Semantics and Scheduling for Machine Knitting Compilers

JENNY LIN, Carnegie Mellon University, USA
VIDYA NARAYANAN, Carnegie Mellon University, USA and Amazon, USA
YUKA IKARASHI, Massachusetts Institute of Technology, USA
JONATHAN RAGAN-KELLEY, Massachusetts Institute of Technology, USA
GILBERT BERNSTEIN, University of Washington, USA
JAMES MCCANN, Carnegie Mellon University, USA

(a) Separated Sheets (b) Stacked Sheets (c) Linked Sheets

Fig. 1. A knitting machine may be programmed to make two opposite-bed sheets (a) at separate needle indices or (b) one in front of the other. However, changing *only* the carriers used in (b) can produce (c) a program that makes sheets linked at the edge. We present the formal foundation required to reason about such subtle equivalences (≅) and distinctions (≇) among knitting programs.

Machine knitting is a well-established fabrication technique for complex soft objects, and both companies and researchers have developed tools for generating machine knitting patterns. However, existing representations for machine knitted objects are incomplete (do not cover the complete domain of machine knittable objects) or overly specific (do not account for symmetries and equivalences among knitting instruction sequences). This makes it difficult to define correctness in machine knitting, let alone verify the correctness of a given program or program transformation. The major contribution of this work is a formal semantics for knitout, a low-level Domain Specific Language for knitting machines. We accomplish this by using what we call the *fenced tangle*, which extends concepts from knot theory to allow for a mathematical definition of knitting program equivalence that matches the intuition behind knit objects. Finally, using this formal representation, we prove the correctness of a sequence of rewrite rules; and demonstrate how these rewrite rules can form the foundation for higher-level tasks such as compiling a program for a specific machine and optimizing for time/reliability, all while provably generating the same knit object under our proposed semantics. By establishing formal definitions of correctness, this work provides a strong foundation for compiling and optimizing knit programs.

CCS Concepts: • **Software and its engineering → Domain specific languages**; • **Applied computing → Computer-aided manufacturing**.

Authors' addresses: Jenny Lin, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, jennylin@cs.cmu.edu; Vidya Narayanan, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA and Amazon, Sunnyvale, California, USA, vidyan@alumni.cmu.edu; Yuka Ikarashi, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, yuka@csail.mit.edu; Jonathan Ragan-Kelley, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, jrk@mit.edu; Gilbert Bernstein, University of Washington, Seattle, Washington, USA, gilbo@cs.washington.edu; James McCann, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, jmccann@cs.cmu.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2023 Copyright held by the owner/author(s).
0730-0301/2023/8-ART143
https://doi.org/10.1145/3592449

Additional Key Words and Phrases: machine knitting, domain specific languages, fabrication, topology, knot theory, program semantics

ACM Reference Format:
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 (August 2023), 26 pages. https://doi.org/10.1145/3592449

1 INTRODUCTION

Machine knitting is an additive fabrication process for soft goods that has experienced a recent surge in popularity due to increased understanding of the scope and complexity of the objects that can be made. V-bed weft knitting machines in particular, which use two parallel rows of needles to create shaped tubes and sheets, have shifted from making relatively simple garments like socks and sweaters, to more complicated shapes such as athletic shoes and architecture [Popescu et al. 2020], to even programmable materials like actuators [Albaugh et al. 2019] and force sensors [Aigner et al. 2022; Ou et al. 2019]. To complement this development, several high-level design and programming systems have been developed to aid in creating increasingly complex objects. Ideally, such systems should be both complete (support everything that a knitting machine can make) and correct (make exactly what the user wants).

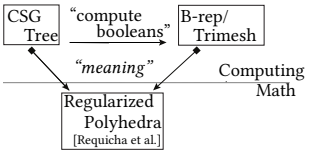
Unfortunately, there is no system that guarantees correctness on the complete scope of machine knitting programs. The cause for this is two-fold: a knit object is a continuous deformation of yarns in space to form an interlocking structure, making it difficult to reason about; and knitting machines have an exponential number of needle configurations which can be used to create a given object. Existing systems deal with this complexity by limiting the scope of knit objects to ones where assigning machine needles, or *scheduling* the object, is tractable. As it turns out, even when the knit object is simple, scheduling it can be surprisingly difficult.

For example, consider a knitting machine program that makes two 50×50 rectangles by alternating between the two of them – first some of sheet A, then some of sheet B – until both sheets are finished at about the same time. This program is written such that Sheet A is scheduled on one row of needles (the *front bed*) at needle indices 1 to 50, while sheet B is scheduled on the other row (the *back bed*) at needles 51 to 100 (Fig. 1(a)). If we only have access to a machine that is 50 needles wide, we would not be able to run this program: back bed needles 51 to 100 do not exist.

A novice programmer might observe that, while sheet A occupies front bed needles 1 to 50, back bed needles 1 to 50 are unoccupied. They might consider rewriting the knitout program so that sheet B is shifted to use back bed needles 1 to 50. As it turns out, however, depending on which machine part delivers the yarn, i.e., which *yarn carrier* is used to knit the sheets, this rewritten program may instead produce two sheets that are linked at their edges (Fig. 1(c)).

The problem at play here is one of program equivalence. For most programs, we say that the "meaning" of the program is "the function it computes." That is, we say two different programs are *functionally equivalent* if they compute the same output for the same input. Theories of functional equivalence allow the definition of semantics that form the basis for systematically testing, debugging, and proving the correctness of program rewrites.

However, programs that control manufacturing machines (e.g., CNC routers, FDM printers, or knitting machines) don't compute functions – they produce physical objects. The "meaning" of a manufacturing program is therefore "the object that it makes." Two different manufacturing programs are *objectively equivalent* if they both denote (i.e., represent) the same object. This raises an important issue: mathematically, what are the objects created by knitting machines?



Mathematically defining physical objects is a surprisingly subtle task. For instance, the analogous "meaning" of constructive solid geometry (CSG) or solid modeling programs was not adequately resolved until Requicha's definition in terms of regular, closed sets [Requicha 1977], which drew on Kuratowski's investigation of closure operators in point-set topology [Kuratowski 1922]. Both the CSG Tree representation (i.e., data structure) and boundary representation (B-rep) – which may be a polygonal mesh or even NURBS surface – "mean" a solid object in Requicha's sense. This definition clarifies the correct behavior of CSG in many edge cases, such as two cubes which intersect in exactly a shared face. The point-set intersection is the shared face (not a solid), but the CSG intersection (regularized intersection) is empty.

Analogously, it has been observed that an adequate mathematical description of knit objects ought to be rooted in knot theory [Grishanov et al. 2009; Markande and Matsumoto 2020; Qu and James 2021]. But similar to the situation in solid modeling, existing formalisms are subtly insufficient for capturing the complete scope of machine knit objects.

In this paper we present the *fenced tangle*, which is an extension of tangles from knot theory, carefully expanded to match the intuition behind machine knitting. Using fenced tangles, we formalize the semantics of the machine knitting language knitout to allow for a mathematical definition of program equivalence. This formalism is complete – it can handle anything a v-bed knitting machine can create – and allows us to reason about correctness – programs are equivalent if they *denote* (i.e., "mean") the same fenced tangle. We then demonstrate how this formalism can be used to prove the correctness of a set of program rewrite rules that can be combined to perform high-level scheduling tasks. The formalization structural overview of the paper is shown in Figure 2.

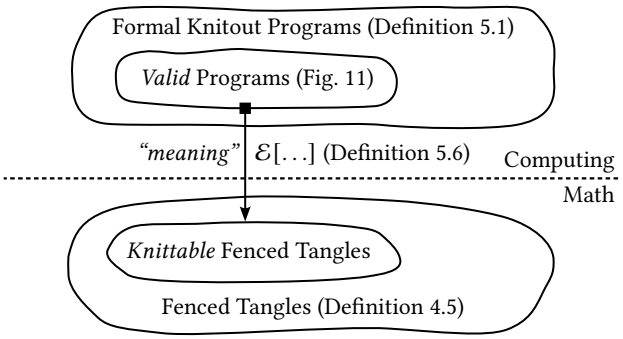
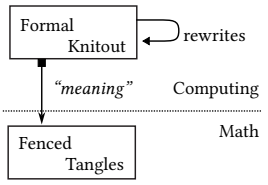


Fig. 2. Formalization approach. The grammar of knitout (Def. 5.1) defines a set of programs, which is narrowed by our validity relation (Fig. 11). Every valid knit program denotes (i.e., "means") a fenced tangle (Def. 4.5) via formal knitout semantics (Def. 5.6, Fig. 12).



The contributions of our paper are as follows:

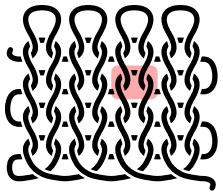
- We propose fenced tangles (Section 4.1) as a mathematical basis for defining machine knit object equivalence. In addition, we provide three operations for composing fenced tangles from simpler primitives.
- We use fenced tangles to define a denotational semantics for knitout (Section 5). This is the first formal semantics that covers the complete space of v-bed knitting machine programs. We believe it may also be the first formal semantics whose denotations are literally pictures/diagrams.
- We demonstrate how our formal semantics can be used to prove the topological correctness of general knitout program rewrite rules (Section 6); and give correctness proofs of several useful low-level rewrite rules (Appendix D).
- We demonstrate how these low-level rewrites can be used to schedule and optimize knitout patterns – including multi-layer objects, which are impossible to create with previous machine knitting design systems (Sections 7 & 8).

2 MACHINE KNITTING BACKGROUND

We begin by providing a brief overview of knit structures and machine knitting; for a detailed description of machine operations we refer the readers to [McCann et al. 2016].

Knitting is the act of taking one or more yarns and manipulating them into a series of interlocking loops that form a stable fabric. The

inset figure shows an example of a knit structure. The yarns used to construct a knit fabric are pliable and can slide along other yarns. This results in soft, deformable fabric structures. Technically, yarn in a knit structure can be unravelled to undo the loops and transformed into a completely different object. However this degree of freedom is counter-productive when trying to characterize the geometric and topological structure of the object. Typically, once the end(s) of the yarn(s) in a knit object have been secured, the loops constituting the object can continue to slide and the fabric can continue to deform in 3D space, but the relationships between yarns that constitute the basic building blocks of the fabric (e.g., the highlighted "knit" stitch) remain fixed. Indeed, this set of fixed loop relationships has been used to accelerate knit simulation [Cirio et al. 2015]. This notion of strands that can be deformed in 3D space but have fixed relationships to their surrounding structures lends itself well to description with the topological notion of 'tangles' – a portion of a knot bounded by a circle with fixed points on its boundary. We will use tangles and extend the idea to a notion of 'fenced' tangles in section 4.1 to formalize the topological structure of knit objects.



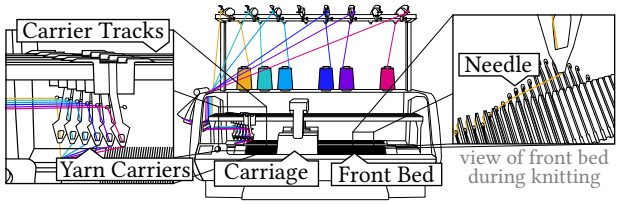


Fig. 3. A v-bed knitting machine creates fabric by using a carriage to actuate needles arranged into front and back beds. The beds are positioned in an inverted "v" shape, with the back bed behind the front bed (and, thus, not visible in this illustration). Yarn is supplied to the needles by yarn carriers which run along carrier tracks. (Figure based on [Sanchez et al. 2023].)

V-bed weft knitting machines (Fig. 3) consist of two facing *beds* (rows) of hook-shaped needles, each of which can hold a stack of loops. Between the two beds runs a number of tracks, each of which has a single yarn carrier that provides yarn. Most machine operations consist of one or more yarn carriers moving to a particular location, the needle at which is then actuated to move forward and pull yarn from the yarn carrier(s) to construct loops on its hook (`tuck`). The needle and associated mechanisms can also pull the new loop through previous loops held on its hook while releasing these held loops (`knit`). While yarn carriers provide yarns for needles to operate with, they also trail yarns between needles. Yarns produced by different carriers can entangle with each other when used on needle beds closer or further away from each track, making the underlying topology of the constructed object challenging to track. In addition to creating and pulling loops through loops, two aligned needles on opposite beds can move loops held on one to the other (`xfer`). This can be done independently or combined with the formation of a new loop (`split`). The back bed can slide left/right (`rack`) to change which needles are aligned. By combining basic loop-making operations with the ability to move loops between needles, knitting machines can produce complex knit structures.

The act of actuating the needle itself is performed by a *carriage* that rides along the length of the needle beds. The carriage encases a configurable cam plate that engages with needles on the needle bed, where each machine operation has a different cam plate setup. This has two important implications. First, any number of stitches may be performed in one carriage *pass* as long as the stitches appear in order and use compatible cam plate setups and yarn carriers. Knitting machine program efficiency is generally increased by decreasing the number of passes – which means that moving knitting instructions without changing program meaning is an important task for knit programmers. Second, any language that allows the use of all cam plate setups and yarn carriers is a complete knitting machine language.

3 RELATED WORK

Knitting (both by hand and machine) and other forms of fabric-making craft have a very rich history [Postrel 2020], with recent research focusing on high-level and 3D design and interaction tools for hand and machine knitting as well as specialized knitted structures for application to various domains. In doing so, a variety of knitting representations have been developed, though most of them do not rigorously characterize the object being made. The work done to mathematically characterize knit objects only apply in more limited settings.

We begin by reviewing the current state-of-the-art knit program generation pipelines and their limitations caused by incomplete characterization of the machine knitting program space. We then go over existing formalizations of knit objects before covering other DSL and semantic approaches to fabrication.

3.1 Knitting Machine Program Generation

Traditionally, knit programming occurs directly in the construction space of the machine – requiring users to figure out the construction location (at which needle must a stitch be created) and construction order of stitches (at what time this stitch must be created) at the same time as they determine stitch type and connectivity. Further, it is the programmer's responsibility to ensure that stitches and transfer instructions are encoded appropriately and efficiently. This is done using either the proprietary languages supported by industrial knitting CAD systems such as KnitPaint [Shima Seiki 2011] and M1 Plus [Stoll 2011] or more recently the knitout language [McCann 2017]. To provide high-level control and support common designs at scale, these CAD systems also support parametric templates for garments such as sweaters and gloves. Libraries of textures are also maintained that can be applied to patterns and further edited [Shima Seiki 2019; Soft Byte Ltd. 1999]. Guidebooks of advanced techniques do exist that can assist with this process [Underwood 2009]. However, the traditional knitting design process still requires the knowledge of an expert machine programmer.

In order to lower the barrier of entry to machine knitting, various systems were developed to decouple knit object design from programming by automatically generating machine knitting programs from high-level representations. Popescu et al. [2018] described

a system that automatically generates a knit representation for topologically-disc-shaped patches, which are later connected manually. However, many intermediate steps including patch segmentation and machine layout remain manual in their system. Narayanan et al. [2018] introduced an automatic pipeline that generates all-knit surfaces from 3D meshes. They later extended the pipeline to handle color and simple textures [Narayanan et al. 2019]. Jones et al. [2021] introduce a system to support patch-level pattern editing while maintaining low-level knittability constraints. Kaspar et al. [2019b] present a system that learns machine knitting instructions by curating a dataset of KnitPaint programs and images of the associated fabricated results. Recently, Nader et al. [2021] presented a graph rewriting based approach for supporting 3D knitting of meshes with textures. Finally, Kaspar et al. [2019a; 2021] presented an interactive design system in the construction space coupled with techniques to compose textures for surface patterning and force-layout based embedding. More recently, they presented an approach to turn cut-and-sew patterns into seamless machine knitting patterns.

Crucially, all these systems assume either explicitly or implicitly that the input representation is a surface where all yarn paths lie within said surface. While it is true many knit objects are amenable to such a representation, techniques such as thick spacer fabrics [Albaugh et al. 2021] and knit integrated tendons [Albaugh et al. 2019] involve yarns that move between what could otherwise be characterized as separate surfaces. In fact, our motivating example of two interlocked sheets (Figure 1) also illustrates a situation where this assumption does not hold. This limitation is primarily due to the difficulty of correctly scheduling knit objects with more complicated yarn routing. Of existing systems, KnitKit [Nader et al. 2021] in theory could be adapted to handle more complicated scheduling problems. However, it still requires an expert to author such a scheduling algorithm, and – indeed – no foundation exists upon which to judge the correctness of such an algorithm.

3.2 Formal Characterization of Knit Objects

How should we mathematically represent an object created by a knitting machine? Unlike rigid machined objects (formalized as regular, closed subsets of $\mathbb{R}^3$ [Requicha 1977]), knit objects are built out of entangled, flexible yarn. While there has been investigation into hand knitting as 2D surfaces, such as Belcastro's [2009] proof that 2D surfaces of any topology can be hand knit, understanding the underlying yarn-level structure of knitting remains an interesting and challenging problem. In addition, it is important to note that human knitters are dexterous and able to form more complex stitches than v-bed machines. Thus it is useful to narrow our focus to specifically objects that can be knit by machines.

Most prior work focused on yarn-level knit topology look to *knot theory* [Adams 1994] for inspiration. However, directly using mathematical "knots" to formalize knit objects runs into three significant problems: (1) Knots are comprised of closed loops, while knit objects have loose ends, and there is no canonical way to close these loops for an arbitrary object. (2) Knot/link diagrams are not composable—meaning that there are no simple operations for building complex knot diagrams out of simpler knot diagrams. (3) The topological equivalence of knot theory does not account for any metric properties of a real knit object, which arise from the looseness/tightness of stitches, as well as non-stitch elements like "misses" (machine-knitting) or "yarn overs" (hand-knitting) that simply let out more yarn between stitches.

To address points (1) and (2), prior work is limited to a subset of knit objects. For example, Grishanov [2009] studied textile structures like knitting and weaving as knots and links on a torus, while Markande and Matsumoto [2020] focused specifically on knit swatches, viewing knit stitches as knots on a thickened torus with an algebra to join them and make a fabric. The choice of the torus as the embedding space addresses issue of loose ends, while the algebra introduced by Markande and Matsumoto allows for a type of composition that provides interesting insight on the periodic nature of common knit structures. However, these particular abstractions can only cover infinite periodic structures, making them ill-suited for describing specific finite programs. Lin and McCann [2021; 2018] have looked specifically at using the Artin braids to formally define transfer plan correctness, which is a subset of knitting instructions. Their choice of the Artin braids for their mathematical formalism means points (1) and (2) are addressed. However, its definition also includes a monotonicity condition that conflicts with the loop formation process in knitting. Thus their approach cannot be extended to all knitting instructions.

As for non-knot theoretic formalizations, TopoKnit is a data structure that uses graph edges and nodes to describe yarn routing and intertwining respectively, thus enabling certain topology checks on machine knit fabrics [Kapllani et al. 2022, 2021]. However, while it covers a large subset of machine operations, it is still incomplete, and not all relevant topological features are captured. Several machine knitting design systems involve graph-based intermediate data structures that in theory could be extended to describe non-planar knits [Kaspar et al. 2019a; Nader et al. 2021; Narayanan et al. 2018]. However, they either assume an input from a planar representation or only consider the planar case.

Our topological formalism is carefully defined to capture the full scope of machine knitting operations while still addressing points (1) and (2). While our formalism does not directly address point (3), we do discuss in Section 7.1.3 how a heuristic approach can be used to develop basic reasoning on metric properties.

3.3 DSLs for Fabrication

Based on the insight that fabrication plans are programs, graphics researchers are applying programming language techniques to solve fabrication problems [Leake et al. 2021; Wu et al. 2019; Zhao et al. 2022]. The Carpentry Compiler uses a set of rewrite rules to perform equality saturation on programs representing different ways of constructing a solid shape from wood [Wu et al. 2019]. This works well because the construction of each sub-component can be constructed *free of the context* of how other sub-components are constructed, and of how that sub-component will be assembled into the whole. By contrast, knitting machines have large amounts of state, and knitting programs are therefore context-sensitive. Consequently, we must state and apply our rewrites in the context of a particular program trace, which exposes our state-dependence.

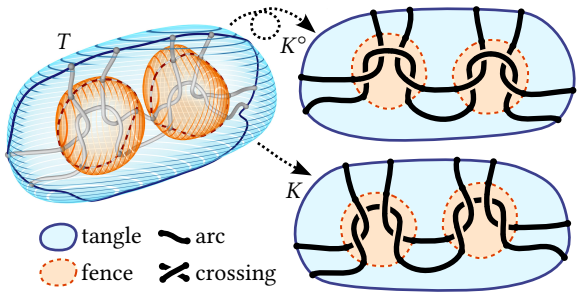


Fig. 4. A fenced tangle, T, and two projections, K and K°, to fenced tangle diagrams which differ in their equator orientation.

