapping. Figure 16 shows an example of our preprocessing on a diagram denoted by the knitout code in Figure 17.

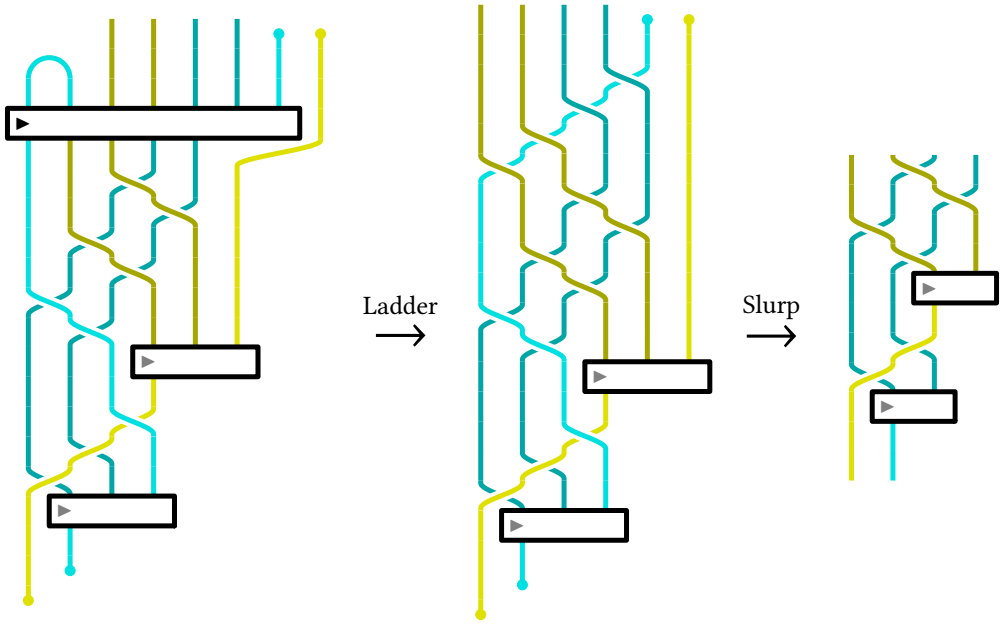


Fig. 16. Example of preprocessing a diagram of a knitted object into $\mathcal{K}$. The left and middle diagrams cannot be represented in $\mathcal{K}$, as they include dropped loops, ins, and outs. The `in`, `out` operations are drawn as bulbs at the ends of strands, and the `drop` operation is drawn as a loop. The `out` operations are slurped to their box morphisms, the `drop` ladders the `split`, and the `in` operations are connected to tucks, so they are attached to the input of the word. If we had attached the `in` operations to the boundary differently (by braiding them together), this would result in a different equivalence class of words in $\mathcal{K}$.

8.1 Grammar of Boxes

Recall that our knit category $\mathcal{K}$ includes a collection of morphisms $\texttt{box} : A \to B$ for $|A| \geq 1$ and $|B| \geq 2$. We define them using a grammar:

$$\texttt{box} ::= \texttt{tuck}\ \mathit{dir}\ y\ \mathit{slurpedOut}$$
$$\mid\ \texttt{stitch}\ \mathit{dir}\ \mathit{bed}\ \mathit{loops}\ \mathit{ys}\ \mathit{slurpedIns}\ \mathit{slurpedOuts}\ \mathit{dropped}$$
$$\mathit{dir} \in \{+, -\} \qquad \mathit{bed} ::= \texttt{f} \mid \texttt{b} \qquad \mathit{slurpedOut}, \mathit{dropped} \in \{\texttt{true}, \texttt{false}\} \qquad y \in \mathbb{N}$$
$$\mathit{ys} ::= \mathbb{N}^+ \text{ without repetition} \qquad \mathit{loops} \in \mathbb{N} \setminus \{0\} \qquad \mathit{slurpedIns}, \mathit{slurpedOuts} \subseteq \mathit{ys}$$

