Formalizing Object Equivalence in Machine Knitting

Jenny Lin

CMU-CS-24-148
August 2024

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
James McCann, Chair
Jan Hoffmann
Scott Hudson
Adriana Schulz, University of Washington

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

This research was sponsored by Shima Seiki Mfg Ltd and the National Science Foundation under award numbers 1955444 and 2319182. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Computer-aided Manufacturing, Computer-aided Design, Program Semantics, Domain Specific Languages, Machine Knitting, Knot Theory

Abstract

Correctness is a desirable property for any program, whether that program computes an equation, controls a machine, or interprets data. Defining what it means for a program to be correct can be surprisingly nuanced, however, especially when that program is used to create a physical object. We can reframe this problem by treating correctness as a question of equivalence. Given some target object, is the result of a fabrication process equivalent to the target object? However, this now requires that we answer the still complicated question of what it means for two objects to be equivalent. In order to do so, we not only need a precise definition of object meaning, but also a strong understanding of how we create and interact with the objects around us.

In this thesis, I tackle this problem of meaning and equivalence for machine knitting programs. Knitting is the act of taking a few strands of yarn and deforming them into interlocking loops that result in a stable structure. While knitting machines are capable of quickly fabricating a vast array of structures with controllable material properties, the complexity of both the machine control process and the resulting physical object makes translating between the two incredibly difficult. This gap prevents existing programing and design tools from accessing the full breadth of its fabrication possibilities. To address this, I formally characterize the complete space of machine knitting programmings. I begin by introducing fenced tangles, a novel mathematical object designed to match intuition about knit object meaning. From there, I use fenced tangles to define a semantics for knitout, which is a low-level language for controlling v-bed knitting machines. The underlying program meaning is then used to reason about the correctness of a set of practical program transformations. I then use this semantic function as guidance for developing Instruction Graphs, which are an intermediate representation of knit objects. Unlike existing knit object representations, Instruction Graphs can capture the full range of machine knittable objects and can be verified as machine knittable using three easy to check graph embedding properties. Finally, I discuss how fabrication constraints may enable an algebraic approach to computing machine knitting program equivalence.

Acknowledgments

Like with many parts of my dissertation, I write these acknowledgements with many strong thoughts and the vague wish that I was better at expressing them. First off, Jim I literally couldn't have done this thesis without you, but even in a more figurative sense, I can't imagine having anyone else as a thesis advisor. Thank you for teaching me about research, food, and life. I'm proud to be your student. Next, I'd like to thank my committee Jan, Scott, and Adriana. Like this thesis, you cover a broad range of specialties, and I really appreciate the broad range of advice you gave me on research and academia. On the topic of academia, I'd like to thank the many professors that gave me professional advice, with a special shoutout to Mor. Thank you for being my supplemental advisor. Your wisdom and confidence helped keep me grounded through the arduous job search.

The Textiles Lab is full of cool people who have been a ton of fun to work with, both in research and more esoteric pursuits. Thank you Nur, Ella, Gabrielle, Yuichi, Catherine, Himalini, Michelle, Holly, Teadora, Pratyay, and others for being amazing labmates. Vidya, thanks for being a cool collaborator, role model, and friend in the Textiles Lab and during my internship at Amazon. Tom, thanks for delving into braid theory with me. Let's get fused braids figured out! Lea, I will try to stop screaming in excitement whenever I see you, but that is probably a lost cause. In addition, I've been lucky to work with many wonderful external collaborators. Thank you Sabetta, Jonathan, Gilbert, Yuka, Cem, and Yura for doing funky knitting research with me. I look forward to doing more in the future. And of course, the Textiles Lab would be in shambles without the administrative support of Jess and then Brian. Thanks for helping me navigate paperwork!

It is possible I could've done my thesis without the support of my friends, but I wouldn't bet on it. I'd like to give a general thanks to all the cool friends I met at board games night, and everyone at Women and NB lunch, with a special thanks to the various organizers throughout the years. I am bad at scheduling hangout time with friends, so it's nice to have that part figured out! One of the things I will miss most about CMU is getting to just hang out and chat with the amazing folks in the graphics lab. Thank you Ticha, Arjun[1], Other Arjun, Dorian, Nicole, Ruta, Chris, Hossein, Olga, Nick, Bailey, Tanli, Nupur, Zoë, Rohan, Sreekar, and everyone else I've forgotten to mention. A special shoutout to graphics Evan, who was an amazing office mate and my Dungeon Master for five(!) years over the course of two(!!) campaigns. It's thanks to you that I got to meet Adam, Sara, Tyo, Crucible, Deriaz, and Drae, and I'm so much happier because of it. And now for the non-exhaustive list of additional friends I didn't want to categorize: thanks to Quanquan, Annie, Jess, Ziv, Sol, Yvonne, David, Marissa, Pallavi, Sara, Ray, Katherine, Justin, PL Evan, Aria, Giulio, Gaurav, and Francisco. Ben, I'm glad we're the sort of friends who can clean out each others fridges. Let's keep stranging voids for years to come. And thank you Mark. Without you, my life would have less laughter, and the notation for horizontal composition of fenced tangles would be $\times$ instead of $\otimes$. Truly, a narrowly avoided disaster.

Finally, thank you to my family. Junjun, you've always been like a cool, older brother to me. Vivi, you are my cool younger sister, and I'm so proud to have you in my life. And thank you Mom, Dad, and Abu, for loving me, and supporting me, and raising me into the person I am today. I love you.

[1] Send more pictures of Snowy

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computational fabrication is the use of computation to help improve the creation of physical objects. It is a research area that encompasses a broad range of subjects from Computer Aided Design (CAD) interfaces to the development of Computer Numerical Control (CNC) machines. In addition to the usage of computers as direct tools for fabrication, computation is also used to study and improve the fabrication process itself. Much like in traditional computer science domains, fabrication is concerned with efficiency, optimality, and feasibility. And with the rise of fabrication tools controlled by computers running instruction sequences, many computer science techniques can find surprising direct applications in fabrication. Underlying all of these research problems is a fundamental question that must be addressed: what does it mean for a fabrication process to have made the right thing?

A principled approach would be to establish a series of desired properties and evaluate whether the object has these properties. If it does, it is "right", and if doesn't, it is "wrong". This turns our question of *correctness* into one of *equivalence*: given some ideal object (either physical or abstract), is some other object the same in all the ways that matter? Of course, changing the question does not necessarily mean we have answered it. First, we must decide what our desired properties are, a task that not only requires understanding of the object at hand, but also a way to characterize these properties in a way amenable to evaluation. For example, imagine your favorite jacket is starting to wear out. I might declare that its size is an important property to consider when we compare it to potential replacements. Yet in order to do this comparison, we must define what the "size" of your jacket is. Is it the length of its sleeves? The volume of material? The size category assigned to it by its manufacturer? The usefulness of our definition is affected by both easy it is to evaluate as well as how reflective it is of reality; measuring multiple features of a jacket is certainly more tedious than checking whether it is medium or large, but given the lack of standardization of fashion sizing, it may still save on heartache in the long run.

Let's say we've decided on a series of measurements, or a list of numbers that describes the size of your jacket. Now imagine that I want to compare your favorite jacket against a sewing pattern; put another way, I want to define equivalence between a set of fabrication instructions, and a physical object. At first blush, this seems impossible, as an equivalence relation is defined on two objects from the same mathematical domain. However, much like we can define a mapping from a physical object to a set of equivalence criteria, we can define a second mapping on our fabrication instructions. So long as we define a function that maps this sewing pattern to a list of the same size, we can compare the lists of numbers and use that to reason about equivalence between your jacket and a sewing pattern.

What I have described is essentially a *denotational semantics*, or a way of formalizing a concrete representation like a computer program (or a sewing pattern) as an abstract mathematical object. We may then reason about properties such as equivalence in the mathematical domain and apply them to

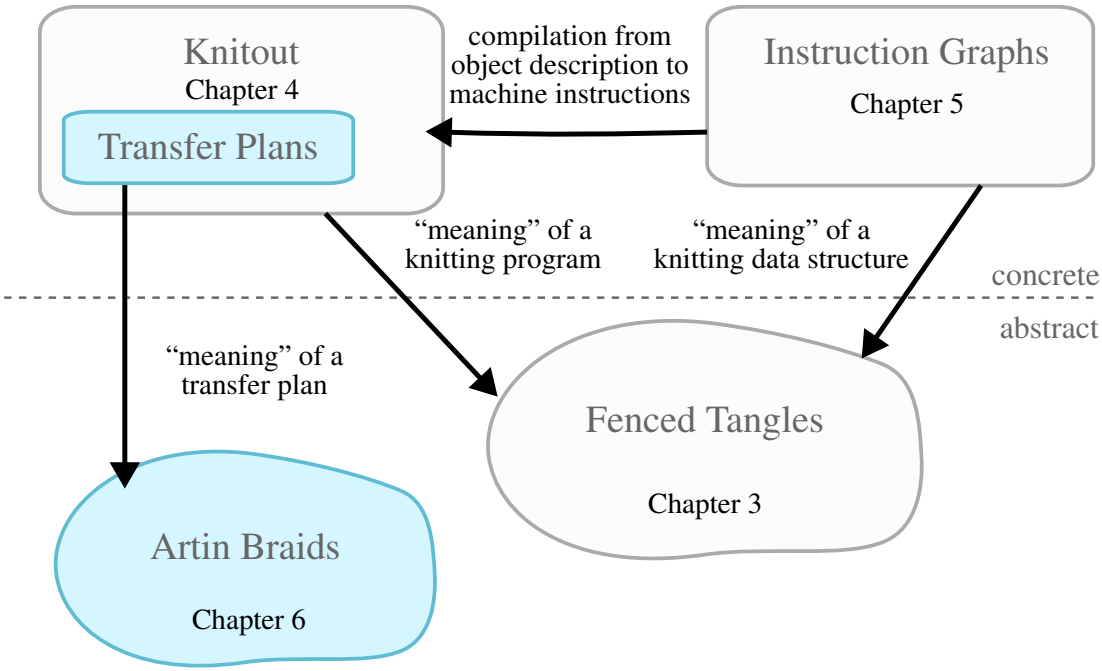


Figure 1.1: The thesis roadmap.

the concrete representation. There are three problems we must keep in mind to ensure our semantics is useful. First, is that of the domain, or what concrete representations we will seek to formalize. Next, is the co-domain, or what abstraction we will use when reasoning about formal equivalence. Finally, is the denotation function itself, or how we map from the concrete representation to the formal abstraction. After all, we may define all sorts of nonsensical mappings, such as insisting all jackets are size ’M’, but if they do not reflect reality, they are of no use. In this thesis, I will tackle these problems specifically within the context of machine knitting.

1.1 Thesis Structure

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 [Ou et al. 2019; Aigner et al. 2022]. To complement this development, several high-level design and programming systems have been developed to aid in creating increasingly complex objects. However, these tools only support a of subset of machine knitting programs, and they suffer from bugs at their edge cases. This is because the complexity of both the machine control process and the resulting physical object makes translating between the two incredibly difficult. Thus the full capabilities of machine knitting remain remain locked to all but a small group of experts.

I will begin by building our intuition about knitting, both the physical material we interact with in the real world, as well as the process of knitting using a machine. This includes both a review of machine

structure as well as how the control instructions, or knitting machine programs are generated. From there, I will move towards discussing a topological knitting equivalence using knot theory. This involves both reviewing prior work on using knot theory to characterize knitting as well as the first contribution of this thesis: the fenced tangle, which is a novel mathematical object defined with the explicit goal of being able to match our intuition of knit object equivalence. This leads us into applications of this idea (Fig. 1.1).

1. Chapter 4 uses the fenced tangle to define a denotational semantics for knitout, a low-level language that describes knitting machine actions. This semantics is then used to reason about semantics-preserving program rewrites and their use in knit programming.
2. Chapter 5 then proposes a new intermediate representation of knit objects called Instruction Graphs. Not only are Instruction Graphs capable of representing any machine knitable object, I prove that three graph properties are necessary and sufficent for toplogically correct lowering to knitting machine instructions.
3. Chapter 6 observes how restricting our attention to a subset of knit programs known as transfer plans and adjusting our program semantics to work with the Artin Braids enables computationally efficient verification of knit program equivalence. This in turn can be used to search for optimal programs.

Finally, I will conclude with some observations on alternative formalizations of knitting, other program properties this may allow us to analyze, as well as how insights from formally defining knit object equivalence might extend to other fabrication domains.

Chapter 2

Machine Knitting

We begin by building our intuition on what a knit object is: what key properties make one knit object distinct from another, with a focus on properties that can be controlled during the knitting process. From there, we review the structure of knitting machines and the basic operations available to it. We can define any language that allows the use of all cam plate setups and yarn carriers to be a *complete knitting machine language*. We then discuss how knitting programs are currently written: either by directly describing the sequence of machine operations with a low-level language, or by automatically generating machine operations from a high-level object representation.

2.1 Knitting Intuition

Knitting is the process of taking a few strands of yarn, deforming them into loops, and pulling those loops through other loops to produce a stable structure. The inset figure shows an example of a knit structure. We might be tempted to think of knitting as surfaces, given the most common examples of knitting around us are thin fabrics. However, this is counterproductive towards our goal of a generalized definition for knit equivalence. A key feature of knitting is how different stitch patterns, or ways loops are entangled with each other, can result in drastically different visual characteristics and macro-scale material properties. For example, in the inset figure, all loops were formed by pulling a loop from the back of an existing loop to its front. However, it is just as reasonable to pull a loop from the front to the back instead. Different arrangements of front and back knits can produce drastically different appearances and material properties. We may also do things like change the order in which loops are knit through, the way yarn connecting loops is routed, and the number of loops that are knit through at once. More advanced techniques can even be utilized to create surprisingly volumetric structures [Albaugh et al. 2021; Aigner et al. 2022]. While one can use a tile-based approach to annotate different regions of a surface with different stitch types, determining the correct set of tiles is actually a very complicated task, which I will discuss in more detail later on. Thus throughout this thesis, I will reason about knitting at the yarn-level, where the specific configuration of those yarn in space is what separates one knit object from another.

The yarns used to construct a knit fabric are typically pliable and can slide along other yarns. This results in soft, deformable fabric structures. Thus in addition to the standard translation and rotation we apply to rigid objects, we want to be able to bend and stretch knitting while preserving the underlying stitch structure. However, intuitively, not all of these transformations should preserve object identity. The yarn in a knit structure can be unravelled to undo the loops and re-knit into a completely configuration. Including this level of deformation would mean that all knit objects with an equal number of yarns would be considered equivalent; this trivializes the problem to the point of being meaningless. In practice, any

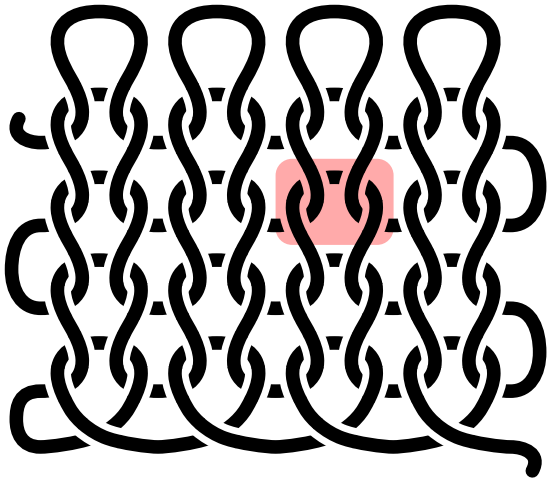


Figure 2.1: A rectangular swatch of knitting. Highlighted is a single knit.

loose ends in a knit object are secured during post-processing to prevent such unravelling. Once the ends are 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. Thus we want our equivalence relations to preserve these basic building blocks.

Finally, the characteristics of the yarn used in knitting has a large impact on the final object. The color, mechanical construction, and materials in the yarn can have a drastic effect on the final object. That said, yarn properties are not modified by the knitting process. Thus while choice of yarn has many implications for the fabrication process, we will not include it in our definition of knit object equivalence. What can be controlled by the knitting process is the amount of yarn used by each loop. Generally, using more yarn makes a loop larger and more flexible, while using less yarn makes it smaller and stiffer. As it turns out, addressing these properties is a surprisingly complicated problem, particularly within the context of knitting machine programs. Thus we leave these remaining properties for future work, which we discuss in Chapter 7.

2.2 Knitting Machine Structure

There are many different types of knitting machines with different architectures, but at a high-level, they all work by using hook-shaped *needles* to form loops out of yarn that is delivered by *yarn carriers*. Yarn carriers can be thought of as specialized tubes where one end of a yarn is threaded through its center and secured at the machine, either by the in-progress knit, or by grippers that hold inactive, or currently unused yarns. When a carrier is activated, or brought `in`, a gripper brings the free end to the needle that uses it and holds it secure until the yarn carrier is used, which, in theory, secures the yarn in the growing fabric and allows it to be safely released. Meanwhile, when a yarn carrier is brought `out`, a gripper grabs the carrier's yarn and cuts it. This detaches the carrier from fabric, allowing it to return to the inactive area. As a carrier's location changes relative to its attach point, yarn slides through the tube, causing it to release or (in a limited capacity) pull back in the same strand of yarn it is attached to. This yarn trailing from the carrier is what is manipulated by a needle to form a new loop. For example, during the `tuck` operation, the needle's hook reaches out and grabs the yarn delivered by a carrier. The needle hook pulls back in, thus creating a new loop on the needle. If loops already exist on the needle, new loops will be stacked

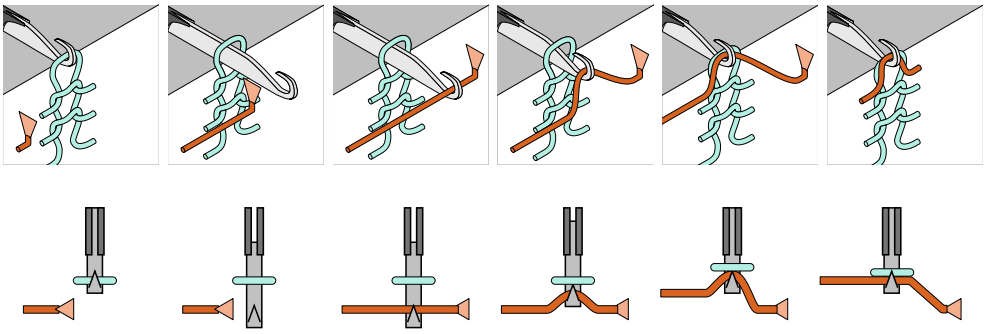


Figure 2.2: A needle performing the `tuck` operation

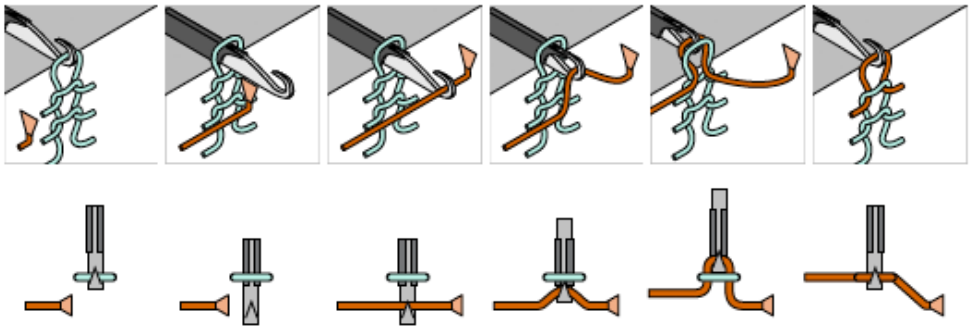


Figure 2.3: A needle performing the `knit` operation

closer to the needle tip. A picture of the `tuck` operation is shown in Fig. 2.2. Note that in the displayed example, the yarn carrier starts to the left of the needle, and moves to the right of it. Similarly, the `knit` operation also involves the needle hook reaching out to catch a yarn. However, the hook pulls further in, which pulls the new loops through all existing loops on the needle Fig. 2.3. The old loops then slide off the needle and hangs underneath it. Once a loop slides off a needle, it cannot be reacquired.

Note that in both these examples, the carrier started to the left of a needle. However, what if the carrier started to the needle's right? One option would be for the needle to grab the yarn as the carrier moves from right-to-left (here on known as the − direction). However, the carrier can also first `miss`, or move to the left of the needle before it is grabbed, and then perform the actual operation while moving left-to-right (the + direction). Because carrier movement always manipulates the location of its associated yarn, + and − variations form different loop geometries. This means any operation that uses carriers must specify which direction the carrier is moving.

A single knitting machine needle can accomplish very little, but by arranging multiple needles in a row called a *bed*, we can start to knit sheets of fabric by forming each loop in a row using a different needle. A wide variety of textures can be knit just by using a single bed and `knit`, `tuck`, and `miss` operations [Twigg-Smith et al. 2024]. However, by introducing a second row of needles, we can greatly increase the variety of knittable objects, both at the local stitch level and the global geometric level. V-bed weft knitting machines (Fig. 2.5) consist of two facing beds. Between the two beds runs a number of tracks, each of which has a single yarn carrier that provides yarn. By knitting on a mixture of front bed and back bed needles, the resulting fabric will have a mixture of front knits and back knits. In addition, a second bed can be used to make tubular structures by flattening the tube and assigning each half to one bed. Finally, we can use pairs of needles on opposing beds to perform an additional operation. The `split`

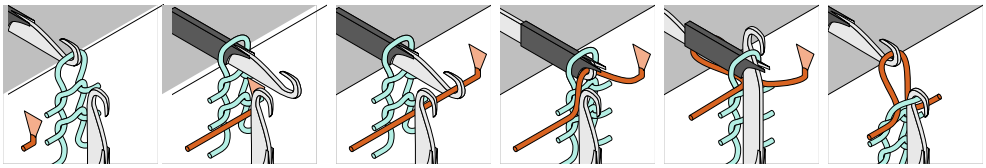


Figure 2.4: Two needles performing the `split` operation

operation also forms new loops using yarn from carriers. But instead of dropping the old loops, they are instead transferred to a different needle Fig. 2.4. The loops are moved without inducing any twist, which means if we look at the order of a loop stack relative to its needle (e.g. from base to tip) the order of the transferred stack is reversed relative to its new needle.

All three of these operations can use anywhere from zero to all of the carriers on the machine. In the case where multiple carriers are used, certain machines can control the relative order of the newly made loops; earlier carriers in the sequence are closer to the base of the needle. The zero carrier case is common enough that `tuck`, `knit`, and `split` have the aliases `amiss`, `drop`, and `xfer`. Of particular interest is the `xfer` operation, which can be used to move existing loops on the machine to new locations. While `split` and `xfer` can only be performed on needles that are directly across from each other, v-bed knitting machines can also perform a `rack` operation, which slides the two beds parallel to each other to change needle alignment between the two beds.

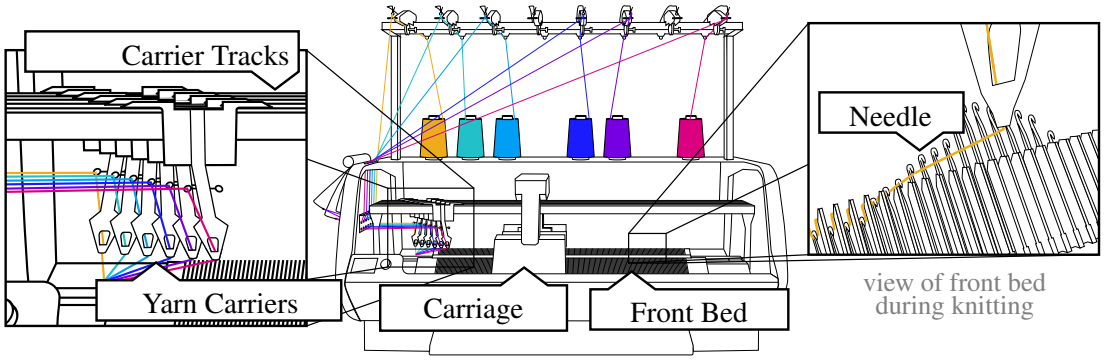


Figure 2.5: A v-bed knitting machine creates fabric by using a carriage to actuate needles arranged into front and back beds. The beds are positioned in an inverted "v" shape, with the back bed behind the front bed (and, thus, not visible in this illustration). Yarn is supplied to the needles by yarn carriers which run along carrier tracks. (Figure based on [Sanchez et al. 2023].)

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 important implications. 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 rearranging knitting instructions without changing program meaning is an important task for knit programmers.

```
1  ;!knitout-2
2  ;;Carriers: 1 2 3 4 5 6 7 8 9 10
3  in 2
4  tuck - f3 2
5  tuck - b2 2
6  tuck - f1 2
7  knit + b2 2
8  knit + f3 2
9  knit - f3 2
10 knit + b2 2
11 knit - f1 2
12 out 2
```

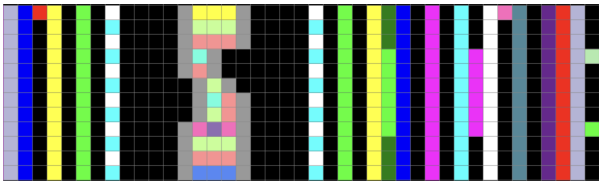


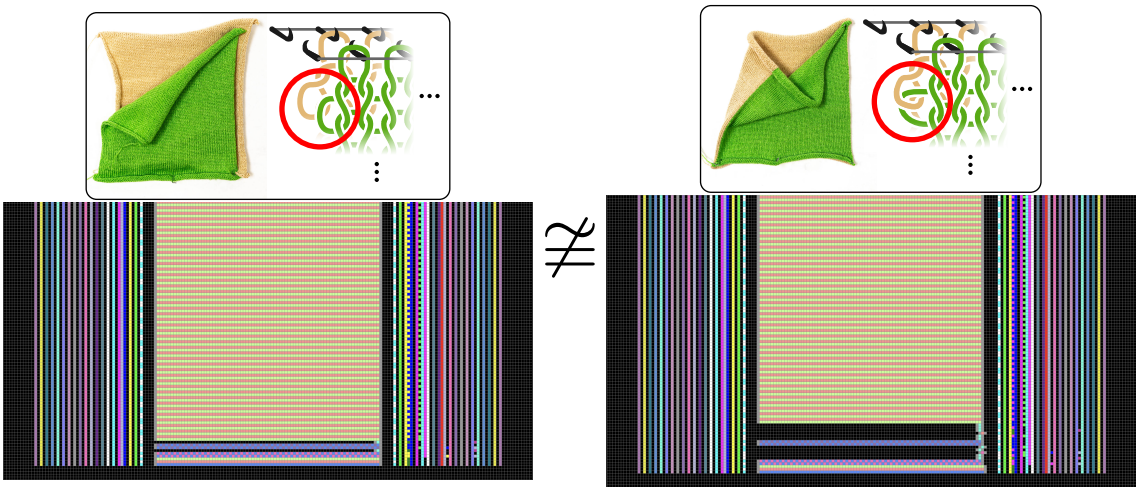
Figure 2.6: A knitout program is merely a sequence of knitting machine operations. When compiled to a DAT program for a Shima SWG machine, operations are consolidated into passes.

2.3 Direct Machine Programming

Knitting programs have traditionally been written using low-level languages that directly control the knitting machine with a sequence of instructions. The most basic language is knitout [McCann 2017], which is a generalized language for v-bed knitting machines. It is purely a sequence of machine operations with no control flow or variables, and it does not explicitly encode pass information.

Proprietary languages from knitting machine vendors, such as KnitPaint [Shima Seiki 2011] and M1Plus [Stoll 2011], are "written" in a two-dimensional grid, where each row is a single carriage pass (i.e. slice in time) and each column represents a machine location. The value in each cell specifies which machine operation is performed at that location, while parameters for a given pass are located to the sides. Machine-specific details are necessary to ensure that all operations in row can actually be actuated during a single carriage pass. The grid-based structure used by KnitPaint and M1Plus can allow for some understanding of the object structure, but is still divorced from the output geometry. While the x-axis corresponds to location on the machine, and is thus correlated with the location of loops within the resulting object, the y-axis is fabrication time, not position in the object. Thus programs with many localized operations that do not span the full width of the object will appear stretched out. Furthermore, depth-wise information about the object is not explicitly visualized and must be inferred from the choice of operation. These factors combined mean that even visual programs are difficult to interpret as objects.

For example, in Fig. 2.7, we see two very similar programs. Both programs alternate knitting between carriers 3 and 5, where one carrier knits on front bed needles 1 to 50, while the other knits on back bed needles 1 to 50, the difference being which carrier is used on which bed. Already, the DAT files looks more like a single striped object instead of two layers of fabric, with the most noticeable difference being a few extra rows at the beginning of program Fig. 2.7b. In fact, the actual difference between the resulting objects is at the edges of the knit. In program Fig. 2.7a, the two layers are unconnected, resulting in two separate sheets, while program Fig. 2.7b connects the layers to form a bi-colored tube.



(a) Stacked Sheets (b) Linked Sheets

Figure 2.7: 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 ($\cong$) and distinctions ($\ncong$) among knitting programs.

These CAD systems do 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 [Soft Byte Ltd. 1999; Shima Seiki 2011]. Guidebooks of advanced techniques also exist that can assist with this process [Underwood 2009]. In addition, meta-programming interfaces like KnitScript [Hofmann et al. 2023] and the JavaScript frontend of knitout [Carnegie Mellon Textiles Lab 2024] provide abstraction and encapsulation, which does reduce the repetitive and laborious tasks associated with low-level programming.

However, using any meta-programming language still requires a detailed understanding of how to assign low-level structures within the knit object to machine resources and how to use machine operations to create these structures. Even if knit fabric simulation was advanced enough to provide rapid, predictive previews of a program's output, this would still require the programmer to repeatedly iterate within the realm of machine operations and evaluate in the object space, where the effect of program changes on the resulting object can be quite unintuitive. A natural question is whether it's possible to compile from a knit object representation to low-level instruction sequence.