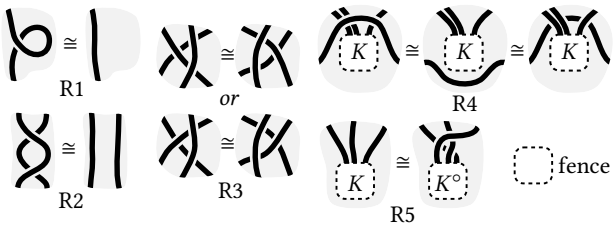


Fig. 5. Equivalent fenced tangle diagrams are connected by sequences of smooth 2D deformations along with Reidemeister moves (R1-3) and fenced-tangle Reidemeister moves (R4, R5), which work regardless of the number of arcs connected to the fence.

The idea of user-scheduling, or decoupling the "algorithm" of what needs to be computed and the "schedule" of how it should be computed, has been popularized as a way of concisely writing high-performance code for CPUs and accelerators [Chen et al. 2018; Ragan-Kelley et al. 2012]. Some of these user-schedulable DSLs have a "rewrite-based" approach, where their scheduling language rewrites one IR to the same IR, which helps make the scheduling language modular and composable [Ikarashi et al. 2022; Steuwer et al. 2017]. In this paper, we also take a rewrite-based scheduling approach, and define a set of scheduling rules that rewrite one knitout representation to another equivalent knitout representation.

4 FENCED TANGLES

In this section, we present a formalism based on a presentation of *tangles*, which, roughly speaking, represent a cut-out piece of a knot diagram. By enriching these tangles with *fences*, we allow internal loose ends and prevent local unravelling (Fig. 4). We then address the issue of composition by defining a standard presentation and three composition operations.

4.1 Fenced Tangles Definition

The following definition uses some technical terms such as *tame*, *homeomorphic*, etc. We provide precise definitions of these terms in Appendix A.1 and recommend a knot theory book [Adams 1994] for a more detailed discussion. In addition, we provide the following brief, intuitive glossary. *Tame* can be understood as meaning "not pathological in strange fractal ways." *Homeomorphic* (meaning there exists a homeomorphism between the two shapes/spaces) can be understood as "has a continuous mapping between them" whereas saying there is an *ambient isotopy* between shapes means "can be continuously deformed into each other without collision". Thus, all closed loops in $\mathbb{R}^3$ are homeomorphic (all equivalently circles in and of themselves) but are not all ambient isotopic – since they can be knotted in different ways.

Definition 4.1 (Tangle)**.** Let $U \subseteq \mathbb{R}^3$ be compact and simply connected (i.e., homeomorphic to a closed ball), with *equator* $Q \subset \mathrm{bd}(U)$, a tame loop homeomorphic to the circle. A *tangle* T in U is a tame embedding of zero or more arcs and loops $\gamma_i : [0,1] \to U$ (continuous and tame), satisfying the following conditions: (i) The interior of each arc is interior to U ($\gamma((0,1)) \subseteq \mathrm{int}(U)$). Either (ii.arc) each endpoint lies interior to U or on the equator ($\gamma(\{0,1\}) \subseteq Q \cup \mathrm{int}(U)$); or (ii.loop) the endpoints are coincident in the interior ($\gamma(0) = \gamma(1) \in \mathrm{int}(U)$). (iii) no two arcs intersect. Two tangles T_1, T_2 are equivalent ($T_1 \cong T_2$) if there is an ambient isotopy of $\mathbb{R}^3$ carrying T_1 to T_2.

Rather than reason about tangle equivalence directly, we will instead work with diagrams.

Definition 4.2 (Tangle Diagram)**.** Let $V \subseteq \mathbb{R}^2$ be compact and simply connected (i.e., homeomorphic to a closed disc). A *tangle diagram* in V is a tame immersion of zero or more arcs and loops $\gamma_i : [0,1] \to V$, and crossing annotations satisfying the following conditions: (i) The interior of each arc is interior to V ($\gamma((0,1)) \subseteq V$). Either (ii.arc) the endpoints may lie anywhere in V; or (ii.loop) the endpoints are coincident in the interior ($\gamma(0) = \gamma(1) \in \mathrm{int}(V)$). (iii) There are a finite number of transversal intersections p_i between the arcs (including self-intersections) with each such "crossing" annotated with one of the two arc segments "passing over" the other.

Two tangle diagrams K_1, K_2 are equivalent ($K_1 \cong K_2$) if K_1 can be transformed into K_2 by some sequence of the following manipulations: ambient isotopy of $\mathbb{R}^2$, or Reidemeister moves 1, 2, or 3 (Fig. 5).

We say that a tangle diagram K is a *projection* of a tangle T, Figure 4, if there is a projection of $\mathbb{R}^3$ to $\mathbb{R}^2$ sending U to V, Q to $\mathrm{bd}(V)$, γ_i in U to γ_i in V, and such that the crossing annotations agree with the ordering of arcs in $\mathbb{R}^3$ as they are projected.

Definition 4.3 (Flip of a Diagram)**.** Note that if K is a projection of T, then K° (the diagram obtained by flipping the order of each crossing, and taking the mirror reflection in $\mathbb{R}^2$) is also a projection of T, but not necessarily an equivalent projection.

Proposition 4.4. *Let T, T' be two tangles and K, K' their projections. Then $T \cong T'$ iff $K \cong K'$ or $K^\circ \cong K'$ (see Figure 4)*

A fenced tangle is defined similarly to a regular tangle, but with the extra data provided by "fences", and one key relaxation of the conditions.

Definition 4.5 (Fenced Tangle (Diagram))**.** Let T be the data for a tangle defined on U. Additionally for reference, let S^2_L be the 2-sphere S^2 along with a distinguished equator $Q_L : S^1 \to S^2_L$. Then a *fenced tangle* on U is defined by the tangle data T, along with a set of tame embeddings of this reference "fenced sphere" $L_i : S^2_L \to U$. These fenced spheres must satisfy the following conditions (i) all

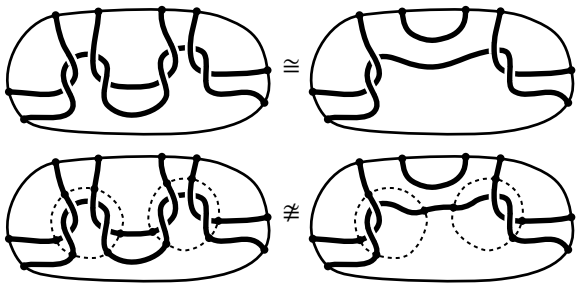


Fig. 6. Tangles without fences (top) can locally "unravel". Fences (bottom) prevent unravelling by restricting the motion of arcs at crossings. This is key to capturing the as-fabricated topology of knit items.

spheres are disjoint in U. (ii) all intersections between arcs and labels are transverse and occur along the equator $L_i(Q_L)$ (fence). Finally, we relax the tangle condition on where endpoints of arcs are allowed to lie. In a fenced tangle, endpoints of arcs are also allowed to lie on fences, as well as on the equator of U or joining up into a loop. Two fenced tangles are equivalent if there is an ambient isotopy between them which also carries fences to fences (Fig. 6).

Given a fenced tangle diagram K on V, let fences be tame embeddings of the circle $L_i : S^1 \to V$ satisfying the following conditions: (i) all fences are disjoint in V. (ii) all intersections between arcs and labels are transverse. (Similarly, arc endpoints are now allowed to lie on the fence circles instead of only forming loops or running to the end of the diagram) A *fenced tangle diagram* is a tangle diagram together with a set of fences. Two fenced tangle diagrams K_1, K_2 are equivalent if K_1 can be transformed into K_2 by some sequence of ambient isotopies of $\mathbb{R}^2$, Reidemeister moves 1, 2, 3, or fenced-tangle Reidemeister moves 4, 5 (Fig. 5).

Similar to plain tangles, a fenced tangle diagram K can be a projection of a fenced tangle T, provided fenced spheres are projected to fences, meaning that the sphere's equator is projected to a diagram fence and the volume enclosed by the fenced sphere is projected to the area enclosed by the fence. A similar proposition holds for K°.

4.2 Fenced Tangle Composition

Having now defined fenced tangles, it is useful to be able to describe them using a composition of simpler fenced tangle diagrams. This enables the proof of several lemmas that can be used to facilitate proofs of fenced tangle equivalence (see appendix B). To do this, we first define a standard diagram presentation that will be used for the rest of the paper:

Definition 4.6 (Slab Presentation)**.** Let K be a fenced tangle diagram defined on R, a rectangle in the plane. Then we say K is an (n, m)-slab if there are n arc endpoints lying on the bottom side of the rectangle and m arc endpoints lying the top side of the rectangle, and no endpoints on the left or right.

Notation 4.7 (Slab Types). It will be useful to refer to the set of (n, m)-slabs by $\mathcal{S}_n^m$, so that we may simply write $K \in \mathcal{S}_n^m$.

We then define three types of tangle concatenation (see Fig. 7 for pictorial intuition).

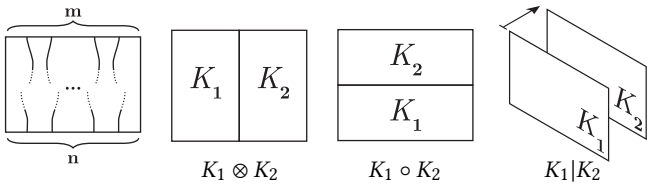


Fig. 7. Slab presentation and three types of fenced tangle concatenation. From left to right, an (n, m)-slab, horizontal concatenation $K_1 \otimes K_2$, vertical concatenation $K_1 \circ K_2$, and layer concatenation $K_1|K_2$

Definition 4.8 (Horizontal Concatenation)**.** Let $K_1 \in \mathcal{S}_n^m$ and $K_2 \in \mathcal{S}_p^q$. By ambient isotopy, we can scale the rectangles to have equal height. Then if we glue the right side of K_1 to the left side of K_2 we get their horizontal concatenation $(K_1 \otimes K_2) \in \mathcal{S}_{n+p}^{m+q}$.

Definition 4.9 (Vertical Concatenation)**.** Let $K_1 \in \mathcal{S}_n^p$ and $K_2 \in \mathcal{S}_p^m$. Again, by ambient isotopy, we may assume that the two rectangles have equal width, and that the p top points of K_1 align with the p bottom points of K_2. Then we can construct their vertical concatenation $(K_1 \circ K_2) \in \mathcal{S}_n^m$ by gluing the two rectangles along the matching top/bottom.

Definition 4.10 (Interleavings)**.** Let $m, n \in \mathbb{N}$. Then, an interleaving ω of m and n ($\omega \in \mathcal{I}_{m,n}$) can be specified as a partition of $[m+n]$ into two sets of size m and n respectively. Let $\omega \subseteq [m+n]$ be the first set, of size m. Let $\overline{\omega} \in \mathcal{I}_{n,m}$ be the opposite interleaving, specified by the second set of ω.

Definition 4.11 (Layer Concatenation)**.** Let $K_1 \in \mathcal{S}_n^m$ and $K_2 \in \mathcal{S}_p^q$, with both defined on the same rectangular region R (also achievable by ambient isotopy). Furthermore let $\iota \in \mathcal{I}_{n,p}$, and $\omega \in \mathcal{I}_{m,q}$ be interleavings of endpoints of K_1 and K_2 on the bottom (input) and top (output) of this common rectangle R. Then, $(K_1|_\iota^\omega K_2) \in \mathcal{S}_{n+p}^{m+q}$ is the layering of K_1 over K_2 according to this interleaving. Let $K_1|_\iota^\omega K_2$ contain all arcs and labels from both diagrams. Any new crossings are annotated such that arcs from K_1 pass over arcs from K_2. Furthermore, $K_1|_\iota^\omega K_2$ is only considered well defined if (i) crossings between arcs and labels from K_1 and K_2 are transverse, (ii) all arcs and labels in K_1 lie outside of all labels in K_2, and (iii) all arcs and labels in K_2 lie outside of all labels in K_1.

Rather than draw out every tangle diagram in full, we will find it useful to define the structure of some common fenced tangle slabs and use those to compose more complex fenced tangles.

Definition 4.12 (Identity Slabs)**.** Let $id_n \in \mathcal{S}_n^n$ consist of n arcs running straight up from the bottom to the top of the slab, called an/the *identity slab*. When n can be inferred from the context, we simply write id. id_0 is also called the *empty tangle*.

Definition 4.13 (Permutation Slab)**.** Let o be a permutation of n things specified (equivalently) as a one-to-one function $o : [n] \to [n]$, which may be notated as a non-repeating list of the numbers in $[n]$ in any order. Then define the slab $\pi_o \in \mathcal{S}_n^n$ as n strands, each running from the i^{th} input point to the $o(i)^{\text{th}}$ output point without crossing itself, and such that whenever the strand starting at input i and the strand starting at input j cross (with $i < j$) i crosses over

j. All such slabs are equivalent. π_o^{-1} is defined as the unique slab s.t. $\pi_o \circ \pi_o^{-1} = id_n$. However, note that in general $\pi_o^{-1} \neq \pi_{o^{-1}}$. So for a given permutation o, the four slabs π_o, π_o^{-1}, $\pi_{o^{-1}}$ and $\pi_{o^{-1}}^{-1}$ are distinct. In particular, $\pi_{o^{-1}}^{-1}$ looks identical to π_o, except the crossings are all right-over-left, rather than left-over-right. (and similarly for the other two cases)

Lastly, we want some way to pick and separate out some number of yarns; and in reverse, a way to merge them back into a group.

Definition 4.14 (Separate and Merge). Let $\iota \in \mathcal{I}_{n,p}$ be an interleaving. Observe that ι defines a permutation function as follows: Let o_ι be the permutation function that sends the subset ι to $[0, n)$ and the subset $\bar{\iota}$ to $[n, n+p)$ with the mapping monotonic within each side of the partition. We define *separate to the left* as $\overleftarrow{V}_\iota = \pi_{o_\iota}$, and *separate to the right* as $\overrightarrow{V}_\iota = \pi_{o_{\bar{\iota}}}$. We define *merge from the left* as $\overrightarrow{\Lambda}_\iota = \pi_{o_\iota}^{-1}$, and *merge from the right* as $\overleftarrow{\Lambda}_\iota = \pi_{o_{\bar{\iota}}}^{-1}$. Thus, the following inverse identities hold: $\overrightarrow{\Lambda}_\iota \circ \overleftarrow{V}_\iota = \overleftarrow{\Lambda}_\iota \circ \overrightarrow{V}_\iota = id_{n+p}$. Examples of the four slabs are given in Fig. 8.

Note that another four similar slabs could have been defined using o_ι^{-1} instead of o_ι. However, we will have no use for them: because of the physical constraints of a knitting machine, lower-numbered yarn carriers must always cross over higher-numbered carriers.

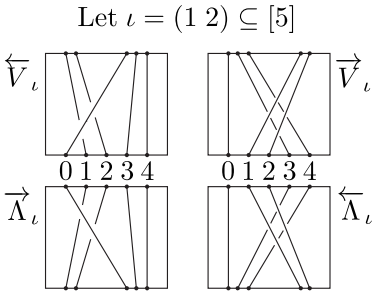


Fig. 8. Given the particular interleaving $\iota = (1\ 2)$ we can define two separate and two merge slabs, varying by the direction in which the yarns identified by ι are merged from or separated to. The arrows act as a mnemonic to tell us which direction the ι yarns are being pulled (reading the slab from bottom to top), and the character acts as mnemonic for whether the yarns are being merged (Λ) or separated (V).

5 FORMAL KNITOUT

Formal definitions of computer programming languages typically consist of at least three major parts: a *grammar* specifying the syntax of the language, a *type system* that further specifies which programs are "valid," and a *semantics* specifying the "meaning" of any valid program (for a more in-depth review of these concepts, we refer you to Appendix A.2). While knitout is a control language for knitting machines, not computers,[1] we can still use the same process to formalize it. We specify the grammar of formal knitout in Definition 5.1 using Backus-Naur form (BNF). In Definition 5.4, we define our type-checking relation $S \xrightarrow{ks} S'$ on abstract machine states S and S'. Not only does this allow us to restrict our attention

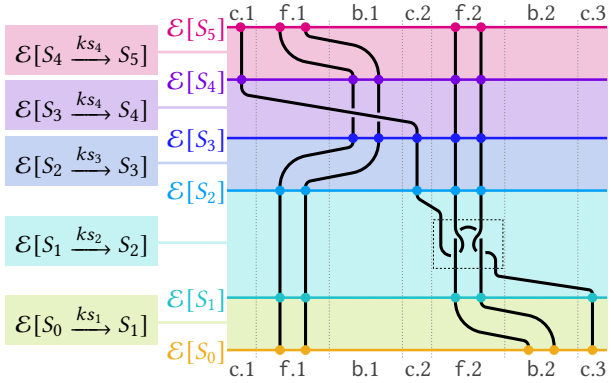
[1]Indeed, knitout does not contain, e.g., variables, function calls, or control flow.

```
1  xfer b.2 f.2;
2  knit - f.2 3.0 (2, 1.0);
3  xfer f.1 b.1;
4  miss - f.1 2;
5  xfer b.1 f.1;
```

(a) Formal knitout program (Definition 5.1)

$$S_0 = (0, [\texttt{f.1} \mapsto 1][\texttt{b.2} \mapsto 1], [2 \mapsto 3], [2 \mapsto \texttt{f.3}])$$
$$\downarrow \texttt{xfer b.2 f.2}$$
$$S_1 = (0, [\texttt{f.1} \mapsto 1][\texttt{f.2} \mapsto 1], [2 \mapsto 3], [2 \mapsto \texttt{f.3}])$$
$$\downarrow \texttt{knit - f.2 3.0 (2, 1.0)}$$
$$S_2 = (0, [\texttt{f.1} \mapsto 1][\texttt{f.2} \mapsto 1], [2 \mapsto 2], [2 \mapsto \texttt{f.2}])$$
$$\downarrow \texttt{xfer f.1 b.1}$$
$$S_3 = (0, [\texttt{b.1} \mapsto 1][\texttt{f.2} \mapsto 1], [2 \mapsto 2], [2 \mapsto \texttt{f.2}])$$
$$\downarrow \texttt{miss - f.1 2}$$
$$S_4 = (0, [\texttt{b.1} \mapsto 1][\texttt{f.2} \mapsto 1], [2 \mapsto 1], [2 \mapsto \texttt{f.2}])$$
$$\downarrow \texttt{xfer b.1 f.1}$$
$$S_5 = (0, [\texttt{f.1} \mapsto 1][\texttt{f.2} \mapsto 1], [2 \mapsto 1], [2 \mapsto \texttt{f.2}])$$

(b) Program trace defined by validity relations (Fig. 11)



(c) Denoted fenced tangle (Fig. 12)

Fig. 9. An excerpt of formal knitout code for knitting linen stitch (a) describes the mechanical actions performed by the machine, but is insufficient for describing the resulting knit topology. Executing the program on initial state S_0 produces a unique trace $S_0 \xrightarrow{kP} S_5$, which proves our program is well-formed (b). Each machine state denotes points on a slab's boundary, while the trace denotes the fenced tangle that connect said points (c).

to only valid formal knitout programs, the information contained in machine states S and S' is useful for defining the meaning of knitout programs (i.e., their semantics). We define the meaning of individual machine states $\mathcal{E}[S]$ in Definition 5.5 as an intermediary step to defining the fenced tangle denoted by a valid knitout program $\mathcal{E}[S \xrightarrow{ks} S']$ (Definition 5.6). For clarity, we do not include some knitout features in the formalization; these differences are explained in Appendix C. An example of our formal definitions applied to a specific program instance is found in Fig. 9.

Definition 5.1 (Knitout)**.** A knitout program ks is defined according to the following context free grammar:

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

Note that l is the size of a loop produced by a stitching operation and s is the length of yarn running between this stitch and the last stitch using said yarn. dir is the direction in which the carrier is moving when executing the operation.

Knitout programs refer to needle locations (on which loops are stored) and yarn carrier locations (at which loose ends of yarn are held). We make a distinction between logical and physical locations (Fig. 10). Knitout programs are written in terms of logical locations, but their validity and semantics are defined in terms of the physical locations. To organize these concepts and avoid confusion, we make the following definitions.