Two box morphisms are equivalent if and only if they are syntactically equal. To canonicalize a word in $\mathcal{K}$, our algorithm requires a total order on every box morphism's inputs and outputs that reverses as the box is flipped by $\overline{(\cdot)}$. We order strands by the direction of knitting: if the direction is + (in our diagrammatic language, the arrow is pointing right), we order both the input and output

```
in - f.0 2;                          yellow carrier 2 on left
in - f.1 1;                          cyan carrier 1 on right
tuck + b.1 5.0 (1, 2.0);             first box
miss + f.0 2;                        move yellow carrier right
miss + f.1 2;                        move yellow carrier right
tuck + f.2 5.0 (2, 2.0);             second box
rack 1;                              cyan loop moves right
xfer b.1 f.2;                        cyan loop joins yellow loop's needle
split + f.2 b.1 5.0 (1, 2.0);        third box
drop f.2;                            output loop of split is dropped
out + f.2 1;                         cyan carrier is dropped
out + f.2 2;                         yellow carrier is dropped
```

Fig. 17. Annotated formal knitout code that denotes the diagram in Figure 16.

left to right. In the − case, we order them right to left. We define box morphisms formally, including their domains and codomains, in Appendix F of our supplementary material.

There are three operations that translate to boxes: tuck is mapped to the tuck variant, knit is mapped to stitch with *dropped* = true, and split is mapped to stitch with *dropped* = false. We call these box operations. Each box operation is guaranteed to have at least one carrier strand y as input and at least one loop (two strands) as output. Hence a straightforward mapping of box operations to box morphisms would satisfy $\mathcal{K}$'s restrictions on box morphisms.

However, we do not directly map box operations to box morphisms. Instead, the translation depends on the in, out, and drop operations that are connected to the box operations via a strand. The *slurpedOut*, *slurpedIns*, *slurpedOuts* and *dropped* fields are populated and updated depending on the nearby in, out, and drop operations in a process we describe next.

8.2 Preprocessing Knitout

The three box operations can be mapped to box morphisms, nop translates to id morphisms, and the three operations miss, xfer, and rack can be represented with braids. The in, out, and drop operations do not have counterparts in $\mathcal{K}$, and this subsection explains how we preprocess a knitout program to remove them by fusing them with box morphisms.

The in operation introduces a carrier strand, out removes a carrier strand, and drop removes one or more loops. These cannot be represented with braids alone, but they also cannot be represented with box morphisms, as $\mathcal{K}$'s restrictions on domain/codomain sizes are not satisfied: each of the three operations has 0 inputs (in) or 0 outputs (out, drop). We instead preprocess them in a step we call *slurping*, dragging them towards a connected box operation and fusing them together. We formalize this in Appendix F as an Abstract Rewriting System (ARS) over formal knitout programs.

The case for out is simplest, so we begin with it. Topologically, an out morphism is a loose end produced when an active carrier's yarn is cut. However, that loose end had to start somewhere – either a box operation, an in operation, or the strand was there at the start of the program. In the final case, we say the out is connected to the boundary of the program. Our rewrite system drags the out towards wherever its strand is connected to.

In the common case that it connects to a box morphism, then we fuse the out with the box operation, reducing the number of output carrier strands the box operation has by 1. Grammatically, we set *slurpedOut* = true when fusing with a tuck and insert the carrier ID into *slurpedOuts* for the stitch variants. Figure 18 shows an example of an out being slurped down to fuse with a tuck. Since every box has at least 2 output loop strands, this still satisfies our codomain restriction. In the uncommon case where the out connects to an in, the operations cancel and are removed, as a loose strand does not change the knitted object. If an out is connected to the boundary, then we remove it from the program, making a note that an out was present at that location that we use when checking program equality. Because the out was slurped to the beginning of the program, its crossings are trivially canonicalized.

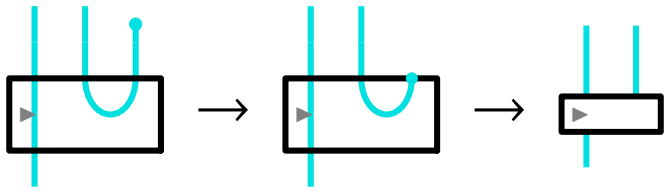


Fig. 18. A tuck fusing with an out. The rightmost figure is the resulting diagrammatic representation in $\mathcal{K}$.

Next, we handle drop. A drop removes all the loop strands on a needle. Even though there are two strands in a loop, we can still slurp the dropped loops to wherever they came from without snagging other strands, as there are no knitout operations that braid a strand through a loop. Similarly to the out case, when dropped loops connect to a box operation, we fuse the drop and the box, reducing the knit operation's loop outputs. Unlike the out case, this may remove *all* outputs from the box operation, making direct translation to $\mathcal{K}$ impossible, as 2 or more outputs are required for every box morphism.