2.4 Automatic Program Generation

By now, I have hopefully made it clear that low-level languages are incredibly basic and unintuitive, akin to assembly and gcode in terms of level of control. The amount of domain expertise required to create even relatively simple objects serves as a high barrier of entry and impeeds experts from leveraging the full capabilities of knitting machines. To simplify this process and make knitting machine programming more accessible, researchers have developed various systems that enable the generation of knitting machine programs from user-friendly, high-level specifications. These systems fall into the category of automatic compilers, which aim to abstract away the complexities of low-level programming and allow users to focus on the design aspects of their knitted objects. These methods all start with data structures that represent the desired object but do not explicitly describe exact machine operations. When converting these data structures into knitting machine programs, these compilers must both specify which instructions to use for

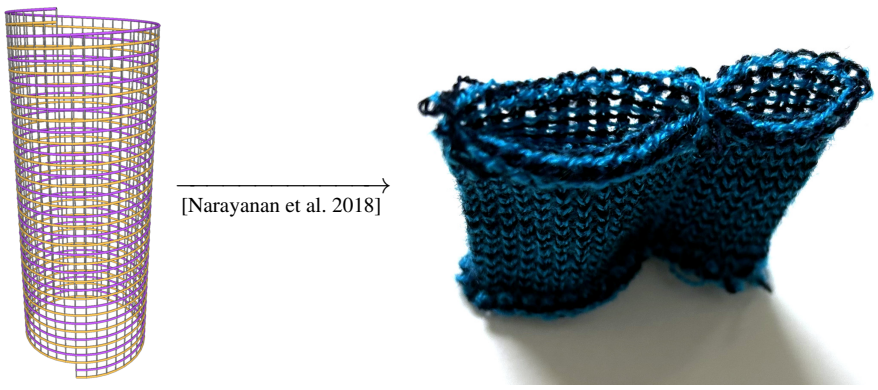


Figure 2.8: Stitch graphs, the post-tracing knit graph representation used in Autoknit [Narayanan et al. 2018], can *represent* a two color striped tube; but their scheduling approach fails to faithfully translate this into machine knitting. Instead, the output contains yarn tangles because their scheduler does not consider carrier crossings.

creating the specified object, as well as when and where on the machine the instructions will occur. For hiistorical reasons, the later problem is known as *scheduling* in the knitting machine literature.

Early systems used primitives that are easy to compile into machine operations. For example, McCann et al. [2016] proposes a set of sheet and tube primitives that can be composed into more complicated shapes. Meanwhile, Popescu et al. [2018] segments a mesh into disc-shaped patches which must be manually seamed to form the final structure. These methods only support one type of knit texture, greatly simplifying instruction generation, and they sidestep the difficult problem of scheduling by working with scheduled primitives or using disc-shaped (thus easy-to-schedule) patches, respectively. The Autoknit system [Narayanan et al. 2018] was the first to include a compiler that converts an unscheduled knitting representation (*stitch graphs*) to scheduled machine instructions (as part of an automatic pipeline to converts a 3D meshes into knit objects). Since then, the Autoknit compiler has received two major developments. Visual Knit [Narayanan et al. 2019] allowed small programs to be attached before and after each node in the graph. These pre- and post- programs can be used during instruction generation to modify and insert knit structures. Knit Sketching [Kaspar et al. 2021] extended Autoknit to perform scheduling of sheets. Many unscheduled knitting design systems essentially propose different high-level specifications that are lowered to a stitch graph, which is then put through the Autoknit compiler [Narayanan et al. 2019; Kaspar et al. 2019; 2021; Jones et al. 2021; Wu et al. 2021; Mitra et al. 2023].

Because Autoknit's compiler was developed within the context of a larger design system, it carries assumptions from that system that can lead to incorrect outputs. For example, the system does not intend multiple yarns to be used at the same time unless they are carefully controlled, which leads to yarn tagging when knitting striped objects (Figure 2.8). At a lower level, the collapse-shift-expand transfer planner used by Autoknit [McCann et al. 2016] does not guarantee any particular stacking order of loops in decreases (an important visual feature in lacework [Lin et al. 2018]). Further, the knitability constraints Narayanan et al. [2018] describe for knit graphs (the data structure that is "traced" to form stitch graphs) are a mixture of hard fabricability constraints, soft metric constraints for approximating the input mesh geometry, and best-principle guidelines for generating more reliable knitting programs. So, while they do form a nice set of intuitive knitability guidelines in the context of a knitting system for 3D meshes, they don't exactly define the borders of what is possible in machine knitting.

In contrast to Autoknit's knit graphs (and their lowered stitch graphs) which only allow a few simple stitch types, *augmented stitch meshes* [Narayanan et al. 2019] store complex knitting sub-programs

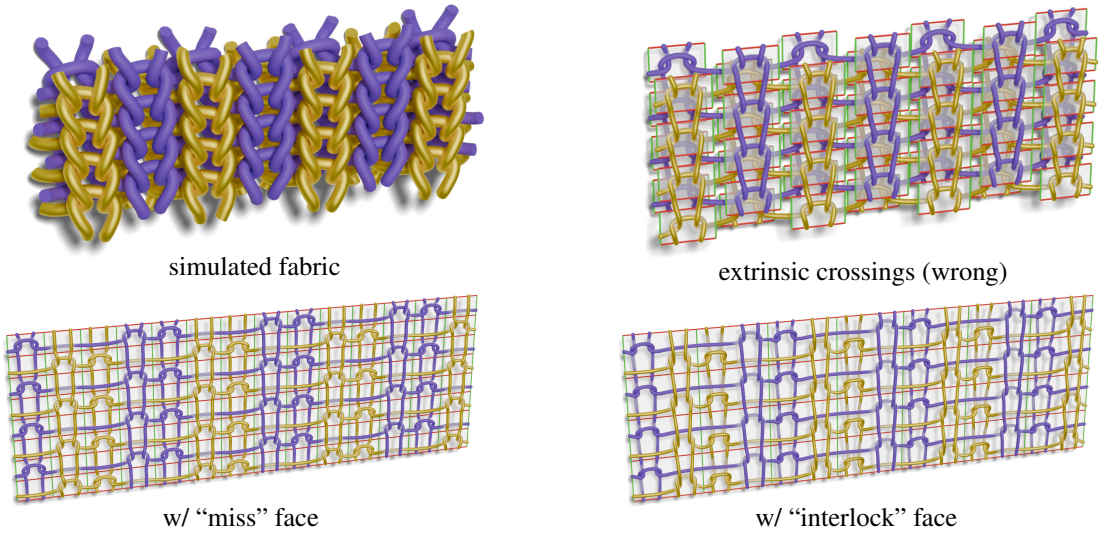


Figure 2.9: Attempting to represent a 7-column, 4-row swatch of interlock fabric with stitch meshes. Right-going yarns are shown in gold; left-going yarns are shown in purple. Loop edges are red, yarn edges are green. Arranging stitch faces as they appear in the final fabric (top right) is aesthetically pleasing, but semantically incorrect, because it doesn't capture important yarn/loop crossings in a face. Capturing these crossings requires either, bottom-left, building a basic block of interlock from sub-stitches; or, bottom-right, capturing a larger block of the interlock pattern in its own face. In either case, the fabric's appearance is distorted by the requirement to route loops and yarns over separate edges.

in mesh faces, and use connections between face edges to indicate resource routing. Augmented stitch meshes provide a nice way to visualize and edit many common knitting structures, but require that all "interesting behavior" in the mesh is captured inside faces or their connectivity (*not* their embedding). This can create problems when depicting certain multi-layer fabrics, like interlock. Interlock is a dense fabric composed of two interleaved layers of knit rib. Building interlock using augmented stitch meshes, Figure 2.9, requires flattening this two-layer fabric in a way that no longer resembles the desired output; since the "natural" approach of using multiple layers hides important yarn/loop crossing information outside the faces and their connectivity.

There are two systems that do not rely on a variant of stitch graphs and autoknit's compiler. The first is a system for program generation with elasticity control by using single-jersey jacquard patterns [Liu et al. 2021]. While they also used a dependency graph generated from a knittable stitch mesh, their library of patterns required a custom compiler that does not generalize to other patterns. The second is KnitKit [Nader et al. 2021], which aims to be a more flexible system for knit program generation from a high-level design. KnitKit bridges high-level mesh representations to low-level machine instructions by using input configurations, or "actions". Input configurations involve expert-designed graph pattern matching, which KnitKit's lowering process handles automatically. The resulting "action graph" is then compiled to machine instructions using expert-authored routines. This method is akin to employing custom compiler passes external to the compiler, similar to LLVM or MLIR passes [Lattner & Adve 2004; Lattner et al. 2021]. Verifying the correctness of KnitKit's compiler requires a general method of verifying action-routine pairs. The expressivity of their system is also linked to the choice of action library; a more diverse action library broadens the range of expressible knitting programs, but also requires careful authoring by an expert.

Chapter 3

Knit Object Equivalence

As a reminder, we have decided that knitting is a collection of yarns, where the specific arrangement of yarns in space is one of its important characteristics. We want some deformations of the object, like stretching, to preserve equivalence, while others, like unraveling, to not preserve equivalence. What I've just described is evocative of knot theory, or the study of one-dimensional objects embedded in space. Indeed, it has been observed that an adequate mathematical characterization of knit objects ought to be rooted in knot theory [Grishanov et al. 2009; Markande & Matsumoto 2020; Qu & James 2021]. But similar to the situation in solid modeling, existing formalisms are subtly insufficient for capturing the complete scope of machine knit objects. We begin with a review of the necessary topology concepts before describing existing knot theoretical approaches to knitting and their limitations for reasoning about object equivalence.

3.1 Topology Terminology

Topology is the study of *topological spaces* and *homeomorphisms* between such spaces. A precise definition of topological spaces relies on point set topology. For more details, please see a standard reference [Munkres 2000]. Intuitively, a topological space is a set of points where each point within the space has some neighborhood of points that are also within the topological space. A *homeomorphism* between topological spaces is a bijective function that locally preserves the neighborhood for points within the space (i.e. it is *continuous*). For example, the open disk S^2 is homeomorphic with the Euclidean plane $\mathscr{R}^2$ because we can stretch the disk out until its boundary is at infinity to produce the Euclidean plane, and we can shrink the plane back down to return to the disk. From the perspective of points within the disc, they are always surrounded by a neighborhood of points, while points near the boundary continue to not have neighbors "past" infinity. In contrast, the torus T^2 is not homeomorphic to the disk, as any mapping inevitably puts a boundary point from the disk on then interior of the torus, which has no boundary. This results in a discontinuity, where the size of the neighborhood suddenly jumps.

While homeomorphism is a kind of equivalence relation, it is not the equivalence relation used for reasoning about knots. This is because homeomorphism is only concerned with the two topological spaces in question and does not care about how a space might overlap with itself as it is transformed. Instead, we work with *embeddings*, which are continuous, injective functions from one topological space to another. Given embeddings $f, g : X \to Y$ (X and Y topological spaces), an *isotopy* is a continuous function $H : X \times [0,1] \to Y$ s.t. $H(x,0) = f(x)$, $H(x,1) = g(x)$, and $H(x,t)$ is an embedding for every t. Intuitively, the second parameter of H can be understood as "time" s.t. the whole isotopy can be understood as a continuous motion or interpolation between f and g. As it turns out, directly reasoning about isotopy

between embeddings is also subtly incorrect for knot theory, as it allows us to start with any knot, shrink down regions of it over $[0,1)$, and then transform the regions to straight line segments at $t = 1$. Instead, knots are defined to be equivalent under *ambient isotopy*. Embeddings $f, g : X \to Y$ are ambiently isotopic if there is an isotopy H from the identity $id : Y \to Y$ to some other homeomorphism $h : Y \to Y$ s.t. $\forall x \in X : H(f(x), 1) = g(x)$. That is, intuitively H is a warp of the entire ambient space H that warps f into g. This, in turn, is the same as being able to bend, stretch, and squash the object described by f to transform it into g in a way that avoids "cutting" the object.

3.2 Knitting and Knot Theory

The standard definition of a mathematical knot is an embedding of the circle in Euclidean space: $S^1 \to \mathscr{R}^3$. Embeddings of multiple circles are called links. Unlike in knitting, knots and links have no loose ends. This is in part because all embeddings of the arc $[0,1]$ in Euclidean space are equivalent under ambient isotopy (as matches our intuition of knit unraveling!). While one can connect loose ends to each other to produce closed circles, different choices of connections produce different kinds of links, making this strategy suboptimal for reasoning about object equivalence. Notably, if we abstract knitting to be infinite two-periodic structures instead of general arrangements of yarn in space, this suggests a canonical method of connecting loose ends. Both Markande & Matsumoto [2020] and Grishanov et al. [2009] observed that two-periodic knit structures can be represented with a link embedded in thickened torus $T^2 \times [0,1]$. Grishanov et al. [2009] then used knot invariants to classify different types of textile structures, which Markande & Matsumoto [2020] used them to reason about knittability properites and basic composition. The ability to use classical knot theory techniques to study knit structures is incredibly powerful. Unfortunately, the two-periodic assumption prevents us from applying this technique to equivalence of knit objects.

An alternative is to represent knitting using other topological objects adjacent to knots. Most promising for our purposes are tangles [Adams 1994].

Definition 3.1 (Tangle)**.** Let U be a topological space homeomorphic to the closed ball S^3 with distinguished equator $Q \subset \text{bd}(U)$, which is homeomorphic to the circle S^1. A tangle is an embedding of zero or more arcs and circles $\gamma_i : [0,1] \to U$ where the endpoint of each arc lies on equator Q. Two tangles T_1 and T_2 are equivalent under ambient isotopy.

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

Definition 3.2 (Tangle Diagram)**.** Let V be a topological space homeomorphic to the closed disc. A *tangle diagram* in V is an immersion of zero or more arcs and circles $\gamma_i : [0,1] \to V$, where there are a finite number of transversal intersections p_i between 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 V, or Reidemeister moves 1, 2, or 3 (Fig. 3.2).

We say that a tangle diagram K is a *projection* of a tangle T, Figure 3.1, if there is a projection of $\mathbb{R}^3$ to $\mathbb{R}^2$ sending U to V, Q to $\text{bd}(V)$, 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 3.3 (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 3.4. 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 3.1)

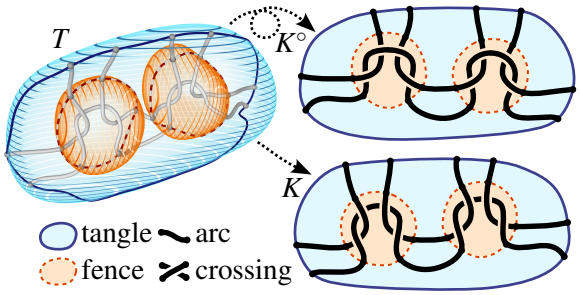


Figure 3.1: A fenced tangle, T, and two projections, K and K°, to fenced tangle diagrams which differ in their equator orientation.

Characterizing knitting as tangles would allow us to attach all end points to the equator of U. This prevents them from moving to the interior to trivially unravel everything. However, when defining our tangle, we must also commit to an ordering of end points on the equator; changing their ordering requires overlapping two end points on the equator, which is not a valid embedding. While we can define a convention for attaching the endpoints of knitting to the equator's boundary, doing so inevitably impacts equivalence between objects, which is less than ideal. What's more, knitting techniques such as plating rely on friction to keep yarns from moving around too much within the knit object. Unrestricted ambient isotopy on the tangle's internals means these techniques can not be captured. To address this, my co-authors and I developed a new mathematical object known as the *fenced tangle*.

3.3 Fenced Tangles

A fenced tangle is defined similarly to a regular tangle, but additional *fences* which surround parts of the tangle and restrict ambient isotopy there. Fences are essentially sub-tangles that are forbidden from intersecting with each other. They are allowed to hold endpoints, which allows fenced tangles to be detached from the larger boundary while preventing uncontrolled unravelling, so long as fences are placed appropriately. The introduction of fences allows for semantically meaningful sections of knitting to be preserved, while still allowing for the kind of flexible deformations expected of knitting. In addition, we can define a standard presentation of fenced tangles that enables composition of fenced tangles. This will be useful in the following chapters, where we will use fenced tangle composition to not only define semantic functions, but also reason about program equivalence, machine knitability, and bounds on program efficiency.

3.3.1 Basic Definitions

Definition 3.5 (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 embeddings of this reference "fenced sphere" $L_i : S^2_L \to U$. These fenced spheres must satisfy the following conditions (i) all spheres are disjoint in U. (ii) all intersections between arcs and fences 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 addition to the equator of U or joined up into a circle, endpoints of arcs may also lie on fences. Two fenced tangles are equivalent if there is an ambient isotopy between them which also carries fences to fences (Fig. 3.3).

Given a fenced tangle diagram K on V, let fences be embeddings of the circles $L_i : S^1 \to V$ satisfying

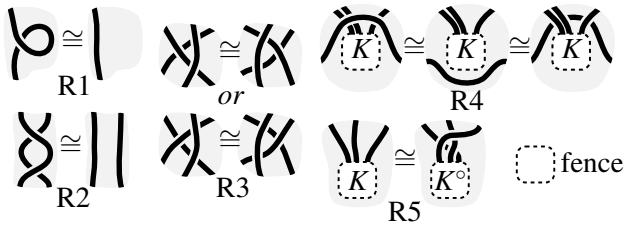


Figure 3.2: Equivalent fenced tangle diagrams are connected by sequences of smooth 2D deformations along with Reidemeister moves (R1-3) and fenced-tangle Reidemeister moves (R4, R5), which work regardless of the number of arcs connected to the fence.

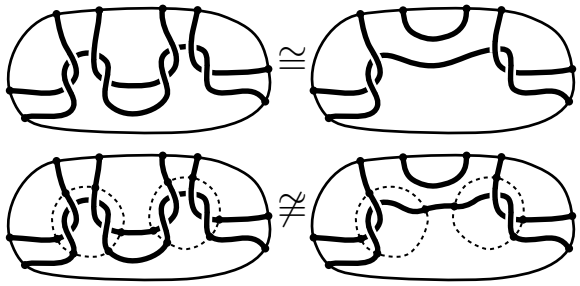


Figure 3.3: Tangles without fences (top) can locally "unravel". Fences (bottom) prevent unravelling by restricting the motion of arcs at crossings. This is key to capturing the as-fabricated topology of knit items.

the following conditions: (i) all fences are disjoint in V. (ii) all intersections between arcs and fences 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. 3.2).

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°.

3.3.2 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. To do this, we first define a standard fenced tangle presentation:

Definition 3.6 (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 3.7 (Slab Types)**.** It will be useful to refer to the set of (n,m)-slabs by $\mathscr{S}_n^m$, so that we may simply write $K \in \mathscr{S}_n^m$.

We then define three types of tangle concatenation (see Fig. 3.4 for pictorial intuition).

Definition 3.8 (Horizontal Concatenation)**.** Let $K_1 \in \mathscr{S}_n^m$ and $K_2 \in \mathscr{S}_p^q$. 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

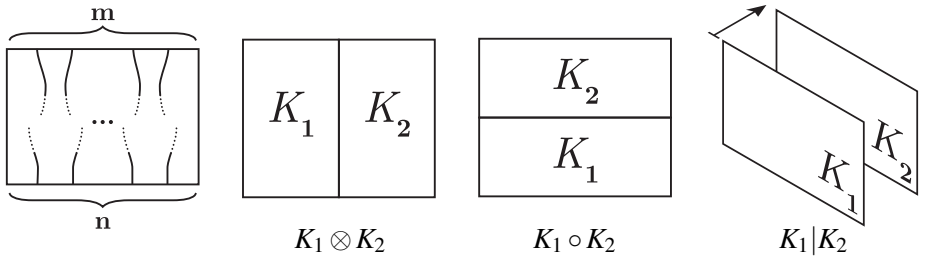


Figure 3.4: Slab presentation and three types of fenced tangle concatenation. From left to right, an (n,m)-slab, horizontal concatenation $K_1 \otimes K_2$, vertical concatenation $K_1 \circ K_2$, and layer concatenation $K_1|K_2$

horizontal concatenation $(K_1 \otimes K_2) \in \mathscr{S}_{n+p}^{m+q}$.

Definition 3.9 (Vertical Concatenation)**.** Let $K_1 \in \mathscr{S}_n^p$ and $K_2 \in \mathscr{S}_p^m$. 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 $(K_1 \circ K_2) \in \mathscr{S}_n^m$ by gluing the two rectangles along the matching top/bottom.

Definition 3.10 (Interleavings)**.** Let $m,n \in \mathbb{N}$. Then, an interleaving ω of m and n ($\omega \in \mathscr{I}_{m,n}$) 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 $\overline{\omega} \in \mathscr{I}_{n,m}$ be the opposite interleaving, specified by the second set of ω.

Definition 3.11 (Layer Concatenation)**.** Let $K_1 \in \mathscr{S}_n^m$ and $K_2 \in \mathscr{S}_p^q$, with both defined on the same rectangular region R (also achievable by ambient isotopy). Furthermore let $\iota \in \mathscr{I}_{n,p}$, and $\omega \in \mathscr{I}_{m,q}$ be interleavings of endpoints of K_1 and K_2 on the bottom (input) and top (output) of this common rectangle R. Then, $(K_1|_\iota^\omega K_2) \in \mathscr{S}_{n+p}^{m+q}$ 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.

Lemma 3.12 (Concatenations are Equivalence-Invariant)**.** Let $K_1 \cong K_1' \in \mathscr{S}_n^m$, $K_2 \cong K_2' \in \mathscr{S}_p^q$, and $K_3 \cong K_3' \in \mathscr{S}_m^p$. 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 3.13. 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. □

3.3.3 Permutation Tangles

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 3.14 (Identity Slabs)**.** Let $id_n \in \mathscr{S}_n^n$ 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 3.15 (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 $\pi_o \in \mathscr{S}_n^n$ 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 3.16 (Separate and Merge)**.** Let $\iota \in \mathscr{I}_{n,p}$ 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 $\overleftarrow{\mathrm{V}}_\iota = \pi_{o_\iota}$, and *separate to the right* as $\overrightarrow{\mathrm{V}}_\iota = \pi_{o_{\bar{\iota}}}$. 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: $\overrightarrow{\Lambda}_\iota \circ \overleftarrow{\mathrm{V}}_\iota = \overleftarrow{\Lambda}_\iota \circ \overrightarrow{\mathrm{V}}_\iota = id_{n+p}$. Examples of the four slabs are given in Fig. 3.5.

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.

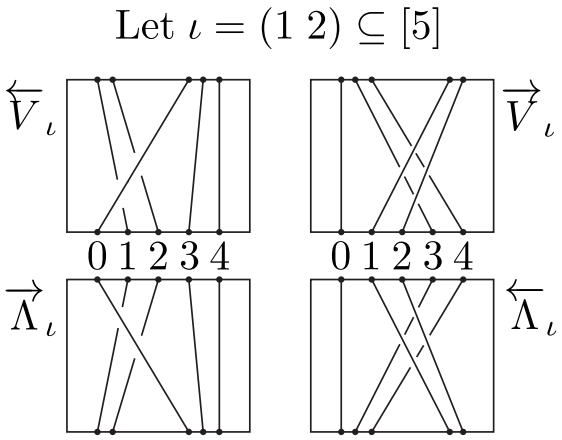


Figure 3.5: Given the particular interleaving $\iota = (1\ 2)$ we can define two separate and two merge slabs, varying by the direction in which the yarns identified by ι are merged from or separated to. The arrows act as a mnemonic to tell us which direction the ι yarns are being pulled (reading the slab from bottom to top), and the character acts as mnemonic for whether the yarns are being merged (Λ) or separated (V).

Chapter 4

Semantics for Knitting Machine Programs

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]. While knitout is a control language for knitting machines, not computers,[1]they too benefit from a programming languages approach. When knitout was first presented as a universal language for v-bed knitting machines, it was essentially defined using a small-step operational semantics, where each knitout operation causes the machine to perform some mechanical action that results in a change in an abstract knitting machine state [McCann et al. 2016]. While this semantics is useful for reasoning about properties of the knitting process, it falls short when reasoning about program equivalence. This is because what we want to preserve is not the exact actions taken by the machine, but the produced knit object.

Thus in this chapter, I formalize a *denotational semantics* of the machine knitting language knitout, where valid knitout programs denote, or "mean" a fenced tangle. This allows fenced tangle equivalence to be used to define an object-focused definition of knit program equivalence. From there, I use this semantics to create a set of *Rewrite Rules*, or program transformations that always preserve the denoted fenced tangle. The formalization structural overview of the chapter is shown in Figure 4.1.

4.1 Formalizing Programming Languages

To illustrate the concepts of used by this chapter 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.)

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

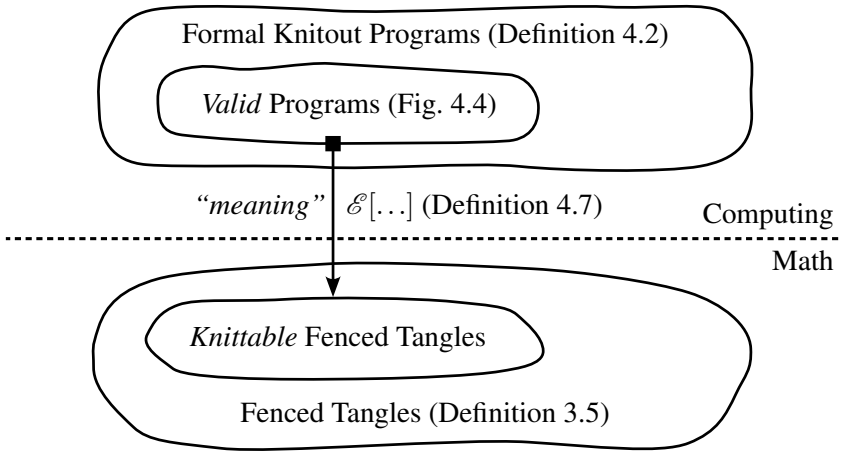


Figure 4.1: Formalization approach. The grammar of knitout (Def. 4.2) defines a set of programs, which is narrowed by our validity relation (Fig. 4.4). Every valid knit program denotes (i.e., "means") a fenced tangle (Def. 3.5) via formal knitout semantics (Def. 4.7, Fig. 4.5).

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

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 4.2.

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 I do not use in this thesis) 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.

For knitout, I will follow a second approach to typing. 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 4.1 (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 $[](a) = \bot$.
- $[a \mapsto b]$ is a *singleton* partial function, defined as $[a \mapsto b](a) = b$ and $[a \mapsto b](a') = \bot$ 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 $\sigma'(a) \neq \bot$, 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 $\sigma(a) \neq \bot$

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."

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

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

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

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

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, the analogue of this type-checking rule can be found in Definition 4.5 and Fig. 4.4. Rather than writing $\Gamma_1 \vdash s \dashv \Gamma_2$, I write $S_0 \xrightarrow{ks} S_1$, where S_0 and S_1 are abstract states of our knitting machine. This is 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 tje 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 $\mathscr{E}[\Gamma_0 \vdash s \dashv \Gamma_1] : \Sigma_{\Gamma_0} \to \Sigma_{\Gamma_1}$. In total, the function $\mathscr{E}$ specifies the semantics of our entire language, rather than a single program. We write $\mathscr{E}$ to suggest "evaluation."

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

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

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

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

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

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

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.

4.2 Formal Knitout

For multiple reasons, we will not formalize knitout as it is defined in the specification [McCann 2017] (hereafter "actual knitout"), but instead work with a more verbose version that imposes stricter validity relations while preserving program expressivity. I will begin by explaining the difference between formal and actual knitout and the reasoning for these changes. From there, I will specify the grammar of formal knitout in Definition 4.2 using Backus-Naur form (BNF). In Definition 4.5, 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 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 $\mathscr{E}[S]$ in Definition 4.6 as an intermediary step to defining the fenced tangle denoted by a valid knitout program $\mathscr{E}[S \xrightarrow{ks} S']$ (Definition 4.7). An example of our formal definitions applied to a specific program instance is found in Fig. 4.2.

4.2.1 Translation Between Formal Knitout and Actual Knitout

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. Needle locations do not have a period dividing the bed and index (`f1` vs `f.1`). There are a two "core" operations in actual knitout that are omitted from formal knitout for simplicity. Fractional racking (`rack`0.5) offsets the beds such that needles are interleaved instead of directly across from each other. This is useful for knitting denser fabrics, but otherwise does not affect object toplogy. Sliders are a feature on some v-bed knitting machines where a needle has a second storage location that cannot form new loops but can hold a stack of loops separate from the loops on its hook. This too is useful for knitting denser fabrics, but can be simulated v-bed machines without sliders by treating every other needle as a storage location.