Definition 5.2 (Locations)**.**

- A *logical needle location* is a pair $n.x \in \mathsf{nLoc}$ where $\mathsf{nLoc} = \{\texttt{f}, \texttt{b}\} \times \mathbb{Z}$ is the set of all logical needle locations. Logical needle locations identify a "front bed" or "back bed" needle location.
- A *logical yarn carrier location* is a pair of a logical needle location and direction $(n.x, dir) \in \mathsf{ycLoc}$, where $\mathsf{ycLoc} = \mathsf{nLoc} \times \{+,-\}$. Intuitively, the direction identifies which side of a needle a yarn carrier is "parked at."
- A *physical needle location* is an integer $z \in \mathbb{Z}$. The physical location corresponding to a logical needle location $n.x$ at racking offset r is $\lfloor \texttt{f}.x \rfloor_r = x$ and $\lfloor \texttt{b}.x \rfloor_r = x + r$.
- A *physical yarn carrier location* is an integer $z \in \mathbb{Z}$. The physical location corresponding to a logical yarn carrier location $(n.x, dir)$ at racking offset r is defined as $\lfloor n.x, + \rfloor_r = \lfloor n.x \rfloor_r + 1$ and $\lfloor n.x, - \rfloor_r = \lfloor n.x \rfloor_r$. Intuitively, yarn carriers immediately to the left of physical needle location z are assigned physical location z, while yarn carriers immediately to the right of physical needle location z are assigned physical location $z+1$. You can think of these as actually sitting at $z - 0.5$ and $z + 0.5$. We use whole numbers for simplicity, and we will sometimes use the notation $c.z$ in diagrams for visual clarity.

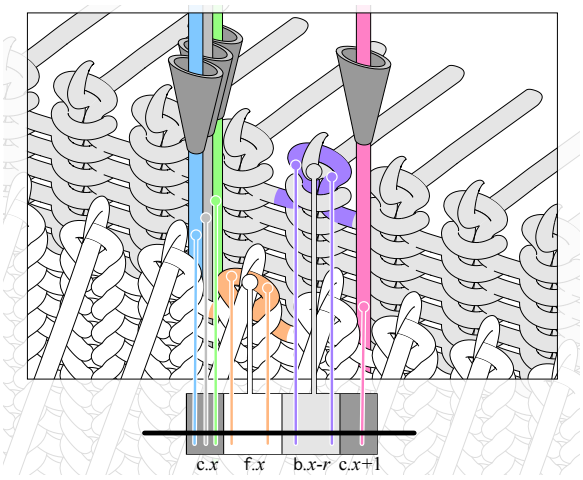


Fig. 10. The knitting machine consists of two beds of needles where at racking r, front bed needle $\texttt{f}.x$ is aligned with back bed needle $\texttt{b}.x - r$. In between the needles are yarn carrier tracks. These logical machine locations are projected from 2D to 1D physical locations using a left-to-right, front-to-back order, where each carrier projects to a single point and each loop projects to two points. These ordered points on a line are what is denoted by a given machine state $\mathcal{E}[S]$.

Each knitout operation creates yarn geometry and manipulates the machine state:

Definition 5.3 (Knitout Machine State)**.** A knitout machine state $S = (r, L, Y, A)$ consists of:

- $r \in \mathbb{Z}$, the racking offset, or the offset of the needles on the back bed relative to the front bed. At offset r, back needle $\texttt{b}.x - r$ is across from front needle $\texttt{f}.x$.
- $L \in \mathsf{nLoc} \to \mathbb{N}$, a partial function with default value 0 that reports the number of loops on each needle.
- $Y \in \mathbb{N} \to \mathbb{Z}$, a partial function that gives the current physical position of the yarn carriers. If the value is $\perp$ (the default value), then we say that the carrier is inactive.
- $A \in \mathbb{N} \to \mathsf{ycLoc}$ a partial function that gives the logical carrier location of where each yarn carrier is attached to a loop. An inactive carrier (with value $\perp$) is not attached.

We define the empty state as $S_\emptyset = (0, [], [], [])$. For a review of partial function notation, see Definition A.1.

Definition 5.4 ((Valid) Knitout Trace)**.** Given a knitout program ks and knitout machine states S, S', we say that executing ks on S produces S' if the relation $S \xrightarrow{ks} S'$ holds (as defined in Figure 11). As a shorthand, we may write $S_0 \xrightarrow{ks_1} S_1 \xrightarrow{ks_2} S_2$ for $S_0 \xrightarrow{ks_1\,;ks_2} S_2$, with the additional information that rule **V-seq** has been instantiated with intermediate state S_1. We also refer to such composite relations as *traces* of knitout programs. We say that a knitout program is *valid* or *well-formed* if it has a trace. We say that a valid knitout program ks is *complete* if it both begins and ends with the empty state $S_\emptyset \xrightarrow{ks} S_\emptyset$. Note that for a given initial state S and knitout statement ks, the resulting state S' is uniquely determined.

$$\lfloor \mathtt{f}.x \rfloor_r := x \qquad \lfloor n.x, + \rfloor_r := \lfloor n.x \rfloor_r + 1 \qquad (Y, yarns) =_r (n.x, dir) \;:=\; \forall y \in yarns :\; Y(y) = \lfloor n.x, dir \rfloor_r$$
$$\lfloor \mathtt{b}.x \rfloor_r := x + r \qquad \lfloor n.x, - \rfloor_r := \lfloor n.x \rfloor_r \qquad n.x \parallel_r n'.x' \;:=\; \lfloor n.x \rfloor_r = \lfloor n'.x' \rfloor_r \;\wedge\; n \neq n'$$

$$\frac{S \xrightarrow{ks_1} S' \qquad S' \xrightarrow{ks_2} S''}{S \xrightarrow{ks_1;ks_2} S''}\ \textbf{V-seq} \qquad \frac{}{S \xrightarrow{\mathtt{nop}} S}\ \textbf{V-nop} \qquad \frac{(Y, yarns) =_r (n.x, \neg dir) \qquad Y' = Y[yarns \mapsto \lfloor n.x, dir \rfloor_r]}{(r, L, Y, A) \xrightarrow{\mathtt{miss}\ dir\ n.x\ yarns} (r, L, Y', A)}\ \textbf{V-miss}$$

$$\frac{(Y, yarns) =_r (n.x, \neg dir) \qquad Y' = Y[yarns \mapsto \lfloor n.x, dir \rfloor_r]}{(r, L, Y, A) \xrightarrow{\mathtt{tuck}\ dir\ n.x\ l\ yarns} (r, L[n.x \mapsto L(n.x) + \#yarns], Y', A[yarns \mapsto n.x])}\ \textbf{V-tuck}$$

$$\frac{(Y, yarns) =_r (n.x, \neg dir) \qquad L(n.x) > 0 \qquad \begin{array}{c} Y' = Y[yarns \mapsto \lfloor n.x, dir \rfloor_r] \\ A' = A[yarns \mapsto n.x] \end{array}}{(r, L, Y, A) \xrightarrow{\mathtt{knit}\ dir\ n.x\ l\ yarns} (r, L[n.x \mapsto \#yarns], Y', A')}\ \textbf{V-knit} \qquad \frac{|r - r'| = 1}{(r, L, Y, A) \xrightarrow{(\mathtt{rack}\ r')} (r', L, Y, A)}\ \textbf{V-rack}$$

$$\frac{n.x \parallel_r n'.x' \qquad L' = L[n.x \mapsto 0][n'.x' \mapsto L(n.x) + L(n'.x')]}{(r, L, Y, A) \xrightarrow{\mathtt{xfer}\ n.x\ n'.x'} (r, L', Y, A[\{y : A(y) = n.x\} \mapsto n'.x'])}\ \textbf{V-xfer} \qquad \frac{Y(n.x) > 0}{(r, L, Y, A) \xrightarrow{\mathtt{drop}\ n.x} (r, L[n.x \mapsto 0], Y, A)}\ \textbf{V-drop}$$

$$\frac{(Y, yarns) =_r (n.x, \neg dir) \qquad n.x \parallel_r n'.x' \qquad \begin{array}{c} Y' = Y[yarns \mapsto \lfloor n.x, dir \rfloor_r] \\ A' = A[\{y : A(y) = n.x\} \mapsto n'.x'][yarns \mapsto n.x] \end{array}}{(r, L, Y, A) \xrightarrow{\mathtt{split}\ dir\ n.x\ l\ n'.x'\ yarns} (r, L[n.x \mapsto \#yarns][n'.x' \mapsto L(n.x) + L(n'.x')], Y', A')}\ \textbf{V-split}$$

$$\frac{Y(y) = \bot \qquad \begin{array}{c} Y' = Y[y \mapsto \lfloor n.x, dir \rfloor_r] \\ A' = A[y \mapsto n.x] \end{array} \qquad A(y) = \bot}{(r, L, Y, A) \xrightarrow{\mathtt{in}\ dir\ n.x\ y} (r, L, Y', A')}\ \textbf{V-in} \qquad \frac{Y(y) = \lfloor n.x, dir \rfloor_r \qquad \begin{array}{c} Y' = Y[y \mapsto \bot] \\ A' = A[y \mapsto \bot] \end{array} \qquad A(y) \neq \bot}{(r, L, Y, A) \xrightarrow{\mathtt{out}\ dir\ n.x\ y} (r, L, Y', A')}\ \textbf{V-out}$$

Fig. 11. Validity relation for knitout programs (see Definition 5.4), where $\#yarns$ is the size of the yarn carrier sequence. Only valid knitout programs denote a fenced tangle. Note that for a fixed S and ks, S' is uniquely determined.

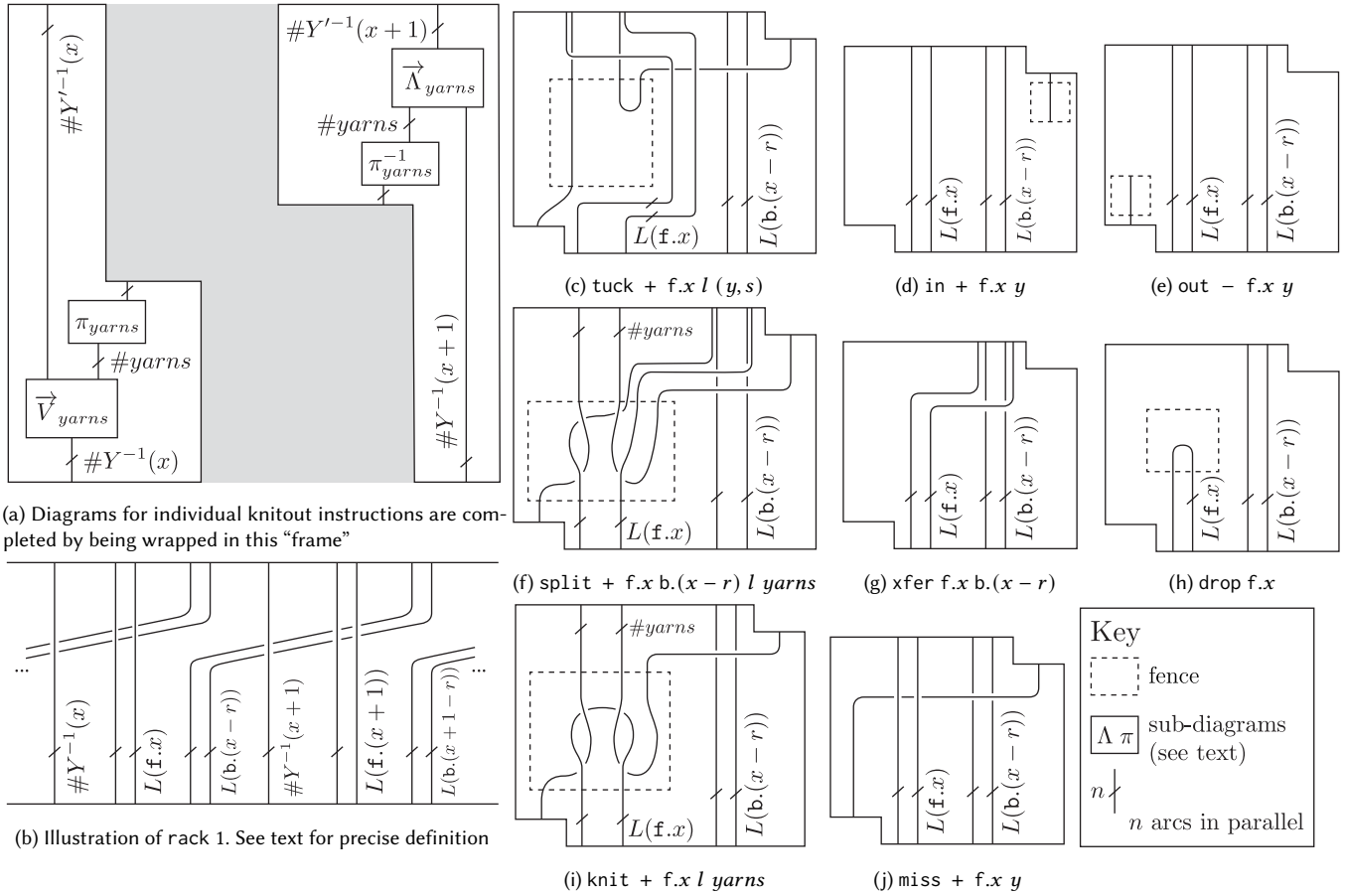
Definition 5.5 (Machine State Denotation)**.** Let $S = (r, L, Y, A)$ be a machine state. Then $\mathcal{E}[S]$, the denotation of S, is a set of points on a line, which is divided into annotated segments as follows (also see Figure 10, bottom):

- for each $i \in \mathbb{Z}$ there is a yarn carrier segment for physical yarn carrier location i, followed by a front needle segment for physical needle location i, followed by a back needle location segment for physical needle location i (corresponding to logical location $i - r$).
- for each $k \in \mathbb{N}$ with $Y(k) \neq \bot$, there is a point in yarn carrier segment $\lfloor Y(k) \rfloor_r = i$. This point is the j^{th} point if there are $(j-1)$ yarns with $l < k$ and $\lfloor Y(l) \rfloor_r = i$.
- for each $nl = (n.x) \in \mathsf{nLoc}$ with $L(nl) = k$, there are $2k$ points in the segment corresponding to needle location $\lfloor nl \rfloor_r$ on the n bed. (These are the k loops on needle nl)

Definition 5.6 (Semantics of Knitout)**.** Let $kT = S_0 \xrightarrow{ks_1} S_1 \rightarrow \cdots \rightarrow S_n$ be a valid knitout program/trace. Then $\mathcal{E}[kT]$ is the fenced tangle which kT denotes, defined inductively. Throughout the definition, we will work with the slab presentation of fenced tangle diagrams. As an invariant, the input (bottom) boundary of $\mathcal{E}[S \xrightarrow{ks} S']$ will match $\mathcal{E}[S]$ and the output (top) boundary will match $\mathcal{E}[S']$.

First, we will address the inductive case. $\mathcal{E}[S \xrightarrow{ks_1} S' \xrightarrow{ks_2} S'']$ is defined to be the vertical concatenation of the two slabs $\mathcal{E}[S \xrightarrow{ks_1} S'] \circ \mathcal{E}[S' \xrightarrow{ks_2} S'']$. This composed diagram is well-defined because its constituent diagrams are well-defined (by induction) and because their shared boundary must identically be $\mathcal{E}[S']$ (by invariant).

The nop instruction does nothing, so $\mathcal{E}[S \xrightarrow{\mathsf{nop}} S] = id$. Next, we handle the rack instruction. Let $kT = S \xrightarrow{\mathsf{rack}\ r} S'$. We define $\mathcal{I}_{<\infty}[S]$ to be the partition of $\mathcal{E}[S]$ into (on the one hand) all yarn carrier points and loop points corresponding to front (f) needle locations, and (on the other hand) all loop points corresponding to back (b) needle locations. We then let $\iota = \mathcal{I}_{<\infty}[S] \in \mathcal{I}_{m,n}$ be the initial interleaving of front-bed loops and yarn carriers on the one hand, with the back-bed loops on the other, and let $\omega = \mathcal{I}_{<\infty}[S'] \in \mathcal{I}_{m,n}$ be the similar final interleaving after the racking operation. Note that by the validity of traces, these partition sizes must match. Then, we define the racking denotation as $\mathcal{E}[kT] = id_m|_\iota^\omega id_n$. (see Fig. 12b for an example illustration)



(a) Diagrams for individual knitout instructions are completed by being wrapped in this "frame"

(b) Illustration of rack 1. See text for precise definition

Fig. 12. Fenced tangles produced by knitout. Part of the definition of knitout semantics (Definition 5.6). Other than rack, all diagrams are wrapped by the "frame" diagram, which defines how the yarn carriers being used in an instruction (*yarns*) are merged (Λ) separated (V) and how they are plated (π). State variables (r, Y, L) are all given with respect to the initial state before an instruction, except for Y' in the frame diagram, which refers to the state after the instruction is done. Note that a group of arcs in parallel annotated as 0-many will disappear from the diagram. Also note that all diagrams here are given for the positive/right-ward knitting direction (+) and in the front-facing variant. The left-ward, back-facing diagrams are flips of these diagrams; and the other two cases are derived via a careful mirroring of the diagrams. All other instruction variation is parametric.

For the remaining operations with trace $kT = S \xrightarrow{ks} S'$, all non-trivial (i.e., not *id*) effects will be restricted to a particular physical needle location x, and its interactions with the yarns immediately to the left and right of the needle (yarn locations x and $x+1$). Given the set of points $\mathcal{E}[S]$, we define $\{\mathcal{E}[S] < pl\}$ to be the subset of all points that correspond to a physical location less than pl, while $\{\mathcal{E}[S] > pl\}$ is all points greater than pl. An examination of the validity relation definition (Fig. 11) makes it clear that $\{\mathcal{E}[S] < \lfloor n.x, -\rfloor_r\} = \{\mathcal{E}[S'] < \lfloor n.x, -\rfloor_r\}$ and $\{\mathcal{E}[S] > \lfloor n.x, +\rfloor_r\} = \{\mathcal{E}[S'] > \lfloor n.x, +\rfloor_r\}$. Thus the denotation of kT can be expressed as $\mathcal{E}[kT] = id_m \otimes T_s \otimes id_n$, where $m = \#\{\mathcal{E}[S] < \lfloor n.x, -\rfloor_r\}$, $n = \#\{\mathcal{E}[S] > \lfloor n.x, +\rfloor_r\}$, and T_s is defined for each operation according to figure 12.

6 TOPOLOGICALLY CORRECT KNITOUT REWRITES

Having defined a formal semantics on knitout using fenced tangles, we can now define what it means for two knitout programs to be topologically equivalent, and use this equivalence to prove correctness of program rewrites. In this section, we focus specifically on swapping the execution order of two knitout operations so we can understand the tools available to us as we both compare specific program instances and prove general rewrite rules; motivation for performing this program transformation is deferred to Section 7. We start by defining what we mean for two knitout programs to be topologically equivalent.

Definition 6.1 (Topological Equivalence of Valid Knitout Programs)**.** Let ks_1 and ks_2 be (partial) knitout programs. If both programs are valid on starting state S and take it to state S' (i.e., $S \xrightarrow{ks_1} S'$ and $S \xrightarrow{ks_2} S'$) and these traces denote the same tangle, $\mathcal{E}[S \xrightarrow{ks_1} S'] \cong \mathcal{E}[S \xrightarrow{ks_2} S']$, we say that ks_1 and ks_2 are equivalent in the context of S and write:

$$S \vdash ks_1 \cong ks_2$$

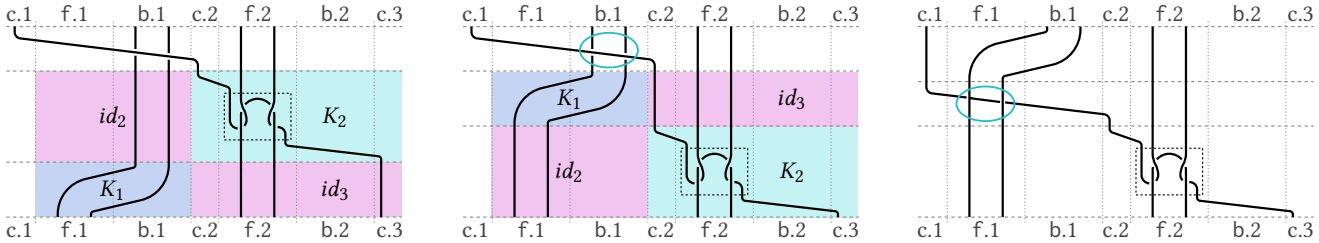


Fig. 13. The fenced tangle diagrams denoted by programs (a) and (b) are topologically equivalent. The diagram transformation is a simple application of ambient isotopy, and their equivalence can also be proven using Lemma 6.5. In contrast, fenced tangles (b) and (c) are not equivalent due to the change in crossing annotations in the circled region.