The topology of knitting offers a solution for this. Figure 19 shows a split operation *laddering*. The stitches in knitting are topologically stable as long as the newly created loops are not dropped. If those new loops are dropped, the box operation ladders into only strand crossings σ, meaning we no longer need the box morphism and $\mathcal{K}$'s constraints are not invalidated. Figure 20 shows a knit laddering. This reveals another implicit drop of the input loop, which in turn needs to be slurped to the box it connects to. Because a box operation is destroyed each time, the laddering of drop operations must eventually terminate. In the boundary case, drop is slurped identically to out. Because laddering removes stitches, it is easier to slurp drop operations before in and out, but the order does not matter.

Finally, we must handle in operations. We would like to handle them similar to out operations, slurping them up (later in the program) instead of down. However, tuck operations only have 1 input, a carrier strand. If an in were slurped and fused with a tuck, this would again violate $\mathcal{K}$'s requirements. In this case, we instead slurp the in *down* and connect it to the input of the word, treating it similarly to an out that has slurped to the boundary. Unfortunately, this process is noncanonical and can break topological equivalence: as the in is slurped down, decisions are made on how to braid the strand with others. In our rewrites, we choose to move the in operation to the start of the program and not modify anything else.

This is an obvious technical limitation, but in the domain of machine knitting, it is not severe: the majority of machine knitting programs do not have complicated use of in operations, and so this has little effect on the value of our canonicalization of machine knitting. We provide further analysis of this limitation in Section 11.

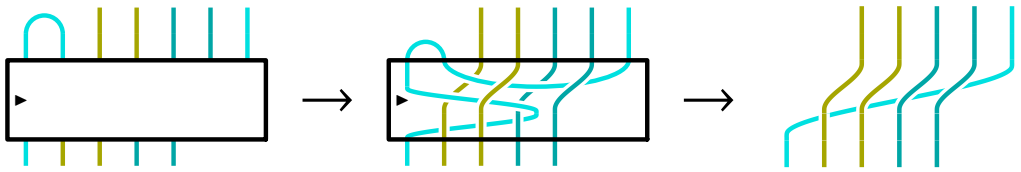


Fig. 19. A `split + f.2 b.1 5.0` $(1, 2.0)$ operation laddering as its output loop is dropped. After laddering, all that remains is the crossing of the carrier thread behind the loops.

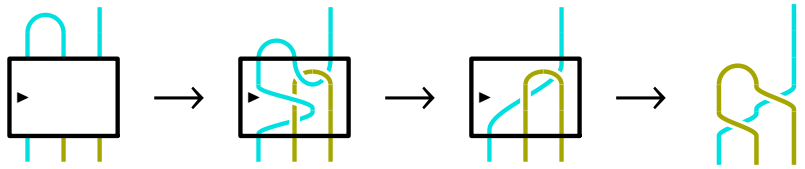


Fig. 20. A `knit + f.`x l (y, s) operation laddering. After laddering, the input loop, which was implicitly dropped by the `knit`, is now explicitly dropped.

Unlike `tuck`, the `knit` and `split` operations are guaranteed to have loop inputs, so `in` operations can be slurped up and fused with them, adding the carrier ID to *slurpedIns* in the grammar. The case for `in` operations connected to the top boundary (the end of the program) is similar to `out` and `drop` connected to the bottom boundary.

8.3 Mapping Processed Knitout to $\mathcal{K}$

After preprocessing knitout, the `in`, `out`, and `drop` operations are either at the beginning or end of the program (ignored by our translation) or fused with some `tuck`, `split`, or `knit` where the fused result can be mapped to a box morphism in $\mathcal{K}$. The `nop` operation maps to `id`, and the remaining three operations only introduce strand crossings so their denotation is trivial: the appropriate braid composed of σ crossings suffices.