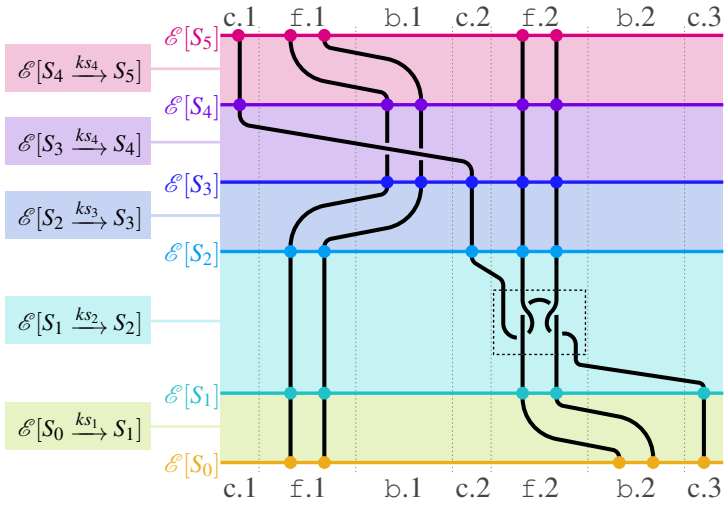
Actual knitout defines the mechanical meaning of operations in a way that simplifies the writing of common case machine actions, but causes the resulting knit structure to be highly dependent on machine

```
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 4.2)

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

(b) Program trace defined by validity relations (Fig. 4.4)



(c) Denoted fenced tangle (Fig. 4.5)

Figure 4.2: 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).

state. For example, in actual knitout the validity of a `knit` operation does not depend on the initial location of its carrier sequence. When the operation is executed, the machine will move any carriers that aren't already in position to the specified needle, during which the carriers will continue to release yarn. Only then can it knit with those carriers. Specifying the fenced tangle denoted by this definition of a `knit` operation not only requires considering all possible carrier locations, it must also consider additional loops and carriers passed by any moved carriers. This quickly becomes tedious to define and read. In contrast, formal knitout defines a `knit` operation to only be valid if its carriers are already in position. Carriers that aren't in position must explicitly be moved with `miss` operations. This not only lets us use the carriers' positions as a precondition when defining the fenced tangle, it also constrains the area of new knitting to the needle where the operation occurs. This makes it much easier to define the class of fenced tangles denoted by an operation as a composition of identity tangles where nothing has changed in the object and small, templated fenced tangle located at an operation's needle. For this reason, a formal knitout `miss` operation is only allowed to move one needle at a time, in contrast to `miss` operations in actual knitout, which can move past multiple needles. Formal knitout also removes other sequential operations: in actual knitout, the `miss`, `tuck`, `in`, and `out` operations can also accept a carrier sequence instead of just a single carrier, a single `rack` can change the racking to any value. Essentially, formal knitout is a de-sugaring of actual knitout that makes implicit carrier movement explicit and unrolls sequences of actions.

There is an additional class of implicit carrier movement that is present in actual knitout. 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 $n'.x'$. Using logical locations for carrier positions often reduces spurious tangles between carriers, which makes it desirable for program writing. This implicit carrier movement can be simulated in formal knitout by tracking logical carrier locations and inserting `miss` operations as is appropriate.

Finally, a few syntactic changes were made to make validity and semantics easier to define on an independent program trace. Formal knitout makes `in` and `out` specify a machine location, while actual knitout infers machine location from the first and last operation that uses the carrier, respectively. Loop size 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).

4.2.2 Formal Knitout Semantics

Definition 4.2 (Formal Knitout)**.** A formal 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). These logical locations specify the location of loops within the machine's structure, the validity and semantics of knitout programs are defined in terms of physical locations in space (Fig. 4.3). The distinction is defined as follows:

Definition 4.3 (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.

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

Definition 4.4 (Knitout Machine State)**.** A knitout machine state $S = (r, L, Y, A)$ consists of:

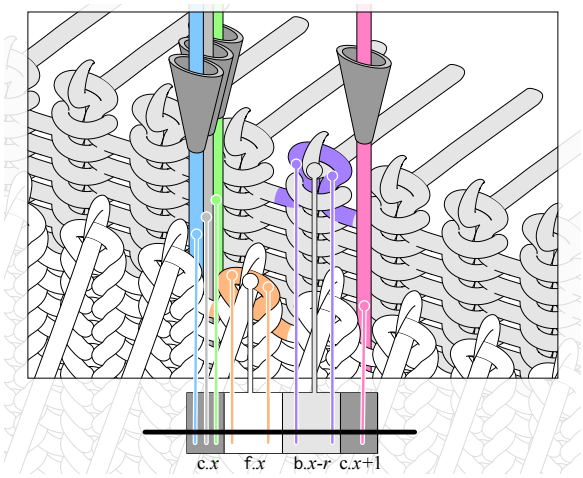


Figure 4.3: The knitting machine consists of two beds of needles where at racking r, front bed needle $\mathtt{f}.x$ is aligned with back bed needle $\mathtt{b}.x-r$. In between the needles are yarn carrier tracks. These logical machine locations are projected from 2D to 1D physical locations using a 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 $\mathscr{E}[S]$.

- $r \in \mathbb{Z}$, the racking offset, or the offset of the needles on the back bed relative to the front bed. At offset r, back needle $\mathtt{b}.x-r$ is across from front needle $\mathtt{f}.x$.
- $L \in \mathsf{nLoc} \to \mathbb{N}$, 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 4.1.

Definition 4.5 ((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 4.4). 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.

Definition 4.6 (Machine State Denotation)**.** Let $S = (r, L, Y, A)$ be a machine state. Then $\mathscr{E}[S]$, the denotation of S, is a set of points on a line, which is divided into annotated segments as follows (also see Figure 4.3, 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 $Y(k) \neq \perp$, 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 $\lfloor Y(l) \rfloor_r = i$.

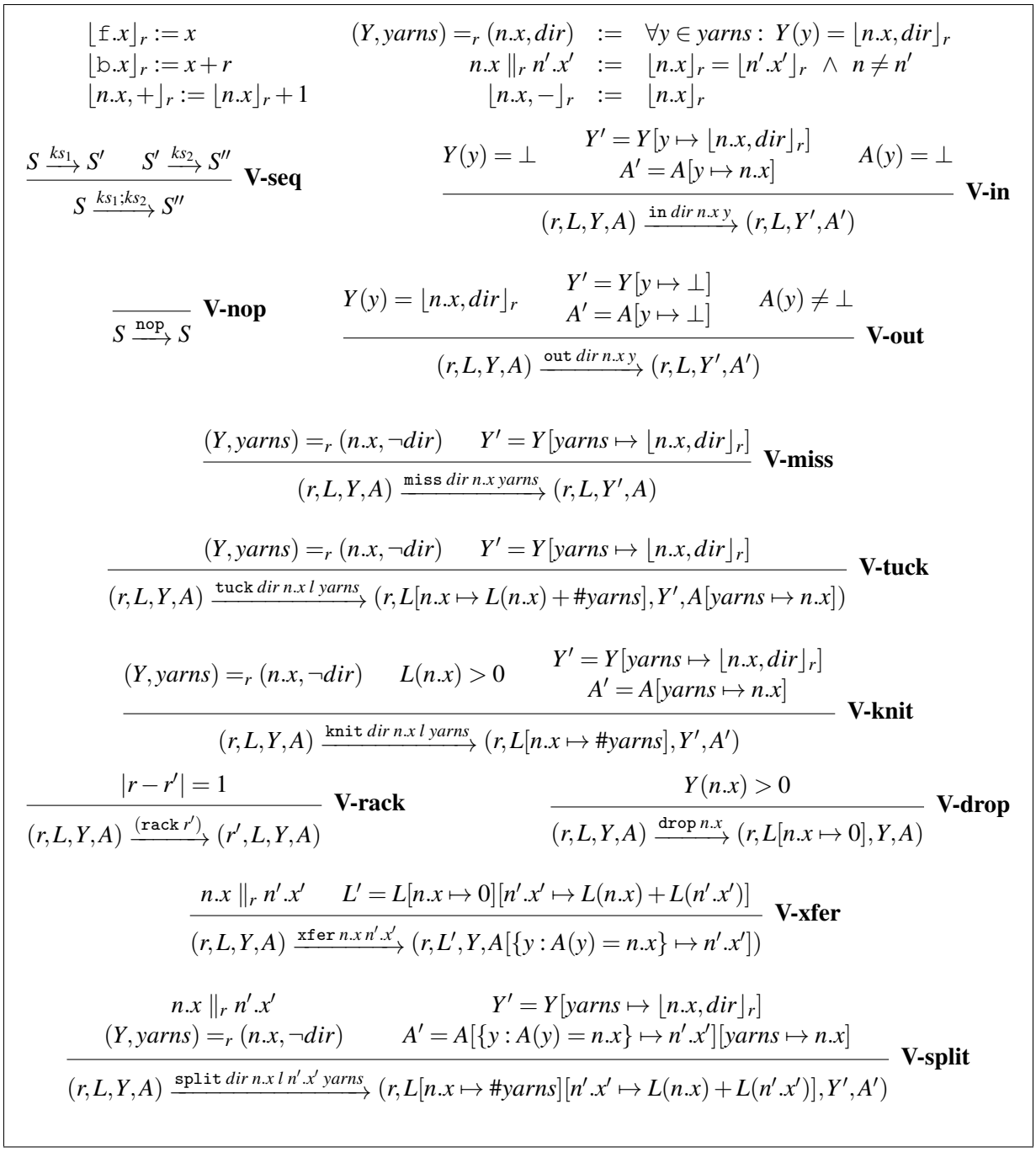
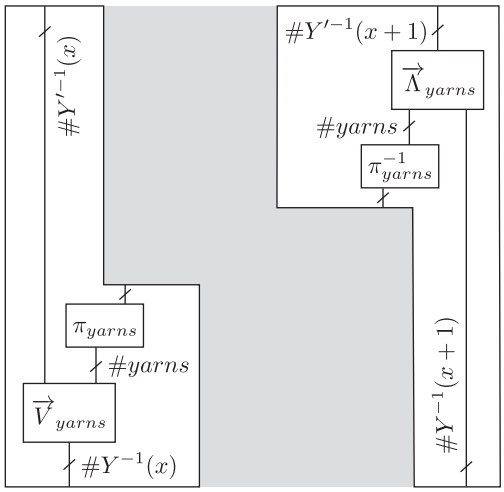


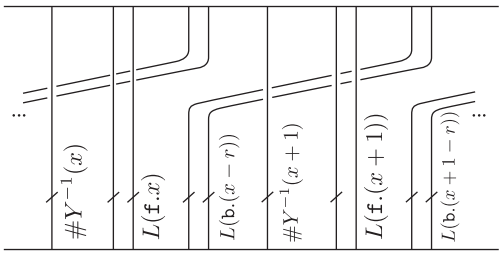
Figure 4.4: Validity relation for knitout programs (see Definition 4.5), where #*yarns* is the size of the yarn carrier sequence. Only valid knitout programs denote a fenced tangle. Note that for a fixed S and ks, S' is uniquely determined.

- 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)

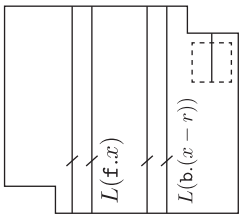
Definition 4.7 (Semantics of Knitout)**.** Let $kT = S_0 \xrightarrow{ks_1} S_1 \to \cdots \to S_n$ be a valid knitout program/trace. Then $\mathscr{E}[kT]$ is the fenced tangle which kT denotes, defined inductively. Throughout the definition, we will work with the slab presentation of fenced tangle diagrams. As an invariant, the input (bottom) boundary



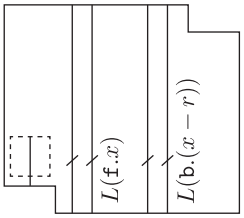
(a) Diagrams for individual knitout instructions are completed by being wrapped in this "frame"



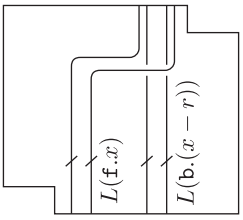
(b) Illustration of `rack` 1. See text for precise definition



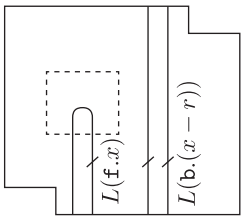
(c) `in` + `f`.*x y*



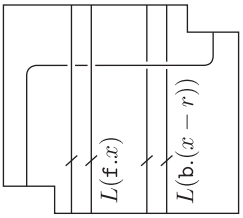
(d) `out` − `f`.*x y*



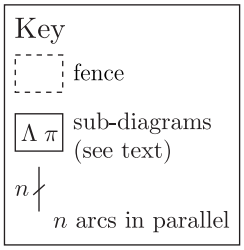
(e) `xfer` `f`.*x* `b`.$(x-r)$

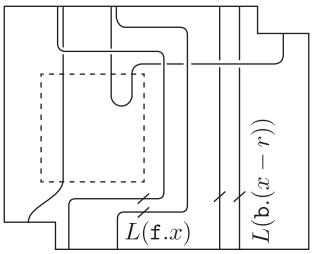


(f) `drop` `f`.*x*

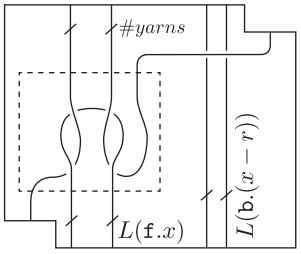


(g) `miss` + `f`.*x y*

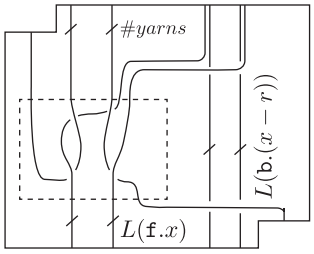




(h) `tuck` + `f`.*x l* (y,s)

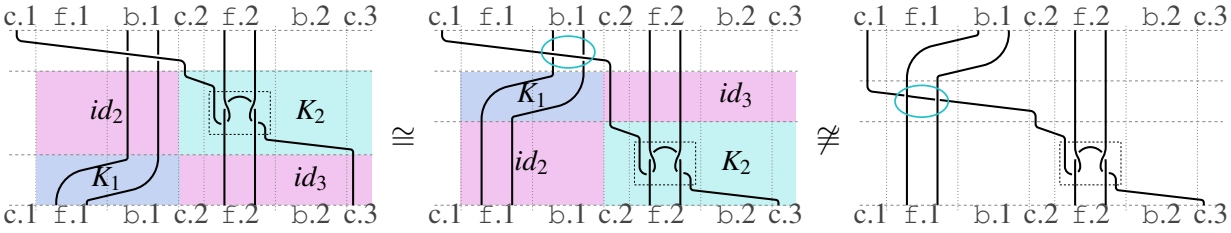


(i) `knit` + `f`.*x l yarns*



(j) `split` + `f`.*x* `b`.$(x-r)$ *l yarns*

Figure 4.5: Fenced tangles produced by knitout. Part of the definition of knitout semantics (Definition 4.7). 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.



(a) $ks_4;ks_3;ks_5$ (b) $ks_3;ks_4;ks_5$ (c) $ks_3;ks_5;ks_4$

Figure 4.6: 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 4.14. In contrast, fenced tangles (b) and (c) are not equivalent due to the change in crossing annotations in the circled region.

of $\mathscr{E}[S \xrightarrow{ks} S']$ will match $\mathscr{E}[S]$ and the output (top) boundary will match $\mathscr{E}[S']$.

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

The `nop` instruction does nothing, so $\mathscr{E}[S \xrightarrow{\texttt{nop}} S] = id$. Next, we handle the `rack` instruction. Let $kT = S \xrightarrow{\texttt{rack}\, r} S'$. We define $\mathscr{I}_{<\infty}[S]$ to be the partition of $\mathscr{E}[S]$ 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 $\iota = \mathscr{I}_{<\infty}[S] \in \mathscr{I}_{m,n}$ 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 $\omega = \mathscr{I}_{<\infty}[S'] \in \mathscr{I}_{m,n}$ 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 $\mathscr{E}[kT] = id_m|^{\omega}_{\iota} id_n$. (see Fig. 4.5b for an example illustration)

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 $\mathscr{E}[S]$, we define $\{\mathscr{E}[S] < pl\}$ to be the subset of all points that correspond to a physical location less than pl, while $\{\mathscr{E}[S] > pl\}$ is all points greater than pl. An examination of the validity relation definition (Fig. 4.4) makes it clear that $\{\mathscr{E}[S] < \lfloor n.x, -\rfloor_r\} = \{\mathscr{E}[S'] < \lfloor n.x, -\rfloor_r\}$ and $\{\mathscr{E}[S] > \lfloor n.x, +\rfloor_r\} = \{\mathscr{E}[S'] > \lfloor n.x, +\rfloor_r\}$. Thus the denotation of kT can be expressed as $\mathscr{E}[kT] = id_m \otimes T_s \otimes id_n$, where $m = \#\{\mathscr{E}[S] < \lfloor n.x, -\rfloor_r\}$, $n = \#\{\mathscr{E}[S] > \lfloor n.x, +\rfloor_r\}$, and T_s is defined for each operation according to figure 4.5.

4.3 Rewriting Knitout Programs

Having defined a formal denotational semantics on knitout using fenced tangles, we can now define what it means for two knitout programs to be equivalent. If we are purely concerned with the output of programs, one immediate definition would be topological equivalence of the denoted fenced tangle.

Definition 4.8 (Topological Equivalence of Knitout Programs)**.** Let kP_1 and kP_2 be the traces of two valid knitout programs. The programs are topologically equivalent if and only if $\mathscr{E}[kP_1] \cong \mathscr{E}[kP_2]$.

This raises the natural question of how do we *prove* whether two programs are equivalent. A general algorithm for determining fenced tangle equivalence is ideal but unlikely, given that solving knot equivalence is NP-hard[Koenig & Tsvietkova 2021]. Instead, we work with a simpler problem: given a program transformation f applied to knitout program ks, is the resulting program $f(ks)$ topologically equivalent?

Table 4.1: Rewriting equivalences (rules) proven in this chapter. Rules with asterisks have preconditions not present in this figure (see associated proof).

Name	Rule	Proof
*Swap**	$ks_1 \cong ks_2$ $ks_2 \cong ks_1$	§4.3.2
*Merge**	$ks_1 \cong$ `nop` ks_2	§4.3.2
Squish	`xfer` $n.x\ n'..x' \cong$ `xfer` $n'.x'\ n.x$ `xfer` $n'.x\ n.x$	§4.3.2
Slide	`tuck` $dir\ n.x\ (y,s) \cong$ `tuck` $dir\ n'.x'\ (y,s)$ `xfer n.x n'.x'` `xfer n.x n'.x'`	§4.3.2

*Conjugate**	$ks(+, \texttt{f}.x) \cong$ `miss - f.`$x-1$ *yarns*
	`SHIFT(f.`$x, r, -1$`)`
	$ks(+, \texttt{f}.x-1)$
	`miss + f.`x *yarns*
	`SHIFT(f.`$x-1, r-1, 1$`)`
		§4.3.2

Because the new program must be both valid and equivalent, we will define a stronger version of program equivalence.

Definition 4.9 (Contextual Equivalence of 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, $\mathscr{E}[S \xrightarrow{ks_1} S'] \cong \mathscr{E}[S \xrightarrow{ks_2} S']$, we say that ks_1 and ks_2 are equivalent in the context of S and write:

$$S \vdash ks_1 \cong ks_2$$

Sub-programs that are contextually equivalent can be used to rewrite larger programs while preserving topological equivalence.

Corollary 4.10 (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 $S \vdash ks_1;ks_2;ks_3 \cong ks_1;ks_2';ks_3$.

By proving small, general statements on program equivalence that can be applied within a larger context, we can develop a powerful tool for reasoning about the correctness of more complicated program transformations.

4.3.1 Rewrite Motivations

Now that we have decided to focus on program rewrites, we must decide on which rewrites to validate. A reasonable starting point would be rewrite rules useful to practical high-level compilation tasks. In the following section, I examine some common motivations for rewriting programs and provide an overview of the relevant rewrite rules. The heading **Rewrite Rule** designates particular lemmas (i.e., propositions) which are stated so that they are immediately applicable to the rewriting/scheduling of knitout programs. A high-level summary of the rewrites and their corresponding proofs in Section 4.3.2 are located in Table 4.1 .

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. Of course, not all operations can be reordered without changing the program meaning.

Rewrite Rule (*Swap*) observes that many knitout operations denote a fenced tangle of the form $id \otimes K \otimes id$ and uses this to define an extent function, which returns the non-identity region of a fenced tangle. The extent is then used to reason about when two operations commute with each other.

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

In addition to the extent analysis performed on general subprograms in Definition 4.23, I perform a special case analysis of the `SHIFT` macro used in Rewrite Rule 5 (Lemma 4.29).

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,

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

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 $L : \mathsf{nLoc} \to \mathbb{N}$ 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 thesis, we can define the following basic metric constraints to ensure physical plausibility:

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 $\texttt{f}.z$ and $\texttt{b}.z-r$, or the pair $\texttt{b}.z-r$ and $\texttt{f}.z$. Then let $ks(n.x) = \texttt{tuck}\ dir\ n.x\ l\ (y,s)$.

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

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 4.11 (Rack and Shift Macros)**.** Let

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

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

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

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 the same ending state. The correct sequence of routing operations is non-trivial to describe and dependent on the operation's initial bed $\{\texttt{f},\texttt{b}\}$, 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 $L(\texttt{b}.x-r)=0$, $L(\texttt{f}.x-1)=0$, and if there are no yarn carriers in the way that we are not using $Y^{-1}(\lfloor \texttt{f}.x,-\rfloor_r) = yarns$, then

$$\begin{aligned} S \vdash ks(+,\texttt{f}.x) \cong\ & \texttt{miss - f.}x-1\ yarns;\ \texttt{SHIFT(f.}x,r,-1);\\ & ks(+,\texttt{f}.x-1);\\ & \texttt{miss + f.}x\ yarns;\ \texttt{SHIFT(f.x-1, r-1, 1)} \end{aligned}$$

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

4.3.2 Rewrite Rule Proofs

Let us consider the example program shown in Fig. 4.2, 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 4.6 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. 4.6a can be transformed into Fig. 4.6b by an ambient isotopy. By contrast, Fig. 4.6c and Fig. 4.6b 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. 4.5, 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 4.12. For any fenced tangle slab $K \in \mathscr{S}_n^m$, vertical concatenation of the identity results in an equivalent fenced tangle:

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

Lemma 4.13 ($\circ$-$\otimes$ Distributivity)**.** Let $K_a \in \mathscr{S}_{n_1}^{m_1}$ and $K_b \in \mathscr{S}_{m_1}^{p_1}$ be one pair of vertically composable fenced tangles, and $K_c \in \mathscr{S}_{n_2}^{m_2}$ and $K_d \in \mathscr{S}_{m_2}^{p_2}$ be a second pair. Then the following compositions are

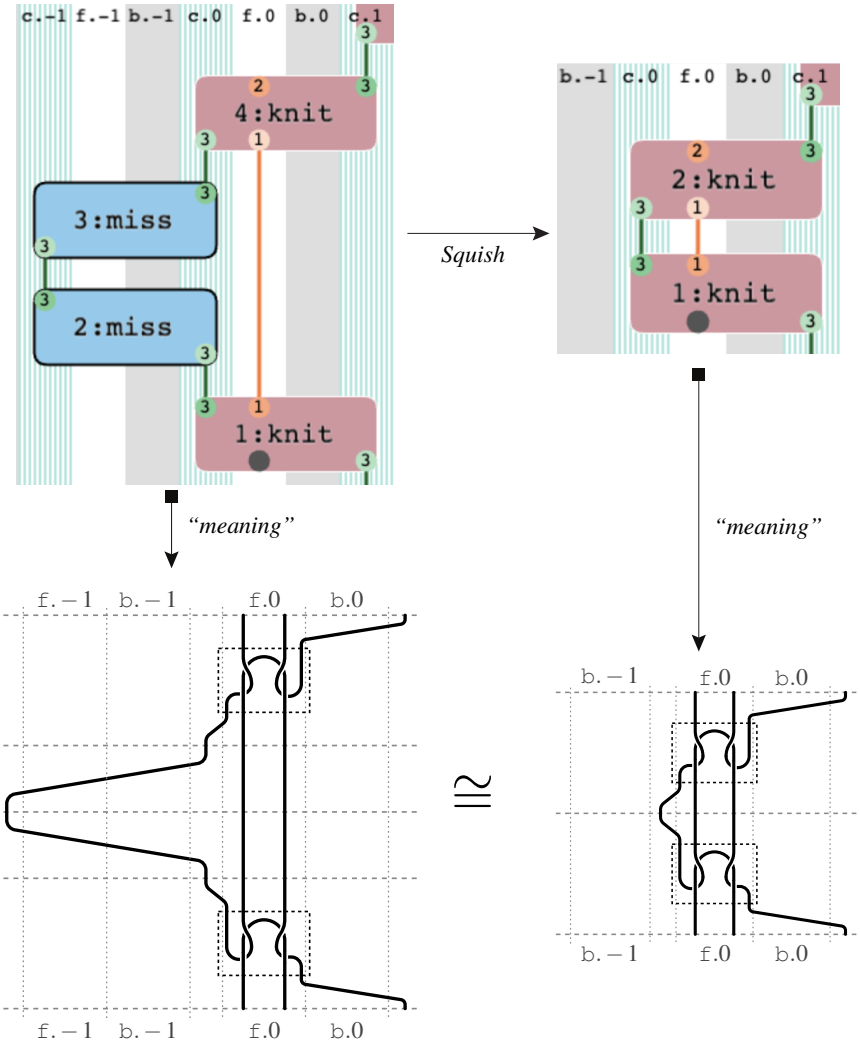


Figure 4.7: Screenshots of the rewrite-editor for *Squish* rewrite rule and the corresponding fenced tangles.

equivalent:

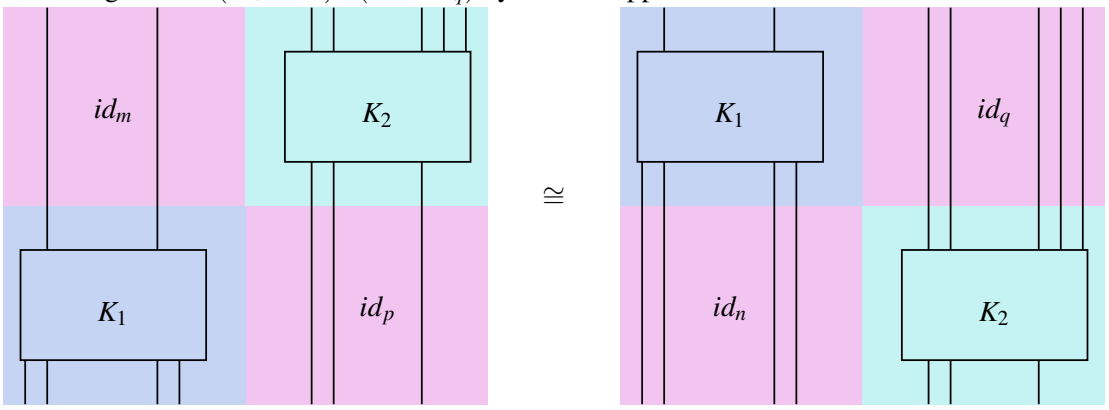
$$(K_a \circ K_b) \otimes (K_c \circ K_d) \cong (K_a \otimes K_c) \circ (K_b \otimes K_d)$$

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

Lemma 4.14 (Commutativity by Horizontal Separation)**.** For any $K_1 \in \mathscr{S}_n^m$ and $K_2 \in \mathscr{S}_p^q$ 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 4.13 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 4.12 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 4.13.



□

Many knitout operations denote (Definition 4.7) a tangle of the form $id \otimes K \otimes id$; and knitout program composition maps to vertical composition ($\circ$) of fenced tangles. Thus, intuitively, we should be able to use Lemma 4.14 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 $\mathrm{ex}(ks)$ (Definition 4.24) 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 $\mathscr{E}[S \xrightarrow{ks} S']$, but its depth-wise decomposition as well, for which we prove a similar commutativity property using Lemma 4.21. Using this extent function, we can state the following generalized Rewrite Rule for swapping knitout subprograms:

Rewrite Rule 6 (Swap)**.** Two operations can be swapped if their extents are disjoint: $S \vdash ks_1; ks_2 \cong ks_2; ks_1$ whenever $\mathrm{ex}(ks_1) \cap \mathrm{ex}(ks_2) = \emptyset$

If we return to our example program rewrite $S_1 \vdash ks_2; ks_3 \cong ks_3; ks_2$, we find that $\mathrm{ex}(ks_2) = [1.5, 2.5] \times [2, \infty]$ and $\mathrm{ex}(ks_3) = \{1\} \times [-\infty, \infty]$, making it an example covered by Rewrite Rule *Swap*. Meanwhile, $\mathrm{ex}(ks_4) = [1.5, 2.5] \times \{2\}$ intersects with $\mathrm{ex}(ks_3)$ in both dimensions. Thus Rewrite Rule *Swap* cannot be applied.

Notation 4.15 (Concatenation of Interleavings)**.** Let $\iota_1 \in \mathscr{I}_{n,p}$ and $\iota_2 \in \mathscr{I}_{m,q}$ be interleavings. Then $\iota_1 \sqcup \iota_2 \in \mathscr{I}_{n+m,p+q}$ is an interleaving defined (using set representations) as $\iota_1 \sqcup \iota_2 = \iota_1 \cup \{i + n + p | i \in \iota_2\}$.

Lemma 4.16 ($|$-$\otimes$ Distributivity)**.** Let $K_a \in \mathscr{S}_{n_1}^{m_1}$, $K_b \in \mathscr{S}_{n_2}^{m_2}$, $K_c \in \mathscr{S}_{p_1}^{q_1}$, and $K_d \in \mathscr{S}_{p_2}^{q_2}$. Furthermore, let $\iota_1 \in \mathscr{I}_{n_1,p_1}$, $\omega_1 \in \mathscr{I}_{m_1,q_1}$, $\iota_2 \in \mathscr{I}_{n_2,p_2}$, and $\omega_2 \in \mathscr{I}_{m_2,q_2}$ 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 4.17 ($\circ$-$|$ Distributivity)**.** Let $K_a \in \mathscr{S}^{m_1}_{n_1}$, $K_b \in \mathscr{S}^{m_2}_{n_2}$, $K_c \in \mathscr{S}^{p_1}_{m_1}$, and $K_d \in \mathscr{S}^{p_2}_{m_2}$. Furthermore, let $\iota \in \mathscr{I}_{n_1,n_2}$, $\mu \in \mathscr{I}_{m_1,m_2}$, and $\omega \in \mathscr{I}_{p_1,p_2}$ be interleaving functions. Then,

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

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 4.14. We begin by noting that the lemma can be trivially extended as follows:

Corollary 4.18 (Commutativity by Horizontal Separation)**.** The preceding two lemmas imply that for any $g \in \mathbb{N}$, $K_1 \in \mathscr{S}^m_n$, and $K_2 \in \mathscr{S}^q_p$ 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 4.19 (No-Overlap Layering)**.** Let $K_1 \in \mathscr{S}^m_n$ and $K_2 \in \mathscr{S}^q_p$. Then,

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

Proof. By Lemma 3.12, 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 4.20 (Layer Decomposition of Separate and Merge)**.** Let $\iota \in \mathscr{I}_{n,p}$ be an interleaving. Then,

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

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 4.21 (Sliding Door Lemma)**.** Let $K_1 \in \mathscr{S}_n^m$, $K_2 \in \mathscr{S}_p^q$ and let $\iota \in \mathscr{I}_{n,p}$, $\omega \in \mathscr{I}_{m,q}$ be interleavings. Then,

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

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

Proof.

$$\begin{aligned} K_1|_\iota^\omega K_2 &\cong (id_n \circ K_1 \circ id_m)|_\iota^\omega (id_p \circ K_2 \circ id_q) \text{ (by Lemma 3.13)} \\ &\cong (id_n|_\iota^{id} id_p) \circ (K_1|_{id}^{id} K_2) \circ (id_m|_{id}^\omega id_q) \text{ (by Lem 4.17)} \\ &\cong \overleftarrow{\nabla}_\iota \circ (K_1|_{id}^{id} K_2) \circ \overrightarrow{\Lambda}^\omega \text{ (by Lemma 4.20)} \\ &\cong \overleftarrow{\nabla}_\iota \circ (K_1 \otimes K_2) \circ \overrightarrow{\Lambda}^\omega \text{ (by Lemma 4.19)} \end{aligned}$$

□

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 4.22 (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 $R_A \subseteq \mathbb{Q}_\infty^2$, $R_B \subseteq \mathbb{Q}_\infty^2$. (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 4.23 (Extent)**.** We define the extent of a valid knitout program $\text{ex}(S \xrightarrow{ks} S') \subseteq \mathbb{Q}_\infty^2$ as a 2D interval (rectangle). Where S and S' can be inferred from context, we will notate the extent as $\text{ex}(ks)$. We will use $[z \pm \frac{1}{2}]$ as shorthand for $[z - \frac{1}{2}, z + \frac{1}{2}]$.

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

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

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

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

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

$\text{ex}(\texttt{split } dir\ n.x\ n'.x'\ l\ yarns)$

[2]Inspection of Lemma 3.13 makes it clear why a third dimension for vertical decomposition is unnecessary.

$$
\begin{aligned}
&= [\lfloor n.x \rceil_r \pm \frac{1}{2}] \times [-\infty, \infty] \\
\mathrm{ex}(\texttt{miss}\ dir\ n.x\ y) &= [\lfloor n.x \rceil_r \pm \frac{1}{2}] \times \{y\} \\
\mathrm{ex}(\texttt{in}\ +\ n.x\ y) &= \{\lfloor n.x \rceil_r + \frac{1}{2}\} \times \{y\} \\
\mathrm{ex}(\texttt{in}\ -\ n.x\ y) &= \{\lfloor n.x \rceil_r - \frac{1}{2}\} \times \{y\} \\
\mathrm{ex}(\texttt{out}\ +\ n.x\ y) &= \{\lfloor n.x \rceil_r + \frac{1}{2}\} \times \{y\} \\
\mathrm{ex}(\texttt{out}\ -\ n.x\ y) &= \{\lfloor n.x \rceil_r - \frac{1}{2}\} \times \{y\} \\
\mathrm{ex}(\texttt{drop}\ \texttt{f}.x\ ) &= \{\lfloor \texttt{f}.x \rceil_r\} \times \{-\infty\} \\
\mathrm{ex}(\texttt{drop}\ \texttt{b}.x\ ) &= \{\lfloor \texttt{b}.x \rceil_r\} \times \{\infty\} \\
\mathrm{ex}(\texttt{xfer}\ n.x\ n'.x'\ ) &= \{\lfloor n.x \rceil_r\} \times [-\infty, \infty] \\
\mathrm{ex}(\texttt{rack}\ r) &= [-\infty, \infty] \times \{\infty\}