Corollary 6.2 (Local Rewrites). *Let $ks_1; ks_2; ks_3$ and $ks_1; ks_2'; ks_3$ be two valid knitout programs, where $S \xrightarrow{ks_1} S'$. If $S' \vdash ks_2 \cong ks_2'$, then $S \vdash ks_1; ks_2; ks_3 \cong ks_1; ks_2'; ks_3$.*

6.1 Proving Fenced Tangle Equivalence

Let us consider the example program shown in Fig. 9, specifically the subprogram $ks_2; ks_3; ks_4$:

```
2  knit - f.2 3.0 (2, 1.0);
3  xfer f.1 b.1;
4  miss - f.1 2;
```

In figure 13 we see the tangle denoted by reordered sub-programs $S_1 \xrightarrow{ks_3;ks_2;ks_4} S_4$ and $S_1 \xrightarrow{ks_2;ks_4;ks_3} S_4$ (note that in this specific example, the knitout trace for both rewrites is valid, but that is not necessarily true for all knitout programs). We see that Fig. 13a can be transformed into Fig. 13b by an ambient isotopy. By contrast, Fig. 13c and Fig. 13b have different crossings between the loop at b.1 and carrier 2 (circled). These diagrams can't be transitioned between using any combination of Reidemeister moves and ambient isotopies. Thus the first pair of fenced tangle diagrams prove that $S_1 \vdash ks_2; ks_3 \cong ks_3; ks_2$ and the second pair seem to strongly suggest that $S_2 \vdash ks_3; ks_4 \ncong ks_4; ks_3$.

Note, however, that these three tangle diagrams are the denotations of these specific three program fragments executed on a specific machine state. Proving that two slightly different program fragments are equivalent would require a new sequence of fenced tangle diagrams, and the correct sequence of Reidemeister moves may be less trivial. While we can (and do) use templated tangle diagrams akin to the ones used in Fig. 12, a purely diagrammatic approach quickly becomes intractable as fenced tangle complexity increases. Fortunately, fenced tangle composition is not only useful for defining fenced tangles, but also for proving topological equivalence. For example, let us consider the following two lemmas (proof left as an exercise for the reader):

Lemma 6.3. *For any fenced tangle slab $K \in \mathcal{S}_n^m$, vertical concatenation of the identity results in an equivalent fenced tangle:*

$$id_n \circ K \cong K \cong K \circ id_m$$

Lemma 6.4 (∘-⊗ Distributivity). *Let $K_a \in \mathcal{S}_{n_1}^{m_1}$ and $K_b \in \mathcal{S}_{m_1}^{p_1}$ be one pair of vertically composable fenced tangles, and $K_c \in \mathcal{S}_{n_2}^{m_2}$ and $K_d \in \mathcal{S}_{m_2}^{p_2}$ be a second pair. Then the following compositions are equivalent:*

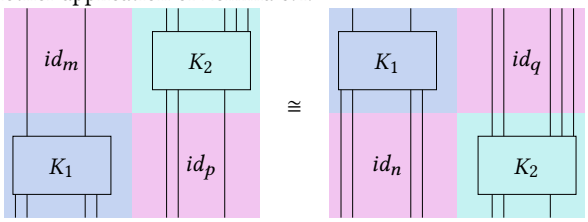
$$(K_a \circ K_b) \otimes (K_c \circ K_d) \cong (K_a \otimes K_c) \circ (K_b \otimes K_d)$$

These lemmas can then be used to prove a general statement about commutativity of horizontally separated sub-tangles:

Lemma 6.5 (Commutativity by Horizontal Separation). *For any $K_1 \in \mathcal{S}_n^m$ and $K_2 \in \mathcal{S}_p^q$ the following equation holds:*

$$(K_1 \otimes id_p) \circ (id_m \otimes K_2) \cong (id_n \otimes K_2) \circ (K_1 \otimes id_q)$$

Proof. We begin by using Lemma 6.4 to rewrite $(K_1 \otimes id_p) \circ (id_m \otimes K_2)$ into $(K_1 \circ id_m) \otimes (id_p \circ K_2)$. Lemma 6.3 can then be used to slide K_1 up and K_2 down to produce fenced tangle $((id_n \circ K_1) \otimes (K_2 \circ id_q))$, which is congruent to $(id_n \otimes K_2) \circ (K_1 \otimes id_q)$ by another application of Lemma 6.4.



□

Many knitout operations denote (Definition 5.6) a tangle of the form $id \otimes K \otimes id$; and knitout program composition maps to vertical composition (∘) of fenced tangles. Thus, intuitively, we should be able to use Lemma 6.5 to prove the correctness of swapping some, but not all, pairs of operations. In fact, we can go one step further and define an extent function $\text{ex}(ks)$ (Definition D.3) that maps any valid knitout program to a rectangle $[R_{xmin}, R_{xmax}] \times [R_{ymin}, R_{ymax}]$ that contains the non-*id* part of its fenced tangle. This rectangle can not only be used to generate the horizontal decomposition of $\mathcal{E}[S \xrightarrow{ks} S']$, but its depth-wise decomposition as well, for which we prove a similar commutativity property using Lemma B.9. Using this extent function, we can state the following generalized Rewrite Rule for swapping knitout subprograms:

Rewrite Rule 1 (Swap). Two operations can be swapped if their extents are disjoint: $S \vdash ks_1; ks_2 \cong ks_2; ks_1$ whenever $\text{ex}(ks_1) \cap \text{ex}(ks_2) = \emptyset$

If we return to our example program rewrite $S_1 \vdash ks_2; ks_3 \cong ks_3; ks_2$, we find that $\text{ex}(ks_2) = [1.5, 2.5] \times [2, \infty]$ and $\text{ex}(ks_3) = \{1\} \times [-\infty, \infty]$, making it an example covered by Rewrite Rule *Swap*. Meanwhile, $\text{ex}(ks_4) = [1.5, 2.5] \times \{2\}$ intersects with $\text{ex}(ks_3)$ in both dimensions. Thus Rewrite Rule *Swap* cannot be applied. By proving small, general statements on program equivalence that can be applied within a larger context, we develop a powerful tool for reasoning about the correctness of more complicated program transformations.

7 IMPLEMENTING A REWRITE-EDITOR

Now that we've demonstrated the importance of a suite of low-level rewrite rules as well as how we can prove their correctness, it is natural to ask what rewrite rules we should prove. A reasonable starting point would be rewrite rules useful to practical high-level compilation tasks. In the following section we examine some common motivations for rewriting programs and provide an overview of the rewrite rules we validated. A high-level summary of our rewrites and their corresponding proofs in the appendix are located in Table 1.

7.1 Rewrite Motivations

7.1.1 Fabrication Time. Recall that the knitting machine has two rows of needles known as beds and a larger piece called the *carriage* that moves along the needle bed and actuates individual operations via a cam system. Each movement of the carriage along the bed is known as a *carriage pass*, and depending on the machine's particular cam sets, different operations can be grouped into a single pass. The amount of time required for a carriage pass is roughly independent of the number of needle operations it contains. This is because much of the pass consists of a constant acceleration/deceleration phase, and the carriage can actuate any needles it passes over at no additional cost. Thus when optimizing a knitting program to reduce fabrication time, the goal is not necessarily to minimize operation count, but to change when operations are executed such that pass count is minimized. Rewrite Rule 1 (*Swap*) is used for changing operation order. In addition to the extent analysis we performed on general subprograms in Definition D.2, we perform a special case analysis of the `SHIFT` macro used in Rewrite Rule 5 (Lemma D.8).

Table 1. Rewriting equivalences (rules) proven in this paper. Rules with asterisks have preconditions not present in this figure (see associated proof).

Name	Rule	Proof
*Swap**	ks_1 ks_2 $\cong$ ks_2 ks_1	§D.1
*Merge**	ks_1 ks_2 $\cong$ nop	§D.2
Squish	xfer *n.x n′..x′* xfer *n′.x n.x* $\cong$ xfer *n′.x′ n.x*	§D.2
Slide	tuck *dir n.x* (y, s) xfer n.x n'.x' $\cong$ tuck *dir n′.x′* (y, s) xfer n.x n'.x'	§D.3
*Conjugate**	miss − f.x − 1 *yarns* SHIFT(f.$x, r, -1$) $ks(+, \text{f}.x) \cong ks(+, \text{f}.x - 1)$ miss + f.x *yarns* SHIFT(f.$x - 1, r - 1, 1$)	§D.3

7.1.2 Program Reliability. While knitting machines are generally quite robust, any operation has some chance of failure. For example, repeated `rack` operations may introduce excess strain on yarn, while `xfer` operations may not cleanly send all loops from source needle $n.x$ to destination needle $n'.x'$. Thus one aspect of improving knit program reliability is to remove unnecessary operations. Rewrite Rule *Merge* does this by considering pairs of operations that are clear inverses:

Rewrite Rule 2 (Merge)**.** Racking in one direction and then back in the other direction is the same as doing nothing.

$$S \vdash (\texttt{rack}\ (r \pm 1);\ \texttt{rack}\ r) \cong \texttt{nop}$$

where r is the initial racking value in S.

Missing at $n.x$ in one direction and then back in the other is the same as doing nothing.

$$S \vdash (\texttt{miss}\ dir\ n.x\ y;\ \texttt{miss}\ \neg dir\ n.x\ y) \cong \texttt{nop}$$

Similarly, Rewrite Rule *Squish* considers how pairs of aligned `xfer` operations cancel:

Rewrite Rule 3 (Squish)**.**

$$S \vdash \texttt{xfer}\ n.x\ n'.x'; \texttt{xfer}\ n'.x'\ n.x \cong \texttt{xfer}\ n'.x'\ n.x$$

Furthermore, when $L(n'.x) = 0$ in initial state S,

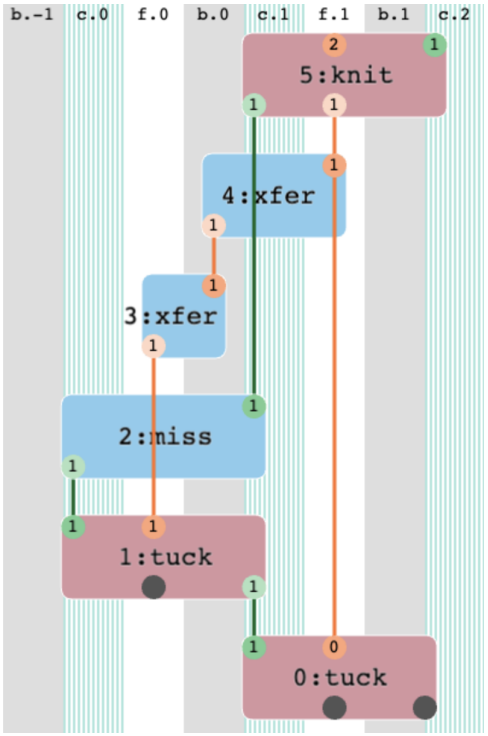
$$S \vdash \texttt{xfer}\ n'.x'\ n.x \cong \texttt{nop}$$

7.1.3 Machine Specific Compatibility. So far, our formalism has assumed an abstract knitting machine with infinitely wide needle beds that can be racked to any value, as well as infinitely many carriers. As a result, there will always be enough space to execute a valid knitting program. In practice, the number of needles and carriers is finite (typically on the order of 10^3 and 10 respectively), and the beds cannot be racked infinitely. These machine constraints can be formalized as follows:

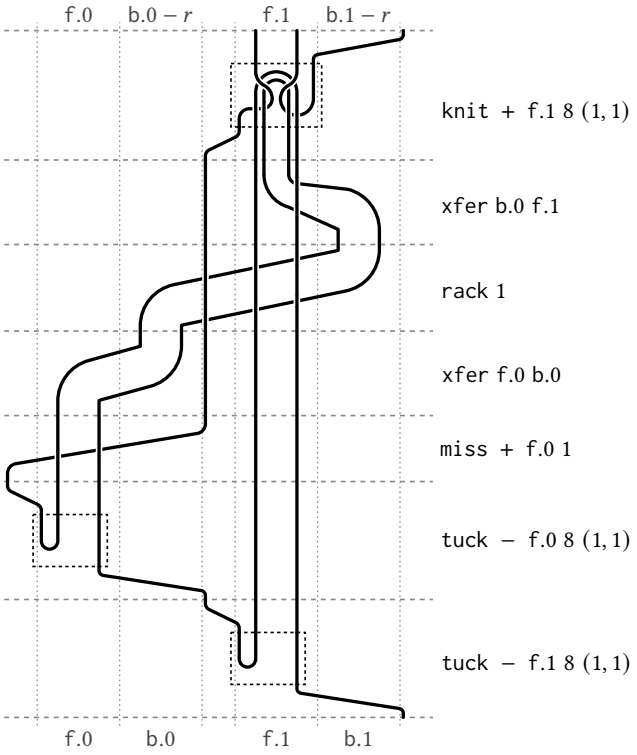
Needle and carrier sets A machine has a finite set of available needles and carriers. Thus, only needles in the range $[x_{min}, x_{max}]$ exist. The loop count state function $L : \text{nLoc} \to \mathbb{N}$ must be zero for any $n.x$ with x outside of $[x_{min}, x_{max}]$. There are also a finite number of yarn carriers y_{count}. So, the yarn carrier state must be a partial function $Y : [y_{count}] \to [x_{min}, x_{max}]$.

Racking Valid racking is constrained to range $[r_{min}, r_{max}]$.

In addition, our semantics is geared towards defining topological correctness; thus it makes no use of loop size parameter l and yarn length parameters s, which control the amount of yarn used for operations. However, specific machines are not only limited to certain l and s values; they have validity conditions that are quite complicated and often state dependent. For example, while yarn may stretch and slide a small amount, it will eventually break when stretched too far. This means that the validity of yarn length parameter s depends on which loops it is attached to. While fully capturing this logic is beyond the scope of this paper, we can define the following basic metric constraints to ensure physical plausibility:



(a) A screenshot of our rewrite-editor. Knitout instructions are shown as nodes in a graph, while loops and yarns are shown as edges.



(b) The fenced tangle and knitout code corresponding to the screenshot. Note that knitout code is read from bottom up to match the fenced tangle presentation. The yarn carrier id is 1 and the yarn lengths are all 1 needle spacing unit.

Fig. 14. Knitout code shown in our rewrite-editor and the corresponding fenced tangle.

Needle width All needles have some width l_{min} that serves as a lower bound for the set of valid loop sizes.

Needle spacing Yarn length y must be greater than the physical distance between the operation and its attach point. Put formally, for each knitout trace $S \xrightarrow{ks} S'$ where ks is a single operation with needle argument $n.x$ and yarn carrier sequence $yarns$, $\forall (y, s) \in yarns : \lambda |Y(y) - A(y)| < s$, where λ is the spacing between needles.

Critically, it is necessary to rewrite a program in a way that preserves the denoted fenced tangle, but changes the elements of the machine state upon which feasible length construction depends. I.e., the needle locations of loops (L), attach points (A) of yarn carriers, and racking (r) when each operation is executed. Changing machine racking can be trivially accomplished with a sequence of `rack` operations, and loops can be moved to the opposite bed with a single `xfer`. This is useful for changing the needle location where `tuck` operations are performed:

Rewrite Rule 4 (Slide)**.** Let $n.x$ and $n'.x'$ be defined such that they are the pair $\mathtt{f}.z$ and $\mathtt{b}.z - r$, or the pair $\mathtt{b}.z - r$ and $\mathtt{f}.z$. Then let $ks(n.x) = \mathtt{tuck}\ dir\ n.x\ l\ (y, s)$.

$$S \vdash ks(n.x);\ \mathtt{xfer}\ n.x\ n'.x' \cong ks(n'.x');\ \mathtt{xfer}\ n.x\ n'.x'$$

Note that this rule does not apply to the `knit` operation. This is because changing which bed a `knit` operation occurs on changes its structure. Moving a loop to a needle on the same bed requires a more involved series of operations:

Definition 7.1 (Rack and Shift Macros)**.** Let

$$\mathtt{RACK}(r, j) := \mathtt{rack}\ r + 1; \mathtt{rack}\ r + 2; \ldots; \mathtt{rack}\ r + j$$

be a knitout program that racks j times to the right starting at racking position r; if $j < 0$, then similarly let `RACK` expand to a sequence of decrementing `rack` instructions. Furthermore, let

$$\begin{aligned} S \vdash \mathtt{SHIFT}(\mathtt{f}.x, r, j) &\cong \mathtt{xfer}\ \mathtt{f}.x\ \mathtt{b}.(x - r); \mathtt{RACK}(r, j); \\ &\quad\ \mathtt{xfer}\ \mathtt{b}.(x - r)\ \mathtt{f}.(x + j) \\ S \vdash \mathtt{SHIFT}(\mathtt{b}.x, r, \mathtt{b}.(x + j)) &\cong \mathtt{xfer}\ \mathtt{b}.x\ \mathtt{f}.(x + r); \mathtt{RACK}(r, -j); \\ &\quad\ \mathtt{xfer}\ \mathtt{f}.(x + r)\ \mathtt{b}.(x + j) \end{aligned}$$

be a knitout program that transfers loops from any one needle to any one other needle on the same bed by using an intermediate needle on the opposite bed.

The `SHIFT` macro and `miss` instructions can be combined to route loops and yarn carriers to a new physical location, where an operation can be performed before re-routing everything back to produce

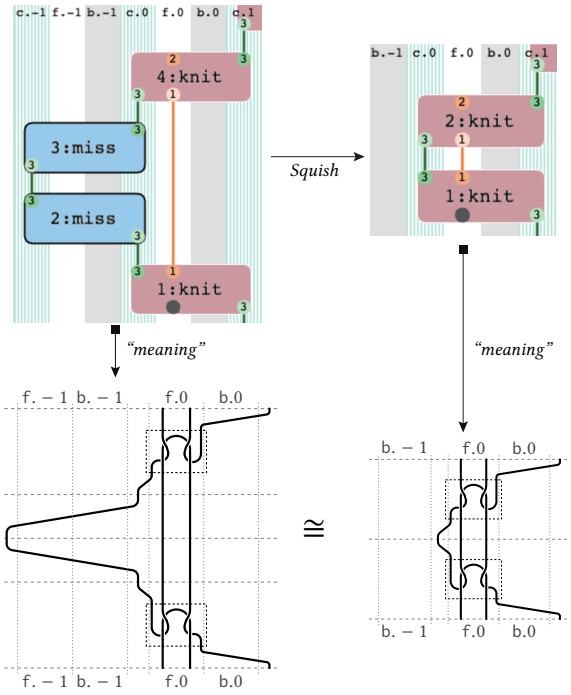


Fig. 15. Screenshots of the rewrite-editor for *Squish* rewrite rule and the corresponding fenced tangles.

the same ending state. The correct sequence of routing operations is non-trivial to describe and dependent on the operation's initial bed $\{\mathtt{f},\mathtt{b}\}$, its *dir* parameter $\{+,-\}$, and whether the physical needle location is incremented or decremented $\{\text{Right},\text{Left}\}$. Thus for clarity, we present only one of six cases here:

Rewrite Rule 5 (Conjugate [f, +, Left])**.** Let $ks(dir, n.x)$ be either a `knit` or `tuck` instruction $ks(dir, n.x)$ = `knit` *dir n.x l yarns* or $ks(dir, n.x)$ = `tuck` *dir n.x l* (y, s) (we will simply refer to (y, s) as *yarns* in the `tuck` case). Let S be the state prior to ks. If the following needles are empty $L(\mathtt{b}.x - r) = 0$, $L(\mathtt{f}.x - 1) = 0$, and if there are no yarn carriers in the way that we are not using $Y^{-1}(\lfloor \mathtt{f}.x, -\rfloor_r) = yarns$, then

$$S \vdash ks(+, \mathtt{f}.x) \cong \mathtt{miss - f}.x - 1\ yarns;\ \mathtt{SHIFT(f}.x, r, -1);$$
$$ks(+, \mathtt{f}.x - 1);$$
$$\mathtt{miss + f}.x\ yarns;\ \mathtt{SHIFT(f.x-1,\ r-1,\ 1)}$$

(where `miss` on multiple *yarns* is simply a sequence of `miss` operations, one for each yarn)

7.2 Knitout Editor Implementation

We implemented an editor for applying our rewrite rules to formal knitout programs. The editor is written in JavaScript and runs on a browser. The interface implements common useful interactions such as multi-select, zoom, drag, etc. The interface of the rewrite-editor is shown in Fig. 14a, and the corresponding fenced tangles and the formal knitout code is shown in Fig. 14b.

Because knitout is time monotonic by definition, it can be visualized as an upward time-dependent graph. Each knitout instruction is visualized as a block that spans needle locations which the instruction uses. For example, 13:tuck shows an instruction that is a `tuck` operation, and is the 13th operation in the program. Note that numbers such as 13 are timestamps, not unique instruction IDs. Therefore they can change after the rewrites. The rewrite-editor visualizes all the knitout instructions except for the `rack` instruction. Instead, the machine's racking value is tracked for each instruction internally.