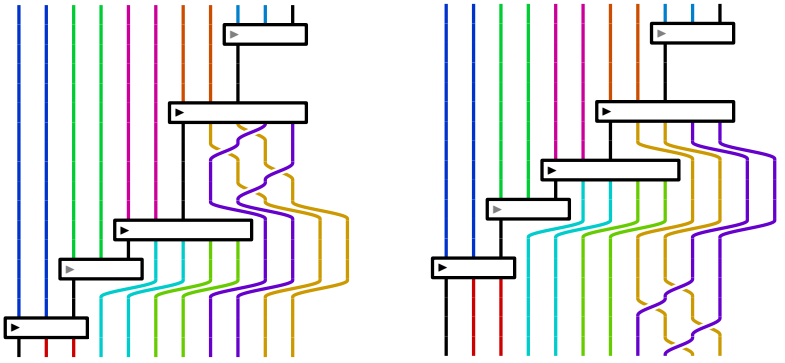
As mentioned previously, Lin et al. [12] mathematically defined the states that knitting machines can have, and how each knitout operation affects machine state. A valid knitout program has a categorical structure, where knitting machine states are objects and knitout operations are morphisms. After our rewrites, this categorical structure still holds. In Appendix F of our supplementary material, we formalize our map from processed knitout to $\mathcal{K}$ using a functor.

9 Applications

In this section we apply our canonicalization algorithm to outputs from two existing knitting compiler systems. We validate the results of one and illustrate a known flaw in the other.

9.1 Validating KODA's Optimization Results

One use case for program equivalence checking is to provide validation of the output of optimizers. We used our canonicalization method to check the first optimization result presented in Hofmann's KODA paper [8]. In their optimization of a lacework example, `xfer` and `rack` instructions are moved past `knit` and/or `tuck` instructions to provide a grouping which is amenable to efficient machine execution. Their optimization reduces the number of carriage passes (linearly correlated with execution time) from 7 to 4. Figure 21 shows that our canonicalization algorithm brings both the input program and the optimized program to the same form, validating KODA's optimization.



(a) Before optimization (b) After KODA optimization, canonical

Fig. 21. Checking the translation result from Figure 6 of KODA [8]. An unoptimized program's semantic denotation is in Figure 21a, and KODA's optimized program's denotation is in Figure 21b. Our canonicalization algorithm shows the two diagrams are equivalent; our canonical form happens to be the efficient diagram in Figure 21b.

9.2 Demonstrating an Autoknit Scheduler Limitation

Our semantics are also useful for studying the output of machine knitting compilers that translate high-level designs into knitout. These compilers are typically complex, multi-step procedures that transform the input through multiple intermediate representations before producing the final machine program.

One such step is *scheduling*, which assigns needles to each knitting operation and moves loops between needles as needed. The scheduler used in Narayanan et al.'s Autoknit system [20] has a known bug: it can introduce half-twists between loops on subsequent rows of knitting as it "rolls" tubes to set up for other operations.

By modifying the re-implementation of this scheduler released by Narayanan et al.,[15] we were able to create two schedules of the same input file to demonstrate this limitation. The input file is a simple five-row, four-stitch-circumference tube. In our first output, we allow the scheduler to choose its default schedule. In the other output we constrained the scheduler to knit one of the intermediate rows of the tube in a different layout, forcing a "roll" of the tube. The semantics of the outputs differ, demonstrating that the scheduler introduced a topological change in the knit output, despite considering both schedules valid.

10 Related Work

Semantic Models for Machine Knitting. The topological semantics introduced in Lin et al. [12] form the basis of our semantics. They represent stitches with *fenced tangles*, analogous to our box morphisms. The authors informally algebraicize their topological semantics, but do not connect it to diagram algebras or monoidal categories. The paper explores how to use the semantics to prove that certain rewrites of programs are permissible; these rewrites improve the runtime efficiency and reliability of knitting programs. We expect that our algebraic semantics, combined with our canonicalization algorithm, will make proving and using similar rewrites easier. The work from

[15]Note that though we modified the code of the scheduler to allow the creation of this example, such schedule changes can also be forced by tube splits and merges in the input object, or by arbitrary tie-breaking in the scheduler's objective function.

Lin et al. [12] is extended in Lin et al. [13] with *instruction graphs*. Lin et al. [13] describe a subset of instruction graphs that are always machine knittable.

Markande and Matsumoto [15] introduce a separate topological semantics for machine knitting. Instead of semantics for programs, they represent semantics for *program snippets*: swatches of stitches embedded on a torus. Input and output loops and strands wrap around because of the toroidal embedding, so fences around stitches are not necessary. However, the semantics are only able to describe swatches with the same number of input/output loops and the same number of input/output carrier strands – most machine knitted objects do not have that form.