\end{aligned}
$$

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 $\mathscr{E}[\![S]\!]$.

Definition 4.24 (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 $\mathscr{E}[\![S]\!]$ the denotation of that machine state (where the extent rectangle for specific programs is defined in Definition 4.23).

Points in $\mathscr{E}[\![S]\!]$ 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 $Y(y) \neq \perp$ 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 4.25 (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 $\mathscr{E}[\![S]\!]$. 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:

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

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 subprogram. 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}).

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

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. 4.5. 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 4.25, we can make the initial decompositions $\mathscr{E}[S_0 \xrightarrow{ks_1} S_1] = id_{n_1^{-x}} \otimes K_1 \otimes id_{n_1^{+x}}$ and $\mathscr{E}[S_1 \xrightarrow{ks_2} S_2] = id_{n_2^{-x}} \otimes K_2 \otimes id_{n_2^{+x}}$ where $K_1 \in \mathscr{S}_{p_1}^{q_1}$ and $K_2 \in \mathscr{S}_{p_2}^{q_2}$. 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 4.18.

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

In the case of depth-wise decomposition, one uses the Sliding Door Lemma (4.21) 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 4.11). 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 4.26 (Move Macro)**.** The `MOVE` macro is the `SHIFT` macro with an additional `RACK` to reset the machine's racking to r:

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

Definition 4.27 (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.

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

where y_{max} is ∞ if $L(\texttt{b.}x-r)>0$, otherwise it is $\max\{y \mid y\in Y^{-1}(x') \text{ and } x<x'\le x+j\}$ 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 $\mathscr{E}[S \xrightarrow{\texttt{MOVE}} S']$ 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 4.28 (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

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

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

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

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

Proof. Consider the case of $ks_{\mathtt{b}}$ first. Since $L(\mathtt{f}.x+r)=0$, 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 $\lfloor \mathtt{b}.x \rfloor_r$ and ending at $\lfloor \mathtt{b}.x+j \rfloor_r$ are perfectly vertical. Therefore, horizontally we can separate out an identity slab of everything to the left of $\lfloor \mathtt{b}.x \rfloor_r$ and an identity of everything to the right of $\lfloor \mathtt{b}.x+j \rfloor_r$.

Now consider the case of $ks_{\mathtt{f}}$, which would ideally be symmetric with $ks_{\mathtt{b}}$. 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 $ks_{\mathtt{f}}$. Therefore, we can layer decompose everything strictly after y_{max} from everything at or before y_{max}. □

Lemma 4.29 (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 $ks_1 = \mathtt{MOVE}(\mathtt{f}.x, r, \mathtt{f}.x+j)$; $L(\mathtt{b}.(x-r))=0$ 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 $ks_1 = \mathtt{MOVE}(\mathtt{b}.x, r, \mathtt{b}.x+j)$; $L(\mathtt{f}.(x+r))=0$ 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 4.28. 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. □

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.

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

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.

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

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

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

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

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

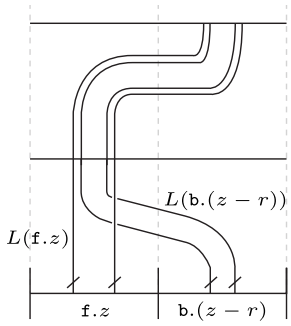
□

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 `xfer f.z 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. □

Table 4.2: 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 $ks(dir, \texttt{b.n})$, conjugate only uses the `SHIFT` macro. In contrast, the front bed case $ks(dir, \texttt{f.n})$ 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`$(\texttt{f.}x, r, -1)$	
	`SHIFT`$(\texttt{f.}x, r, -1)$	`miss` $-$ `f.`x *yarns*	`SHIFT`$(\texttt{b.}x, r, -1)$
	$ks(+, \texttt{f.}x-1)$	$ks(-, \texttt{f.}x-1)$	$ks(dir, \texttt{b.}x-1)$
	`miss` $+$ `f.`x *yarns*	`SHIFT`$(\texttt{f.}x-1, r-1, 1)$	`SHIFT`$(\texttt{b.}x-1, r-1, 1)$
	`SHIFT`$(\texttt{f.}x-1, r-1, 1)$	`miss` $+$ `f.`$x-1$ *yarns*	
Right	`SHIFT`$(\texttt{f.}x, r, 1)$	`miss` $+$ `f.`$x+1$ *yarns*	
	`miss` $+$ `f.`x *yarns*	`SHIFT`$(\texttt{f.}x, r, +1)$	`SHIFT`$(\texttt{b.}x, r, 1)$
	$ks(+, \texttt{f.}x+1)$	$ks(-, \texttt{f.}x+1)$	$ks(dir, \texttt{b.}x+1)$
	`SHIFT`$(\texttt{f.}x+1, r+1, -1)$	`miss` $-$ `f.`x *yarns*	`SHIFT`$(\texttt{b.}x+1, r+1, -1)$
	`miss` $-$ `f.`$x+1$ *yarns*	`SHIFT`$(\texttt{f.}x+1, r+1, -1)$	

Subprogram Machine Location

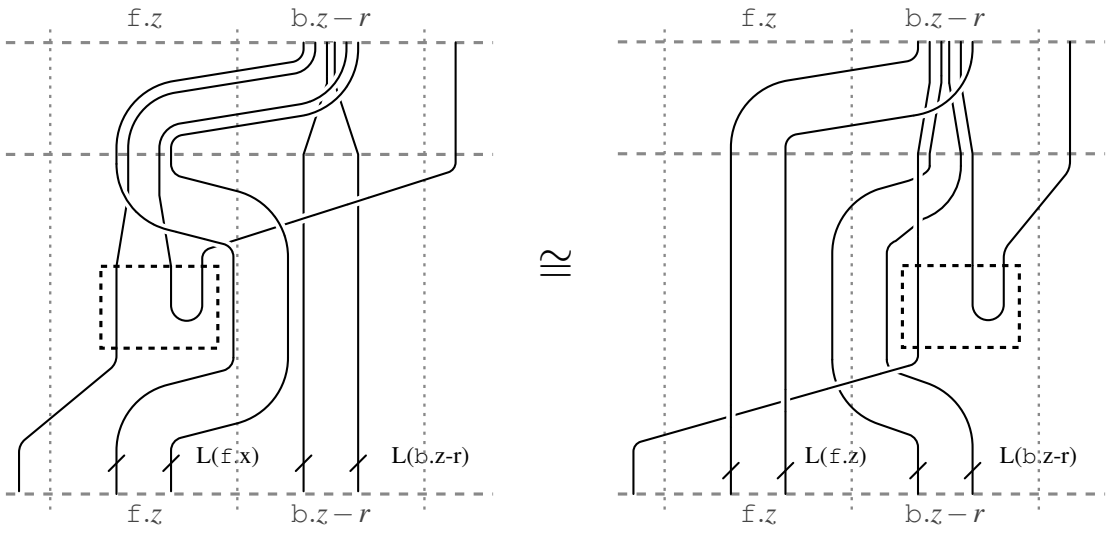
We have now explained how to cancel and commute various knitout instructions relative to each other. This allows us to change the order in which we perform operations and remove redundant operations. However, it doesn't yet allow us to "reschedule" programs in the sense of adjusting gauge or changing 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 $ks(n.x) =$ `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.



□

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 4.2. 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 $L(\texttt{b}.x - r) = 0$, $L(\texttt{f}.x - 1) = 0$, and if there are no yarn carriers in the way that we are not using $Y^{-1}(\lfloor \texttt{f}.x, -\rfloor_r) = yarns$, then

$$
\begin{aligned}
S \vdash ks(+, \texttt{f}.x) \cong\ & \texttt{miss - f.}x-1\ yarns;\ \texttt{SHIFT}(\texttt{f}.x, r, -1); \\
& ks(+, \texttt{f}.x-1); \\
& \texttt{miss + f.}x\ yarns;\ \texttt{SHIFT(f.x-1, r-1, 1)}
\end{aligned}
$$

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

Proof. The proof is given in figure 4.8. We briefly expound on details here.

Because only and exactly *yarns* are present at $\lfloor \texttt{f}.x, -\rfloor_r$, the unconjugated diagram has no initial $\overrightarrow{\mathrm{V}}_{yarns}$; 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 $\lfloor \texttt{f}.x-1, -\rfloor_r$. The rest of the diagram fairly trivially deforms back to the unconjugated diagram, using standard Reidemeister moves. □

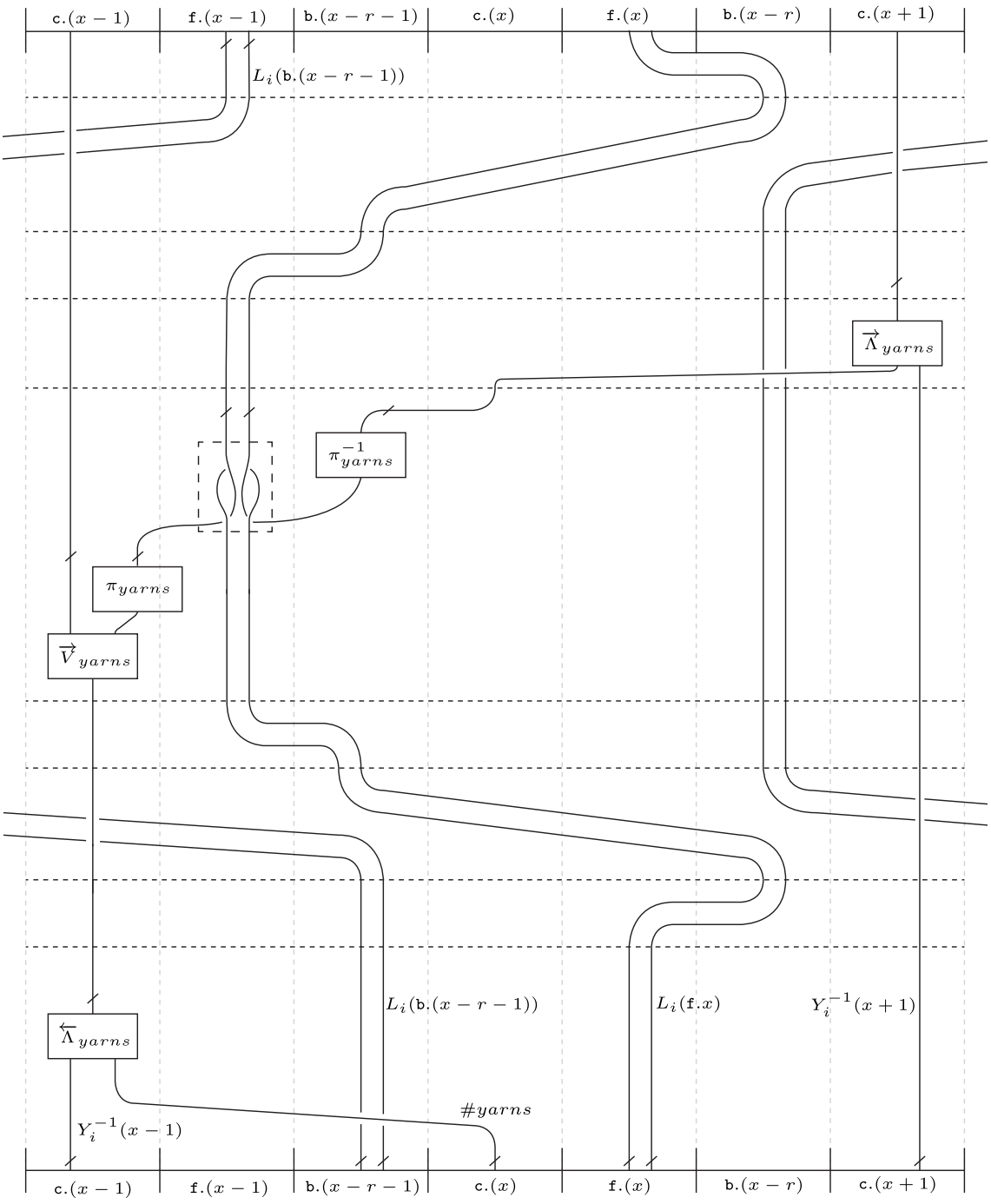
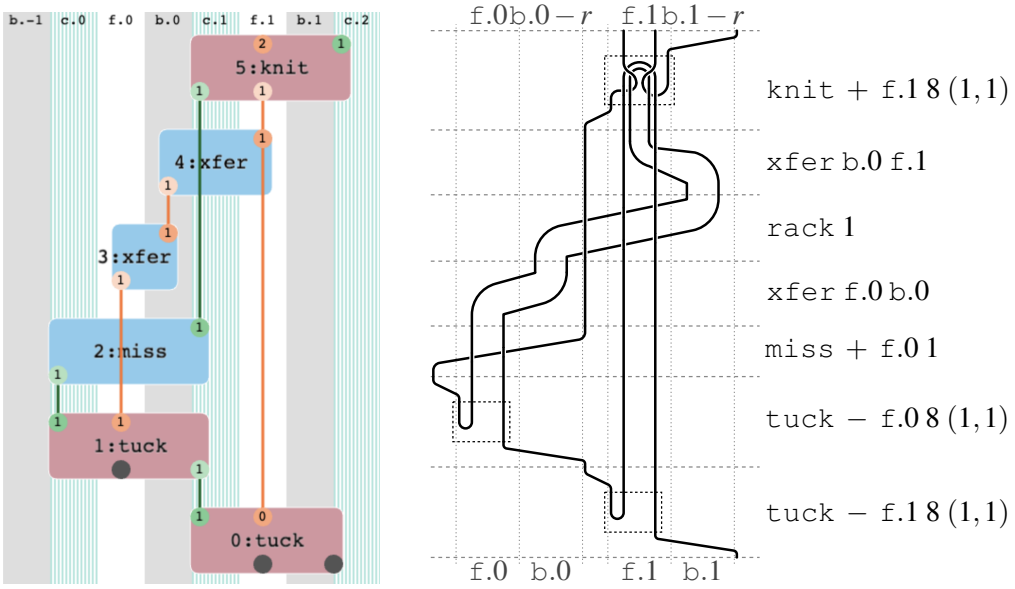


Figure 4.8: The fenced tangle denoted by the conjugate left program. Note it is equivalent to the fenced tangle for `knit` + `f`.*x l yarns* as seen in figure 4.5i.



(a) A screenshot of the 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.

Figure 4.9: Knitout code shown in the rewrite-editor and the corresponding fenced tangle.

4.4 Results

To demonstrate the usefulness of these rewrite rules, my co-authors and I 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. 4.9a, and the corresponding fenced tangles and the formal knitout code is shown in Fig. 4.9b.

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. 4.7). The rewrite is applied only if it is correct given the program context.

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

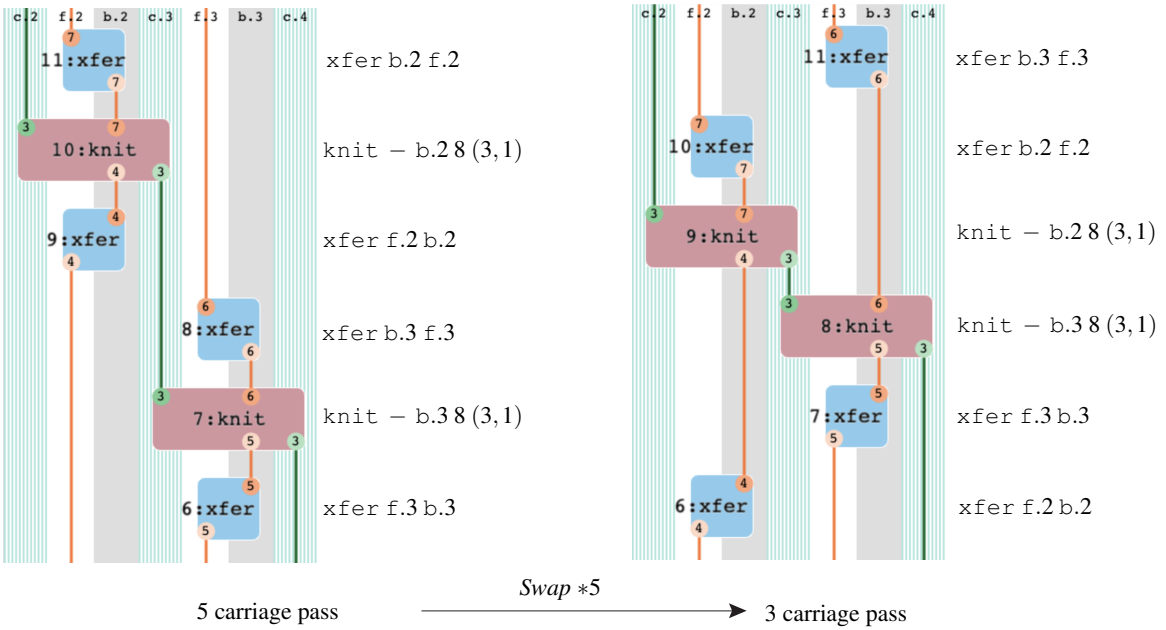


Figure 4.10: Rewrite-editor screenshot and the corresponding knitout code of the pass optimization example. See Definition 4.2 for the formal knitout syntax. Left is typical knitout that novices tend to write, which is correct but inefficient due to unnecessary carriage passes. After applying the *Swap* rewrite rule five times, we can consolidate `knit` and `xfer` instructions so that the number of carriage passes becomes three. The impact of pass consolidation increases as the size of the program gets larger.

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).

4.4.1 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. 4.10 (left). Such per-stich interleavings are inefficient because `knit` and `xfer` operations require separate carriage passes. Re-ordering the code to group `knit`s and `xfer`s into separate blocks results in fewer passes and a shorter knitting time.

We optimized the carriage passes by applying a sequence of *Swap* operations on the original knitout code. Fig. 4.10 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 `xfer`s and `knit`s 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 $3600/240 = 15$.

4.4.2 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.

In the following example, we use a rewrite sequence pattern for moving the instructions to a neighboring needle, which is illustrated in Fig. 4.12. We first apply the rule *Conjugate Right* to two `knit` instructions `3:knit` and `0:knit`. *Conjugate Right* will insert `miss`es and `xfer`s as described in section Section 4.3. Then, we *Swap* the `xfer`s until they are next to each other and apply *Squish* to cancel redundant `xfer`s.

We scheduled a full gauge sheet (Fig. 4.11a) to a half gauge sheet (Fig. 4.11b), and a full gauge tube (Fig. 4.11c) to a half gauge tube (Fig. 4.11d) 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.

4.4.3 Sheet Stacking

Recall the example discussed in Chapter 2, 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. 2.7a) so that they were correctly stacked (Fig. 2.7b) 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. 4.12. Then, we used the *Swap* rule to swap `knit` instructions until the sheets were correctly interleaved.

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

(a) Full-gauge sheet

(b) Sheet transformed to half-gauge

(c) Full-gauge tube

(d) Tube transformed to half-gauge

Figure 4.11: Examples of sheets and tubes converted from full gauge to half gauge using rewrite rules to guarantee topological equivalence.

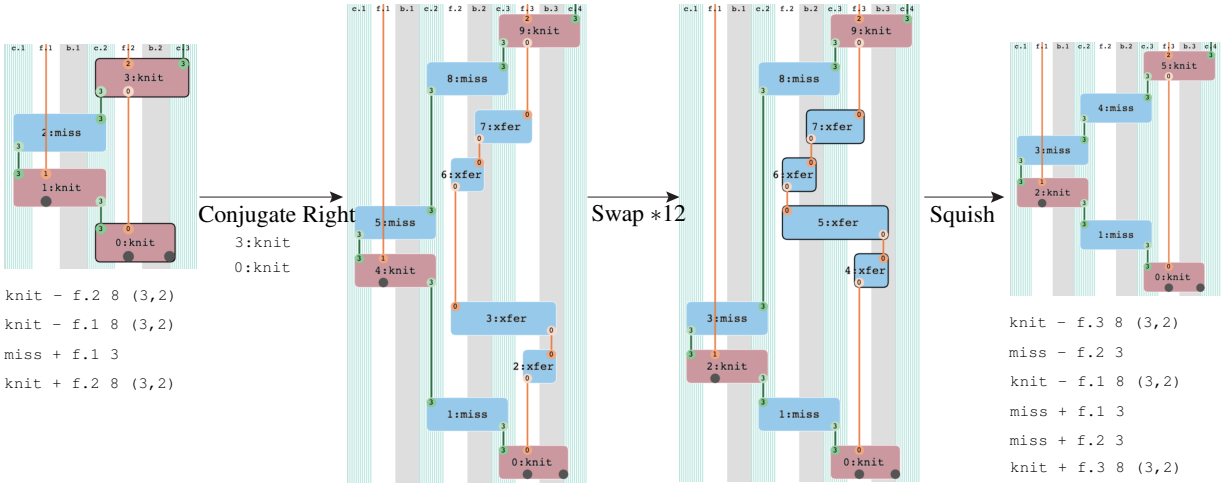


Figure 4.12: Rewrite-editor screenshot and the corresponding knitout of the rewrite sequence for moving the `knit`/`tuck` instruction one to the left or right. The sequence of rewrites in this example moves two `knits` (`3:knit` and `0:knit`) from needle `f.2 to f.3`.

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 2.7c.

4.4.4 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.

I, an experienced knitting machine programmer, wrote a knitout program 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 $1/4$th gauge (each layer uses one out of every four needle indices). However, knitting at $1/4$th 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. 4.13a). 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 $1/4$th gauge to $2/3$rd 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.4.12). 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. 4.13b).

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

Figure 4.13: Photos of the fabricated pleated tube examples

Chapter 5

Compilation of Unscheduled Knitting Representations

Last chapter, I presented a denotational semantics for knitout. This is essentially a mapping from low-level machine control operations to a knit object (specifically fenced tangles). The knit compilers described in Chapter 2 are interested in the inverse direction: mapping from knit objects to low-level control operations. Robust knit compilers enable users to design and essentially "write" knitting programs solely by working with intuitve, unscheduled object reprsentations, which would greatly simplify the knit programming process.

That said, we cannot simply invert our denotation function and directly use fenced tangles to represent knit objects. For one, a fenced tangle is a mathematical object, not a data structure we can manipulate on a computer. To be precise, while the knitout semantic function defined last chapter maps from a program to a specific presentation of a fenced tangle, the actual semantic domain is the equivalence class of fenced tangles under ambient isotopy. Rather than try to express an abstract mathematical concept on a finite, physical computer, we instead need to define a data structure, or computer *representation* that also denotes a fenced tangle. From there, we can reason about the correctness of compilation functions between the knit object representation (i.e. *unscheduled representation*) and knitout programs by appealing to the equivalence of their respective denoted fenced tangles.

This leads to the question of what that representation should be. A natural choice might be a data structure that describes a presentation of a fenced tangle, but this runs into the issue that most fenced tangles are not machine knittable; for a simple example, consider any fenced tangle that includes an embedded circle. Ideally, we want a knit representation that is both complete (represents the result of any valid sequence of machine operations) and sufficient (only represents objects that can be made with a sequence of machine operations). This is precisely the goal of this chapter.

In the following sections, I will introduce *instruction graphs*, which are an unscheduled representation of machine knitting. Unlike earlier graph-based representations of knitting, which are only concerned with graph connectivity, instruction graphs treat the embedding of the graph in space as important semantic information. This results in a knit representation that can precisely describe the topology of the object as fenced tangles. I show that instruction graphs with certain easy-to-check properties (upward, forward, and ordered) are *exactly* the ones that can be machine knit. This allows us to define a compilation function from instruction graphs to knitout that is sound and complete: it will correctly lower an any instruction graph that can be knit to an equivalent knitout program. An overview of the chapter structure is shown in Fig. 5.1.

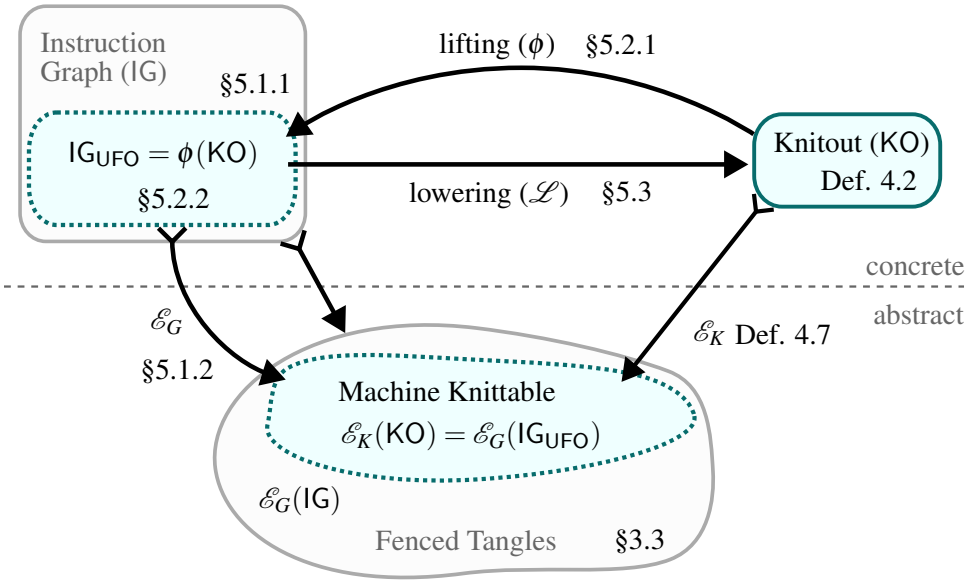


Figure 5.1: The chapter overview. We first present instruction graphs, which are a high-level knitting representation that also denote fenced tangles (Section 5.1). By defining the function ϕ that lifts all valid knitout programs to instruction graphs while preserving fenced tangle equivalence (Section 5.2.1), we observe that all graphs in the co-domain of ϕ have three graph properties: upward, forward, and ordered (Section 5.2.2). We then define function $\mathscr{L}$ that lowers all UFO instruction graphs to an equivalent formal knitout program (Section 5.3). Because we made sure the lifting/lowering functions commute with the denotation functions, we prove that there is an equivalence-preserving bijection between UFO instruction graphs and valid knitout programs, i.e. all UFO instruction graphs are machine knittable.

5.1 Instruction Graphs

Before formally defining instruction graphs, let's discuss their design goals. As the primary objective is to express all machine-knittable structures, instruction graphs should, at a minimum, be capable of expressing every fenced tangle denoted by a valid formal knitout program ($\mathscr{E}_K[kP]$). If we examine the semantic definition of each primitive knitout operation, we notice: first, there is a countable family of tangles that can appear inside fences; and, second, there are two types of arc bundles between fences: those that are parameterized by loop count L, and those that are parameterized by yarn carrier position Y. External to fences, the type of an arc bundle never changes. This allows us to partition machine knittable fenced tangles into three key structures:

1. Fences, which contain some templated tangle that depends on the associated operation. All arcs that exit the top of a fence either enter the bottom of a different fence or exit the top of the bounding slab. All arcs that enter the bottom of a fence either exited the top of a different fence or the bottom of the bounding slab.
2. Loop stacks, which are an even number of arcs that move as a parallel bundle. All arcs in a loop stack connect the same pair of fences. Except for when two loop stacks are merged by ending up on the same needle, anything that crosses one arc in a loop stack crosses all arcs in a loop stack.
3. Yarns, which move independently but have restrictions on their crossing order with other yarns.

We define the instruction graph (G) in Section 5.1.1, specify its semantics as a denotation to fenced tangles ($\mathscr{E}_G$), and define an equivalence relation ($\cong_G$) in Section 5.1.2.

5.1.1 Instruction Graph Definitions

A knit instruction graph has three features: a set of nodes, along with arcs and ribbons that connect them. The nodes are affine transforms of a countable set of *exemplars*(Definition 5.2), which correspond to specific machine operations; the arcs are paths in space corresponding to yarns; and the ribbons are framed paths in space corresponding to loop stacks. Annotations on arcs and ribbons are used to track carrier id and loop stack counts respectively. Intersection is not permitted between these primitives except at specific connection points on node boundaries.

Definition 5.1 (Instruction Graph)**.** An instruction graph is a tuple $G = (N, A, R)$ where:

Nodes each node $n \in N$ consists of a reflection-free, non-degenerate affine transformation of some particular exemplar.

Arcs each arc $a \in A$ consists of a piecewise-linear path embedded in space $a : [0,1] \to \mathbb{R}^3$, along with a yarn-carrier annotation $\mathrm{id}(a) \in \mathbb{N}$

Ribbons each ribbon $r \in R$ a framed piecewise-linear path embedded in space $r = (p, \hat{n})$, with underlying path $p : [0,1] \to \mathbb{R}^3$, and unit vector $\hat{n} : [0,1] \to \mathbb{R}^3$.

Furthermore, an instruction graph must satisfy

- for each arc, one of its endpoints is coincident with some exemplar's arc input port, and the other with some arc output port. The exemplar and arc must agree on yarn carrier id.
- for each ribbon, two opposite sides (beginning and ending) intersect nodes at exactly some exemplar's output ribbon port and exactly some input ribbon port (respectively). The ribbon and ports must agree on loop count, and the dot product of the ribbon's binormal and the exemplar's local z-axis must be zero.
- for each node, all ports should be coincident to exactly one arc or ribbon (no more, no less).
- other than these ports, no arcs, ribbons, or nodes intersect in space.

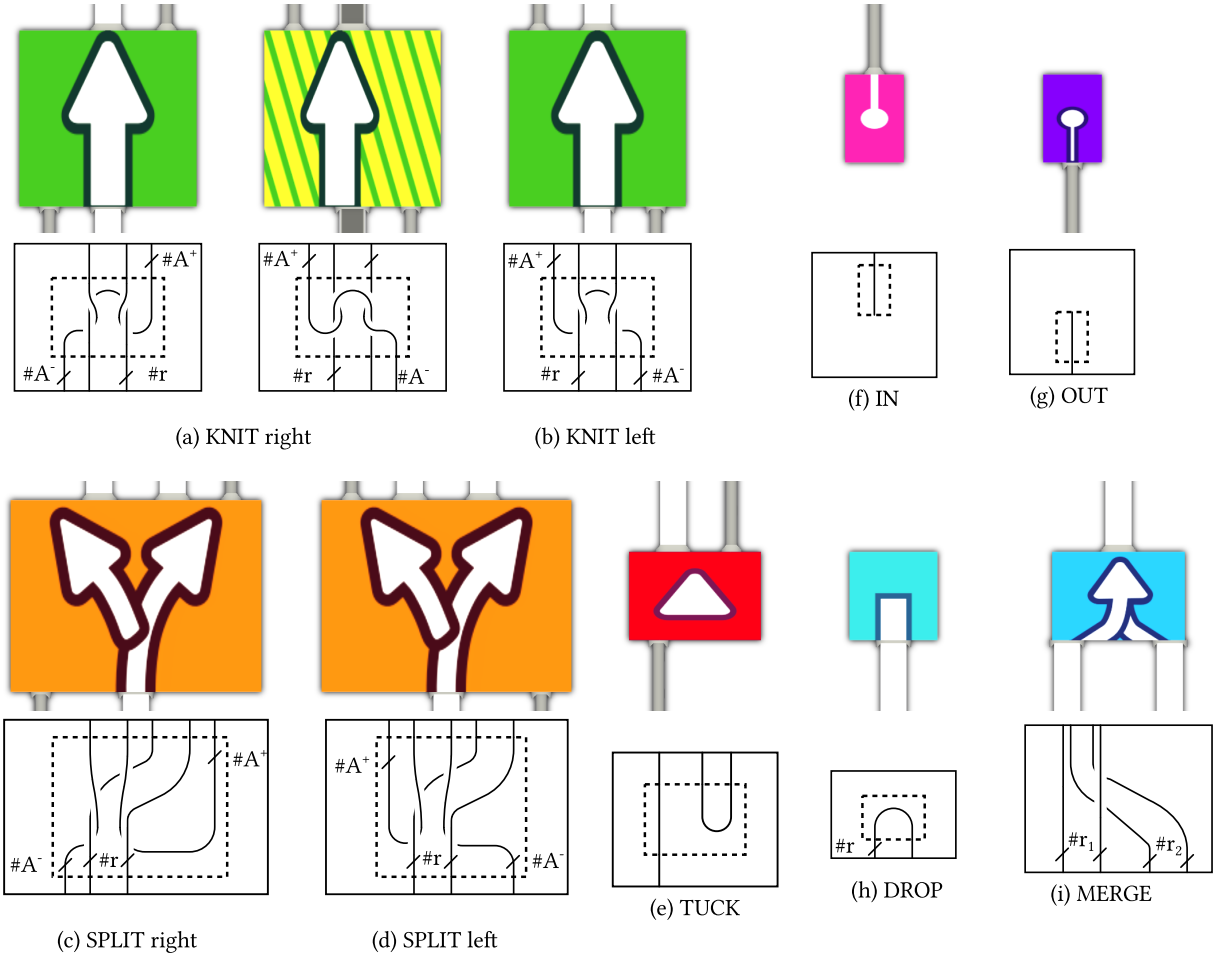


Figure 5.2: Knit instruction graph nodes draw their contents from a countable set of exemplars, where each exemplar denotes a fenced tangle. As seen in KNIT-right (a), rotating the exemplar 180° around the y axis rotates the fenced tangle as well to produce the mirror image with all crossing annotations flipped. This is a different fenced tangle than KNIT-left (b). Similarly, we need a left and right variation for SPLIT (c,d), while all remaining exemplars (e-i) are equivalent to their mirror image.

An exemplar is a carefully-chosen snippet of yarn topology that represents regions of knitting that cannot be denoted using arcs and ribbons: fenced regions places where loop stacks are combined. Each exemplar consists of an axis-aligned rectangular domain with a set of input ports on a line along its lower face and a set of output ports on a line along its upper face.

Definition 5.2 (Exemplar)**.** An instruction graph exemplar is a fenced tangle defined on $[-1,1]^3$ with an equator at $z=0$. Exemplars must be drawn from a finite set of classes (defined in Figure 5.2) and have all endpoints on the equator grouped and annotated as follows:

Arc input ports Each $a_i^- \in A^-$ is an end-point lying on the lower equator line segment $[-1,1]\times\{-1\}\times\{0\}$; $\mathrm{id}(a_i^-)\in\mathbb{N}$ is the yarn carrier id of this port.

Arc output ports Each $a_i^+ \in A^+$ is an end-point lying on the upper equator line segment $[-1,1]\times\{1\}\times\{0\}$; $\mathrm{id}(a_i^+)\in\mathbb{N}$ is the yarn carrier id of this port. The number of end-points in A^- is the same as the number of end-points in A^+, and for all i, $id(a_i^-) = id(a_i^+)$.

Ribbon input ports $r_i^- \in R^-$ is an end-point lying on the lower equator. $\#r_i^- = n$ is the "number of loops" associated with this port.

Ribbon output ports $r_i^+ \in R^+$ is an end-point lying on the upper equator. $\#r_i^+ = n$ is the number of loops, similarly.

Much like with fenced tangles, we will find it useful to construct more complicated Instruction Graphs out of simpler ones. To this end, we define Partial Instruction Graphs and method for composing them.

Definition 5.3 (Partial Instruction Graph)**.** A partial instruction graph is an instruction graph defined within $\mathbb{R}\times[0,1]\times[0,1]$, in which the endpoints of arcs and ribbons are allowed to to lie on either the input boundary $\mathbb{R}\times\{0\}\times\{0\}$ or the output boundary $\mathbb{R}\times\{1\}\times\{0\}$. Boundaries also have ports, an interleaved sequence of n arc points and m ribbon points in $\mathbb{R}$, where arcs are annotated with a yarn carrier id y_c, and ribbons are annotated with loop count l. Arcs and ribbons attached to the boundary must agree on carrier id and loop count annotations respectively. A partial instruction graph with no inputs nor outputs is simply an instruction graph.