Instruction nodes are augmented with orange circles such as 1, which annotate each loop with an id and the location of the incoming and outgoing loop, and green circles such as 3, which visualize the yarn carrier id and the location of the incoming and outgoing yarn. Empty loops and yarns are visualized as gray circles. When one loop or yarn connects two instructions, we draw a vertical dependency line with the corresponding color.

For each needle location, the front bed is visualized as a white column and the back bed is visualized as a grey column. The yarn carrier location exists on both sides of the needle locations, and is visualized as a green column. Program rewrites are performed by selecting instructions followed by the appropriate rule (Fig. 15). The rewrite is applied only if it is correct given the program context.

8 RESULTS

To demonstrate the expressivity of the rewrite rules, we programmed four examples using the rewrite-editor and knit them on a Shima Seiki SWG091N2 (15 gauge) two-bed knitting machine. All the inputs to the rewrite-editor were either handwritten or produced by simple JavaScript code, and all the rewrites were performed on the rewrite-editor to produce the output. Since the rewrite rules apply on individual instructions, scheduling large knitout programs can be challenging. Thus, we did rewrites on small versions of each knitout program and then expanded them using a Python script that duplicates and enlarges the program. Length annotation parameter units are based on the spacing between needles (approximately 1.7mm on our machine). Thus a length annotation of 1 can be respected if the yarn connects adjacent needle location, but is invalid if the loops are two needles apart. Loop length parameters are kept constant throughout the examples at 8 (which we decided would correspond to our machine's default loop length setting).

8.1 Pass Optimization

When teaching, we have noticed that machine knitting novices tend to write knitout that is correct but inefficient. For example, when writing knitout instructions to back bed `knit` (a 'purl' in hand-knitting) several loops held on the front bed, a novice will write a transfer-knit-transfer sequence for each loop. This sequence is visualized on the rewrite editor in Fig. 16 (left). Such per-stich interleavings are inefficient because `knit` and `xfer` operations require separate carriage passes. Re-ordering the code to group `knits` and `xfers` into separate blocks results in fewer passes and a shorter knitting time.

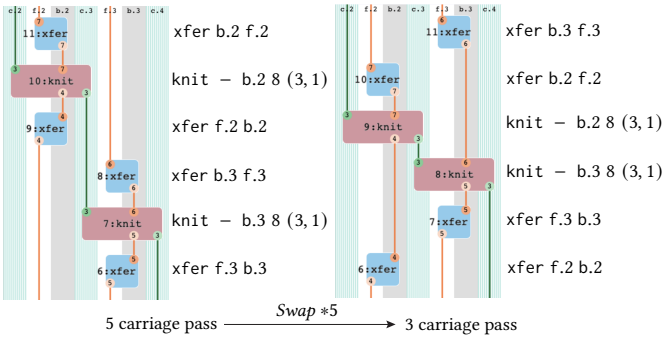


Fig. 16. Rewrite-editor screenshot and the corresponding knitout code of the pass optimization example. See Definition 5.1 for the formal knitout syntax. Left is typical knitout that novices tend to write, which is correct but inefficient due to unnecessary carriage passes. After applying the *Swap* rewrite rule five times, we can consolidate `knit` and `xfer` instructions so that the number of carriage passes becomes three. This is a small example, but the impact of the optimization increases as the size of the program gets larger.

We optimized the carriage passes by applying a sequence of *Swap* operations on the original knitout code. Fig. 16 shows a small example of how such optimization can be done in the rewrite-editor using the following sequence of *Swap* operations. Note that numbers in nodes are not unique IDs but are timestamps.

(1) *Swap* (`8:xfer`, `9:xfer`)
(2) *Swap* (`9:xfer`, `10:knit`)
(3) *Swap* (`10:xfer`, `11:xfer`)
(4) *Swap* (`7:knit`, `8:xfer`)
(5) *Swap* (`6:xfer`, `7:xfer`)

The input knit structure had 60 rows and 30 columns. Before scheduling, there were two `knit-xfer` switches per row and column. Therefore, the initial number of passes was $2 * 60 * 30 = 3600$. After applying the rewrite *Swap*, there are only four `knit-xfer` switches per row, because `xfers` and `knits` are consolidated across columns. Therefore, the number of passes is $60 * 4 = 240$.

The manufacturer's design software for our knitting machine [Shima Seiki 2011] estimates the original code's runtime at 50 minutes 30 seconds while the optimized version needs only 3 minutes 26 seconds. The rewrite optimized version is 14.7x faster, which roughly corresponds to the ratio of the number of passes, which is $3600/240 = 15$.

8.2 Full to Half Gauge

Consider a tightly knit sheet of knit fabric, constructed on a contiguous sequence of machine needles. The same sheet can also be produced by using needles that are further spaced out, for example using every other needle (i.e., on 'half-gauge'). While this change in gauge affects the ability of a machine to respect yarn length parameters, the topology of the underlying structure remains intact. Adjusting the gauge and moving instructions to desired locations while preserving topological equivalence is a ubiquitous task in machine knitting. Given this, we demonstrate how our rewrite rules can be used to transform a full-gauge fabric to half-gauge.

(a) Full-gauge sheet (b) Sheet transformed to half-gauge

(c) Full-gauge tube (d) Tube transformed to half-gauge

Fig. 17. Examples of sheets and tubes converted from full gauge to half gauge using rewrite rules to guarantee topological equivalence.

In the following example, we use a rewrite sequence pattern for moving the instructions to a neighboring needle, which is illustrated in Fig. 18. We first apply the rule *Conjugate Right* to two `knit` instructions `3:knit` and `0:knit`. *Conjugate Right* will insert `misses` and `xfers` as described in section Section 7. Then, we *Swap* the `xfers` until they are next to each other and apply *Squish* to cancel redundant `xfers`.

We scheduled a full gauge sheet (Fig. 17a) to a half gauge sheet (Fig. 17b), and a full gauge tube (Fig. 17c) to a half gauge tube (Fig. 17d) by moving each `knit` and `tuck` instruction to the right. Note that the half-gauge examples are wider than their full-gauge despite having the same topology. This is because the increased spacing in the half gauge example prevents the annotated yarn length from being respected.

8.3 Sheet Stacking

Recall the example discussed in the introduction, where a novice attempted to reschedule two sheets with interleaved construction passes so that instead of lying adjacent on the machine, one sheet was directly in front of the other. We scheduled two separate, adjacent sheets (Fig. 1a) so that they were correctly stacked (Fig. 1b) using the rewrite-editor. We performed this scheduling task by first moving all the knit instructions in the back bed sheet to use the same physical needle locations as the front bed sheet, using the same sequence of rewrite rules as Fig. 18. Then, we used the *Swap* rule to swap `knit` instructions until the sheets were correctly interleaved.

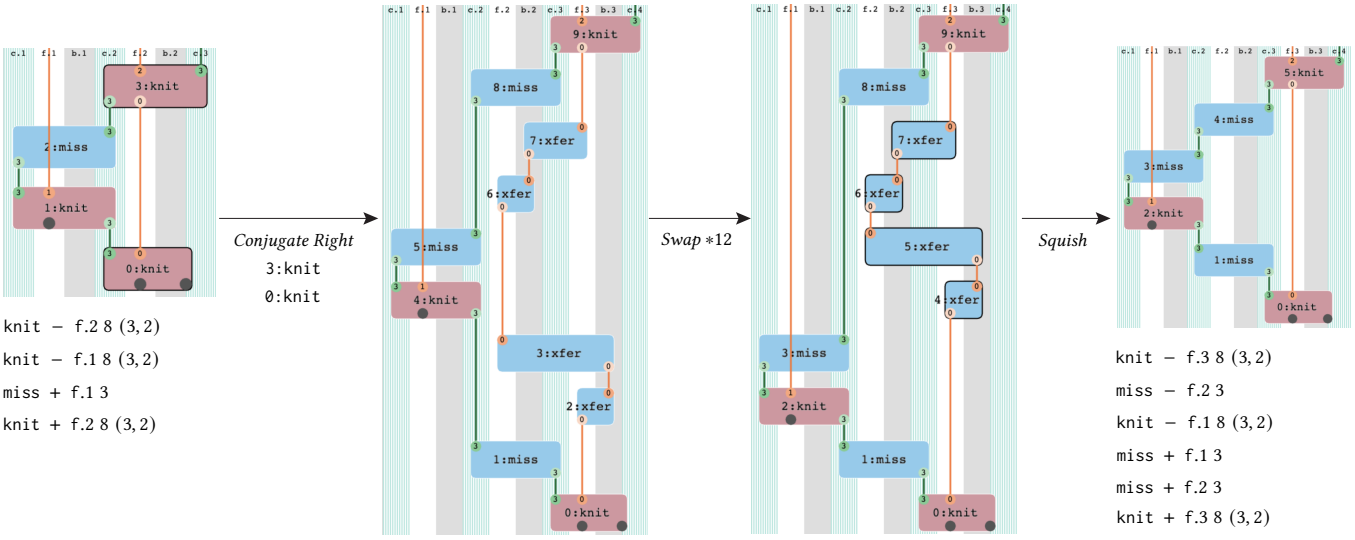


Fig. 18. Rewrite-editor screenshot and the corresponding knitout of the rewrite sequence for moving the `knit`/tuck instruction one to the left or right. The sequence of rewrites in this example moves two `knits` (`3:knit` and `0:knit`) from needle `f.2` to `f.3`.

(a) 1/4th gauge, before rewrites (b) 2/3rd gauge, after rewrites

Fig. 19. Photos of the fabricated pleated tube examples

Note that the proof for the *Swap* rule relies on the swapped instructions having disjoint extents. This scenario requires repeated swapping of instruction `knit` *dir* `f.`x 8 $(y_f, 1)$ with `knit` *dir* `b.`x 8 $(y_b, 1)$. In our original program, $y_f = 3$ and $y_b = 4$. This means the extents of the operations are disjoint, and the *Swap* rule is safe to perform. If instead $y_f = 5$, the instructions would no longer be disjoint, making the rewrite unsafe. Executing the program with this change in carriers results in the error seen in Fig 1c.

8.4 Pleated Tube

Existing knit design systems that automatically schedule knitout programs all have the limitation that they cannot schedule structures that require overlapping more than two sheets at the same physical needle location. This excludes structures like pleats, where a fold in the fabric is secured at one end. However, using a technique known as fractional gauging, it is possible to machine knit such structures. At a high level, n separate sheets can be scheduled to the machine by abstracting the needle bed as bins of width n needles. The i-th sheet in the stack is then assigned to the i-th needle in a bin. This technique requires careful usage of transfers to keep the sheets from intertangling. Therefore, it normally involves much trial and error by an experienced knitting machine programmer.

We got an experienced knitting machine programmer to make a tube with pleats. The tube has locations where there are 4 layers at the same time. Therefore, the program was written in 1/4th gauge (each layer uses one out of every four needle indices). However, knitting at 1/4th gauge means the machine is forced to put more yarn between each loop. Put another way, $s \geq 4$ for all s parameters in the program. We can see this extra yarn in the fabricated result (Fig. 19a). In addition, the program had many extraneous transfers, which reduces fabrication reliability. These ideally should be removed.

To address these issues, we rewrote the program from 1/4th gauge to 2/3rd gauge (each layer uses one out of every three needle indices, where two layers share the same index). This gauge adjustment used high-level rewrite strategies similar to the full to half gauge and sheet intersection examples (see Fig.18). Extraneous transfers were removed using *Squish* and *Slide*. In the resulting pleated tube, we can see that it is narrower and that the bottom of the tube, where most of the extra transfers occurred, looks neater (Fig. 19b).

9 DISCUSSION

Design Decisions. While the topology of the arcs used to define the knitout semantics does match the yarn topology produced by the machine, the choice of fence location was just that: a choice. Knitted objects do not have little boxes around each loop that constrain their range of motion, and arguments can be made for a different choice of fence location and granularity. That said, in order to have a mathematical object with a useful definition of equivalence, *some* choice of constraints must be made to prevent unravelling. We believe that our particular semantics manages to strike the balance between preserving local patterns important in machine knitting

while not excessively constraining transformations that physical knit objects would undergo. However, different applications might be better served by different decisions. Hand knitting, for example, is both more flexible and more prone to variation than machines. Thus, an attempt to formalize hand knit objects using fenced tangles might decide on a different set of topological features as important to preserve and define labels appropriately. We have demonstrated the strength of fenced tangles as a mathematical tool for machine knitting semantics and believe that it would be productive future work to use them to define equivalence on other yarn-based structures.

Metric Correctness. As alluded to in Section 7.1.3, our semantics focuses on formalizing topological correctness, but has no direct consideration for metric correctness, which is crucial for compilation tasks such as respecting machine compatibility. While one could apply metric annotations to arc segments within a fenced tangle and then use said annotations to define heuristics for valid loop and stitch size parameters, fully capturing the complexity of a given machine's metric constraints will inevitably require confronting the continuous nature of yarn. Again, there is no physical box around each loop that prevents yarn from sliding in and out of it. Yet in practice, arbitrary lengths of yarn do not slide around the object; if it did, all the length could slide out one end and essentially unravel the knit. There are many potential avenues for addressing metric correctness, and it will be interesting to see how our work with fenced tangles influences any future approaches.

Rewrite Expressivity. Our presented set of rewrites is not complete; given two knitout programs that denote the same fenced tangle, it is not always possible to move between them using just these rewrites. For example, `knit + f.`x and `knit - b.`x produce topologically equivalent fenced tangles, but we did not introduce a rewrite to transform between them. Furthermore, because our rewrite rules are all fairly low-level, manually performing high-level program rewriting tasks with our editor is tedious and time-intensive (10's to 100's of minutes for the examples shown). We consider our rewrite editor to be a proof-of-concept demonstration of edits that should be used as the foundation of higher-level user-facing tools.

Unscheduled Machine Knitting. This paper formally defines the semantics of knitout, which is a low-level, scheduled representation of machine knitting. However, many machine knitting design tools work with unscheduled representations, where machine instructions have not yet been assigned to specific needles. Extending this work to unscheduled representations would not only enable reasoning about transformations within those representations, it is critical for developing provably correct compilers from unscheduled representations to scheduled machine instructions.

In addition, while prior works make claims about machine knittability constraints [Narayanan et al. 2018], what they actually define are constraints on the types of knit representations for which they can schedule a "correct" program. Our pleated tube is an example of an object that violates said constraints but in reality can be machine knit. It would be interesting if insights from knitout semantics could be used to prove a tighter bound on the machine knittability of unscheduled knitting representations.

ACKNOWLEDGMENTS

The authors would like to thank Himalini Gururaj and Lea Albaugh for their help generating figures, with extra thanks to Lea for knitting samples when the authors were sick. In addition, we would like to thank the reviewers, particularly reviewer 3 for their insightful guidance on improving paper readability. This material is based upon work supported by the National Science Foundation under Grant No. 1955444. Yuka Ikarashi acknowledges the generous support of the Funai Overseas Scholarship and the Masason Foundation Fellowship.

REFERENCES

C.C. Adams. 1994. *The Knot Book.* W.H. Freeman, New York, NY.

Roland Aigner, Mira Alida Haberfellner, and Michael Haller. 2022. SpaceR: Knitting Ready-Made, Tactile, and Highly Responsive Spacer-Fabric Force Sensors for Continuous Input. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology* (Bend, OR, USA) *(UIST '22)*. Association for Computing Machinery, New York, NY, USA, Article 68, 15 pages. https://doi.org/10.1145/3526113.3545694

Lea Albaugh, Scott Hudson, and Lining Yao. 2019. Digital Fabrication of Soft Actuated Objects by Machine Knitting. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–13.

Lea Albaugh, James McCann, Scott E. Hudson, and Lining Yao. 2021. Engineering Multifunctional Spacer Fabrics Through Machine Knitting. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 498, 12 pages. https://doi.org/10.1145/3411764.3445564

Sarah-Marie Belcastro. 2009. Every Topological Surface Can Be Knit: A Proof. *Journal of Mathematics and the Arts* 3, 2 (2009), 67–83.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-end Optimizing Compiler for Deep Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, Berkeley, CA, USA, 579–594. http://dl.acm.org/citation.cfm?id=3291168.3291211

Gabriel Cirio, Jorge Lopez-Moreno, and Miguel A. Otaduy. 2015. Efficient Simulation of Knitted Cloth Using Persistent Contacts. In *Proceedings of the 14th ACM SIGGRAPH / Eurographics Symposium on Computer Animation* (Los Angeles, California) *(SCA '15)*. Association for Computing Machinery, New York, NY, USA, 55–61. https://doi.org/10.1145/2786784.2786801

Ryan Doenges, Mina Tahmasbi Arashloo, Santiago Bautista, Alexander Chang, Newton Ni, Samwise Parkinson, Rudy Peterson, Alaia Solko-Breslin, Amanda Xu, and Nate Foster. 2021. Petr4: Formal Foundations for P4 Data Planes. *Proc. ACM Program. Lang.* 5, POPL, Article 41 (jan 2021), 32 pages. https://doi.org/10.1145/3434322

Sergei Grishanov, Vadim Meshkov, and Alexander Omelchenko. 2009. A topological study of textile structures. Part I: An introduction to topological methods. *Textile Research Journal* 79, 8 (2009), 702–713.

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (may 2001), 396–450. https://doi.org/10.1145/503502.503505

Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. 2022. Exocompilation for Productive Programming of Hardware Accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 703–718. https://doi.org/10.1145/3519939.3523446

Benjamin Jones, Yuxuan Mei, Haisen Zhao, Taylor Gotfrid, Jennifer Mankoff, and Adriana Schulz. 2021. Computational Design of Knit Templates. *ACM Trans. Graph.* 41, 2, Article 16 (dec 2021), 16 pages. https://doi.org/10.1145/3488006

Levi Kapllani, Chelsea Amanatides, Genevieve Dion, and David E. Breen. 2022. Loop Order Analysis of Weft-Knitted Textiles. *Textiles* 2, 2 (2022), 275–295. https://doi.org/10.3390/textiles2020015

Levi Kapllani, Chelsea Amanatides, Geneviève Dion, Vadim Shapiro, and David E. Breen. 2021. TopoKnit : A Process-Oriented Representation for Modeling the Topology of Yarns in Weft-Knitted Textiles. *CoRR* abs/2101.04560 (2021), 22 pages. arXiv:2101.04560 https://arxiv.org/abs/2101.04560

Alexandre Kaspar, Liane Makatura, and Wojciech Matusik. 2019a. Knitting Skeletons: A Computer-Aided Design Tool for Shaping and Patterning of Knitted Garments. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology* (New Orleans, LA, USA) *(UIST '19)*. Association for Computing Machinery, New York, NY, USA, 53–65. https://doi.org/10.1145/3332165.3347879

Alexandre Kaspar, Tae-Hyun Oh, Liane Makatura, Petr Kellnhofer, and Wojciech Matusik. 2019b. Neural Inverse Knitting: From Images to Manufacturing Instructions. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, Long Beach, California, USA, 3272–3281.

Alexandre Kaspar, Kui Wu, Yiyue Luo, Liane Makatura, and Wojciech Matusik. 2021. Knit Sketching: From Cut & Sew Patterns to Machine-Knit Garments. *ACM Trans. Graph.* 40, 4, Article 63 (jul 2021), 15 pages. https://doi.org/10.1145/3450626.3459752

Casimir Kuratowski. 1922. Sur l'opération Ā de l'Analysis Situs. *Fundamenta Mathematicae* 3, 1 (1922), 182–199. http://eudml.org/doc/213290

Mackenzie Leake, Gilbert Bernstein, Abe Davis, and Maneesh Agrawala. 2021. A Mathematical Foundation for Foundation Paper Pieceable Quilts. *ACM Trans. Graph.* 40, 4, Article 65 (jul 2021), 14 pages. https://doi.org/10.1145/3450626.3459853

Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert – A Formally Verified Optimizing Compiler. In *ERTS 2016: Embedded Real Time Software and Systems*. SEE, Toulouse, France. http://xavierleroy.org/publi/erts2016_compcert.pdf

Jenny Lin and James McCann. 2021. An Artin Braid Group Representation of Knitting Machine State with Applications to Validation and Optimization of Fabrication Plans. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*. Institute of Electrical and Electronics Engineers, New York, NY, USA, 1147–1153. https://doi.org/10.1109/ICRA48506.2021.9562113

Jenny Lin, Vidya Narayanan, and James McCann. 2018. Efficient Transfer Planning for Flat Knitting. In *Proceedings of the 2Nd ACM Symposium on Computational Fabrication (SCF '18)*. ACM, New York, NY, USA, 1:1–1:7.

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

James McCann. 2017. The "Knitout" (.k) File Format. [Online]. Available from: https://textiles-lab.github.io/knitout/knitout.html.

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 (July 2016), 49:1–49:11.