Machine knitting design tools have historically used representations that are not formally defined using topology. Autoknit [20] uses a *knit graph* to represent how stitches are connected by loops and carrier strands. Their abstraction does not encode crossings of loops and strands, as the authors use Autoknit to generate knitting machine instructions to knit any manifold and orientable triangle mesh: crossings between loops and strands are irrelevant in their semantics. As such, knit graphs cannot differentiate between programs that are topologically distinct.

KODA [8] is an optimizing compiler that translates knitout to knitout. It does this by first lifting knitout to an *expanded knit graph*, which includes information about crossings between loops and strands by recording loop-loop and loop-strand crossings individually. While the expanded knit graph representation does capture enough information to recreate a topologically equivalent knitout program, KODA cannot represent all possible topological transformations. For example, equalities by L_1 are not considered by KODA. Additionally, KODA does not record movements of carrier strands due to `miss` operations, so it only supports sound optimization on the subset of knitout without `miss`.

Programming Languages in Fabrication. There is a rich history of applying ideas from programming languages to the domain of fabrication. Nandi et al. [19] proposes the use of programming languages techniques to the CAD process for 3D printing. The authors treat solid geometry as a programming language, and provide a verified compiler from CAD code to meshes. They additionally supply a synthesis algorithm that translates meshes into CAD designs for easier editing. Sottile and Tekriwal [24] detail the development of a verified interpreter for G-code for additive manufacturing.

In a unique application, Zhu et al. [27] develop a language for machine knitting that is embedded *inside* machine knit objects. They designed their machine knitting language with the intent of developing quines for machine knitting – knitted objects that denote themselves. Clark and Bohrer [3] introduce an imperative language for sewn quilts, with semantics inspired by homotopy type theory.

Monoidal Categories in Programming Languages. There are many applications of monoidal categories to programming languages, often to represent semantics of computation. Bonchi et al. [1] formulate the rewriting of string diagrams for symmetric monoidal categories using hypergraphs. Choudhury et al. [2] reduce reversible boolean circuits to canonical forms using symmetric rig groupoids (a special case of symmetric monoidal categories).

Outside of symmetric monoidal categories, Hasegawa and Lechenne [6] explore *ribbon combinatory algebras*, relating to the braided untyped linear lambda calculus and framed oriented tangles. Their work expresses the geometric side of combinatory logic, and they study both braided and balanced structures. The work in Joyal and Street [10] forms the foundation of our paper, connecting monoidal categories and the topology of their string diagrams. Selinger [22] provides an accessible survey of monoidal categories and their string diagrams.

Polynomial-Time Program Equivalence. Most programming languages are not canonicalizable in polynomial time, but one classic computational context is regular expressions, described by finite automata [9]. Regular expressions still receive serious research attention today [17, 26].

11 Conclusion

We have presented an algebraic semantics for machine knitting and an algorithm to fully canonicalize those semantics into normal forms. Our canonicalization runs in polynomial time, and we prove our algorithm's correctness.

Our work inherits a limitation from the topological semantics in Lin et al. [12]; we do not capture metric properties. Machine knitting is a physical process, and metrics such as the spacing between stitches or the amount of slack on threads is not captured by our representation. Now that we have developed the means to efficiently classify the topology of machine knitted objects, we hope to incorporate metric properties into our algebraic representation in future work.

While our canonicalization algorithm is proven correct over words in $\mathcal{K}$, our preprocessing step makes noncanonical decisions as it maps knitout code into $\mathcal{K}$. When it encounters an `in` operation attached to a `tuck` that does not ladder, it attaches that `in` to the boundary. Because `in` operations that connect directly to `tucks` tend to only appear in the beginning of machine knitting programs, we believe that this preprocessing limitation does not affect the majority of use cases in machine knitting. When performing local optimizations, the portion of the program containing `in` operations can be ignored. In cases where our preprocessing is insufficient, a user could modify their program to route the `in` operations to the boundary themselves to guarantee properties of the canonicalization algorithm. We recognize this limitation as arising from a trade-off between capturing all programs and equivalences in machine knitting using the full breadth of knot theory, and capturing *almost* all programs and equivalences while maintaining polynomial-time canonicalization.