Definition 5.4 (Vertical Composition of Partial Instruction Graphs)**.** We define the vertical composition $G_0 \circ G_1$ of partial instruction graphs $G_0 = (N_0, A_0, R_0)$ and $G_1 = (N_1, A_1, R_1)$ as $G = (N, A, R)$ where: $N = N_0 \cup N_1$; A is the result of joining A_0 and A_1 at any common endpoints; and R the result of joining R_0 and R_1 at any common endpoints. For this to make sense, the output arc and ribbon endpoints of G_0 must be in one-to-one correspondence with input arc and ribbon endpoints of G_1; the correspondence must match the ordering along input/output lines. Furthermore, the vertical composition of partial instruction graphs is only well-defined when all matching arcs agree on id, and all matching ribbons agree on loop count annotation and frame at the joined endpoints.

5.1.2 Instruction Graph Semantics

While formal knitout represents fenced tangles coupled with an explicitly-specified set of machine operations and relies on machine semantics to provide spatial information, instruction graphs directly encode spatial information without including machine-specific details (e.g., which needle will be used to create each structure). Consequently, instruction graphs can represent a larger space of fenced tangles than those that can be machine-knit. Therefore, defining the semantics of instruction graphs is crucial. Similar to how the semantics of formal knitout were defined as a denotation on fenced tangles ($\mathscr{E}_K$), we define the semantics of an instruction graph as a denotation of fenced tangles ($\mathscr{E}_G$).

Definition 5.5 (Instruction Graph Denotation)**.** Let (N, A, R) be a partial instruction graph. We define the number of inputs n and number of outputs m as follows. Let p be a ribbon endpoint (for ribbon r) on the input or output line. Then, let $\#p = 2\cdot\#r$ (twice the loop count). Then n is the number of input arc

endpoints plus $\sum_{p \in \text{input}} \#p$, and m is the number of output arc endpoints plus $\sum_{p \in \text{output}} \cdot$.

Then, the denotation of our partial instruction graph is a slab, $\mathscr{E}_G[(N,A,R)] \in \mathscr{S}_n^m$ defined as follows. Let every arc $a \in A$ be sent to an arc $a \in A'$; let every ribbon $r \in R$ be sent to a set of $k = 2 \cdot \#r$ arcs $r_i \in R'$ $(0 \leq i < k)$, where the binormal of the ribbon is used to offset the ribbon's path p by $i * \varepsilon$, where ε is small enough to prevent any additional intersections. Lastly, let every node n be sent to arcs $a \in A'$ and fences $L_i \in \mathbf{L}'$ specified by n's exemplar, and transformed by the affine transformation of n. Wherever there are ports on the boundary of a node/exemplar, the corresponding arcs/ribbons external to the node and arcs internal to the node the endpoints will be coincident. Finally, we define a fenced tangle by joining together all arcs with coincident endpoints into contiguous arcs. Along with the labels $\mathbf{L}'$, this defines a fenced tangle on an (n,m)-slab.

Observe that the denotation forgets all length annotations, similarly to the denotation of knitout.

Definition 5.6 (Instruction Graph Equivalence)**.** Two instruction graphs are equivalent if and only if their denoted fenced tangles are equivalent.

Given this definition, it also trivially follows that any transformation on the instruction graph under ambient isotopy of $\mathbb{R}^3$ is also equivalent. This is because any transformation under ambient isotopy of an instruction graph also transforms the denoted fenced tangle under ambient isotopy. Since ambient isotopy preserves the structure and relationships between components in 3-dimensional space, it also preserves the equivalence of fenced tangles. Therefore, applying an ambient isotopy to an instruction graph results in an equivalent fenced tangle. A GUI implementation of such equivalence-preserving transformation is described in Section 5.4.1.

Corollary 5.7 (The Denotation of Vertical Composition is Vertical Composition)**.** Let $G = G_1 \circ G_2$ be a partial instruction graph vertically composed out of G_1 and G_2. Then, $\mathscr{E}_G[G_1 \circ G_2] \cong \mathscr{E}_G[G_1] \circ \mathscr{E}_G[G_2]$.

5.2 Machine Knitablity Implies UFO Instruction Graph

Now that we've defined instruction graphs, we will proceed to demonstrate that all valid formal knitout programs denote a fenced tangle that is equivalent to one denoted by an instruction graph with a few easy-to-check properties (upward, forward, and ordered). We will proceed by developing a function to "lift" valid formal knitout programs to instruction graphs and then examine the properties of these lifted instruction graphs.

5.2.1 Lifting Knitout to Instruction Graphs

In order to demonstrate that instruction graphs can represent all machine-knittable structures, we define a lifting, $\phi : \mathsf{KO} \to \mathsf{IG}$, from valid formal knitout programs to instruction graphs that preserves the meaning (denoted tangles) of the programs.

Our definition for ϕ closely follows the structure of the definition of the formal knitout denotation function $\mathscr{E}_K$ (Definitions 4.6, 4.7, and Figure 4.5):

Definition 5.8 (Lifting Function)**.** Let $\phi : \mathsf{KO} \to \mathsf{IG}$ be lifting function that translates valid formal knitout programs into instruction graphs, as follows:

1. The lifting of a program is the vertical concatenation of the lifting of each step of the program trace:

$$\phi\left(kP\right) \equiv \phi\left(S_0 \xrightarrow{ks_1} S_1\right) \circ \cdots \circ \phi\left(S_{n-1} \xrightarrow{ks_n} S_n\right) \tag{5.9}$$

   where the ks_* are individual formal knitout statements (i.e., $kP = ks_1; \ldots; ks_n$), and the S_* are machine states along the execution trace of kP.

2. The lifting of a given step is an instruction graph slab that implements the instruction and passes through uninvolved loops/yarns on either side:

$$\phi(S \xrightarrow{ks} S') \equiv \text{See Figure 5.3} \tag{5.10}$$

With ϕ defined, it is straightforward to state our main result from this section:

Theorem 1 (Everything Machine Knittable is an Instruction Graph)**.** For any valid formal knitout program $kP \in \mathsf{KO}$, there exists an instruction graph $G \in \mathsf{IG}$ such that $\mathscr{E}_G[G] \cong \mathscr{E}_K[kP]$.

Proof. We show that $G = \phi(kP)$ satisfies the hypothesis. Notice that both $\phi(kP)$ and $\mathscr{E}_K[kP]$ are built from vertical slabs, one per instruction. So (by Corollary 5.7) it suffices to inspect the denotation of each slab in $\phi(kP)$ and confirm that it is equivalent to the slab in $\mathscr{E}_K[kP]$ – i.e., to check that

$$\mathscr{E}_G[\phi(S_{i-1} \xrightarrow{ks_i} S_i)] \cong \mathscr{E}_K[S_{i-1} \xrightarrow{ks_i} S_i]$$

for every step ks_* in the execution of kP.

This can be verified by visual inspection of each case in Figure 5.3.

□

5.2.2 Upward, Forward, Ordered

Having defined a semantics-preserving lifting ϕ from valid formal knitout programs to instruction graphs, we now have a clearer picture of exactly the instruction graphs that correspond to valid machine knitting programs. We call this subset $\mathsf{IG}_{UFO} \subset \mathsf{IG}$ (the subscript's meaning will become obvious anon):

$$\mathsf{IG}_{UFO} \equiv \{\phi(kP) \mid kP \in \mathsf{KO}\} \tag{5.11}$$

Observe that any instruction graph that denotes a knittable fenced tangle must be equivalent to one in this set. Therefore, it is interesting to see if we can find any properties that characterize IG_{UFO}. We define three such properties below which (coincidentally) map to fabrication constraints of knitting machines. For now, it is easy to verify by inspection of the definition of ϕ that every instruction graph $G \in \mathsf{IG}_{UFO}$ has these properties; in the next section we will show that *any* instruction graph with all three of these properties is equivalent to one in IG_{UFO}.

Definition 5.12 (Upward)**.** All nodes have their local y axis aligned with the global y axis, and all arcs and ribbons travel strictly upward (are strictly increasing in y coordinate).

Knitting is a sequence of instructions where later instructions depend on the results of earlier instructions. They cannot depend on operations that have not yet occurred. The knitting machine does not have the ability to rotate anything, which means the result of each operation has a fixed orientation that cannot be changed.

Definition 5.13 (Forward)**.** All node transforms have their local z axis facing along the global $+z$ or $-z$ axis. The normal vector of all ribbons is always aligned with the global $+z$ or $-z$ axis.

Knitting machines have only front-bed and back-bed needles, not needles at other orientations, so they can only perform operations in these two orientations. Again, because the knitting machine cannot rotate anything, there is no formal knitout operation that can insert twist into loops. Thus all arcs in a loop run in parallel and never cross each other.

Definition 5.14 (Ordered)**.** All arcs that share the same y coordinate must have distinct carrier IDs. If two arcs also share the same x coordinate, the arc with the smaller carrier ID must also have the smaller z coordinate (carrier crossing order is consistent with carrier ID).

Figure 5.3: Our semantics-preserving lifting function, ϕ, is defined by the per-statement liftings illustrated here. Each case is illustrated in specific but should be considered as a general template as per Lin et al.'s construction. (Front/back and left/right variations are not shown, but are constructable following a similar pattern). The grey boxes labeled id_l and id_r represent identity instruction graphs that connect uninvolved loops and yarns from the bottom to the top boundaries. The grey L-shapes in the labeled tangles follow Lin et al.'s figure in including both identity tangles and yarn routing. Our illustrated instruction graphs include yarn routing explicitly; the definition for `split` includes annotations showing which part of the figure implements each routing action.

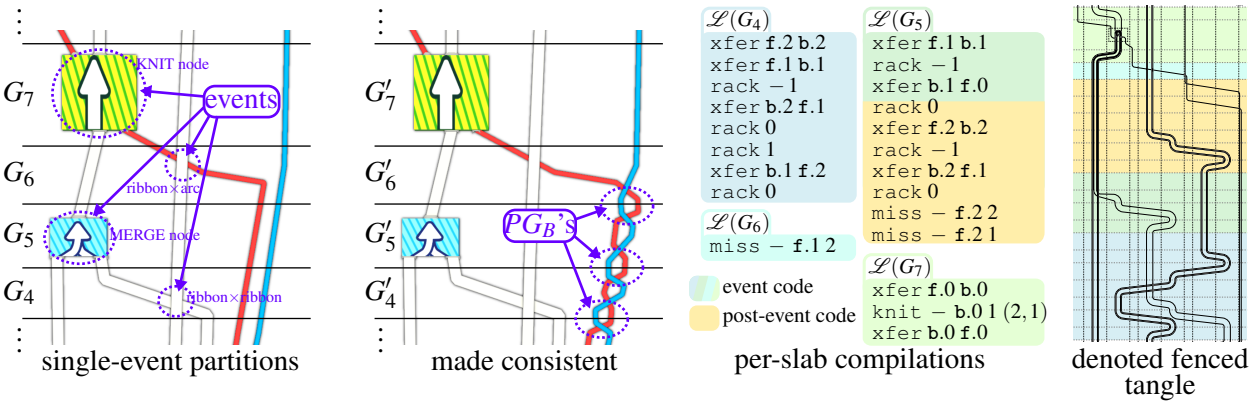


Figure 5.4: Running our compilation function ($\mathscr{L}$) to convert an UFO instruction graph $G \in \mathsf{IG}_{UFO}$ into knitout code. The graph is partitioned between events; then partition boundaries are locally adjusted for consistent yarn-carrier ordering; finally, each partition is then transformed into a formal knitout program. The tangle denoted by the knitout code matches the tangle denoted by the instruction graph. (Note that the leftmost ribbon crossing the lower boundary of the illustrated portion of the instruction graph has a loop count of two because of earlier operations.)

The ordered property captures two restrictions: first that a given carrier cannot be used by operations that occur at two separate physical locations. Second, the constraint that the carrier with the smaller id crosses in front of the carrier with the larger id corresponds to how carriers sit on parallel rails, thus restricting the carrier to movement along only one axis. This fact is also visible in the fenced tangles denoted by knitout operations: all crossings between arcs denoted by carriers are defined using separation and permutation slabs, which in turn are uniquely determined by permutations on carrier ids.

5.3 UFO Instruction Graphs are Machine Knittable

The previous section showed that the UFO properties will always be satisfied by some presentation of an instruction graph that represents a machine-knittable structure. In this section, I demonstrate that the UFO properties are also *sufficient* to guarantee machine knittability. Similarly to the proof in Section 5.2.1, I do so by defining a total mapping $\mathscr{L} : \mathsf{IG}_{\mathsf{UFO}} \to \mathsf{KO}$ that preserves fenced tangle equivalence.

Theorem 2 (Every UFO Instruction Graph is Machine Knittable)**.** For all UFO instruction graphs $G \in \mathsf{IG}_{\mathsf{UFO}}$, there exists a valid knitout program $kP \in \mathsf{KO}$ where $\mathscr{E}_G[G] \cong \mathscr{E}_K[kP]$

At a high level, $\mathscr{L} : \mathsf{IG}_{\mathsf{UFO}} \to \mathsf{KO}$ is a divide-and-conquer algorithm that flattens input graph G along the z axis and segments it into partial graphs until a partial graph has only a single *event*: either a node, or a ribbon crossing with an arc or a ribbon. The x-order of ribbons and arcs on the boundary of the partial graph will define machine states between events; while each event will map to a carefully-defined subprogram that transitions between machine states and includes any necessary fenced regions. These subprograms are then concatenated to a full program. Each subprogram consists of a core, which captures the fenced tangle denoted by the event, and potentially a pre-program that allocates space for the core operation or a post-program that fills in empty space to ensure the program is compatible with the machine states before and after the event. An example of $\mathscr{L}$ applied to a partial instruction graph can be seen in Fig. 5.4.

Definition 5.15 (Compilation of Instruction Graphs)**.** Given UFO instruction graph G, lowering function $\mathscr{L}$ decomposes the graph into partial instruction graphs $G \cong G_1 \circ G_2 \circ \ldots \circ G_n$, where each partial instruc-

tion graph contains a single event. It returns $\mathscr{L}(G_1) \circ \mathscr{L}(G_2) \circ \ldots \circ \mathscr{L}(G_n)$, where $\mathscr{L}(G_i) = S \xrightarrow{ks_i^-;ks_i^*;ks_i^+} S'$ Knitout program ks_i is then a case decomposition depending on the event in partial instruction graph.

5.3.1 Knitting Machine State

We begin by defining the mapping from instruction graph boundaries to knitting machine states.

Definition 5.16 (Lowering IG boundaries)**.** Let B be an instruction graph boundary with n arc ports and m ribbon ports, where each point is annotated with a yarn carrier id y_c, and each interval is annotated with loop count l. $\mathscr{M}(B)$ is defined as follows.

Beginning with $S = S_\emptyset$ we use $i = 0$ to track the next free needle, and iterate along the sequence of ports. If the port is a ribbon, then $L = L[\mathtt{f}.i \mapsto l]$, and i is incremented (i.e., there are l loops on front-needle i). If the port is an arc, then $Y = Y[y_c \mapsto \lfloor \mathtt{f}.i, - \rfloor_0]$ (i.e., the yarn carrier is parked to the left of the next free needle). In short, all loop stacks and carriers are scheduled into a single contiguous block, where all loop stacks are placed on the front bed, leaving the back bed empty.

This results in the following invariant for machine states generated from this mapping:

Definition 5.17 (Front-bed packed)**.** Machine state $S = (0, L, Y, A)$ is front-bed packed if for $\forall x \in [0, m]$: $L(\mathtt{f}.x) > 0$ and $L(n) = 0$ for all other needles n. In addition, for all active carriers y_c, $Y(y_c) \in [0, m+1]$.

Note that the order of carriers on B is *not* necessarily the same as the order of carriers in $\mathscr{E}[\mathscr{M}(B)]$. This is because formal knitout's denotation puts all yarn carriers at the same machine location in ascending order by carrier id. Meanwhile, instruction graphs boundaries do not necessarily have this property. However, it is always possible to segment a UFO instruction graph so its boundary is consistent with knitting machine states.

Definition 5.18 (Consistent)**.** Let $B(i)$ be the i-th element of B, The boundary is consistent if for every pair $B(i)$ and $B(i+1)$ that are both arc endpoints, $id(B(i)) < id(B(i+1))$. Put another way, all contiguous sequences of arc endpoints have increasing yarn carrier annotations. A UFO partial instruction graph is consistent if both its boundaries are consistent. An instruction graph with no inputs nor outputs is also consistent.

Lemma 5.19. [Consistent UFO Graph Decomposition] Let G be a consistent UFO instruction graph. For any decomposition $G_1 \circ G_2 \cong G$ with shared boundary B, there exists a corresponding decomposition $G_1' \circ G_2' \cong G$, where $G_1' = G_1 \circ PG_B^{-1}$ and $G_2' = PG_B \circ G_2$ are consistent.

Proof. Let $c(B)$ be the permutation that takes a instruction graph boundary and makes it consistent only by permuting arc ports. Recall that any permutation o can be used to define the unique permutation tangle π_o. This means we can define UFO permutation graph PG_B that has no events and has B as its input boundary and $c(B)$ as its output. More specifically, PG_B is the unique instruction graph up to equivalence where $\mathscr{E}_G[PG_B] \cong \pi^{-1}_{c(B)^{-1}}$. Similarly, PG_B^{-1} is the UFO permutation graph with no nodes that has $c(B)$ as its input boundary and B as its output. It denotes $\pi_{c(B)^{-1}}$. We know that vertically concatenating a permutation tangle with its inverse results in a tangle equivalent to the identity. Thus $\mathscr{E}_G[PG_B^{-1}] \circ \mathscr{E}_G[PG_B] \cong id$, and $G_1' \circ G_2' \cong G_1 \circ G_2$ □

5.3.2 Instruction Generation

Using Lemma 5.19, we can assume that we are always working with consistent partial instruction graphs. We now need to define $\mathscr{L}(G)$ for every partial instruction graph, and prove that it is equivalent to G and valid with respect to its input and output states. The exact definitions are found in Section 5.3.4, but at a high level, the intuition is as follows:

Let G be our consistent UFO instruction graph with a single event. We can always decompose the graph into $G = PG^{-1}_{B^{*-}} \circ G^* \circ PG_{B^{*+}}$, where G^* is the graph with no arc crossings and only a single event, with B^{*-} and B^{*+} as its input and output boundary. Each program $\mathscr{L}(G)$ is defined so that the denoted tangle can be vertically decomposed into $K_1 \circ \mathscr{E}_G[G^*] \circ K_2$. Because $PG^{-1}_{B^{*-}}$ and $PG_{B^{*+}}$ are permutation graphs, we can prove that they denote K_1 and K_2 by showing they are also permutation slabs with the same permutation and crossing annotation. By definition, any yarn-yarn crossings in K_1 and K_2 will have the correct crossing order. Thus when defining our programs, it is sufficient to show that except for the slice that maps to G^*, the surrounding tangles only contain crossings between yarns. Put more intuitively, the ordered property means that the exact crossings between yarn carriers can be ignored so long as the yarn is routed to and from the right destinations.

Now let us describe the details of each program. $\mathscr{L}(G)$ is a valid program $kP = S \xrightarrow{ks^-;ks^*;ks^+} S'$. An event e either surrounds a node or a crossing. Either way, it can be bounded in a rectangle where a set of arcs A^- and ribbons R^- enters at the bottom, and a set of arcs A^+ and ribbons R^+ exits at the top. The size of each ribbon set is at most 2. Events where R^- and R^+ are equal are relatively straightforward. Crossings events only require swapping the physical locations of the yarns/loop stacks with `miss`/`xfer` instructions in a way that produces the correct crossing order. IN and OUT nodes have empty ribbon sets, and are just the corresponding knitout operation on $id(A^-)$ and $id(A^+)$ respectively. For KNIT nodes, the orientation and type of node is used to determine the dir and needle parameter n, while $yarns = id(A^-)$. For all these examples, both the pre and post program are nop.

For events SPLIT and TUCK, where $\#R^- < \#R^+$, a similar procedure is used to determine the arguments to the core program. However, one of the needles $\mathtt{f}.x$ used by the program is already full, which would create a spurious merge if not addressed. Thus $ks^- = SHIFT\_RIGHT(S,x)$ moves all loops and carriers to the right of $\mathtt{f}.x$ over by one to free it up (Procedure 1). In contrast, events DROP and MERGE leave behind an empty needle at $\mathtt{f}.x$, which produces a state that violates our invariant on machine states Definition 5.17. Thus $ks^+ = SHIFT\_LEFT(S^*,x)$ is used to shift loops and yarn to fill in the empty needle, where S^* is the result of running $ks^-;ks^*$ on S (Procedure 2).

Note that the machine state invariant is not only necessary for ensuring valid program composition, it is helpful for ensuring the denoted fenced tangle is equivalent. Leaving all back bed needles free ensures the SHIFT programs can always be used to correctly move loops around, and it removes the potential for extra crossings with back bed loops.

5.3.3 Program Composition

Assuming general position, it is always possible segment a UFO graph so that the resulting partial graphs each contains a single event. The final step in proving our mapping is correct is showing instruction graph composition maps to knitout program composition.

Lemma 5.20 (Instruction Graph Composition is Knitout Composition)**.** Given consistent graph $G = G_1 \circ G_2$, $kP_1 = \mathscr{L}(G_1)$ and $kP_2 = \mathscr{L}(G_2)$, $kP_1;kP_2$ is a valid knitout program, where $\mathscr{E}_K[kP_1;kP_2] \cong \mathscr{E}_G[G]$

Proof. Our mapping $\mathscr{M} : B \to S$ takes partial instruction graph boundaries to knitting machine states. The lowering function $\mathscr{L} : G \to S \xrightarrow{kP} S'$ maintains the invariant that $S = \mathscr{M}(B^-)$ and $S' = \mathscr{M}(B^+)$, where B^- and B^+ are the input and output boundaries of G, respective. The shared boundary between G_1 and G_2 maps to the shared machine state between kP_1 and kP_2, which guarantees the composition of the programs is valid. □

Procedure 1 *SHIFT_RIGHT*

Input: S, Machine state x, needle index