J.R. Munkres. 2000. *Topology*. Prentice Hall, Incorporated. https://books.google.com/books?id=XjoZAQAAIAAJ

Georges Nader, Yu Han Quek, Pei Zhi Chia, Oliver Weeger, and Sai-Kit Yeung. 2021. KnitKit: A Flexible System for Machine Knitting of Customizable Textiles. *ACM Trans. Graph.* 40, 4, Article 64 (jul 2021), 16 pages. https://doi.org/10.1145/3450626.3459790

Vidya Narayanan, Lea Albaugh, Jessica Hodgins, Stelian Coros, and James McCann. 2018. Automatic Machine Knitting of 3D Meshes. *ACM Trans. Graph.* 37, 3 (Aug. 2018), 35:1–35:15.

Vidya Narayanan, Kui Wu, Cem Yuksel, and James McCann. 2019. Visual knitting machine programming. *ACM Transactions on Graphics (TOG)* 38, 4 (2019), 1–13.

Jifei Ou, Daniel Oran, Don Derek Haddad, Joseph Paradiso, and Hiroshi Ishii. 2019. SensorKnit: Architecting textile sensors with machine knitting. *3D Printing and Additive Manufacturing* 6, 1 (2019), 1–11.

Mariana Popescu, Matthias Rippmann, Andrew Liew, Lex Reiter, Robert Johann Flatt, Tom Van Mele, and Philippe Block. 2020. Structural design, digital fabrication and construction of the cable-net and knitted formwork of the KnitCandela concrete shell. *Structures* 31 (2020), 1287–1299.

Mariana Popescu, Matthias Rippmann, Tom Van Mele, and Philippe Block. 2018. Automated Generation of Knit Patterns for Non-developable Surfaces. In *Humanizing Digital Reality*, De Rycke K. et al. (Ed.). Springer, Singapore.

Virginia Postrel. 2020. *The fabric of civilization : how textiles made the world.* Basic Books, Hachette Book Group, New York.

Ante Qu and Doug L. James. 2021. Fast Linking Numbers for Topology Verification of Loopy Structures. *ACM Trans. Graph.* 40, 4, Article 106 (jul 2021), 19 pages. https://doi.org/10.1145/3450626.3459778

Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2012. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. *ACM Trans. Graph.* 31, 4, Article 32 (jul 2012), 12 pages. https://doi.org/10.1145/2185520.2185528

Aristides A.G. Requicha. 1977. Mathematical Models of Rigid Solid Objects. *Production Automation Project, University of Rochester, Rochester, New York 14627* 28 (1977), 74 pages.

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* n/a, n/a (2023), 2212541. https://doi.org/10.1002/adfm.202212541 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/adfm.202212541

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

Shima Seiki. 2019. SDS-ONE Apex4. [Online]. Available from: https://www.shimaseiki.com/product/design/.

Soft Byte Ltd. 1999. Designaknit. [Online]. Available from: https://www.softbyte.co.uk/designaknit.htm.

Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: A Functional Data-Parallel IR for High-Performance GPU Code Generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization* (Austin, USA) *(CGO '17)*. IEEE Press, New York, NY, USA, 74–85.

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

Jenny Underwood. 2009. *The design of 3D shape knitted preforms.* Ph. D. Dissertation. Fashion and Textiles, RMIT University.

Chenming Wu, Haisen Zhao, Chandrakana Nandi, Jeffrey I. Lipton, Zachary Tatlock, and Adriana Schulz. 2019. Carpentry Compiler. *ACM Transactions on Graphics* 38, 6 (2019), Article No. 195. presented at SIGGRAPH Asia 2019.

Haisen Zhao, Max Willsey, Amy Zhu, Chandrakana Nandi, Zachary Tatlock, Justin Solomon, and Adriana Schulz. 2022. Co-Optimization of Design and Fabrication Plans for Carpentry. *ACM Trans. Graph.* 41, 3, Article 32 (mar 2022), 13 pages. https://doi.org/10.1145/3508499

A PRELIMINARY DEFINITIONS

The following section provides a more in-depth review of concepts used by this paper.

A.1 Topology Terminology

For the basic definitions of, e.g., a *topological space* and *continuous functions* between such spaces, please see a standard reference [Munkres 2000]. A *homeomorphism* between topological spaces is a bijective function that is continuous in both directions. Despite the similar sounding name, a homotopy does not necessarily express the equivalence of two things. A *homotopy* between two continuous functions $f, g : X \to Y$ (X and Y topological spaces) is a continuous function $H : X \times [0, 1] \to Y$ s.t. $H(x, 0) = f(x)$ and $H(x, 1) = g(x)$. Intuitively, the second parameter of H can be understood as "time" s.t. the whole homotopy can be understood as a continuous motion or interpolation between f and g. If f and g are also embeddings (meaning they are both continuous and injective) then we say H is an *isotopy* between f and g if $H(\cdot, t)$ is an embedding for every t.

An *ambient isotopy* between two embeddings $f, g : X \to Y$ is an isotopy H from the identity $id : Y \to Y$ to some other homeomorphism $h : Y \to Y$ s.t. $H(f(x), 1) = g(x)$. That is, intuitively H is a warp of the entire ambient space H that warps f into g.

A *tame arc* in Y (for $Y = \mathbb{R}^2$ or $Y = \mathbb{R}^3$) is any embedding $\gamma : [0, 1] \to Y$ s.t. γ is ambiently isotopic to a straight line segment. A *tame loop* in Y (same as before) is any embedding of the circle $\gamma : S^1 \to Y$ s.t. the restriction of γ to any closed subinterval of the circle is a tame arc. There are a number of simpler and more intuitive properties which are sufficient to ensure that a knot/arc is tame. For instance, if we require all of our embeddings to be smooth, then they are necessarily tame. If we require all of our embeddings to be composed of a finite number of piecewise linear segments, then they are necessarily tame. The usual examples of non-tame (aka. wild) knots/arcs use constructions similar to the Topologist's sine curve ($\sin(\frac{1}{x})$), in which knotted bits of the path occur infinitely frequently as one limits towards some particular point.

A.2 Formalizing Programming Languages

The formal study of programming languages developed in order to unambiguously specify programming languages and prove properties about them. At one extreme, such theories have allowed us to

construct mechanically verified C compilers [Leroy et al. 2016]. Even without such mechanized proofs, formalization has influenced the design of major programming languages such as Java [Igarashi et al. 2001] and newer domain-specific-languages, such as the network configuration language P4 [Doenges et al. 2021].

To illustrate the concepts of used by this paper as well as our notational conventions, we will describe a simple language. For instance, consider the following program in an assembly-like language. It compares two numbers held in variables `R.1` and `R.2`, and subtracts the smaller variable from the larger.

```
LT R.3 R.1 R.2 ;
IF R.3 {
  SWAP R.1 R.2
} ;
SUB R.4 R.1 R.2
```

To specify a language including such a program, we must first specify the grammar. We do this using the well-known Backus-Naur form (BNF) for a context-free grammar. In the following grammar, we specify that a program or statement (s) is defined to be either a sequence of two other statements or one of four instructions. (A non-toy example would include more primitive instructions.)

$$\begin{array}{rcl} s & ::= & i_1\,;\, s_1 \;\mid\; i_1 \\ i & ::= & \texttt{LT}\ r_1\ r_2\ r_3 \\ & \mid & \texttt{SUB}\ r_1\ r_2\ r_3 \\ & \mid & \texttt{SWAP}\ r_1\ r_2 \\ & \mid & \texttt{IF}\ r\ \{\ s\ \} \\ r & ::= & \texttt{R}.n \\ n & \in & \mathbb{N} \end{array}$$

Grammars are one example of a *structurally inductive definition*. Formally, the grammar is defining a set of strings (or equivalently, ASTs) via induction. To be explicit, let $S_0 = \emptyset$ be the set of all height-0 ASTs. Then, S_1 is the set of all 1-instruction programs. In general S_i is the set of all programs that can be constructed from the grammar rules, assuming $s_1 \in S_{i-1}$. The set of all grammatical statements is then the union (or "least fixed point") of all S_i, namely $S = \bigcup_{i=0}^{\infty} S_i$. Analogously, the syntax for our formalization of knitout can be found in Definition 5.1.

In general, not every grammatical program may be error-free. In fact, we may not even be able to say what every grammatical program means. For example, the `LT` instruction computes and stores a Boolean value into r_1, and the `IF` instruction branches based on a Boolean value. We could define every non-0 value to be "truthy" as in languages like C or Javascript, but for the sake of our example, let's instead say that using an integer where we expect a Boolean is an error.

We now have a decision to make. How do we formalize errors in our language? One approach (which we do not use in this paper) is to specify the meaning of errors via some kind of error state. If we were to go down this route, then we might expect to prove that a *type-system* for our language prevents such errors.

In this paper, we follow a second approach to typing. For us, the *type-system* serves to restrict our attention to a subset $W \subseteq S$ of "valid" programs. Then we will only worry about specifying the meaning (i.e., semantics) of these valid programs. Additionally, the typing will *annotate* our AST with additional information that makes it easier to specify the meaning of our programs.

Let $\Gamma : \{r\} \to \{\mathsf{Int}, \mathsf{Bool}\}$ be a partial function mapping register names to types, where our partial function notation is as follows:

Definition A.1 (Partial function notation)**.** Let A and B be sets with a distinguished *default* element $\perp$ of B. Then a partial function $\sigma \in A \to B$ is a function from A to B with the following notational conventions and operations defined.

- $[]$ is the *empty* partial function defined as $[](a) = \perp$.
- $[a \mapsto b]$ is a *singleton* partial function, defined as $[a \mapsto b](a) = b$ and $[a \mapsto b](a') = \perp$ when $a \neq a'$.
- Given $\sigma \in A \to B$, $\sigma[a \mapsto b]$ is an *extension* of a partial function defined as $\sigma[a \mapsto b](a) = b$ and $\sigma[a \mapsto b](a') = \sigma(a')$ when $a \neq a'$.
- Given two partial functions $\sigma, \sigma' \in A \to B$, $\sigma\sigma'$ is their *concatenation* (not function composition) defined as $\sigma\sigma'(a) = \sigma'(a)$ if $\sigma'(a) \neq \perp$, and $\sigma\sigma'(a) = \sigma(a)$ otherwise. (i.e., first lookup in σ' and then lookup in σ if that fails)
- For a partial function $\sigma \in A \to B$, we say that $a \in \sigma$ if $\sigma(a) \neq \perp$

We call this the typing environment. Then we can define a type-checking *relation* $\Gamma_1 \vdash s \dashv \Gamma_2$, which says that if the registers hold values with types specified by Γ_1, and program s is run, then it will run successfully and leave the registers holding values with types specified by Γ_2. Like the grammar itself, we define this typing relation via structural induction. For historical and conventional reasons, we do this using a horizontal line, known as *sequent notation*: A rule of the form $\frac{A \quad B}{C}$ is equivalent to the logical statement "If A and B, then C."

$$\frac{\Gamma_1 \vdash i \dashv \Gamma_2 \qquad \Gamma_2 \vdash s \dashv \Gamma_3}{\Gamma_2 \vdash i\,;\, s \dashv \Gamma_3}\ \textbf{T-Seq} \qquad \frac{\Gamma(r_1) = \Gamma(r_2)}{\Gamma \vdash (\texttt{SWAP}\ r_1\ r_2) \dashv \Gamma}\ \textbf{T-SWAP}$$

$$\frac{\Gamma(r_2) = \mathsf{Int} \qquad \Gamma(r_3) = \mathsf{Int}}{\Gamma \vdash (\texttt{LT}\ r_1\ r_2\ r_3) \dashv \Gamma[r_1 \mapsto \mathsf{Bool}]}\ \textbf{T-LT}$$

$$\frac{\Gamma(r_2) = \mathsf{Int} \qquad \Gamma(r_3) = \mathsf{Int}}{\Gamma \vdash (\texttt{SUB}\ r_1\ r_2\ r_3) \dashv \Gamma[r_1 \mapsto \mathsf{Int}]}\ \textbf{T-SUB}$$

$$\frac{\Gamma(r) = \mathsf{Bool} \qquad \Gamma \vdash s \dashv \Gamma}{\Gamma \vdash (\texttt{IF}\ r\ \{\ s\ \}) \dashv \Gamma}\ \textbf{T-IF}$$

Using these rules, our original example program is well-typed with initial typing environment $\Gamma_0 = [\texttt{R.1} \mapsto \mathsf{Int},\ \texttt{R.2} \mapsto \mathsf{Int}]$ and final typing environment $\Gamma' = \Gamma_0[\texttt{R.3} \mapsto \mathsf{Bool},\ \texttt{R.4} \mapsto \mathsf{Int}]$. (We omit the derivation to save space.)

For knitout, our analogue of this type-checking rule can be found in Definition 5.4 and Fig. 11. Rather than writing $\Gamma_1 \vdash s \dashv \Gamma_2$, we write $S_0 \xrightarrow{ks} S_1$, where S_0 and S_1 are abstract states of our knitting machine. We use this notation because type-checking of knitting programs is equivalent to performing a kind of abstract execution or simulation of the knitting machine – sufficient to determine whether all resources are always present in the correct places for an execution of the machine to make sense. Despite our use of arrows ($\to$) this is not a specification of knitting program semantics.

To complete the definition of our toy language, we must specify what the programs actually mean. The meaning of most computational programs is the function which that program computes. In particular, let $\sigma : \{r\} \to (\mathbb{Z} \cup \mathbb{B})$ be a partial function mapping register names to integers or Booleans. We call σ the store, and use Σ_Γ to mean the set of all possible stores whose values are consistent with the typing environment Γ. Then given a well-typed program $\Gamma_0 \vdash s \dashv \Gamma_1$, the *denotation* (aka. *meaning* or *semantics*) of the program is a function between stores $\mathcal{E}[\![\Gamma_0 \vdash s \dashv \Gamma_1]\!] : \Sigma_{\Gamma_0} \to \Sigma_{\Gamma_1}$. In total, the function $\mathcal{E}$ specifies the semantics of our entire language, rather than a single program. We write $\mathcal{E}$ to suggest "evaluation."

Like every other part of the language, we again use structural induction to define the function $\mathcal{E}$.

$$\mathcal{E}[\![\Gamma_0 \vdash i;\ s \dashv \Gamma_2]\!](\sigma) = \begin{pmatrix} \mathcal{E}[\![\Gamma_1 \vdash s \dashv \Gamma_2]\!] \circ \\ \mathcal{E}[\![\Gamma_0 \vdash i \dashv \Gamma_1]\!] \end{pmatrix}(\sigma)$$

$$\mathcal{E}[\![\Gamma_0 \vdash \texttt{LT}\ r_1\ r_2\ r_3 \dashv \Gamma_1]\!](\sigma) = \sigma[r_1 \mapsto (\sigma(r_2) < \sigma(r_3))]$$

$$\mathcal{E}[\![\Gamma_0 \vdash \texttt{SUB}\ r_1\ r_2\ r_3 \dashv \Gamma_1]\!](\sigma) = \sigma[r_1 \mapsto (\sigma(r_2) - \sigma(r_3))]$$

$$\mathcal{E}[\![\Gamma_0 \vdash \texttt{SWAP}\ r_1\ r_2 \dashv \Gamma_1]\!](\sigma) = \sigma[r_1 \mapsto \sigma(r_2),\ r_2 \mapsto \sigma(r_1)]$$

$$\mathcal{E}[\![\Gamma \vdash \texttt{IF}\ r\ \{\ s\ \} \dashv \Gamma]\!](\sigma) = \begin{cases} \mathcal{E}[\![\Gamma \vdash s \dashv \Gamma]\!], & \sigma(r) = \mathit{true} \\ \sigma, & \text{otherwise} \end{cases}$$

There are other (non-denotational) approaches to programming language semantics. However, for our formalization of knitout, the denotational approach made the most sense. Unlike usual denotational semantics, where programs denote functions (e.g., the toy language of this section), knitout programs denote mathematical representations of the objects they manufacture. One of the main concerns of this paper is to find a suitable mathematical object for knitout programs to denote.

Finally, observe that if we wanted to optimize programs in our toy language, we would be able to prove that certain rewritings of programs are correct – by appeal to the semantics we have just defined. For example, it should be the case that swapping the contents of two registers, and then immediately swapping those contents back is equivalent to the identity function (or empty program). A real programming language may allow us to deduce many such equivalences, or *rewrite rules*. Such rules form an important part of compilers, but are tricky to get right in general. Among other uses, formal language semantics allow us to precisely determine the validity of such rules, and thus develop more reliable and powerful compilers for a language.

B FENCED TANGLE LEMMAS

We begin with some basic properties of fenced tangles elided in the main text for clarity before progressing to more complicated lemmas important to our proofs of rewrite rule correctness.

Lemma B.1 (Concatenations are Equivalence-Invariant)**.** *Let $K_1 \cong K_1' \in \mathcal{S}_n^m$, $K_2 \cong K_2' \in \mathcal{S}_p^q$, and $K_3 \cong K_3' \in \mathcal{S}_m^p$. Then $K_1 \circ K_3 \cong K_1' \circ K_3'$; $K_1 \otimes K_2 \cong K_1' \otimes K_2'$; and for any choice of ι and ω, $K_1|_\iota^\omega K_2 \cong K_1'|_\iota^\omega K_2'$.*

Proof. For $K_1 \circ K_3$ and $K_1 \otimes K_2$, this follows trivially from disjointness of the two composite diagrams in the plane. For $K_1|_\iota^\omega K_2$ the argument is less trivial. K_1 and K_2 can be unprojected into fenced tangles T_1 and T_2 on regions U_1 and U_2, sharing a common equator and a boundary disk in common. The interiors of U_1 and U_2 are disjoint, and so can be arbitrarily modified with ambient isotopies before being reprojected into a layered diagram. □

Lemma B.2. *The three concatenation operators are associative, and each has a unit slab: $id_0 \otimes K \cong K \cong K \otimes id_0$; $id_0|_{id}^{id} K \cong K \cong K|_{id}^{id} id_0$; and for K an (n, m)-slab, $id_n \circ K \cong K \cong K \circ id_m$. Therefore it is justified to omit parentheses when repeatedly concatenating in the same way. Furthermore, $id_n \otimes id_m \cong id_n|id_m \cong id_{n+m}$*

Proof. Immediate from drawing diagrams for the relevant equations. □

Notation B.3 (Concatenation of Interleavings). Let $\iota_1 \in \mathcal{I}_{n,p}$ and $\iota_2 \in \mathcal{I}_{m,q}$ be interleavings. Then $\iota_1 \sqcup \iota_2 \in \mathcal{I}_{n+m,p+q}$ is an interleaving defined (using set representations) as $\iota_1 \sqcup \iota_2 = \iota_1 \cup \{i + n + p | i \in \iota_2\}$.

Lemma B.4 (|-⊗ Distributivity)**.** *Let $K_a \in \mathcal{S}_{n_1}^{m_1}$, $K_b \in \mathcal{S}_{n_2}^{m_2}$, $K_c \in \mathcal{S}_{p_1}^{q_1}$, and $K_d \in \mathcal{S}_{p_2}^{q_2}$. Furthermore, let $\iota_1 \in \mathcal{I}_{n_1,p_1}$, $\omega_1 \in \mathcal{I}_{m_1,q_1}$, $\iota_2 \in \mathcal{I}_{n_2,p_2}$, and $\omega_2 \in \mathcal{I}_{m_2,q_2}$ be interleavings. Then,*

$$(K_a|_{\iota_1}^{\omega_1} K_c) \otimes (K_b|_{\iota_2}^{\omega_2} K_d) \cong (K_a \otimes K_b)|_{\iota_1 \sqcup \iota_2}^{\omega_1 \sqcup \omega_2} (K_c \otimes K_d)$$

Proof. immediate from picture □

Lemma B.5 (∘-| Distributivity)**.** *Let $K_a \in \mathcal{S}_{n_1}^{m_1}$, $K_b \in \mathcal{S}_{n_2}^{m_2}$, $K_c \in \mathcal{S}_{m_1}^{p_1}$, and $K_d \in \mathcal{S}_{m_2}^{p_2}$. Furthermore, let $\iota \in \mathcal{I}_{n_1,n_2}$, $\mu \in \mathcal{I}_{m_1,m_2}$, and $\omega \in \mathcal{I}_{p_1,p_2}$ be interleaving functions. Then,*

$$(K_a \circ K_c)|_\iota^\omega (K_b \circ K_d) \cong (K_a|_\iota^\mu K_b) \circ (K_c|_\mu^\omega K_d)$$

Proof. immediate from picture □

Since our semantics will assign a fenced tangle to each knitout program, we will want to know under what circumstances different sub-programs can be re-ordered (i.e., commute). The following lemmas will help us develop such commutativity principles by allowing cleaner reasoning about various kinds of sub-diagrams. Recall Lemma 6.5. We begin by noting that the lemma can be trivially extended as follows:

Corollary B.6 (Commutativity by Horizontal Separation)**.** *The preceding two lemmas imply that for any $g \in \mathbb{N}$, $K_1 \in \mathcal{S}_n^m$, and $K_2 \in \mathcal{S}_p^q$ the following equation holds, permitting the vertical commuting of horizontally non-overlapping sub-tangles.*

$$(K_1 \otimes id_{g+p}) \circ (id_{m+g} \otimes K_2) \cong (id_{n+g} \otimes K_2) \circ (K_1 \otimes id_{g+q})$$

In principle we also ought to be able to commute operations occurring in wholly different layers. However, we can develop even stronger machinery. In many cases, we can explicitly convert composition by layer into horizontal composition.