We are excited to use these semantics in future programming languages research for machine knitting. Our work develops algebraic semantics and a computable canonical form. A natural next step is to develop an optimizing compiler for machine knitting with the formalisms and algorithm of this paper. We expect that optimizing a language with polynomial-time equivalence checking will produce fruitful results, but incorporating desired physical properties like metrics and fabrication constraints may prove challenging. Finally, we hope to design a domain-specific language for machine knitting that denotes our algebraic semantics directly. Such a language could serve as an intermediate representation for design tools, enabling creators to formally and declaratively describe machine knitted objects without specifying how to fabricate them on a knitting machine.

Acknowledgments

We thank Hannah Fechtner for detailed mathematical review and guiding us towards twisted involutive monoidal categories, and thank Ryan Zambrotta for helpful discussion. This material is based upon work partially supported by the National Science Foundation under Grant No. 2319181 and 2319182.

References

[1] Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Paweł Sobociński, and Fabio Zanasi. 2016. Rewriting modulo symmetric monoidal structure. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science* (New York, NY, USA) *(LICS '16)*. Association for Computing Machinery, New York, NY, USA, 710–719. doi:10.1145/2933575.2935316

[2] Vikraman Choudhury, Jacek Karwowski, and Amr Sabry. 2022. Symmetries in reversible programming: from symmetric rig groupoids to reversible programming languages. *Proc. ACM Program. Lang.* 6, POPL, Article 6 (Jan. 2022), 32 pages. doi:10.1145/3498667

[3] Charlotte Clark and Rose Bohrer. 2023. Homotopy Type Theory for Sewn Quilts. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Functional Art, Music, Modelling, and Design* (Seattle, WA, USA) *(FARM 2023)*. Association for Computing Machinery, New York, NY, USA, 32–43. doi:10.1145/3609023.3609803

[4] Patrick Dehornoy. 2008. Efficient solutions to the braid isotopy problem. *Discrete Applied Mathematics* 156, 16 (2008), 3091–3112. doi:10.1016/j.dam.2007.12.009 Applications of Algebra to Cryptography.

[5] J. M. Egger. 2011. On Involutive Monoidal Categories. *Theory and Applications of Categories* 25 (2011), 368–393. http://www.tac.mta.ca/tac/volumes/25/14/25-14.pdf

[6] Masahito Hasegawa and Serge Lechenne. 2024. Braids, Twists, Trace and Duality in Combinatory Algebras. In *Proceedings of the 39th Annual ACM/IEEE Symposium on Logic in Computer Science* (Tallinn, Estonia) *(LICS '24)*. Association for Computing Machinery, New York, NY, USA, Article 42, 14 pages. doi:10.1145/3661814.3662098

[7] Joel Hass, Jeffrey C. Lagarias, and Nicholas Pippenger. 1999. The computational complexity of knot and link problems. *J. ACM* 46, 2 (March 1999), 185–211. doi:10.1145/301970.301971

[8] Megan Hofmann. 2024. KODA: Knit-program Optimization by Dependency Analysis. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology* (Pittsburgh, PA, USA) *(UIST '24)*. Association for Computing Machinery, New York, NY, USA, Article 64, 15 pages. doi:10.1145/3654777.3676405

[9] John Hopcroft. 1971. An n log n algorithm for minimizing states in a finite automaton. In *Theory of Machines and Computations*, Zvi Kohavi and Azaria Paz (Eds.). Academic Press, Cambridge, MA, USA, 189–196. doi:10.1016/B978-0-12-417750-5.50022-1

[10] André Joyal and Ross Street. 1991. The geometry of tensor calculus, I. *Advances in Mathematics* 88, 1 (1991), 55–112. doi:10.1016/0001-8708(91)90003-P

[11] Greg Kuperberg. 2014. Knottedness is in NP, modulo GRH. *Advances in Mathematics* 256 (2014), 493–506. doi:10.1016/j.aim.2014.01.007

[12] Jenny Lin, Vidya Narayanan, Yuka Ikarashi, Jonathan Ragan-Kelley, Gilbert Bernstein, and James Mccann. 2023. Semantics and Scheduling for Machine Knitting Compilers. *ACM Trans. Graph.* 42, 4, Article 143 (July 2023), 26 pages. doi:10.1145/3592449

[13] Jenny Han Lin, Yuka Ikarashi, Gilbert Louis Bernstein, and James McCann. 2024. UFO Instruction Graphs Are Machine Knittable. *ACM Trans. Graph.* 43, 6, Article 206 (Nov. 2024), 22 pages. doi:10.1145/3687948

[14] Logica. 2023. PaintKnit. [Online]. Available from: https://www.paintknit.com.