```
yarns ← {y : Y(y) ≠ ⊥}
i ← max({n : L(f.n) > 0})
while i ≥ x do
    for y_c ∈ yarns do
        if Y(y_c) = i + 1 then
            miss + f.i + 1 y_c;
    xfer f.i b.i;
    rack 1;
    xfer b.i f.i + 1;
    rack 0;
    i ← i − 1
```

Procedure 2 *SHIFT_LEFT*

Input: S: Machine state x: needle index

```
yarns ← {y : Y(y) ≠ ⊥}
i ← x + 1
j ← max({n : L(f.n) > 0})
if Y(y_c) = i then
    miss − f.x y_c;
while i ≤ j do
    xfer f.i b.i;
    rack −1;
    xfer b.i f.i − 1;
    rack 0;
    for y_c ∈ yarns do
        if Y(y_c) = i + 1 then
            miss − f.i y_c;
    i ← i + 1
```

5.3.4 Converting Events to Knitout

In this section, we define $\mathscr{L}(G)$ for all consistent UFO instruction graphs with a single event. For each example, we need to show that the resulting trace is valid, and that the denoted fenced tangles are equivalent. This could be proven by carefully drawing out the templated fenced tangle diagram for each picture and comparing it against $\mathscr{E}_G[G]$.

Definitions and Lemmas

Recall the high level strategy is to show that for program $kP = \mathscr{L}(G)$, $\mathscr{E}_K[kP] = K_1 \circ \mathscr{E}_G[G^*] \circ K_2$, where G^* is the slice of G that only contains an event and no other crossings, and K_1 and K_2 are the permutation tangles that connect the appropriate instruction graph boundaries of G and G^*. It is straightforward to find the knitout program that contains G^*, as we can look to ϕ for guidance. However, we can see in the knitout denotation function that any single knitout operation denotes not just the sliver $\mathscr{E}_G[G^*]$, but additional permutation tangles in the frame. If we vertically decompose that tangle, we get two permutation tangles, but the composition of those with other permutation tangles is not necessarily a permutation tangle. We need a slightly stronger mathematical object, which we define below.

Definition 5.21 (Ordered Permutation Slab)**.** A permutation slab is the unique tangle that routes strand i to position $o(i)$, where strand i crosses over j for any strand $q(i) < q(j)$, where q is some annotation that establishes a total order on its input.

Lemma 5.22 (Ordered Permutation Slab Concatenation)**.** The vertical composition of two ordered permutation slabs that share the same annotation scheme is also an ordered permutation slab.

Note that a permutation slab is just a special case of the ordered permutation slab where q maps i to itself. Thus we can show that $\mathscr{E}_K[kP] \cong \mathscr{E}_G[G]$ simply by showing that kP is a valid program that can be decomposed into $ks_1; ks_2; ks_3$, where ks_1 and ks_3 are the correct ordered permutation tangles and ks_2 contains $\mathscr{E}_G[G^*]$. Towards that goal, we make the following observations:

Lemma 5.23 (Miss Permutation)**.** Let $S = (r, L, Y, A)$ be a machine state where $L(\texttt{f}.x) = 0$ and $L(\texttt{b}.x = 0)$. kP_l is the valid trace that results from running `miss f.`x + y_l on S, where y_l is some carrier parked to the

left of $\texttt{f}.x$ ($Y(y_l) = x$). Similarly, kP_r runs $\texttt{miss f}.x - y_l$ on S, where y_r is some carrier parked to the right ($Y(y_r) = x+1$). $\mathscr{E}_K[kP_l]$ and $\mathscr{E}_K[kP_r]$ are both ordered permutation tangles.

This lemma can be combined with the rewrite rules proven in Section 4.3.2 to prove the following:

Lemma 5.24 (*SHIFT_RIGHT* Permutation)**.** Let S be a front-bed packed machine state. Then for any x, $SHIFT\_RIGHT(S,x)$ is an ordered permutation tangle.

Lemma 5.25 (*SHIFT_LEFT* Permutation)**.** Let S be a front-bed packed machine state, and $S' = (0, L[\texttt{f}.x \mapsto 0], Y, A)$. Then for any x, $SHIFT\_LEFT(S',x)$ is an ordered permutation tangle.

Instruction Graph Events

In the following section, we look at programs $\mathscr{L}(G) = S_0 \xrightarrow{ks^-} S_1 \xrightarrow{ks^*} S_2 \xrightarrow{ks^+} S_3$, where G a consistent partial instruction graph with a single event and S_0 and S_3 were mapped from from the input and output boundaries B^- and B^+ using the mapping in Definition 5.16. We use the notation $ribbon(B,i)$ for the i-th ribbon port in boundary B and $arcs(B,i)$ for the set of arcs between $ribbon(B,i-1)$ and $ribbon(B,i)$

For all cases, we know exactly which ribbon and arc ports on the boundary are connected to each event, as connecting to a different port would involve at least one crossing involving a ribbon (a second event). In addition, we know that we can create the correct permutation tangles with an operation on a single yarn carrier sequence derived from the node, as all nodes by definition have the same carrier id sequence for their input and output arc ports. We get this sequence as follows:

For any node n with arc ports $a_i \in A^-$, if the node is facing forward, $yarns = [id(a_i), s]$ for increasing i. If it is facing backwards, *yarns* is instead ordered by decreasing i.

Because this is only a proof about fenced tangle denotation, we can ignore anything related to metric properties. Thus for brevity, we will use constants s and l in all programs and leave A out of the machine state, with the understanding that it is being filled out as is appropriate.

Case: $R^- = R^+$ In this situation, there is no need to reallocate loops, so $ks^- = \texttt{nop}$ and $ks^+ = \texttt{nop}$. This means $S_1 = \mathscr{M}(B^-)$ and $S_2 = \mathscr{M}(B^+)$.

ribbon-ribbon crossings Let r_1 be the ribbon that connects $ribbon(B^-,x)$ and $ribbon(B^+,x+1)$. r_2 is the ribbon that connects $ribbon(B^-,x+1)$ and $ribbon(B^+,x)$.

If r_1 crosses in front of r_2:

$$\begin{aligned} ks^* = \quad & xfer\ \texttt{f}.i\ \texttt{b}.i; \\ & xfer\ \texttt{f}.i+1\ \texttt{b}.i+1; \\ & rack\ 1; \\ & xfer\ \texttt{b}.i\ \texttt{f}.i+1; \\ & rack\ 0; \\ & rack\ -1; \\ & xfer\ \texttt{b}.i+1\ \texttt{f}.i; \\ & rack\ 0 \end{aligned}$$

Otherwise:

$$\begin{aligned} ks^* = \quad & xfer\ \texttt{f}.i\ \texttt{b}.i; \\ & xfer\ \texttt{f}.i+1\ \texttt{b}.i+1; \\ & rack\ -1; \\ & xfer\ \texttt{b}.i+1\ \texttt{f}.i; \\ & rack\ 0; \\ & rack\ 1; \\ & xfer\ \texttt{b}.i\ \texttt{f}.i+1; \\ & rack\ 0 \end{aligned}$$

We know that both $arcs(B^-, x+1)$ and $arcs(B^+, x+1)$ are empty, as that implies the existence of at least one arc-ribbon crossing, which is a second event. Thus in both cases, $\nexists y : Y_1(y) = x+1$. This paired with the front-bed packed condition means these programs denote exactly the correct crossing between loops.

ribbon-arc crossings Let a be the arc that crosses ribbon r, which connects ports $ribbon(B^-, x)$ and $ribbon(B^+, x)$. There are four cases to consider.

a connects $arc(B^-, x)$ to $arc(B^+, x+1)$ and crosses over r:

$$\begin{array}{l} xfer\ \texttt{f.}x\ \texttt{b.}x \\ miss\ +\ \texttt{f.}x\ id(a) \\ xfer\ \texttt{b.}x\ \texttt{f.}x \end{array}$$

a connects $arc(B^-, x)$ to $arc(B^+, x+1)$ and crosses under r:

$$miss\ +\ \texttt{f.}x\ id(a)$$

a connects $arc(B^-, x+1)$ to $arc(B^+, x)$ and crosses under r:

$$miss\ -\ \texttt{f.}x\ id(a)$$

a connects $arc(B^-, x+1)$ to $arc(B^+, x)$ and crosses over r:

$$\begin{array}{l} xfer\ \texttt{f.}x\ \texttt{b.}x \\ miss\ -\ \texttt{f.}x\ id(a) \\ xfer\ \texttt{b.}x\ \texttt{f.}x \end{array}$$

Crossing order between loops and yarns in knitout is determined by whether the loop is on the front or the back bed. Thus the crossed loop is temporarily transfered to the back bed as appropriate.

IN and OUT The IN node is rotationally symmetric and has a single input arc port connected to $arcs(B^+, x)$ by a:

$$ks_{IN} = \quad in\ -\ \texttt{f.}x\ id(a);$$

Similarly, the OUT node has a single output arc port connected to $arcs(B^-, x)$ by a:

$$ks_{OUT} = \quad out\ -\ \texttt{f.}x\ id(a);$$

In both cases there is only one strand connected to the fence, so it is trivial to slide it to the correct location.

knit There are two KNIT nodes (right, left) with two rotational orientations (+, -). It has a single in/out ribbon port (r^- and r^+) and a sequence of in/out arc ports (A^- and A^+) on both its in and out boundaries. In all cases, the node's input ribbon is connected to $ribbon(B^-, x)$, and its output ribbon is connected to $ribbon(B^+, x)$.

If the node is a KNIT-right facing forward, there are arcs connecting ports in $arc(B^-, x)$ to A^-, and arcs connecting A^+ to ports in $arc(B^+, x)$.

$$ks^* = \quad knit\ +\ \texttt{f.}x\ l\ yarns;$$

If the node is a KNIT-left facing forward, there are arcs connecting $arc(B^-, x+1)$ to A^-, and arcs connecting A^+ to $arc(B^+, x)$.

$$ks^* = \quad knit\ -\ \texttt{f}.x\ l\ yarns;$$

If the node is a KNIT-right facing backward, there are arcs connecting $arc(B^-, x+1)$ to A^-, and arcs connecting A^+ to $arc(B^+, x)$. Transfers are performed so the `knit` is performed on the back bed before being returned to the front bed.

$$\begin{aligned} ks^* = \quad & \texttt{xfer}\ \texttt{f}.x\ \texttt{b}.x \\ & knit\ -\ \texttt{b}.x\ l\ yarns; \\ & \texttt{xfer}\ \texttt{b}.x\ \texttt{f}.x \end{aligned}$$

If the node is a KNIT-left facing backward, there are arcs connecting $arc(B^-, x)$ to A^-, and arcs connecting A^+ to $arc(B^+, x)$. As another backwards node, it must also be made on the back bed.

$$\begin{aligned} ks_{KNIT-left} = \quad & \texttt{xfer}\ \texttt{f}.x\ \texttt{b}.x \\ & knit\ +\ \texttt{b}.x\ l\ yarns; \\ & \texttt{xfer}\ \texttt{b}.x\ \texttt{f}.x \end{aligned}$$

Constructing the right fence for the backwards fence requires knitting on the back bed needle. The loop is immediately transferred back to front bed, ensuring $S_2 = \mathscr{M}(B^+)$.

Case: $R^- < R^+$ In this case, the core program ks^* makes use of some needle $\texttt{f}.x$ that is already occupied in $S_0 = \mathscr{M}(B^-)$. Thus $ks^- = SHIFT\_RIGHT(S_0, x)$ is run, which results in $S_1 = (0, L_0[\texttt{f}.x \mapsto 0][\forall i > x : \texttt{f}.i \mapsto L_0(\texttt{f}.i-1)], Y_0[\forall Y_0(y) > i : y \mapsto i+1])$. Because *SHIFT_RIGHT* is being applied to a front-bed packed machine state, it will always denote an ordered tangle.

tuck There is only one TUCK node with two rotational orientations (+, -). It has a single arc port on its in and out faces and a single ribbon port on its out face. The node's output ribbon connects to $r^+ = ribbon(B^+, x)$. Because the tuck node can appear between a sequence of arcs, there is a set of arcs A_{move} that connect $arc(B^-, x)$ to $arc(B^+, x+1)$. This yarn carrier set $y_m = \{a \in A_{move} : id(a)\}$ will be moved before the `tuck` operation.

For the + case, a^- is the arc that connects a point in $arc(B^-, x)$ to the node's input port, while a^+ connects the node's output port to a point in $arc(B^+, x+1)$:

$$\begin{aligned} ks^* = \quad & miss\ +\ \texttt{f}.x\ y_m; \\ & tuck\ +\ \texttt{f}.x\ id(a^-); \end{aligned}$$

For the - case, a^- is the arc that connects a point in $arc(B^-, x)$ to the node's input port, while a^+ connects the node's output port to a point in $arc(B^+, x)$. *SHIFT_RIGHT* and the additional `miss` operations on y_m have cleared out space around $\texttt{f}.x$, but we need an additional `miss` to move $id(a^-)$ into position:

$$\begin{aligned} ks^* = \quad & miss\ +\ \texttt{f}.i\ y_{move}; \\ & miss\ +\ \texttt{f}.i\ id(a); \\ & tuck\ -\ \texttt{f}.i\ id(a); \end{aligned}$$

In both these cases, the additional `miss` operations move past an empty needle, so we know from Lemma 5.23 that they denote ordered permutations. The resulting state $S_2 = \mathscr{B}^+$ in both cases, so $ks^+ = \texttt{nop}$.

split There are two SPLIT nodes (left,right) that can face forwards or backwards, for four cases total. They have a single input ribbon port r^- and two output ribbon ports r_1^+ and r_2^+, as well as a sequence of arc ports A^- and A^+ on both the in and out faces. A `split` operation results in one stack of loops on a front bed needle, and another stack of loops on the back bed needle. Thus after executing the appropriate `split` operation, ks^* must also transfer the back bed loop to the front bed in a way that results in no loop crossings.

If the node is a SPLIT-left facing forward, there are arcs connecting ports in $arc(B^-,x)$ to A^- and arcs connecting A^+ to $arc(B^+,x-1)$. Ribbons connect $ribbon(B^-,x-1)$ to r^-, r_1^+ to $ribbon(B^+,x-1)$, and r_2^+ to $ribbon(B^+,x)$.

$$\begin{aligned} ks^* = \;& split\ -\ \texttt{f}.x-1\ \texttt{b}.x-1\ l\ yarns; \\ & rack\ 1; \\ & xfer\ \texttt{b}.x-1\ \texttt{f}.x; \\ & rack\ 0 \end{aligned}$$

If the node is a SPLIT-right facing forward, there are arcs connecting ports at $arc(B^-,x-1)$ to A^- and arcs connecting A^+ to $arc(B^+,x+1)$. Ribbons connect $ribbon(B^-,x-1)$ to r^-, r_1^+ to $ribbon(B^+,x-1)$, and r_2^+ to $ribbon(B^+,x)$. Here, the `split` operation is immediately followed by `miss` operations to move *yarns* to the right of both output loops.

$$\begin{aligned} ks^* = \;& split\ +\ \texttt{f}.x-1\ \texttt{b}.x-1\ l\ yarns; \\ & miss\ +\ \texttt{f}.x\ yarns; \\ & rack\ 1; \\ & xfer\ \texttt{b}.x\ \texttt{f}.x+1; \\ & rack\ 0 \end{aligned}$$

If the node is a SPLIT-right facing backward, there are arcs connecting ports at $arc(B^-,x)$ to A^- and arcs connecting A^+ to $arc(B^+,x-1)$. Ribbons connect $ribbon(B^-,x-1)$ to r^-, r_1^+ to $ribbon(B^+,x)$, and r_2^+ to $ribbon(B^+,x-1)$. Note how rotating the node means the smaller ribbon port relative to the exemplar's axis attaches to the larger boundary port. Otherwise there would be a ribbon-ribbon crossing.

$$\begin{aligned} ks^* = \;& xfer\ \texttt{f}.x-1\ \texttt{b}.x-1; \\ & split\ -\ \texttt{b}.x\ \texttt{f}.x\ l\ yarns; \\ & rack\ 1; \\ & xfer\ \texttt{b}.x-1\ \texttt{f}.x; \\ & rack\ 0 \end{aligned}$$

If the node is a SPLIT-left facing backward, there are arcs connecting ports at $arc(B^-,x-1)$ to A^- and arcs connecting A^+ to $arc(B^+,x)$. Ribbons connect $ribbon(B^-,x-1)$ to r^-, r_1^+ to $ribbon(B^+,x)$, and r_2^+ to $ribbon(B^+,x-1)$.

$$\begin{aligned} ks^* = \;& xfer\ \texttt{f}.x-1\ \texttt{b}.x-1; \\ & split\ +\ \texttt{b}.x-1\ \texttt{f}.x-1\ l\ yarns; \\ & miss\ +\ \texttt{f}.x\ yarns; \\ & rack\ 1; \\ & xfer\ \texttt{b}.x\ \texttt{f}.x+1; \\ & rack\ 0 \end{aligned}$$

Following the logic used in the ribbon crossing case, we know that $arcs(B^-,x)$ and $arcs(B^+,x)$ are empty, and before we run ks^*, yarn carrier position x is empty. In all these cases, any additional `miss` operations are performed after `split` and before moving the loop on $\texttt{b}.x-1$ to $\texttt{f}.x$. This means $\texttt{f}.x$ is empty during the `miss`, and the operations denote ordered permutations. In addition, when the loop is moved to fill in the empty needle, yarn carrier position x is still empty, and no yarn-loop crossings occur. The resulting state $S_2 = \mathscr{M}(B^+)$, so $ks^+ = \texttt{nop}$.

Case: $R^- > R^+$ In this situation, all loops are in position for the core program, so $ks^- = \texttt{nop}$, and $S_1 = \mathscr{M}(B^-)$. However, running these core programs results in state $S_2 = (0, L_1[\texttt{f}.x \mapsto 0], Y_1)$, where the empty needle at $\texttt{f}.x$ means S_2 is not front-bed packed. Thus $ks^+ = SHIFT\_LEFT(S_2,x)$ is used to fill in the empty needle and produce state $S_3 = (0, L_2[\forall i \geq x : \texttt{f}.i \mapsto L_2(\texttt{f}.i+1)], Y_0[\forall Y(y) > i : y \mapsto i+1])$. This results in $S_3 = \mathscr{M}(B^+)$.

merge The MERGE node has two input ribbon ports (r_1^- and r_2^-) and a single output ribbon port r^+. Since the node is rotationally symmetric, we only need to consider one case. Let r_1^- be connected to $ribbon(B^-,x-1)$, r_2^- be connected to $ribbon(B^-,x)$, and r^+ be connected to $ribbon(B^+,x)$.

$$\begin{aligned} ks_{MERGE} = \quad & xfer\ \texttt{f}.x\ \texttt{b}.x; \\ & rack\ -1; \\ & xfer\ \texttt{b}.x\ \texttt{f}.x-1; \\ & rack\ 0; \end{aligned}$$

Much like the argument for the ribbon crossing case, we know that $\nexists y : Y_1(y) = x+1$ because the graph has a single event. Thus the program stacks the loops without introducing additional crossings.

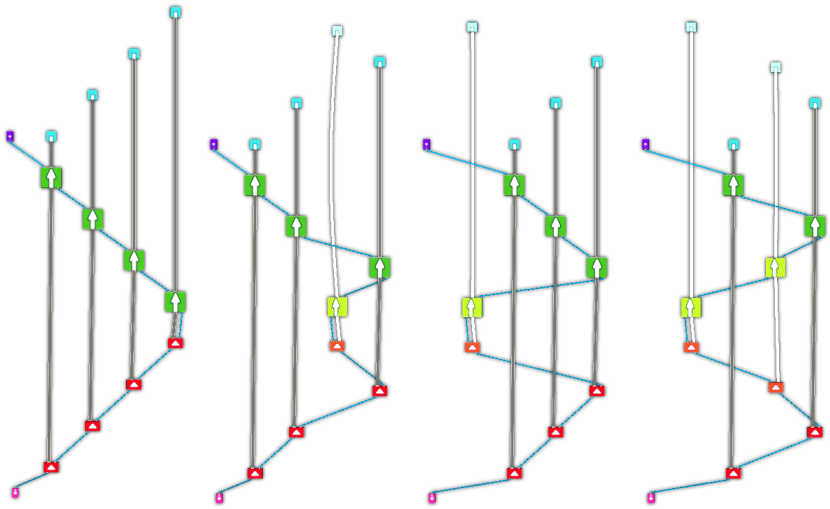
drop A DROP node has only has a single input ribbon port r^-, and it is rotationally symmetric along its y-axis. Thus there is a single case. Let r be the ribbon connecting r^- and $ribbon(B^-,x)$.

$$ks_{IN} = \quad drop\ \texttt{f}.x;$$

5.4 System Implementation

In the previous sections, we established that all machine-knittable structures can be represented as UFO instruction graphs, and all UFO instruction graphs are machine-knittable. In this section, we present the implementation of our compiler and an associated instruction graph construction and editing system, which demonstrates the practical applications of our theoretical framework. Our compiler possesses two key properties: completeness and soundness. Completeness ensures that the compiler can handle all machine-knittable structures as input, while soundness guarantees that if the compilation process succeeds, the output will be machine-knittable. We present the implementation of our system in three subsections, and then talk about the system limitation.

Our system is implemented in C++, and provides visualization (during editing) using the OpenGL API. The data structures to represent an instruction graph are reasonably straightforward. Each node stores a position, orientation, and scale, along with a pointer to an exemplar which defines the node type. Yarn arcs and loop ribbons store references to the nodes they connect to, with loop ribbons additionally storing a flag to indicate the orientation of their connection. Yarn arcs are stored as polylines with associated metadata



(a) full gauge (b) mixed gauge (c) long float (d) half gauge

Figure 5.5: The instruction graph of a stockinette sheet can be laid out such that all knit faces are aligned with the viewing axis (a) This compiles to a full-gauge sheet. If we take a column of nodes and fold it over, the result can be a program with mixed gauge (b) or an excessively long float (c) depending on the relative locations of the nodes. Two layers with the same number of columns can be alternated to produce half-gauge code (d). It is important to note that all these graphs are equivalent from a topological perspective. The variations (b), (c), and (d) were generated by applying deformations to the original graph (a) under ambient isotopy using our GUI implementation (see section 5.4.1).

for (e.g.) construction lengths; loop ribbons are also stored as polylines with metadata, including a twist value which indicates the total deviation of their frame from the minimum-twist frame along their length.

5.4.1 Semantic Preserving Graph Rewrite

Our system supports semantics-preserving instruction graph rewriting (e.g., Figure 5.5). Users can translate, rotate, and scale portions of the instruction graph, while the system runs a rudimentary physics system to prevent interpenetration. Particularly, each frame, the system dices arcs and ribbons into chains of spheres with equal separation, moves each sphere toward its neighbors to smooth/shorten the chains, and resolves collisions by incrementally pushing the spheres apart. Pull-through is avoided by limiting the step size. Twist values are updated incrementally by recomputing the minimum-twist frame along each ribbon and choosing the minimum rotation possible that keeps the loop aligned to its connection point.

Semantics-preserving rewrites are useful because, as we will explore in depth in the next section, there exist instruction graphs that are not UFO but can be deformed into UFO, making them fabricable. This step is analogous to *user-scheduling* [1] in rewrite-based user-schedulable languages ([Ikarashi et al. 2022; Hagedorn et al. 2020]). In user-schedulable languages, scheduling performs loop transformations that preserve the semantic equivalence of the input program. Similarly, in our context, the user performs graph transformations that preserve the semantic equivalence of the input graph.

[1] not to be confused with the resource allocation problem in knitting, which uses the same term but has a different meaning

5.4.2 UFO Check

The lowering process is only well-defined on an instruction graph if that instruction graph satisfies the upward, forward, and ordered properties. Therefore, our system checks these properties before running its lowering code. Since all machine-knittable structures are UFO, if the input graph does not meet these criteria, the system can confidently emit an error. This step is analogous to backend checks, such as data-race detection and memory access checks, in traditional compiler implementations.

Upward. To check upwardness, the system to iterates through all nodes, yarns, and loops, locally checking orientation (nodes) and monotonicity (yarns, loops).

Forward. Forwardness checking is also easy, and involves checking node orientations and iterating along every ribbon to look at the local frame's y-axis. (In the case of the ribbon's local frame, a tolerance is used to allow for floating point error when computing the orientations.)

Ordered. The ordered check is the only one that requires global information about the instruction graph (particularly, all live yarn edges at a given y-coordinate and all yarn-yarn crossings when the graph is projected along the z-axis). This check is deferred until the sweep-line-based planar arrangement construction detailed in the next section.

5.4.3 Lowering ($\mathscr{L}$)

So far, using our GUI, we have demonstrated that instruction graphs can be rewritten into a UFO presentation if possible, and the UFO check can ensure that the graph indeed satisfies the UFO properties. The final part is implementing the lowering function ($\mathscr{L}$), which translates the input instruction graphs into a formal knitout. This step is analogous to backend code generation in traditional compiler implementations.

Our lowering implementation detects events by incrementally constructing a planar arrangement using a sweep-line algorithm. To make the computation easier to think about, all positions are rounded to an integer grid, with the size of the grid chosen to be small relative to the feature size of the instruction graph in order to avoid aliasing. Nodes are replaced by a single point at their centers, with extra edges stretching out to the connected arcs and ribbons (during event processing, the purpose of each connected ribbon can be deduced by its order in entering/exiting the node).

The core of the function is an event processing loop that iterates through events by ascending z-coordinate, storing a next-event queue and a sorted list of current yarn and loop edge segments. At each event (crossing or node), it calls an event processing function and updates its next-event queue and current segments list. Our code uses the GMP library's exact rationals to compute exact results where appropriate (e.g., when sorting event times).

The event processing functions proceed as described in Section 5.3, generating the appropriate formal knitout instructions to implement each node and update the on-machine layout of all loops. The ordered property is also incrementally checked by running code on arc-arc crossings to check carrier ids, and by looking through the list of active yarns whenever a new yarn is brought in.

5.4.4 System Limitations

Our GUI system is both sound and complete. It is complete because users can deform any machine-knittable structure into a fabricable UFO presentation using semantics-preserving rewrites. This ensures that our compilation process can handle all machine-knittable structures as input. The system is sound

because the UFO check verifies that the instruction graph satisfies the UFO properties, ensuring it meets the necessary conditions for machine knittability, and the lowering function correctly translates the instruction graph into a valid and executable knitout. However, it is important to note that the soundness and completeness of our system are only with respect to topological equivalence. There are many features, described below, that we did not consider in our current implementation.

Program Efficiency

In order to guarantee a topologically correct knit program for any UFO input, our compiler makes several assumptions that are bad for knit program efficiency. For example, leaving everything stored on the front bed by default means the program is constantly performing transfers that could be removed entirely. To fabricate our results, we applied the local rewrites from Section 4.3 to improve the pass consolidation in the compiled knitout. Modifying the compiler to allow for long term back-bed storage and simple pass consolidation is a clear improvements.

Metric Properties

In the prior sections of this chapter, we described a compilation process from an instruction graph to formal knitout that guarantees topological correctness as defined with fenced tangles. However, in order to generate fabricable programs, we also need to ensure the metric properties, or amount of yarn in different regions of the object, is appropriate. These properties are controlled by the l and s parameters in loop-making operations. Small l and s values generally result in smaller, stiffer stitches, while large l and s values produce larger, looser stitches. Extreme metric parameters beyond the capabilities of the yarn/machine will often result in fabrication failure. If the s parameter is too small relative to the attach point of the yarn carrier, excess tension can tear the yarn and even damage the machine. Meanwhile, an excessively large s parameter will result in too little tension, which makes loop formation less reliable. A general solution for satisfying metric constraints is beyond the scope of this thesis.

Thus, in our implementation, we defer this problem by putting additional metric annotations on ribbons and arcs that are propagated to the l and s parameters of the appropriate knitout operations. This places the burden of checking compatibility between metric annotations and the scheduled program on the user. However, despite these restrictions, we note several techniques to help users generate reasonable metric results.

Prior work on generating knit programs from unscheduled input often has a 'regularity' property, where the desired amount of yarn per stitch is approximately constant throughout the knit [Narayanan et al. 2018; Nader et al. 2021]. Put another way, there is a (mostly) constant l and s used throughout the entire program. Because our compiler schedules instruction graphs by packing events as closely together as possible, it is limited in the types of layouts it can generate. However, we can exert some control on the metric layout.

Let us consider a simple instruction graph that denotes a sheet that is four loops wide and only uses front bed operations (Fig. 5.5). Properly tracking float length for arbitrary programs is difficult, as transfers and misses can be used to wrap ongoing floats around other yarns or loops. For these simple examples, we can approximate float length $f = |Y_j(y) - A_j(y)|w + \varepsilon$ for $ks_j \in [\texttt{tuck}, \texttt{knit}, \texttt{split}]$, where the width of a needle is w and ε accounts for the small distance between needles. An immediately obvious layout is to make it so that all nodes face forward and lie in adjacent columns, Fig. 5.5a. When scheduled, this results in a full-gauge sheet, where $f = \varepsilon$ for all floats. We can also use the transformations implemented in Section 5.4.1 to take the rightmost edge and fold it over Fig. 5.5b. The resulting instruction graph is still UFO, but it now constructs the fourth column on `f2`, and the third column on `f3`. This results in a