Lemma B.7 (No-Overlap Layering)**.** *Let $K_1 \in \mathcal{S}_n^m$ and $K_2 \in \mathcal{S}_p^q$. Then,*

$$K_1|_{id}^{id} K_2 \cong K_1 \otimes K_2$$

Proof. By Lemma B.1, we may assume that the entirety of K_1 and K_2 are disjoint, with no overlaps, since there are no interleavings of their loose ends. Consequently the sub-diagrams of $K_1|_{id}^{id} K_2$ are horizontally separated—and can therefore equally well be interpreted as $K_1 \otimes K_2$. □

Lemma B.8 (Layer Decomposition of Separate and Merge)**.** *Let $\iota \in \mathcal{I}_{n,p}$ be an interleaving. Then,*

$$\begin{aligned}
\overleftarrow{V}_\iota &= id_p \,|^{id}_{\bar{\iota}}\, id_n \\
\overrightarrow{V}_\iota &= id_n \,|^{id}_{\iota}\, id_p \\
\overrightarrow{\Lambda}_\iota &= id_p \,|^{\bar{\iota}}_{id}\, id_n \\
\overleftarrow{\Lambda}_\iota &= id_n \,|^{\iota}_{id}\, id_p
\end{aligned}$$

Proof. By the definition of a permutation slab, all crossings must be oriented consistently in merge and separation slabs. Furthermore, because the permutation o_ι derived from the interleaving is required to be monotonic within each half of the partition, we know that the diagram viewed on each such subset of the yarns must be the identity slab. Therefore, all of these slabs must decompose into a layering of two identity slabs. Inspection of the four cases confirms the above formulas as correctly specifying the various interleavings. □

The following lemma allows us to convert layering composition into horizontal composition in general by "sliding apart" the different layers composing a diagram. This makes it easy to modify layers independently and separately from the concerns of interleaving patterns.

Lemma B.9 (Sliding Door Lemma)**.** *Let $K_1 \in \mathcal{S}^m_n$, $K_2 \in \mathcal{S}^q_p$ and let $\iota \in \mathcal{I}_{n,p}$, $\omega \in \mathcal{I}_{m,q}$ be interleavings. Then,*

$$K_1|^\omega_\iota K_2 \cong \overleftarrow{V}_\iota \circ (K_1 \otimes K_2) \circ \overrightarrow{\Lambda}_\omega$$

(note: $\overleftarrow{\Lambda}$ may be used instead of $\overrightarrow{\Lambda}$)

Proof.

$$\begin{aligned}
K_1|^\omega_\iota K_2 &\cong (id_n \circ K_1 \circ id_m)|^\omega_\iota (id_p \circ K_2 \circ id_q) \text{ (by Lemma B.2)} \\
&\cong (id_n|^{id}_\iota id_p) \circ (K_1|^{id}_{id} K_2) \circ (id_m|^\omega_{id} id_q) \text{ (by Lem B.5)} \\
&\cong \overleftarrow{V}_\iota \circ (K_1|^{id}_{id} K_2) \circ \overrightarrow{\Lambda}^\omega \text{ (by Lemma B.8)} \\
&\cong \overleftarrow{V}_\iota \circ (K_1 \otimes K_2) \circ \overrightarrow{\Lambda}^\omega \text{ (by Lemma B.7)}
\end{aligned}$$

□

C TRANSLATION BETWEEN FORMAL KNITOUT AND ACTUAL KNITOUT

Formal knitout differs from knitout [McCann 2017] (hereafter "actual knitout") in a few specifics discussed below. These details do not change the expressively of the language, but do make formal knitout slightly easier to reason about. We include our implementation of a translator from actual knitout to formal knitout in our supplemental material, and we characterize the differences between the two in this section.

Actual knitout is a UTF-8-encoded text file where operations are new-line separated, and comments are annotated with the character `;`. Optional headers may be used to assign carriers string-based aliases as well as provide optional definitions such as target machine model and yarn type. The operation syntax is also less verbose: `in` and `out` only have a carrier ID parameter, and location is inferred from the first operation that uses that carrier. Needle locations do not have a period dividing the bed and index (`f1` vs `f.1`). Stitch length l is a global state parameter that is set with the command `stitch` or extension `x-stitch-length`, while yarn length s is implicit (though s can be somewhat controlled via a combination of `tuck` and `drop` operations). Formal knitout doesn't support fractional racking (`rack 0.5`) or transferring to sliders, but as the former does not affect object topology, and the latter can be simulated using existing transfers, we chose to omit them for simplicity.

In actual knitout, the `miss`, `in`, and `out` operations can also accept a carrier sequence instead of just a single carrier. In addition, a single `miss` is allowed to move past multiple needles, and a single `rack` can change the racking to any value. These can all be represented in formal knitout with a sequence of formal operations.

Finally, while formal knitout treats each operation as updating a carrier's physical location, actual knitout operations set a logical location and update the physical location to match as needed. This affects the `rack` operation, where in actual knitout, back-bed referenced carriers will move to maintain the same relative location to their back-bed needles. Furthermore, in actual knitout `xfer` and `split` operations update a carrier's logical location: for all carriers *not* in the yarn carrier sequence, if their logical location is relative to source needle $n.x$, it is updated to be relative to target needle $n'.x'$. This can be simulated in formal knitout by tracking logical carrier locations and inserting `miss` operations as is appropriate.

D KNITOUT PROGRAM REWRITES

We will use the heading **Rewrite Rule** to designate particular lemmas (i.e., propositions) which are stated so that they are immediately applicable to the rewriting/scheduling of knitout programs.

D.1 Subprogram Commutativity

Intuitively, if two instructions have "disjoint" effects, then they should commute ($ab = ba$). In order to capture this intuition, we will define instruction *extents*, which allow us to narrowly confine their non-trivial (i.e., non-id) behavior to a rectangle. Intuitively, the two dimensions of the extent rectangle correspond to horizontal and depth-wise decomposition respectively[2]. To be able to define the extent of any valid program, we can conservatively take the join of the extents of the underlying subprograms.

Definition D.1 (Join of Rectangles)**.** Let

$$\begin{aligned}
R_A &= [A_{xmin}, A_{xmax}] \times [A_{ymin}, A_{ymax}] \\
R_B &= [B_{xmin}, B_{xmax}] \times [B_{ymin}, B_{ymax}]
\end{aligned}$$

be two rectangles $R_A \subseteq \mathbb{Q}^2_\infty$, $R_B \subseteq \mathbb{Q}^2_\infty$. (where $\mathbb{Q}_\infty = \mathbb{Q} \cup \{-\infty, \infty\}$) Their *join* is defined as the smallest rectangle enclosing both R_A and R_B:

$$\begin{aligned}
R_A \sqcup R_B = {} & [\min(A_{xmin}, B_{xmin}), \max(A_{xmax}, B_{xmax})] \times \\
& [\min(A_{ymin}, B_{ymin}), \max(A_{ymax}, B_{ymax})]
\end{aligned}$$

Definition D.2 (Extent)**.** We define the extent of a valid knitout program $\text{ex}(S \xrightarrow{ks} S') \subseteq \mathbb{Q}^2_\infty$ as a 2D interval (rectangle). Where S and S' can be inferred from context, we will notate the extent as

[2]Inspection of Lemma B.2 makes it clear why a third dimension for vertical decomposition is unnecessary.

$\mathrm{ex}(ks)$. We will use $[z \pm \frac{1}{2}]$ as shorthand for $[z - \frac{1}{2}, z + \frac{1}{2}]$.

$$\mathrm{ex}(S \xrightarrow{ks_1} S' \xrightarrow{ks_2} S'') = \mathrm{ex}(S \xrightarrow{ks_1} S') \sqcup \mathrm{ex}(S' \xrightarrow{ks_2} S'')$$

$$\mathrm{ex}(\texttt{tuck}\ dir\ \texttt{f}.x\ l\ (y,s)) = [\lfloor \texttt{f}.x \rfloor_r \pm \frac{1}{2}] \times [-\infty, y]$$

$$\mathrm{ex}(\texttt{tuck}\ dir\ \texttt{b}.x\ l\ (y,s)) = [\lfloor \texttt{b}.x \rfloor_r \pm \frac{1}{2}] \times [y, \infty]$$

$$\mathrm{ex}(\texttt{knit}\ dir\ \texttt{f}.x\ l\ yarns) = [\lfloor \texttt{f}.x \rfloor_r \pm \frac{1}{2}] \times [-\infty, y_{max}]$$

$$\mathrm{ex}(\texttt{knit}\ dir\ \texttt{b}.x\ l\ yarns) = [\lfloor \texttt{b}.x \rfloor_r \pm \frac{1}{2}] \times [y_{min}, \infty]$$

$$\mathrm{ex}(\texttt{split}\ dir\ n.x\ n'.x'\ l\ yarns) = [\lfloor n.x \rfloor_r \pm \frac{1}{2}] \times [-\infty, \infty]$$

$$\mathrm{ex}(\texttt{miss}\ dir\ n.x\ y) = [\lfloor n.x \rfloor_r \pm \frac{1}{2}] \times \{y\}$$

$$\mathrm{ex}(\texttt{in}\ +\ n.x\ y) = \{\lfloor n.x \rfloor_r + \frac{1}{2}\} \times \{y\}$$

$$\mathrm{ex}(\texttt{in}\ -\ n.x\ y) = \{\lfloor n.x \rfloor_r - \frac{1}{2}\} \times \{y\}$$

$$\mathrm{ex}(\texttt{out}\ +\ n.x\ y) = \{\lfloor n.x \rfloor_r + \frac{1}{2}\} \times \{y\}$$

$$\mathrm{ex}(\texttt{out}\ -\ n.x\ y) = \{\lfloor n.x \rfloor_r - \frac{1}{2}\} \times \{y\}$$

$$\mathrm{ex}(\texttt{drop}\ \texttt{f}.x\ ) = \{\lfloor \texttt{f}.x \rfloor_r\} \times \{-\infty\}$$

$$\mathrm{ex}(\texttt{drop}\ \texttt{b}.x\ ) = \{\lfloor \texttt{b}.x \rfloor_r\} \times \{\infty\}$$

$$\mathrm{ex}(\texttt{xfer}\ n.x\ n'.x'\ ) = \{\lfloor n.x \rfloor_r\} \times [-\infty, \infty]$$

$$\mathrm{ex}(\texttt{rack}\ r) = [-\infty, \infty] \times \{\infty\}$$

where y_{max} and y_{min} are the minimum and maximum across *yarns*.

To use these operation extents to decompose the denoted fenced tangles, we begin by defining 2D coordinates for every point in $\mathcal{E}[S]$.

Definition D.3 (Coordinates of State Denotations)**.** Let

$$R = [x_{min}, x_{max}] \times [y_{min}, y_{max}] \subseteq \mathbb{Q}^2_\infty$$

be an extent rectangle, S be a machine state, and $\mathcal{E}[S]$ the denotation of that machine state (where the extent rectangle for specific programs is defined in Definition D.2).

Points in $\mathcal{E}[S]$ either arise from loops at physical needle locations or active yarn carriers at physical carrier locations. For each of these points, define "coordinates" $p \in \mathbb{Q}^2_\infty$ as follows: a point arising from $L(\texttt{f}.x) > 0$ has coordinates $(x, -\infty)$; a point arising from $L(\texttt{b}.(x - r)) > 0$ has coordinates (x, ∞); finally, the point for an active yarn $Y(y) \neq \bot$ has coordinates $\left(Y(y) - \frac{1}{2}, y\right)$.

Put in words, loops are depth-located in front of or behind everything else, at the specified whole number needle. Meanwhile, yarns are located in depth according to their yarn id, and at $\frac{1}{2}$ between needles.

Lemma D.4 (Extent Decomposition)**.** *Let $S \xrightarrow{ks} S'$ be a valid knitout program with extent $R = \mathrm{ex}(ks)$. First, we can define various partitions of the set of points $\mathcal{E}[S]$. Let R^{-x} consist of all points with x-coordinate less than R and R^{+x} similarly points with x-coordinate greater than R; meanwhile, let $R^{|x}$ be the remaining set of points whose x-coordinate overlaps R. The tri-partition R^{-y}, R^{+y} and $R^{|y}$ may similarly and independently be defined using y-coordinates.*

Then there exist both horizontal and depth-wise decompositions:

$$\mathcal{E}[S \xrightarrow{ks} S'] = id_{n^{-x}} \otimes K \otimes id_{n^{+x}} = id_{n^{-y}}|_{\iota^-}^{\omega^-} \left(K' \,|_{\iota^+}^{\omega^+} id_{n^{+y}}\right)$$

where $n^{-x} = \#R^{-x}$, and similarly for other $n^\bullet$; the points in each id slab corresponding to the appropriate partition of the state by the extent.

Proof. The proof proceeds inductively.

First, consider the case of $ks_1; ks_2$. Let $R_1 = \mathrm{ex}(ks_1)$ and $R_2 = \mathrm{ex}(ks_2)$ be the extents of each sub-program. We will prove the case of the horizontal decomposition; the depth-wise case proceeds similarly. Let $n^{-x} = \min(n_1^{-x}, n_2^{-x})$, and $n^{+x} = \max(n_1^{+x}, n_2^{+x})$. Then, both decompositions can be rectified with each other to share this common trivial slab on the outside, since (e.g.) $id_{n_1^{-x}} = id_{n^{-x}} \otimes id_{n_1^{-x} - n^{-x}}$ (and similarly for n_1^{+x}, n_2^{-x}, and n_2^{+x}).

$$\begin{aligned}
&\mathcal{E}[S \xrightarrow{ks_1} S' \xrightarrow{ks_2} S''] \\
&\quad = \mathcal{E}[S \xrightarrow{ks_1} S'] \circ \mathcal{E}[S' \xrightarrow{ks_2} S''] \\
&\quad = \left(id_{n^{-x}} \otimes id_{n_1^{-x} - n^{-x}} \otimes K_1 \otimes id_{n_1^{+x} - n^{+x}} \otimes id_{n^{+x}}\right) \circ \\
&\qquad \left(id_{n^{-x}} \otimes id_{n_2^{-x} - n^{-x}} \otimes K_2 \otimes id_{n_2^{+x} - n^{+x}} \otimes id_{n^{+x}}\right) \\
&\quad = id_{n^{-x}} \otimes \begin{pmatrix} id_{n_1^{-x} - n^{-x}} \otimes K_1 \otimes id_{n_1^{+x} - n^{+x}} \circ \\ id_{n_2^{-x} - n^{-x}} \otimes K_2 \otimes id_{n_2^{+x} - n^{+x}} \end{pmatrix} \otimes id_{n^{+x}} \\
&\quad = id_{n^{-x}} \otimes K \otimes id_{n^{+x}}
\end{aligned}$$

All other cases concern individual instructions. We justify these by examining the preceding definition of the extent function, the preceding definition of coordinates, and the denotations in Fig. 12. The cases of `tuck`, `knit`, `split`, and `miss` are all justified because both the denotation diagram and the extent encompass the needles at a location and yarns before and after that needle location. The cases of `in`, `out`, `drop`, and `xfer` require closer inspection to observe that all non-trivial behavior in the diagram is confined more narrowly to a single yarn, or single needle location (front and back). Finally the `rack` instruction acts non-trivially on the entire back bed but leaves the front-bed fixed. □

Having laid the groundwork with the extent function, we can now prove our first rewrite rule in earnest.

Rewrite Rule 1 (Swap)**.**

$$S \vdash ks_1; ks_2 \cong ks_2; ks_1$$

whenever $\mathrm{ex}(ks_1) \cap \mathrm{ex}(ks_2) = \emptyset$

Proof. If $\mathrm{ex}(ks_1) \cap \mathrm{ex}(ks_2) = R_1 \cap R_2 = \emptyset$, the R_1 and R_2 must be horizontally disjoint or depth-wise disjoint. Without loss of generality, assume they are horizontally disjoint. Furthermore without loss of generality assume that K_2 occurs to the right of K_1. Let

$S_0 = S$, so that we begin with the trace $S_0 \xrightarrow{ks_1} S_1 \xrightarrow{ks_2} S_2$. By Lemma D.4, we can make the initial decompositions $\mathcal{E}[S_0 \xrightarrow{ks_1} S_1] = id_{n_1^{-x}} \otimes K_1 \otimes id_{n_1^{+x}}$ and $\mathcal{E}[S_1 \xrightarrow{ks_2} S_2] = id_{n_2^{-x}} \otimes K_2 \otimes id_{n_2^{+x}}$ where $K_1 \in \mathcal{S}^{q_1}_{p_1}$ and $K_2 \in \mathcal{S}^{q_2}_{p_2}$. In addition, since the operations are horizontally disjoint, there must exist some number of yarns, $g \in \mathbb{N}$ in-between the output of K_1 and the input of K_2 such that $n_1^{+x} = g + p_2 + n_2^{+x}$ and $n_2^{-x} = n_1^{-x} + q_1 + g$. The middle step follows from Corollary B.6.

$$\begin{aligned}
&\mathcal{E}[S_0 \xrightarrow{ks_1} S_1 \xrightarrow{ks_2} S_2] \\
&\quad = ((id_{n_1^{-x}} \otimes K_1) \otimes id_{g+p_2+n_2^{+x}}) \circ (id_{n_1^{-x}+q_1+g} \otimes (K_2 \otimes id_{n_2^{+x}})) \\
&\quad = (id_{n_1^{-x}+p_1+g} \otimes (K_2 \otimes id_{n_2^{+x}})) \circ ((id_{n_1^{-x}} \otimes K_1) \otimes id_{g+q_2+n_2^{+x}}) \\
&\quad = \mathcal{E}[S_0 \xrightarrow{ks_2} S_1'] \circ \mathcal{E}[S_1' \xrightarrow{ks_1} S_2]
\end{aligned}$$

In the case of depth-wise decomposition, one uses the Sliding Door Lemma (B.9) to convert depth-wise composition to horizontal composition, thus reducing to the already handled case. □

While this swap rewrite handles most permissible exchange between non-interacting instructions, the extent-based analysis is far too conservative when encountering `rack` instructions, which have an extent of $[-\infty, \infty] \times \{\infty\}$ due to how racking affects the whole back bed. However, racking can be combined with `xfer` operations in the `SHIFT` macro to *locally* rearrange loops between needles (Definition 7.1). Unless we have some way to localize the effect of this pattern, `rack` will form an insurmountable barrier to our attempts to reschedule knitting programs. To streamline the proof of this special case extent function, we define the following modified macro:

Definition D.5 (Move Macro)**.** The `MOVE` macro is the `SHIFT` macro with an additional `RACK` to reset the machine's racking to r:

$$\begin{aligned}
\texttt{MOVE}(\texttt{f}.x, r, \texttt{f}.(x+j)) &:= \texttt{SHIFT}(\texttt{f}.x, r, \texttt{f}.(x+j)); \\
&\quad\ \texttt{RACK}(r+j, -j) \\
\texttt{MOVE}(\texttt{b}.x, r, \texttt{b}.(x+j)) &:= \texttt{SHIFT}(\texttt{b}.x, r, \texttt{b}.x+j) \\
&\quad\ \texttt{RACK}(r-j, j)
\end{aligned}$$

Definition D.6 (Move Extent)**.** Let ks be exactly a `MOVE` sub-program as just defined. Then the move-extent of ks is a rectangle, like for a basic extent. However, unlike basic extents, move-extents are context-sensitive: their definition depends on the state S of the knitting machine immediately prior to the `MOVE` sub-program.

$$\begin{aligned}
\mathrm{ex}_m(S, \texttt{MOVE}(\texttt{f}.x, r, \texttt{f}.(x+j))) &= [x, x+j] \times [-\infty, y_{max}] \\
\mathrm{ex}_m(S, \texttt{MOVE}(\texttt{b}.x, r, \texttt{b}.(x+j))) &= [x-r, x-r+j] \times [y_{min}, \infty]
\end{aligned}$$

where y_{max} is ∞ if $L(\texttt{b}.x - r) > 0$, otherwise it is $\max\{y \mid y \in Y^{-1}(x') \text{ and } x < x' \leq x + j\}$ or $-\infty$ if there are no yarn-carriers parked between x and $x + j$ in the state S. Similarly, y_{min} is $-\infty$ if $L(\texttt{b}.x + r) > 0$ and ∞ otherwise.