[15] Shashank G Markande and Elisabetta Matsumoto. 2020. Knotty Knits are Tangles in Tori. In *Proceedings of Bridges 2020: Mathematics, Art, Music, Architecture, Education, Culture*, Carolyn Yackel, Robert Bosch, Eve Torrence, and Kristóf Fenyvesi (Eds.). Tessellations Publishing, Phoenix, Arizona, 103–112. http://archive.bridgesmathart.org/2020/bridges2020-103.html

[16] James McCann, Lea Albaugh, Vidya Narayanan, April Grow, Wojciech Matusik, Jennifer Mankoff, and Jessica Hodgins. 2016. A compiler for 3D machine knitting. *ACM Trans. Graph.* 35, 4, Article 49 (July 2016), 11 pages. doi:10.1145/2897824.2925940

[17] Dan Moseley, Mario Nishio, Jose Perez Rodriguez, Olli Saarikivi, Stephen Toub, Margus Veanes, Tiki Wan, and Eric Xu. 2023. Derivative Based Nonbacktracking Real-World Regex Matching with Backtracking Semantics. *Proc. ACM Program. Lang.* 7, PLDI, Article 148 (June 2023), 24 pages. doi:10.1145/3591262

[18] K. Murasugi and B. Kurpita. 1999. *A Study of Braids*. Springer Netherlands, Norwell, MA, USA. https://books.google.com/books?id=tbi_FDbSJo0C

[19] Chandrakana Nandi, James R. Wilcox, Pavel Panchekha, Taylor Blau, Dan Grossman, and Zachary Tatlock. 2018. Functional programming for compiling and decompiling computer-aided design. *Proc. ACM Program. Lang.* 2, ICFP, Article 99 (July 2018), 31 pages. doi:10.1145/3236794

[20] Vidya Narayanan, Lea Albaugh, Jessica Hodgins, Stelian Coros, and James McCann. 2018. Automatic Machine Knitting of 3D Meshes. *ACM Trans. Graph.* 37, 3, Article 35 (Aug. 2018), 15 pages. doi:10.1145/3186265

[21] Vanessa Sanchez, Kausalya Mahadevan, Gabrielle Ohlson, Moritz A. Graule, Michelle C. Yuen, Clark B. Teeple, James C. Weaver, James McCann, Katia Bertoldi, and Robert J. Wood. 2023. 3D Knitting for Pneumatic Soft Robotics. *Advanced Functional Materials* 33, 26 (2023), 2212541. doi:10.1002/adfm.202212541

[22] P. Selinger. 2011. *A Survey of Graphical Languages for Monoidal Categories*. Springer Berlin Heidelberg, Berlin, Heidelberg, 289–355. doi:10.1007/978-3-642-12821-9_4

[23] Shima Seiki. 2011. SDS-ONE Apex3. [Online]. Available from: http://www.shimaseiki.com/product/design/sdsone_apex/flat/.

[24] Matthew Sottile and Mohit Tekriwal. 2024. Design and Implementation of a Verified Interpreter for Additive Manufacturing Programs (Experience Report). In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Software Architecture* (Milan, Italy) *(FUNARCH 2024)*. Association for Computing Machinery, New York, NY, USA, 10–17. doi:10.1145/3677998.3678221

[25] Stoll. 2011. M1Plus pattern software. [Online]. Available from: http://www.stoll.com/stoll_software_solutions_en_4/pattern_software_m1plus/3_1.

[26] Ian Erik Varatalu, Margus Veanes, and Juhan Ernits. 2025. RE#: High Performance Derivative-Based Regex Matching with Intersection, Complement, and Restricted Lookarounds. *Proc. ACM Program. Lang.* 9, POPL, Article 1 (Jan. 2025), 32 pages. doi:10.1145/3704837

[27] Amy Zhu, Adriana Schulz, and Zachary Tatlock. 2023. Exploring Self-Embedded Knitting Programs with Twine. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Functional Art, Music, Modelling, and Design* (Seattle, WA, USA) *(FARM 2023)*. Association for Computing Machinery, New York, NY, USA, 25–31. doi:10.1145/3609023.3609805

Received 2025-02-27; accepted 2025-06-27