mixed gauge sheet, where $f = w$ for some needles, while the $f = \varepsilon$. This transformation can be taken to an extreme by stretching the end column even further until it is at the leftmost edge of the sheet Fig. 5.5c. This results in a float of length $3w + \varepsilon$ between `f.0` and `f.3`, which is likely to strain the involved yarn. However, if we rotate and translate an additional column so that front and back bed alternate, we get a half-gauge schedule, which like full-gauge, has a mostly consistent distance of $w + \varepsilon$ between attached loops. Thus we see how an expert user can leverage their choice of UFO graph embedding to control the metric properties of the resulting program. Note that through all of these manipulations, our system our system guaranteed that the toplogical properties remained consistent.

User Interface

We presented a GUI meta-programming interface for instruction graphs, enabling users to interactively manipulate and transform these graphs. Many potential improvements could enhance the user experience, particularly in the user interface for semantic-preserving graph rewrites (Section 5.4.1). Although it is theoretically possible to generate UFO presentations for all the examples in Section 5.5 using the deformation rewrite feature of our editor, this process proved to be excessively tedious in practice. As a result, all the instruction graphs in the case studies were directly meta-programmed using C++ code. This cumbersome manual process at the stitch level is unsurprising, given that instruction graphs are designed to function as an IR rather than a user-facing abstraction.

Designing a user interface to leverage its capabilities effectively is future work. This will involve determining the appropriate meta-programming abstractions for these transformations, such as clipping to an axis or repeating patterns. Future features could include additional transformations, heuristics for optimal UFO embedding, and the development of a lifting function to convert the instruction graph back into a human-readable knitting pattern. Moreover, while enriching the frontend meta-programming interface could be an interesting avenue for exploration, it may not be strictly necessary if the instruction graph becomes an IR targeted by other knitting compiler systems.

5.5 Case Studies

This section will explore the actual fabricated results and analyze how UFO properties manifest in knitting structures. We will discuss the importance of topological correctness, the relevance of this work to practical knitting, and highlight the practical implications of UFO properties.

5.5.1 Interlock Pocket

As discussed in Chapter 2, existing systems struggle to fabricate interlocking structures. The Interlock Pocket shown in in Figure 5.6 is an example of a structure that existing systems could only create using unverified escape hatches, which distort the geometry of the desired object. In order to fabricate the Interlock Pocket using these systems, you would need to specify a design that doesn't resemble the intended final product (Fig. 2.9). However, the interlocking structure that stitch graphs struggle with can be easily handled using instruction graphs. With instruction graphs, we can successfully employ the naive strategy used in Fig. 2.9 to lay out the geometry of nodes and connections in space. Instruction graphs treat the routing of yarns and crossings as explicit information to be compiled, ensuring that the necessary crossings are generated accurately. This approach enables the successful fabrication of interlocking structures. Furthermore, we can transition between interlock and a folded stockinette sheet to create a pocket-like structure. To make the instruction graph UFO, we can simply stretch it out along the z-axis. The fab-

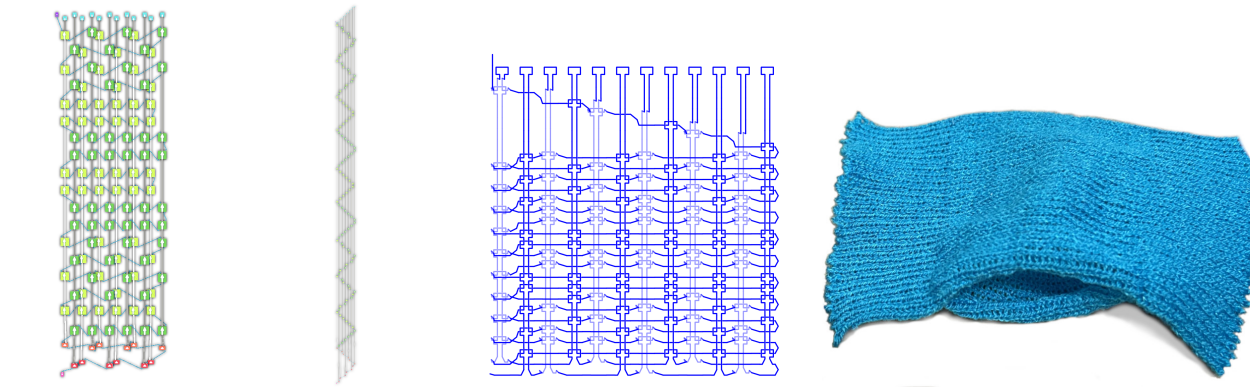


Figure 5.6: Instruction graphs are a *complete* unscheduled representation of machine knitting. An instruction graph is machine knittable if and only if it has an upward, forward, and ordered (UFO) presentation. Instruction graphs can represent complex knit structures outside the capabilities of other unscheduled knit representations. For example, this interlock sheet that contains a stockinette pocket (opening to the right in the instruction graph and knitout code). Knit object is from a larger version of the same pattern.

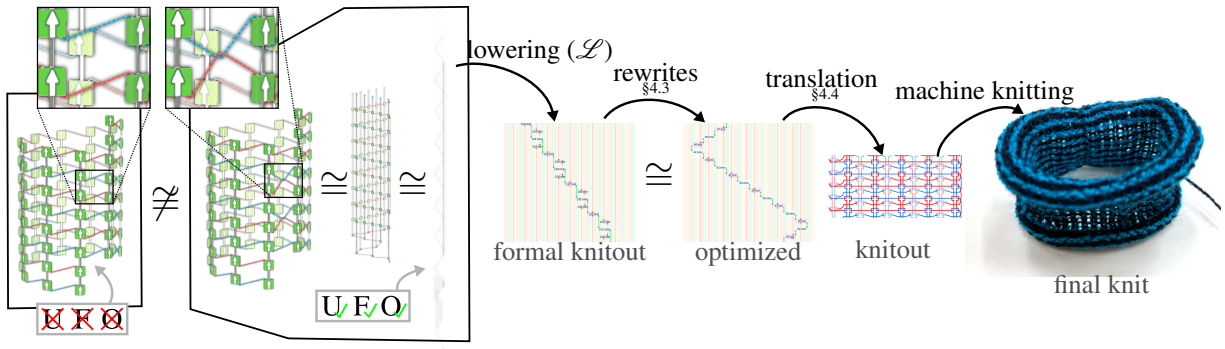


Figure 5.7: Making a striped tube. A basic two-color striped tube is difficult to machine knit because there is no way to create an ordered presentation graph using only rotation and flattening. However, a similar design that adds a twist between the yarns *can* be flattened to an equivalent upward, forward, and ordered presentation. We used our system to convert a higher-stitch-count version of this instruction graph to formal knitout; manually simplified the formal knitout using rewrites; and produced knitout code. When up-scaled and run on an industrial knitting machine the code produces a spiral tube as desired.

ricated result of the Interlock Pocket can be seen in Fig. 5.6, demonstrating the capability of instruction graphs to handle complex interlocking structures that challenge existing systems.

5.5.2 Barber Pole

The barber pole example is a structure that autoknit's stitch graph can describe but cannot schedule correctly – when scheduled by autoknit, rather than resulting in a hollow tube, the tube is "tagged" together by carrier crossings resulting in a figure-of-8 shape (Fig. 2.8). If, on the other hand, we transform this example barber pole into an instruction graph (Figure 5.7), we encounter non-ordered crossings that are

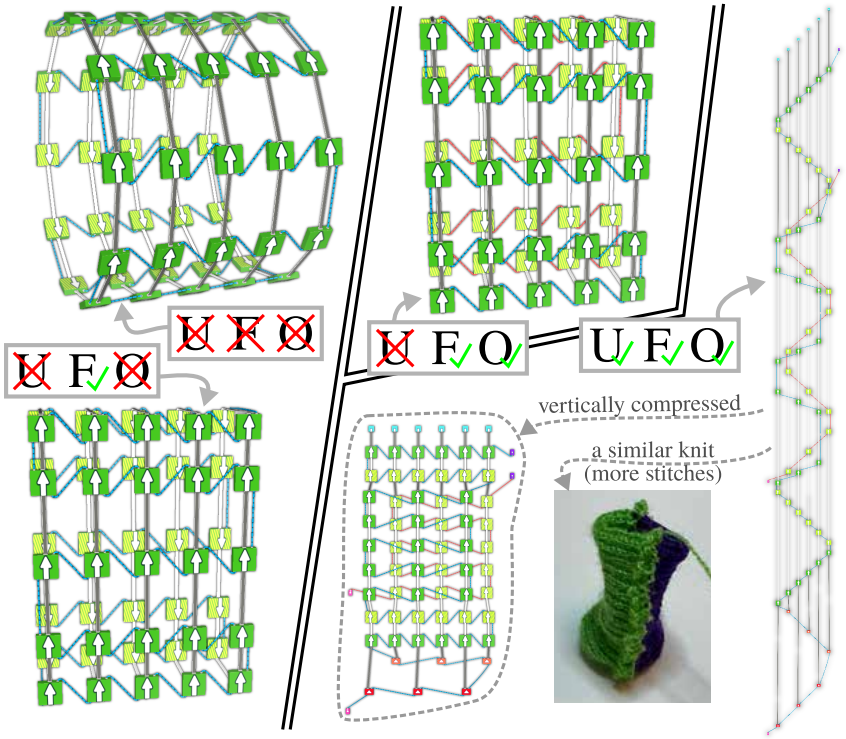


Figure 5.8: A naive infinity scarf (upper-left) can be flattened into two layers to produce a Forward graph (lower-left). The cyclic yarn and stitch dependences prevent transforming the graph to be Ordered and Upward. This requires changing the Instruction Graph to use two different yarns (upper middle) allow it to be made Ordered; and, futher changing the graph by modifying the orientation of the nodes (right; with height-compressed version in dotted circle) allows it to be made fully UFO, and thus knittable. The photograph shows the results of knitting a vesion of the UFO instruction graph with more rows and columns of stitches added.

difficult to remove solely through the deformation of the object. Rather than knitting the wrong thing, the instruction graphs system shows us a knitability flaw in our specification[2]. Adding twists between the yarn carriers in a single column allows us to make the example UFO, and, further, gives us the control to place this crossings on the edge of the object where they will not glue the front and back together.

5.5.3 Infinity Scarf

In this section, we walk through how the UFO properties can guide us in transforming an unfabricable design – a cyclic infinity scarf, Figure 5.8 – into something that can actually be machine knit. Instead of pursuing an impossible transformation, we create a different instruction graph that maintains the same high-level geometric properties (a tube knit sideways). To create an infinity scarf, one might start by envisioning the structure like the upper left in Fig. 5.8. Using our GUI editor, it is possible to deform the graph to satisfy the forward property (lower-left). However, the cyclic dependencies inhibit both the ordered property by forcing a single yarn to be in two places at the same time. Thus this object cannot be

[2]It is possible that there exists a way to transform the original input without changing its topology. However, finding such a transformation or proving its existence is challenging, and we did not pursue this avenue in our work. Moreover, even if we find a transformation, it might be impossible to invert on the physical object.

made on a knitting machine. To satisfy the ordered property, we need to construct a different graph (upper middle) in Fig. 5.8 using two yarns instead of one. This modification eliminates the cyclic dependency and allows the graph to be ordered. However, this graph still has a problem; it is not upward because the stitches on the back (depicted in light green) are facing downwards. No knitting machine can create a knit upside-down. To address this issue, we have to create yet another graph, shown in the bottom-middle of Fig. 5.8. This graph can be made UFO by stretching it along the z-axis (rightmost figure), making it fabricable. By transforming this modified graph into a UFO presentation and knitting it, we successfully produce the desired infinity scarf.

Chapter 6

Practical Verification of Program Equivalence

In prior chapters, we have used fenced tangles to prove the correctness of a mapping between knitting representations: first a set of rewrite rules of knitout programs, and then a compilation function from instruction graphs to knitout programs. However, what if we are not presented with a mapping? What if we are presented with two knitting programs of unknown origin and asked to evaluate whether they are equivalent? As stated before, a general solution for knot equivalence is NP-hard [Koenig & Tsvietkova 2021]; since knots are a subset of fenced tangles, a general solution to fenced tangle equivalence feels unlikely. However, what we need is not a general solution to fenced tangle equivalence, but a solution that works on *machine knittable* fenced tangles. In fact, if we restrict our set of knitting programs further, we can work with a much simpler topological object with multiple polytime solutions: the Artin Braids.

Braid theory is the topological study of groups whose elements are intertwining strands with fixed endpoints [Murasugi & Kurpita 2012]. More precisely, braids are the subset of n,n slab tangles where all arcs move monotonically from the input to output boundary. While most knitout operations do not denote braids, three do: `xfer`, `rack`, and `miss`. This chapter uses this observation to optimize *transfer plans*, or subprograms that contain only `xfer` and `rack` operations (though we will discuss how this work can be extended to `miss` operations). The Artin Braid Group not only enables fast equivalence checks on arbitrary pairs of transfer plans, but also a measure of complexity that can be used as a lower bound for transfer plan complexity. These combine to enable search for optimal transfer plans.

6.1 The Artin Braids

Definition 6.1 (The Artin Braids)**.** Let $b \in \mathscr{S}_n^n$ be a tangle where the movement of strands are monotonic along the time axis.

Much like with fenced tangles, braids can be composed with each other to form more complex braids. Put another way, we can decompose a braid into smaller, simpler braids, where each braid only has a single crossing. These braids can then be used to *generate* any braid, and any composition of these generators will result in another braid. The Artin Braid Group also has a corresponding algebraic definition that describes the composition of generators as a word:

Definition 6.2 (The Artin Braid Group)**.** The Artin Braid Group B_n on $n > 1$ strands is the group generated

by generators $\sigma_1^+ \dots \sigma_{n-1}^+$ with the equivalence relations:

$$\begin{aligned} \sigma_i^+ \sigma_j^+ &= \sigma_j^+ \sigma_i^+ && \text{for } |i-j| > 1 \\ \sigma_i^+ \sigma_j^+ \sigma_i^+ &= \sigma_j^+ \sigma_i^+ \sigma_j^+ && \text{for } |i-j| = 1 \end{aligned}$$

When writing and drawing braids, we use the convention that positive crossing, σ_i^+, represents the ith strand crossing *over* the $i+1$st strand, and the inverse (negative) crossing, σ_i^-, represents the ith strand crossing *under* the $i+1$st strand. Braid words are products of generators, where the leftmost generator is the most recently executed generator, and the rightmost generator is least recent.

An inverse of a word can be quickly found as follows:

Definition 6.3 (Braid word inverse)**.** Given the braid word $W = \sigma_i^{\pm} \sigma_j^{\pm} \dots \sigma_k^{\pm}$, the word $W^{-1} = \sigma_k^{\mp} \dots \sigma_j^{\mp} \sigma_i^{\mp}$ is its inverse, where the product $W^{-1}W$ is equivalent to the trivial identity braid with no crossings, ε.

Due to the equivalence relations, each member y of the braid group B_n is an equivalence class of braid words, all of which represent the same underlying topological braid. The word problem on braids asks whether two braid words W and W' belong to the same equivalence class, i.e. are the same topological braid. There exist a variety of solutions to the word problem, of which several use what are known as simple positive braids:

Definition 6.4 (Simple positive braid)**.** The simple positive braids are the set of braids where all crossings are positive, and every pair of strands crosses at most once.

The symmetric normal form [Dehornoy 2008] rearranges each braid word into a carefully ordered sequence of simple positive braids and inverse simple positive braids. This sequence is proven to be unique for each member of the braid group, thereby reducing the word problem to a question of strict equality.

6.2 State Representation

Because this chapter is based on an earlier paper focused on transfer planning [Lin & McCann 2021], it uses a different convention from Chapter 4 for mapping the machine state in 3D to a 2D ordering necessary for fenced tangle diagrams. Note that here, loops are ordered left-to-right, back-to-front, as seen in Figure 6.1 (note how the assignment of loops 1 and 2 swaps). Traditionally, transfer planning ignores the presence of carriers. We may justify this by imagining that all carriers are functionally inactive, or parked outside the extent of the transfer plan. Unlike with full knitting programs, transfer plans only move existing loops and never change the existing number of loops on the machine. Thus instead of representing machine state as a partial function from needles to loop count, we instead instantiate a constant number of loops L, where $L[i]$ is a loop's current needle location. Furthermore, no operation separates the two legs of a loop from each other, nor induces twist between them. Thus we can treat each loop as a single strand in the braid. From there, a loop's needle location is used to establish its position within the braid. In addition, this work looks at actual knitout instead of formal knitout. In practice, this only means that `rack` operation used throughout the chapter is in reality closer to the *RACK* macro (Definition 4.11).

Observe that any transfers that occur at racking r do not change the loop ordering. It is only when the machine's racking value changes that any loops in back bed b would be sent to different needles in front bed f, potentially changing the loop ordering. Furthermore, loops on the same bed never change order relative to each other; a loop in f can only potentially cross a loop in b. Thus the resulting braid word from a racking operation can be found by tracking the differences in loop ordering using Algorithm 3.

Let V be the braid word produced by a single racking operation and Y be the symmetric normal braid representing the previous state's ordering. The braid word VY represents the new state. Note that a single

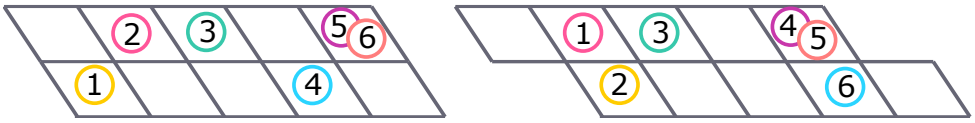


Figure 6.1: The left-to-right, back-to-front ordering on loops (colored circles) on a machine at zero racking (left) and -1 racking (right). Note how the relative order of loops on the same bed does not change. This includes loops on the same needle.

Procedure 3 Racking Operation

Input: ordered list of loops L, old racking r, new racking r'
Output: braid word V

$V \leftarrow \varepsilon$
if $r' < r$ **then**
  for $i = 0 \ldots n$ **do**
    if $L[i] \in b$ **then**
      for $j = i-1 \ldots 0$ **do**
        if $L[j] \in f \wedge L[i] + r' \leq L[j]$ **then**
          $V \leftarrow \sigma_j V$
else if $r' > r$ **then**
  for $i = n \ldots 0$ **do**
    if $L[i] \in b$ **then**
      for $j = i+1 \ldots n$ **do**
        if $L[j] \in f \wedge L[i] + r' > L[j]$ **then**
          $V \leftarrow \sigma_{j-1}^{-} V$
return V

rack operation causes any two strands to cross only at most once, and resulting crossings are either all positive or all negative. Thus V is either a simple positive braid or its inverse. Therefore, the symmetric normal form of VY can be calculated in $O(n\log(n))$ operations [Epstein et al. 1992].

Given an initial list of loop locations and an initial braid, the resulting state from any list of transfer operations can be determined. Because the ordering used for the braid word also captures the ordering of loops in a stack, it is sufficient for a state to only store each loop's needle position instead of explicitly storing the loop list of each needle.

6.3 Optimal A* search

Given this discrete representation of transfer plan meaning, I now use search techniques to find minimum-length transfer plans. The search is based on A*, where a state's immediate neighbors are those reachable via any number of transfers followed by a single racking operation, and the goal has a braid equivalent to input braid W and loops at target needle locations L. We now model the constraints of the machine knitting process to restrict the search, and propose several modifications to the state representation to improve search performance.

6.3.1 Constraints

Loops on a knitting machine are constructed using a single continuous strand of yarn, where the amount of yarn between adjacent loops can be varied. This physical connection can break when connected loops are moved too far apart. To account for this, we define a slack constraint $[s_-, s_+]$ on the distance between connected loops l_a and l_b. In other words:

Definition 6.5 (Slack Constraint)**.** Connected loops l_a and l_b respect slack when $s_- \leq pos(l_a) - pos(l_b) \leq s_+$, where

$$pos(l) ::= \begin{cases} i & \text{if } l \in f_i \\ i + r & \text{if } l \in b_i \end{cases}$$

Note that, for loops on opposite beds, the machine's racking affects whether the loops respect slack. Thus the set of all connected loops which lie on opposite beds can be used to define a valid racking range for a given machine state.

We also explicitly define which needles are available, as certain needles on the machine may be occupied by loops that should not be moved by the transfer plan. This must also be taken into account when determining whether a transfer is reversible.

6.3.2 Cost Model

Recall that the machine can execute any number of transfers in a single transfer pass as long as they occur at the same racking. Therefore, it's useful to think of transitionsbetween machine states as $(\{xfer\}, rack)$ pairs, where $\{xfer\}$ is one of the 2^n subsets of n total distinct transfer operations for a given state, and *rack* is one of the valid racking operations for the state post $\{xfer\}$. The cost of a transition is 0 if $xfer$ is empty, and 1 otherwise.

6.3.3 State Equivalence

A* search stores visited states in memory in order to avoid expanding the same state multiple times. This makes fast state equality checks essential. In fact, we can do better than just strict equality. For two different states which produce the exact same set of subsequent states under expansion, we can prune the state space by searching only one of them. In this section I define a function, *canonicalize*, to identify such equivalence classes in which this property holds true.

Recall that $\texttt{xfer}(n, n')$ can be reversed by $\texttt{xfer}(n', n)$ if n' is an empty needle. Consider two states S and S', which are identical except for a single loop that is on needle n in state S, and needle n' in state S'. If $\texttt{xfer}(n, n')$ is reversible and the transfer set from S to some state includes $\texttt{xfer}(n, n')$, then that transfer can be reversed with $\texttt{xfer}(n', n)$ to acquire the transfer set that would reach the same state from S'. If the transfer set from S does not include $\texttt{xfer}(n, n')$, then S' can use the same set plus $\texttt{xfer}(n', n)$. This means that in a search, no matter which of the two states are expanded first, it will visit all states that can be reached from the other state, making the second expansion redundant.

Now consider some non-empty transfer set X at racking r. This set would have cost 1. Any number of additional transfers can be performed before and after X, and as long as they also occur at racking r, they can be rolled into the existing transfer pass for cost 0. Thus if we define some reversible transfer function that sends all equivalent loop states to a single, canonical loop state, the function can be applied before the duplicate check, making it a simple equality check. These reversible transfers would then combine with the $xfer$ portion of the expansion, making it a zero-cost transformation. Let *canonicalize* denote an operation on a given state, which performs all reversible transfers on back bed loops, essentially loading as many loops as possible to the front bed without performing any irreversible transfers (Figure 6.2).

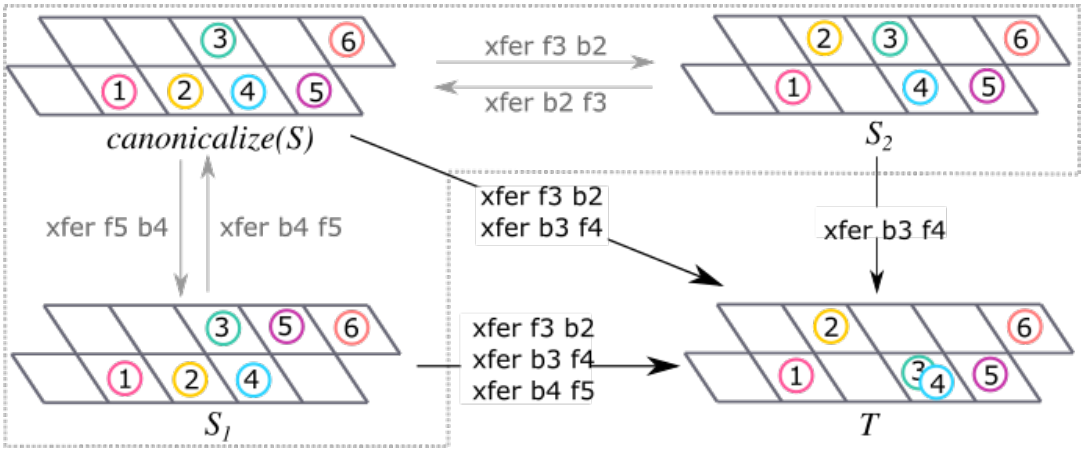


Figure 6.2: A transition from S_i to state T costs a single transfer pass even if it is preceded by some number of reversible transfers. Thus the states are equivalent, and we can use a single canonical state $canonicalize(S)$ when representing them in a search.

Then, during the search, our code stores visited states and checks against them all under application of *canonicalize*.

6.3.4 Heuristics

To guide the search, we provide the following heuristics based on the braid word and the loop locations.

Braid Word Length

Let $len(Y)$ give the number of simple braids contained in the symmetric normal form Y. This notion of braid length changes by at most one after multiplication by another simple braid [Epstein et al. 1992]. Rather than start with the identity braid ε and check equivalence against the target braid W, we can let the initial state be W^{-1} and let ε be the target braid. The resulting braid produced by the plan will still be the target braid W, and $len(Y)$, where Y is the symmetric canonical braid stored in the state, will be a consistent heuristic.

Offsets

If we consider a relaxed version of the transfer planning problem that has infinite slack, reversible loop stacking, and is only concerned with needle position and not relative ordering, then a solution can be found by solely considering each loop's offset, or the single racking value that would move a loop from its current location to its destination. We can define $\mathscr{O}_{p,r}$ – the set of sets of offsets that can be brought to zero in p steps starting at racking r – using the following recurrence:

$$\mathscr{O}_{p+1,r'} \equiv \left\{ \{x, x+(r'-r) \mid x \in S\} \mid S \in \mathscr{O}_{p,r} \right\} \tag{6.6}$$

$$\mathscr{O}_{0,r} \equiv \begin{cases} \{\{0\}\} & \text{if } r = 0 \\ \emptyset & \text{otherwise} \end{cases} \tag{6.7}$$

Notice that the number of offsets at most doubles every step. Thus we can establish the following lower bound:

Theorem 3. Any problem with n unique non-zero offsets requires are least $\lfloor log_2(n+1) \rfloor$ passes to solve.

Our offset table heuristic goes further, computing and storing $\mathscr{O}_{p,r}$ (effectively, a pattern database [Felner et al. 2007]) for $r \in [-8,8]$ and $p \leq 8$. In order to facilitate fast lookups in $\mathscr{O}$, our code builds a table – for every racking r – of all maximal[1] achievable offset-sets and their associated minimum step count. This table is sorted by step count. To look up a query set in the table for the current racking value, the code examines entries in order until a containing set is found and returns the associated step count. This lookup is accelerated by maintaining a "skip" value for each offset alongside each row in the table, indicating the next table entry in which that offset appears. These skip values are used by our code to avoid needing to check every row.

6.4 Results

All experiments were performed on a mid-range workstation-class computer running Debian GNU/Linux with an Intel Core i7-8700K 3.7GHz / 12-thread CPU – though our search is single-threaded – and 64GB of RAM.

Optimal plans We compare the optimal plans produced by canonical-node A* search against two existing transfer planning algorithms: schoolbus+sliders (`sb+s`) and collapse-shift-expand (`cse`). Both algorithms have limitations on the types of problems they can and are best suited to solve, so we used three test sets for comparison.

For each test case, our test harness first ran the existing algorithm to generate a transfer sequence, then used that transfer sequence to construct a target state, and, finally, ran our search algorithm with that target state. (This procedure is needed because `cse` chooses stacking and rotation direction without user control, so it may plan to one of several possible output states.)

The first set, `flat-lace`, consists of all 28,696 unique transfer problems with eight loops, stacks of at most three loops, distance between loops increasing by at most one, and no loop crossings (cables/twists). Problems that only differ by translation are considered equivalent. In other words, this is the set of eight-loop problems that `sb+s` can solve. Results are shown in Figure 6.3. As expected, `sb+s` is able to solve these cases in relatively few (< 15) passes, and, indeed, is optimal in 2942 of the cases ($\approx 10\%$ of trials which finished). However, there is still room to improve, since in the remaining cases, `sb+s` uses up to $2.8\times$ the passes of the optimal solution.

The second set, `simple-tubes`, contains all 2113 problems on eight-loop tubes, where pairs of adjacent loops can either remain adjacent, be stacked atop each other, or be separated by an empty needle, and the overall tube can be rotated. These mimic the basic shaping operations used, e.g., by [Narayanan et al. 2018]; and, thus, the problems that `cse` was designed to solve. Figure 6.4 shows that `cse` produces optimal solutions in a much smaller fraction of problems: only 36 ($\approx 1.7\%$) were optimal, and solutions were sometimes more than $4\times$ slower. This is not unexpected, as `cse` may require up to $O(n^2)$ passes in the worst case [Lin et al. 2018].

For the final set, `cable-tubes`, we constructed 1183 transfer sequences by prepending 1x1 and 2x2 cables (loop crossings) to plans generated by `cse` for eight-loop tubes with rotations. The comparison with optimal transfer sequences (Figure 6.5) shows that there are significant fabrication speed gains to be made by combining cable and rotation transfers instead of performing them sequentially, as might be done by a programmer putting together an ad-hoc solution.

[1] I.e., if both $S, R \in \mathscr{O}_{p,r}$ and $S \subset R$, then only R is stored.