Much like the previously defined extents on individual operations, ex_m is used to horizontally and layer decompose $\mathcal{E}[S \xrightarrow{\texttt{MOVE}} S']$ as an intermediate step in proving when a `MOVE` subprogram can be swapped with another program. A more detailed proof of this is as follows.

Lemma D.7 (Move Decomposition)**.** *Let $ks_\texttt{f} = \texttt{MOVE}(\texttt{f}.x, r, \texttt{f}.(x+j))$ and let $ks_\texttt{b} = \texttt{MOVE}(\texttt{b}.x, r, \texttt{b}.(x+j))$. Let S be an initial state s.t. in the case of $ks_\texttt{f}$, $L(b.(x-r)) = 0$; and in the case of $ks_\texttt{b}$, $L(f.(x+r)) = 0$. $ks_\texttt{f}$ admits horizontal and layer decompositions, of the forms*

$$\begin{aligned}
\mathcal{E}[S \xrightarrow{ks_\texttt{f}} S'] &= id_{n^-} \times K \times id_{n^+} \\
&= K'|_t^\omega id_m
\end{aligned}$$

where for $R = ex_m(S, ks_\texttt{f})$, $n^- = \#R^{-x}$, $n^+ = \#R^{+x}$, and $m = \#R^{+y}$. $ks_\texttt{b}$ admits horizontal and layer decompositions, of the forms

$$\begin{aligned}
\mathcal{E}[S \xrightarrow{ks_\texttt{b}} S'] &= id_{n^-} \times K \times id_{n^+} \\
&= id_m|_t^\omega K'
\end{aligned}$$

where for $R = ex_m(S, ks_\texttt{b})$, $n^- = \#R^{-x}$, $n^+ = \#R^{+x}$, and $m = \#R^{-y}$.

Proof. Consider the case of $ks_\texttt{b}$ first. Since $L(\texttt{f}.x + r) = 0$, no loops transferred to the temporary front-bed needle will get stacked together with any other loops. Any loops that are temporarily transferred to the front bed, all loops on the front bed, and all yarn carriers remain horizontally stationary over all of the `MOVE` operation prior to the final `rack` operation. Consequently, after the second `xfer` operation, the entire back-bed can be layer-separated from the rest of the denoted tangle, and the final racking respects this decomposition. Thus, the second decomposition is justified. In the case of the first decomposition, observe that in this final diagram all paths except those starting at $\lfloor \texttt{b}.x \rfloor_r$ and ending at $\lfloor \texttt{b}.x + j \rfloor_r$ are perfectly vertical. Therefore, horizontally we can separate out an identity slab of everything to the left of $\lfloor \texttt{b}.x \rfloor_r$ and an identity of everything to the right of $\lfloor \texttt{b}.x + j \rfloor_r$.

Now consider the case of $ks_\texttt{f}$, which would ideally be symmetric with $ks_\texttt{b}$. Unfortunately, the transferred loops no longer remain stationary during the racking in-between the transfers. Rather, they and the entire back bed move in between the two transfers. Once the transferred loops are back on the front-bed and the racking undone, the same basic argument as above justifies the horizontal decomposition. However, the layer decomposition is less obvious. Since the transferred loops move in tandem with all of the back-bed loops, no crossings between them are introduced prior to the final racking sequence. At this point we can introduce a layer decomposition of the back bed. This would justify consistency with a rectangular interval of $[-\infty, \infty)$ in the y-coordinate. However, observe that by definition there are no yarns between y_{max} and ∞ inside the horizontal extent of $ks_\texttt{f}$. Therefore, we can layer decompose everything strictly after y_{max} from everything at or before y_{max}. □

Lemma D.8 (Move Swap)**.** *Let ks_1 be a `MOVE` subprogram and ks_2 some other knitout program. Let S be an initial state, s.t. $S \xrightarrow{ks_1} S' \xrightarrow{ks_2} S''$ is valid. We consider the two `MOVE` cases separately.*

If $ks_1 = \texttt{MOVE}(\texttt{f}.x, r, \texttt{f}.x + j)$; $L(\texttt{b}.(x-r)) = 0$ in S and S'; and $\mathrm{ex}_m(S, ks_1) \cap \mathrm{ex}(ks_2) = \emptyset$; then

$$S \vdash ks_1; ks_2 \cong ks_2; ks_1$$

If $ks_1 = \texttt{MOVE}(\texttt{b}.x, r, \texttt{b}.x + j)$; $L(\texttt{f}.(x + r)) = 0$ *in* S *and* S'; *and* $\mathrm{ex}_m(S, ks_1) \cap \mathrm{ex}(ks_2) = \emptyset$; *then*

$$S \vdash ks_1; ks_2 \cong ks_2; ks_1$$

Proof. The proof is structurally the same as for Rewrite Rule 1, with the additional use of Lemma D.7. Because of the additional preconditions and context-sensitivity of the move decomposition lemma, we must ensure that the preconditions are satisfied in both S and S'; but these are already explicit preconditions of this rewrite. □

D.2 Canceling Subprograms

Next we consider programs which in some way cancel each other, akin to the algebraic law $a^{-1}a = id$ in group theory. As one might expect, these rules all involve operations which do not produce fences since it can be trivially proven that equivalent fenced tangles must have an equal number of fences.

Rewrite Rule 2 (Merge)**.** *Racking in one direction and then back in the other direction is the same as doing nothing.*

$$S \vdash (\texttt{rack}\ (r \pm 1);\ \texttt{rack}\ r) \cong \texttt{nop}$$

where r *is the initial racking value in* S.

Missing at $n.x$ *in one direction and then back in the other is the same as doing nothing.*

$$S \vdash (\texttt{miss}\ dir\ n.x\ y;\ \texttt{miss}\ \neg dir\ n.x\ y) \cong \texttt{nop}$$

Proof. First we consider merging the two rack operations $ks_1; ks_2$. Recall that $\mathcal{E}[S \xrightarrow{\texttt{rack r}} S'] = id_n|^{\iota'}_{\iota} id_m$ where $\iota = \mathcal{I}_{<\infty}[S]$ and $\iota' = \mathcal{I}_{<\infty}[S']$. Then, again by distributivity

$$\begin{aligned}
\mathcal{E}[S \xrightarrow{\texttt{rack}\ (r\pm1);\texttt{rack}\ r} S] &= \left(id_n|^{\iota'}_{\iota} id_m\right) \circ \left(id_n|^{\iota}_{\iota'} id_m\right) \\
&= (id_n \circ id_n)|^{\iota}_{\iota}(id_m \circ id_m) \\
&= id_{n+m} \\
&= \mathcal{E}[S \xrightarrow{\texttt{nop}} S]
\end{aligned}$$

Similarly, the proof for merging the two miss operations $m_1; m_2$ for yarn y proceeds by observing that both miss operations have a compatible layer decomposition, and that the composition within each layer is simply id.

Let S' be the state between the two miss operations. Let $\iota_< = \mathcal{I}_{<y}[S] \in \mathcal{I}_{n,m+1}$ be the interleaving of all yarns and loops in front of yarn y and $\iota_> = \mathcal{I}_{>y}[S] \in \mathcal{I}_{1,m}$ be the interleaving of the yarn y with all of the yarns and loops behind yarn y. Similarly, let $\iota'_< = \mathcal{I}_{<y}[S']$ and $\iota'_> = \mathcal{I}_{>y}[S']$. Then $\mathcal{E}[S \xrightarrow{m_1} S'] = id_n|^{\iota'_<}_{\iota_<}(id_1|^{\iota'_>}_{\iota_>} id_m)$ and $\mathcal{E}[S' \xrightarrow{m_2} S] = id_n|^{\iota_<}_{\iota'_<}(id_1|^{\iota_>}_{\iota'_>} id_m)$. Therefore, by distributivity

$$\begin{aligned}
\mathcal{E}[S \xrightarrow{m_1;m_2} S] &= \left(id_n|^{\iota'_<}_{\iota_<}(id_1|^{\iota'_>}_{\iota_>} id_m)\right) \circ \left(id_n|^{\iota_<}_{\iota'_<}(id_1|^{\iota_>}_{\iota'_>} id_m)\right) \\
&= (id_n \circ id_n)|^{\iota_<}_{\iota_<}\left((id_1 \circ id_1)|^{\iota_>}_{\iota_>}(id_m \circ id_m)\right) \\
&= id_{n+1+m} \\
&= \mathcal{E}[S \xrightarrow{\texttt{nop}} S]
\end{aligned}$$

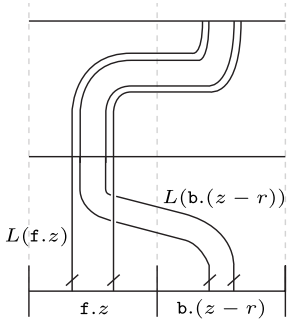
□

Rewrite Rule 3 (Squish)**.**

$$S \vdash \texttt{xfer}\ n.x\ n'.x'; \texttt{xfer}\ n'.x'\ n.x \cong \texttt{xfer}\ n'.x'\ n.x$$

Furthermore, when $L(n.x) = 0$ *in initial state* S,

$$S \vdash \texttt{xfer}\ n.x\ n'.x' \cong \texttt{nop}$$



Proof. Either $n.x = \texttt{f}.z$, in which case $n'.x' = \texttt{b}.(z - r)$; or $n.x = \texttt{b}.(z - r)$ and $n'.x' = \texttt{f}.z$. Without loss of generality, assume the latter case (the former is symmetric via flipping the diagrams 180°). The non-trivial part of the diagram denoted by this composite program is shown above. Once composed, this is the same diagram that $\texttt{xfer f}.z\ \texttt{b}.(z - r)$ denotes. If $L(\texttt{f}.z) = 0$ in S, then this sub-diagram is simply id_{2n}, which is denoted by nop as well—demonstrating the second claim. □

D.3 Subprogram Machine Location

We have now explained how to cancel and commute various knitout instructions relative to each other. This allows us to change the order in which we perform operations and remove redundant operations. However, it doesn't yet allow us to "reschedule" programs in the sense of adjusting gauge or changing which needle we use to perform a substantive operation (e.g., knit, tuck).

In the case of the tuck operation, which produces the same yarn topology independent of the bed it occurs in, a single xfer operation can be used to change the needle argument to the same physical location on the opposite bed.

Rewrite Rule 4 (Slide)**.** *Let* $n.x$ *and* $n'.x'$ *be defined such that they are the pair* $f.z$ *and* $b.z - r$, *or the pair* $b.z - r$ *and* $f.z$. *Then let* $ks(n.x) = \texttt{tuck}\ dir\ n.x\ l\ (y, s)$.

$$S \vdash ks(n.x);\ \texttt{xfer}\ n.x\ n'.x' \cong ks(n'.x');\ \texttt{xfer}\ n.x\ n'.x'$$

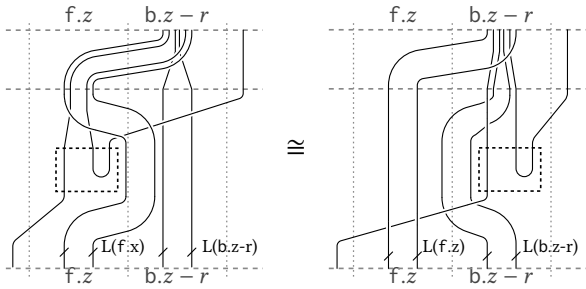
Proof. Without loss of generality, assume $n.x = \texttt{f}.z$, $n'.x' = \texttt{b}.z - r$, and $dir = +$ (the other cases involve symmetric diagrams of equivalent complexity). We can see that a simple application of Reidemeister moves R3 and R4 can be used to slide the fence produced by tuck under front bed loops and over the back bed loops to transform from the diagram on the left to the diagram on the right. Further, observe that in the case where $L(n.x) = 0$, the diagram can be further simplified and a xfer instruction removed via Rewrite Rule 3.

Table 2. When performing operation $ks(dir, n.x)$ = `knit` $dir\ n.x\ l\ yarns$ or $ks(dir, n.x)$ = `tuck` $dir\ n.x\ l\ (y, s)$, conjugate either moves ks one needle to the *left* (n.x-1) or one needle to the *right* (n.x+1). In the back bed case $ks(dir, \mathtt{b}.n)$, conjugate only uses the `SHIFT` macro. In contrast, the front bed case $ks(dir, \mathtt{f}.n)$ requires additional `miss` instructions to route yarns to the correct physical carrier location. The correct ordering of `miss` and `SHIFT` operations that prevents intertwining of loops and carriers depends on the dir parameter, producing two extra cases each. Note that all six cases require preconditions similar to those described in the proof of Rewrite Rule 5.

	Front		Back
	+	-	any

Left	`miss - f.`$x-1$ $yarns$
	`SHIFT(f.`$x, r, -1$`)`
	$ks(+, \mathtt{f}.x-1)$
	`miss + f.`x $yarns$
	`SHIFT(f.`$x-1, r-1, 1$`)`
		`SHIFT(f.`$x, r, -1$`)`
		`miss - f.`x $yarns$
		$ks(-, \mathtt{f}.x-1)$
		`SHIFT(f.`$x-1, r-1, 1$`)`
		`miss + f.`$x-1$ $yarns$
			`SHIFT(b.`$x, r, -1$`)`
			$ks(dir, \mathtt{b}.x-1)$
			`SHIFT(b.`$x-1, r-1, 1$`)`

Right	`SHIFT(f.`$x, r, 1$`)`
	`miss + f.`x $yarns$
	$ks(+, \mathtt{f}.x+1)$
	`SHIFT(f.`$x+1, r+1, -1$`)`
	`miss - f.`$x+1$ $yarns$
		`miss + f.`$x+1$ $yarns$
		`SHIFT(f.`$x, r, +1$`)`
		$ks(-, \mathtt{f}.x+1)$
		`miss - f.`x $yarns$
		`SHIFT(f.`$x+1, r+1, -1$`)`
			`SHIFT(b.`$x, r, 1$`)`
			$ks(dir, \mathtt{b}.x+1)$
			`SHIFT(b.`$x+1, r+1, -1$`)`



□

In order to move an operation to a different needle on the same bed, we must use a sequence of operations. All resulting loops and yarn carriers produced by the operation would then need to be moved back to match the appropriate end state S'. This is particularly important for the `knit` instruction, which has a mirrored structure depending on which bed it's performed on (the difference between a knit and a purl in hand-knitting). Continuing the algebraic analogy to group theory, we might expect a structure similar to ghg^{-1} (the conjugation of h by g) and h to be similar or equivalent given a suitably trivial g. In fact, this is the right way to think about moving operations around, though the exact knitout operations in g and g^{-1} vary depending on the operation being conjugated. Due to the asymmetry in machine operations `rack`, `knit`, and `tuck`, generating the correct sequence of routing operations requires breaking conjugate into six different cases seen in Table 2. All six cases use similar logic for transforming between the fenced tangle diagrams. Thus we walk through the proof of only one case below.

Rewrite Rule 5 (Conjugate [f, +, Left])**.** *Let* $ks(dir, n.x)$ *be either a* `knit` *or* `tuck` *instruction* $ks(dir, n.x)$ = `knit` $dir\ n.x\ l\ yarns$ *or* $ks(dir, n.x)$ = `tuck` $dir\ n.x\ l\ (y, s)$*. (we will simply refer to* (y, s) *as yarns in the* `tuck` *case). Let* S *be the state prior to* ks*. If the following needles are empty* $L(\mathtt{b}.x - r) = 0$, $L(\mathtt{f}.x - 1) = 0$*, and if there are no yarn carriers in the way that we are not using* $Y^{-1}(\lfloor \mathtt{f}.x, - \rfloor_r) = yarns$*, then*

$$S \vdash ks(+, \mathtt{f}.x) \cong \mathtt{miss\ -\ f.}x-1\ yarns;\ \mathtt{SHIFT(f.}x, r, -1\mathtt{)};\ ks(+, \mathtt{f}.x-1);\ \mathtt{miss\ +\ f.}x\ yarns;\ \mathtt{SHIFT(f.x-1,\ r-1,\ 1)}$$

(where `miss` *on multiple yarns is simply a sequence of* `miss` *operations, one for each yarn)*

Proof. The proof is given in figure 20. We briefly expound on details here.

Because only and exactly $yarns$ are present at $\lfloor \mathtt{f}.x, - \rfloor_r$, the unconjugated diagram has no initial $\overrightarrow{V}_{yarns}$; only the final $\overleftarrow{\Lambda}_{yarns}$. In the conjugation diagram, this final merge cancels against the initial separation of the knit instruction, allowing $yarns$ to become separated from the other yarns initially parked at $\lfloor \mathtt{f}.x - 1, - \rfloor_r$. The rest of the diagram fairly trivially deforms back to the unconjugated diagram, using standard Reidemeister moves. □

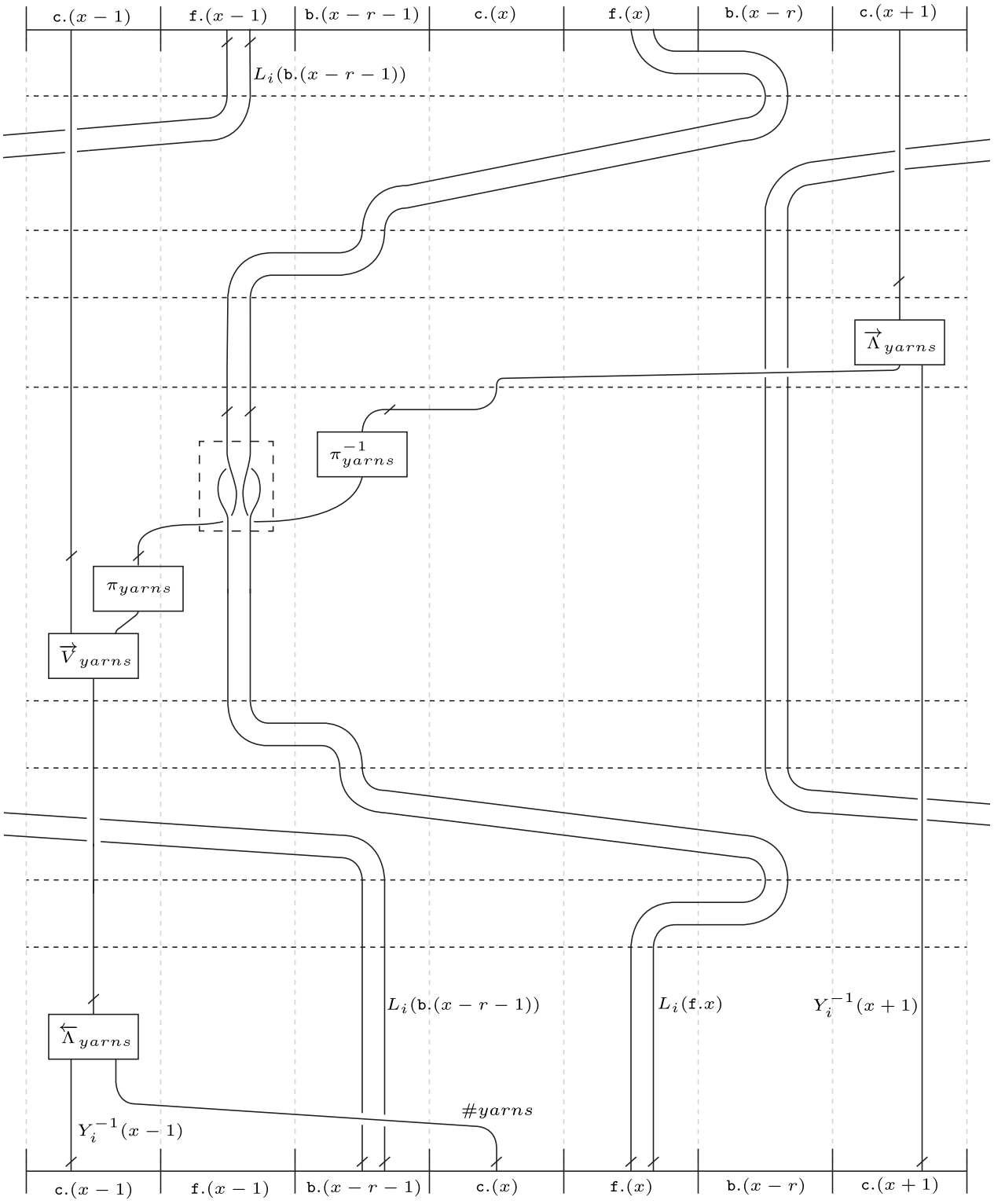


Fig. 20. The fenced tangle denoted by the conjugate left program. Note it is equivalent to the fenced tangle for `knit + f.`x l *yarns* as seen in figure 12i.