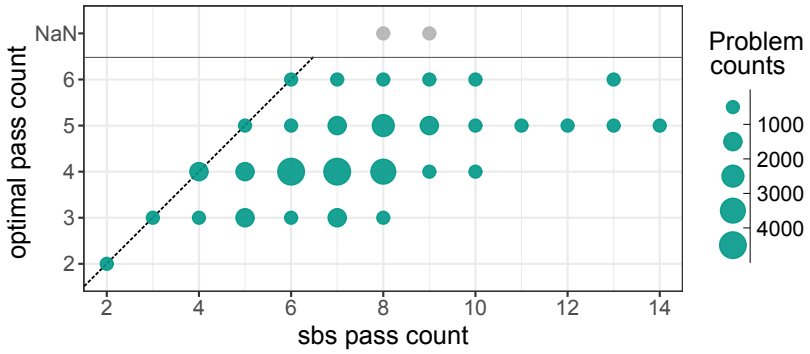


Figure 6.3: The Schoolbus + Sliders (`sb+s`) algorithm [Lin et al. 2018] produces transfer plans within a factor of 3× of optimal on a set of 6-loop lace-like patterns (`flat-lace`).

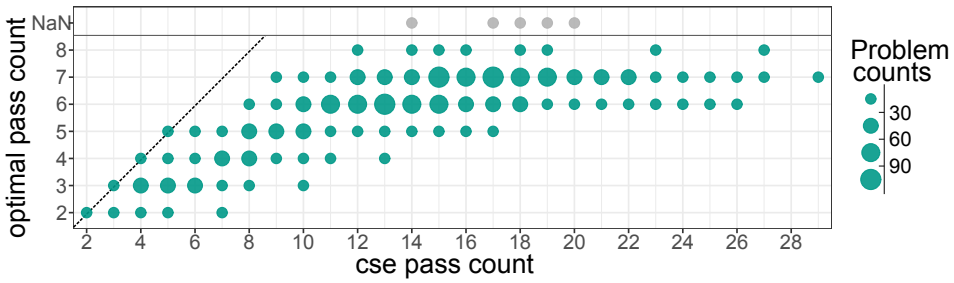


Figure 6.4: The Collapse-Shift-Expand (`cse`) algorithm [McCann et al. 2016] produces transfer plans that stray relatively far from optimal on a set of 8-loop shaped tube problems (`simple-tubes`).

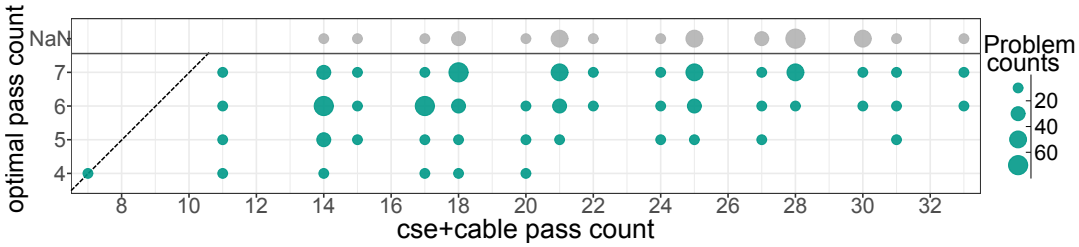


Figure 6.5: The ad-hoc strategy of concatenating cable and tube rotation plans (dataset `cable-tubes`) presents many opportunities for algorithmic optimization.

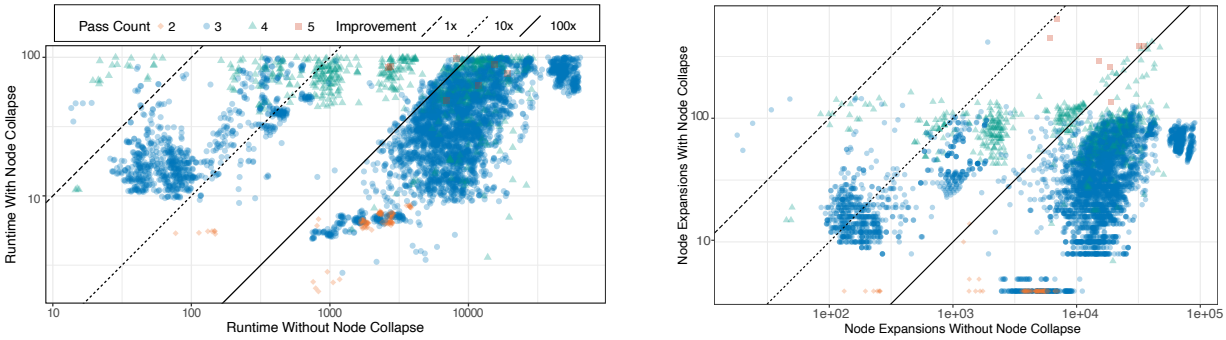


Figure 6.6: Our canonical-node optimization results in an approximately two-order-of-magnitude reduction in both the search runtime and in the number of nodes expanded in our tests on the `all-short` dataset.

Scenario	Nodes	Time (s)	Speedup
No Heuristic	471451589	582264.3	1×
+ Braid	281291965	330931.9	1.8×
+ Log Offsets	2500142	2612.2	223×
+ Braid+Log	1663664	1688.6	344×
+ Prebuilt	259985	274.1	2124×
+ Braid+Prebuilt	184265	189.8	3067×

Table 6.1: Total sum of nodes expanded and time taken for various combinations of heuristics over the 5002 problems that all heuristics finished. Heuristics were combined using $max(h_1, h_2)$.

Canonical Node We also examined the effect of canonicalizing on the time and memory requirements of the search on `all-short`, the subset of all datasets for which our search found a solution in $< 100ms$. As can be seen in Figure 6.6, the number of node expansions improves by a factor of 3165×, and runtime by 3828×, with larger problems experiencing more improvement. Of the few problems where canonical-node is slower (34 out of 5902 total problems), much of the slow down can be attributed to additional overhead from the canonicalize operation. We conjecture that the 10 problems where canonical-node expands more nodes is due to tie breaking between equally weighted states.

Heuristics In addition, we looked at the performance of various heuristics for the A* search (Table 6.1). Our combined offset table and braid word length heuristic provide a three-order-of-magnitude reduction in both memory usage and search time compared to using no heuristic, and a one-order-of-magnitude reduction compared to the simpler combined braid word length and log offset heuristic. Building the offset table took 5.2 seconds, making it a strict improvement for larger problems even without amortizing the time required to build the table across multiple problems. Furthermore, taking the maximum of braid word length (which only looks at the braid) and offset table (which only looks at loop positions) provides improvements over using either heuristic alone.

Chapter 7

Conclusion

In this thesis, I presented the first formal characterization of knit object equivalence that encompasses the complete domain of machine knitting programs. This result allows us to finally answer the fundamental question of whether two knitting programs are the same. This in turn enables many additional research directions on both the theory of machine knitting as well as the practice of developing systems and tools for machine knitting practitioners. For example, the program optimality discussed in Chapter 6 was only possible because program equivalence was well defined. Furthermore, I believe insights from how I developed this definition is not only useful for developing a more nuanced semantics of machine knitting, but can also guide formalizations for other computational fabrication techniques as well. I conclude with some immediate applications and extensions of this formalization, followed by some thoughts on formalization within the greater field of computational fabrication.

7.1 Practical Knit Programming Tools

There reamins much future work and interesting problems that must be addressed before the formalizations proposed in this work can be integrated into practical systems. For example, integrating the instruction graph into other automatic knitting compiler systems as an IR would ensure that existing compilers will also have a sound and complete lowering and scheduling process. As mentioned earlier, the instruction graph semantics only ensure topological correctness, which we believe is an essential property that all knitting compiler systems should possess. However, other knitting compiler systems might be concerned with additional properties, such as material properties. In such cases, a naive copy-paste of the instruction graph would be insufficient, and developers of those systems would need to augment the property specifications in the formalization of the instruction graph to accommodate their specific requirements.

Furthermore, while the proofs in this work provide a theoretical foundation for compiler verification, they are not mechanized. To create an end-to-end verified knitting compiler, it would be necessary to formalize the semantics and proofs using proof assistants such as Coq, Lean, or Agda. To the best of our knowledge, no existing knitting compilers are fully mechanized, making this an interesting avenue for future research. We believe that the work represents a step towards the development of fully verified knitting compilers, and we hope that it will inspire further advancements in this field.

The editors presented in this thesis also raise interesting questions as to what would be an appropriate user interface for guiding the compilation and optimization process. The current implementations of the knitout rewrite editor and instruction graph editor are fully manual, which makes scheduling and optimization a tedious task. An immediate question is how these transformations could be automated. However, this also raises the question of what operations shouldn't be fully automated in order to provide

more user control. An interesting point about instruction graphs is that while any deformation of the graph under ambient isotopy preserves object equivalence, this deformation must also be performed on the physical object, which may not always be feasible. A more interactive scheduler may be useful not only for building intuition about the automated knitting process, but also for reasoning about post-processing effort and material constraints.

7.2 Formalizing the Full Fabrication Pipeline

In this thesis, I examined two representations used within the machine knitting pipeline: knitout programs, which are a sequence of machine control operations, and instruction graphs, which are an intermediate representation of a knit object. These representations are only a slice of the full fabrication pipeline. Extending formal semantics to representations used in the design process and physical fabrication systems would allow for provable guarantees about the complete fabrication pipeline.

For example, while instruction graphs are useful for precisely describing the exact topology of a specific knit object, such level of detail may be excessive during the initial design phase. While these design specifications may be less precise, that does not make them less important for specifying the correctness of an object. For example, 2D surfaces are useful as a simplified representation of many knit objects, and a given designer might be uninterested in the precise stitch topology so long as the global topology is respected. These degrees of freedom may then be leveraged when generating an optimal machine knitting program. If these high-level designs can be formalized as a set of verifiable object properties, this would not only allow for verification of program generation process, but also improve the downstream fabrication process. In addition, a denotational semantics for design representations can also be used to characterize which programs a design system can create as well as which programs a system can distinguish between. Design systems must juggle expressibility and simplicity, and a formal definition of system expressibility could be used to better characterize this trade off.

In addition, the current operational semantics for knitting machine programs assumes an idealized knitting machine with idealized materials. In practice, knitting occurs on a physical machine with physical materials that may introduce all sorts of errors during the fabrication process. Different machines then have different ways of addressing these errors. For example, machines with sophisticated tension mechanisms may perform more drastic shaping operations, and elastic yarns more closely approximate the infinitely stretchy arcs in our definition of fenced tangle equivalence. While the UFO condition must necessarily be satisfied for any v-bed knitting machine, practical machine knitability is impacted by the specific machine architecture and material properties of the yarn.

7.3 Alternative Knitting Semantics

Throughout this thesis, I have focused on preserving topological properties when compiling and optimizing knitting machine programs. However, this is just one of the many properties of knit objects that are important. One that begs immediate attention from a practical fabrication standpoint is metric properties, or how much yarn is used to construct knit structures. An approach I've taken to provide some reasoning about metric properties is to essentially annotate regions of Instruction Graphs and fenced tangles with the length of yarn that should be in that region. However, a truly practical knit compiler should consider additional fabrication constraints, such as metric measurements, program efficiency, and program reliability (i.e., how likely an instruction set can be executed without error). Furthermore, allowing any transformation under ambient isotopy is likely too broad a set of transformations when searching the space of equivalent instruction graph embeddings, as those transformations must then be inverted in the real world.

Including metric properties as a first-class semantic property is the next big step in practical verification of the machine knitting pipeline.

In addition, there are additional topological properties that are not addressed by the work in this thesis. Stability, or whether a constructed loop remains a loop or unravels into a different structure. For example, the fenced tangle depicting two knit stitches in Fig. 3.3 actually will unravel into the single larger loop depicted by a regular tangle. Novices frequently create unstable structures by mistake, while experts will sometimes use controlled unraveling to adjust metric properties of a knit or to create support structures that increase fabrication reliability. Further more, stability is less a binary categorization and more a continuum along which certain structures require intervetion post-fabrication to overcome internal friction, while others will immediately unravel during the knitting process and result in cascading failures. Capturing this property will likely require a semantic function that is less generous with its placement of fences in the fenced tangle or a different semantic object entirely.

What's more, there are knitting techniques that cannot be captured using ambient isotopy on fenced tangles. Those familiar with machine knitting might be concerned that the forward property forbids the formation of twisted loops. This is demonstrably untrue, as twisted loops are a common technique/artifact in machine knitting. In fact, we can see them in the subtle seam of "spin" scheduling example in figure 7 of McCann et al. [2016]. The explanation for this is rather nuanced. First, twisted loops can be formed so long as there is a semantic preserving rewrite that untwists the loops so that the graph is in UFO presentation. This transformation is what causes the twisted loops in McCann et al. [2016]. However, there are additional techniques for twisting loops that are not captured by equivalence on fenced tangles. These techniques are a potential way to locally remove twists from ribbons, which is particularly interesting given that we can prove certain configurations of twisted ribbons and nodes can never be made forward using the existing definition of instruction graph equivalence. Subtle transformations like these raise additional questions on the correct mathematical definition for knit objects.

Finally, in Chapter 6 I examined how an alternative semantic function mapping to the Artin Braids allows for a more compact presentation as well as an efficient general solution to transfer plan equivalence. A natural followup question is whether this approach can be extended to all valid machine knitting programs. If we examine the fenced tangles denoted by the remaining knitout operations, however, we see that they include non-monotonic arcs as well as fences, which have no clear parallel in the Artin Braids. One useful observation is that knitting machine programs denote a subset of fenced tangles, where all arcs outside of fenced regions are monotonic. This suggests that much like how fenced tangles served as a useful extension of tangles for representing knitting, an extension of the Artin Braids may allow for a more computationally tractable abstraction of machine knitting.

7.4 Formalizing Fabrication At Large

The knot theory contributions in this work suggest several immediate applications in other textiles-adjacent fabrication methods. The translation between hand knitting instructions and output object are similarly opaque, and humans are both more dexterous and less repeatable than machines. Textile techinques like crochet, braiding, and solid knitting [Hirose et al. 2024] also involve the execution of a sequence of operations that deforms a strand of yarn into a specific topology that could be easily unravelled. This makes fenced tangles a natural semantic domain to consider for these fabrication methods. However, I believe that the real insight of these thesis lies with the question I raised back in the introduction: what does it mean for a fabrication process to have made the right thing?

Let us return to the toy example I raised, where we tried to define equivalence on the size of your favorite jacket. Now, imagine how we might evaluate other properties, such as its color, or texture, or the

way it feels on your body. While it is certainly possible to convert these properties into more measurable quantaties, these subtlties lead into our second problem: what are the desired properties that make an object distinct? Put another way, what properties can change while still preserving an object's identity? For example, we may take the viewpoint that all properties are important, but this quickly brings us to a situation where nothing is equivalent, not even an object compared with itself miliseconds into the future. After all, age is something we can measure. Taken to another extreme, we can insist that an object is always the same as itself, but this neglects any transformations that have happened to an object over time. If you loan me your jacket, and the next day I return it with a big ink stain on the front, is that the same jacket? What if instead I'd cut it into pieces? Sewn those pieces back together? Or even just washed it and used the wrong detergent? Not only are these qualities hard to quantify, they are highly situational. Depending on the events and people involved, there might be drastically different answers as to whether two objects are the same. In some ways, coming up with a single canonical definition of object equivalence may be impossible.

Yet it is precisely because object meaning is so nuanced varied that I believe that it is critical to develop precise, mathematical characterizations of object fabrication. Having a mathematical definition of object equivalence means we can precisely define the boundaries of what is and is not the same. We can examine the edge cases where this definition deviates from our intuition of the physical phenomenon. And we can precisely communicate how the entire fabrication pipeline, from input design, to machine control program, to final object, preserve or change the underlying meaning. While formal categorization of inherently subjective phenomenon will inevitably abstract away the nuances in how humans interact with the world around them, the precision of these abstract models provide us with useful insight. And that, dear reader, is important no matter how or what it is that you want to create.

Bibliography

Adams, C. (1994). *The Knot Book*. New York, NY: W.H. Freeman. ISBN: 9780821886137.

Aigner, R., Haberfellner, M. A., and Haller, M. (2022). "Spacer: knitting ready-made, tactile, and highly responsive spacer-fabric force sensors for continuous input". *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*. UIST '22. Bend, OR, USA: Association for Computing Machinery. ISBN: 9781450393201. DOI: 10.1145/3526113.3545694. URL: https://doi.org/10.1145/3526113.3545694.

Albaugh, L., Hudson, S., and Yao, L. (2019). "Digital fabrication of soft actuated objects by machine knitting". *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI '19. Glasgow, Scotland Uk: Association for Computing Machinery, pp. 1–13. ISBN: 9781450359702. DOI: 10.1145/3290605.3300414. URL: https://doi.org/10.1145/3290605.3300414.

Albaugh, L., McCann, J., Hudson, S. E., and Yao, L. (2021). "Engineering multifunctional spacer fabrics through machine knitting". *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. CHI '21. Yokohama, Japan: Association for Computing Machinery. ISBN: 9781450380966. DOI: 10.1145/3411764.3445564. URL: https://doi.org/10.1145/3411764.3445564.

Carnegie Mellon Textiles Lab (2024). *Knitout-frontend-js*. [Online]. Available from: https://github.com/textiles-lab/knitout-frontend-js.

Dehornoy, P. (2008). "Efficient solutions to the braid isotopy problem". *Discrete Applied Mathematics* 156.16. Applications of Algebra to Cryptography, pp. 3091–3112. ISSN: 0166-218X. DOI: https://doi.org/10.1016/j.dam.2007.12.009. URL: http://www.sciencedirect.com/science/article/pii/S0166218X08000437.

Doenges, R., Arashloo, M. T., Bautista, S., Chang, A., Ni, N., Parkinson, S., Peterson, R., Solko-Breslin, A., Xu, A., and Foster, N. (Jan. 2021). "Petr4: formal foundations for p4 data planes". *Proc. ACM Program. Lang.* 5.POPL. DOI: 10.1145/3434322. URL: https://doi.org/10.1145/3434322.

Epstein, D. B. A., Paterson, M. S., Cannon, J. W., Holt, D. F., Levy, S. V., and Thurston, W. P. (1992). *Word Processing in Groups*. Natick, MA, USA: A. K. Peters, Ltd. ISBN: 0867202440.

Felner, A., Korf, R. E., Meshulam, R., and Holte, R. C. (2007). "Compressed pattern databases". *Journal of Artificial Intelligence Research* 30, pp. 213–247.

Grishanov, S., Meshkov, V., and Omelchenko, A. (2009). "A topological study of textile structures. part i: an introduction to topological methods". *Textile Research Journal* 79.8, pp. 702–713.

Hagedorn, B., Lenfers, J., Koehler, T., Gorlatch, S., and Steuwer, M. (2020). *A language for describing optimization strategies*. arXiv: 2002.02268 [cs.PL].

Hirose, Y., Gillespie, M., Bonilla Fominaya, A. M., and McCann, J. (July 2024). "Solid knitting". *ACM Trans. Graph.* 43.4. DOI: 10.1145/3658123. URL: https://doi.org/10.1145/3658123.

Hofmann, M., Albaugh, L., Wang, T., Mankoff, J., and Hudson, S. E. (2023). "Knitscript: a domain-specific scripting language for advanced machine knitting". *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. UIST '23. San Francisco, CA, USA: Association for Computing Machinery. ISBN: 9798400701320. DOI: 10.1145/3586183.3606789. URL: https://doi.org/10.1145/3586183.3606789.

Igarashi, A., Pierce, B. C., and Wadler, P. (May 2001). "Featherweight java: a minimal core calculus for java and gj". *ACM Transactions on Programming Languages and Systems* 23.3, pp. 396–450. ISSN: 1558-4593. DOI: 10.1145/503502.503505. URL: http://dx.doi.org/10.1145/503502.503505.

Ikarashi, Y., Bernstein, G. L., Reinking, A., Genc, H., and Ragan-Kelley, J. (2022). "Exocompilation for productive programming of hardware accelerators". *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, pp. 703–718. ISBN: 9781450392655. DOI: 10.1145/3519939.3523446. URL: https://doi.org/10.1145/3519939.3523446.

Jones, B., Mei, Y., Zhao, H., Gotfrid, T., Mankoff, J., and Schulz, A. (Dec. 2021). "Computational design of knit templates". *ACM Trans. Graph.* 41.2. ISSN: 0730-0301. DOI: 10.1145/3488006. URL: https://doi.org/10.1145/3488006.

Kaspar, A., Makatura, L., and Matusik, W. (2019). "Knitting skeletons: a computer-aided design tool for shaping and patterning of knitted garments". *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. UIST '19. New Orleans, LA, USA: Association for Computing Machinery, pp. 53–65. ISBN: 9781450368162. DOI: 10.1145/3332165.3347879. URL: https://doi.org/10.1145/3332165.3347879.

Kaspar, A., Wu, K., Luo, Y., Makatura, L., and Matusik, W. (July 2021). "Knit sketching: from cut & sew patterns to machine-knit garments". *ACM Trans. Graph.* 40.4. ISSN: 0730-0301. DOI: 10.1145/3450626.3459752. URL: https://doi.org/10.1145/3450626.3459752.

Koenig, D. and Tsvietkova, A. (2021). "Np–hard problems naturally arising in knot theory". *Trans. Amer. Math. Soc. Ser. B* 8.15, pp. 420–441. ISSN: 2330-0000. DOI: 10.1090/btran/71.

Lattner, C. and Adve, V. (2004). "Llvm: a compilation framework for lifelong program analysis & transformation". *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. CGO '04. Palo Alto, California: IEEE Computer Society, p. 75. ISBN: 0769521029.

Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., and Zinenko, O. (2021). "Mlir: scaling compiler infrastructure for domain specific computation". *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '21. Virtual Event, Republic of Korea: IEEE Press, pp. 2–14. ISBN: 9781728186139. DOI: 10.1109/CGO51591.2021.9370308. URL: https://doi.org/10.1109/CGO51591.2021.9370308.

Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., and Ferdinand, C. (2016). "Compcert – a formally verified optimizing compiler". *ERTS 2016: Embedded Real Time Software and Systems*. Ed. by SEE. Toulouse, France: SEE.

Lin, J. and McCann, J. (2021). "An artin braid group representation of knitting machine state with applications to validation and optimization of fabrication plans". *2021 IEEE International Conference on Robotics and Automation (ICRA)*. New York, NY, USA: Institute of Electrical and Electronics Engineers, pp. 1147–1153. DOI: 10.1109/ICRA48506.2021.9562113.

Lin, J., Narayanan, V., and McCann, J. (2018). "Efficient transfer planning for flat knitting". *Proceedings of the 2nd ACM Symposium on Computational Fabrication*. SCF '18. Cambridge, Massachusetts: As-

sociation for Computing Machinery. ISBN: 9781450358545. DOI: 10.1145/3213512.3213515. URL: https://doi.org/10.1145/3213512.3213515.

Liu, Z., Han, X., Zhang, Y., Chen, X., Lai, Y.-K., Doubrovski, E. L., Whiting, E., and Wang, C. C. (2021). "Knitting 4d garments with elasticity controlled for body motion". *ACM Transactions on Graphics (TOG)* 40.4, pp. 1–16.

Markande, S. G. and Matsumoto, E. (2020). "Knotty knits are tangles in tori". *Proceedings of Bridges 2020: Mathematics, Art, Music, Architecture, Education, Culture*. Ed. by Yackel, C., Bosch, R., Torrence, E., and Fenyvesi, K. Phoenix, Arizona: Tessellations Publishing, pp. 103–112. ISBN: 978-1-938664-36-6. URL: http://archive.bridgesmathart.org/2020/bridges2020-103.html.

McCann, J. (2017). *The "knitout" (.k) file format*. [Online]. Available from: https://textiles-lab.github.io/knitout/knitout.html.

McCann, J., Albaugh, L., Narayanan, V., Grow, A., Matusik, W., Mankoff, J., and Hodgins, J. (July 2016). "A compiler for 3d machine knitting". *ACM Trans. Graph.* 35.4, 49:1–49:11.

Mitra, R., Makatura, L., Whiting, E., and Chien, E. (2023). "Helix-free stripes for knit graph design". *ACM SIGGRAPH 2023 Conference Proceedings*. SIGGRAPH '23. , Los Angeles, CA, USA, Association for Computing Machinery. ISBN: 9798400701597. DOI: 10.1145/3588432.3591564. URL: https://doi.org/10.1145/3588432.3591564.

Munkres, J. (2000). *Topology*. Featured Titles for Topology. Prentice Hall, Incorporated. ISBN: 9780131816299. URL: https://books.google.com/books?id=XjoZAQAAIAAJ.

Murasugi, K. and Kurpita, B. (2012). *A Study of Braids*. Mathematics and Its Applications. Springer Netherlands. ISBN: 9789401593199. URL: https://books.google.com/books?id=VLTnCAAAQBAJ.

Nader, G., Quek, Y. H., Chia, P. Z., Weeger, O., and Yeung, S.-K. (July 2021). "Knitkit: a flexible system for machine knitting of customizable textiles". *ACM Trans. Graph.* 40.4. ISSN: 0730-0301. DOI: 10.1145/3450626.3459790. URL: https://doi.org/10.1145/3450626.3459790.

Narayanan, V., Albaugh, L., Hodgins, J., Coros, S., and McCann, J. (Aug. 2018). "Automatic machine knitting of 3d meshes". *ACM Trans. Graph.* 37.3, 35:1–35:15.

Narayanan, V., Wu, K., Yuksel, C., and McCann, J. (July 2019). "Visual knitting machine programming". *ACM Trans. Graph.* 38.4. ISSN: 0730-0301. DOI: 10.1145/3306346.3322995. URL: https://doi.org/10.1145/3306346.3322995.

Ou, J., Oran, D., Haddad, D. D., Paradiso, J., and Ishii, H. (2019). "Sensorknit: architecting textile sensors with machine knitting". *3D Printing and Additive Manufacturing* 6.1, pp. 1–11.

Popescu, M., Rippmann, M., Liew, A., Reiter, L., Flatt, R. J., Mele, T. V., and Block, P. (2020). "Structural design, digital fabrication and construction of the cable-net and knitted formwork of the knitcandela concrete shell". *Structures* 31, pp. 1287–1299.

Popescu, M., Rippmann, M., Van Mele, T., and Block, P. (2018). "Automated generation of knit patterns for non-developable surfaces". *Humanizing Digital Reality*. Ed. by al., D. R. K. et. Singapore: Springer.

Qu, A. and James, D. L. (July 2021). "Fast linking numbers for topology verification of loopy structures". *ACM Trans. Graph.* 40.4. ISSN: 0730-0301. DOI: 10.1145/3450626.3459778. URL: https://doi.org/10.1145/3450626.3459778.

Sanchez, V., Mahadevan, K., Ohlson, G., Graule, M. A., Yuen, M. C., Teeple, C. B., Weaver, J. C., McCann, J., Bertoldi, K., and Wood, R. J. (2023). "3d knitting for pneumatic soft robotics". *Advanced Functional Materials* n/a.n/a.

Shima Seiki (2011). *Sds-one apex3*. [Online]. Available from: http://www.shimaseiki.com/product/design/sdsone_apex/flat/.

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

Stoll (2011). *M1plus pattern software*. [Online]. Available from: http://www.stoll.com/stoll_software_solutions_en_4/pattern_software_m1plus/3_1.

Twigg-Smith, H., Whiting, E., and Peek, N. (2024). "Knitscape: computational design and yarn-level simulation of slip and tuck colorwork knitting patterns". *Proceedings of the CHI Conference on Human Factors in Computing Systems*. CHI '24. , Honolulu, HI, USA, Association for Computing Machinery. ISBN: 9798400703300. DOI: 10.1145/3613904.3642799. URL: https://doi.org/10.1145/3613904.3642799.

Underwood, J. (2009). "The design of 3D shape knitted preforms". PhD thesis. Fashion and Textiles, RMIT University.

Wu, K., Tarini, M., Yuksel, C., Mccann, J., and Gao, X. (2021). "Wearable 3d machine knitting: automatic generation of shaped knit sheets to cover real-world objects". *IEEE Transactions on Visualization and Computer Graphics*.