Linear Logic: Project Report
Knitting Math Formalized

Chelsea Battell
7899250

Abstract

This work presents the formalization and encoding of a fragment of the language of knitting patterns in linear logic in such a way that pattern correctness can be proven. Motivating this work is the observation that errors in a knitting pattern can be exceedingly frustrating for a knitter working from it and may cause a large amount of time to be lost because the existence of the error will likely not be realized before reaching that point in the pattern. Other than a course project by the author, there does not appear to be any work attempting to achieve the goals proposed here. The problem will be reduced to a variant of a Blocks World problem for a simplified analysis before being mapped back to the knitting application area. The encoding will not be implemented as part of this work since that is out of scope for the available timeline, but the theoretical construction will direct implementation, given an appropriate linear logic programming language. Encoding knitting patterns and their resource consumption in linear logic so that correctness can be verified results in a system that can help reduce stress and time loss in an otherwise psychologically beneficial hobby, leading to positive social outcomes.

1 Introduction and Background

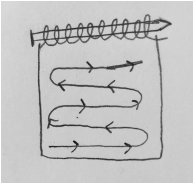
Knitting is a relaxing activity resulting in the production of cloth and garments that has been practiced for over 650 years [11, p.9]. Numerous benefits of knitting have been found, including self-reported increased relaxation, perceived happiness and reduced stress [10], a reduction in anxious preoccupations related to eating disorders [6], relief from compassion fatigue in nurses [2], and it can be useful for stress relief in both individual and group therapy [12].

All of the studies mentioned above refer to hand knitting, where one knitting needle is held in each hand. The fabric is constructed by working new loops into the existing fabric, passing the stitches from one needle to the other. We will briefly describe this process later, but first we need to consider the alternate major knitting form: machine knitting.

Knitting machines are mainly used in industrial settings but are becoming increasingly available for domestic use. They construct fabric far more quickly than by hand and are interesting in their own right, but they are not known to provide the same benefits listed above. In addition, the process for constructing the fabric is sufficiently different from hand knitting that the work described here does not apply to knitting machines. So whenever we talk about knitting without prefacing this with “hand” or “machine”, we are referring to hand knitting.

1.1 Knitting Basics

Before going further, it is helpful for the reader to have a basic idea of how knitting works. Knit fabric is formed by making interconnected loops from a length of yarn, working back and forth along rows.



To begin, we assume there is a way to get the first row on the needles; there are many techniques, but their description is tangential to the theory described in this work. This initial row looks like a row of loops on one needle with trailing yarn connecting to the yarn source (ball of yarn). This needle is held in the left hand.

To create a *stitch*, the needle held in the right hand is put through one (or more) loops on the left needle and the yarn is wrapped around the tip of the right needle; the different options for direction through the loop and yarn wrapping techniques result in a multitude of possible new stitches. As an example, steps in making a knit stitch are shown in Figure 1

Figure 1: Steps to make a knit stitch (code `K`)

Next, the tip of the right needle is pulled through the loop(s) that the right needle originally travelled through. Finally, the loops on the left needle that the right needle travelled through are dropped off the left needle because they are now supported by a loop on the right needle. The impact of this operation is that the left needle has fewer loops and the right needle has more. Once there are no loops remaining on the left needle, the needles switch hands, and the next *row* begins.

We see that the main elements we consider in creating a knit project are *stitches* and *rows*, where stitches are the content of a row. Given this introduction to the process of knitting, we can move on to the aspect of knitting that we wish to improve with this work. This is related to recording instructions for knitting, typically called a *knitting pattern*.

1.2 Knitting Patterns

When instructions to knit an object become complex, it is no longer practical to hold them in memory. Knitting patterns preserve instructions but also communicate them across time and space. A complicated knitting project such as an intricate lace shawl can be a rewarding project to complete, but designing or modifying its pattern can be very challenging. From the knitters perspective, spending a substantial amount of time knitting a pattern before discovering that the pattern has an error that is difficult to remedy is an extremely frustrating experience. What we wish to accomplish here is to formalize a method for determining if a pattern is correct.

The language used for knitting patterns is pseudo-formalized; the Craft Yarn Council has a document recommending how to use knitting pattern notation [7]. There is notation for each kind of stitch and for operations on stitches, such as sequences and repeats. We wish to push these standards to a new level of formalization and use logic tools to reason about knitting patterns.

Pattern correctness can be checked manually by computing how many stitches the pattern uses for each row and comparing this to the available stitches at the beginning of the row. Performing these computations manually is very tedious and error prone, and is an ideal task for computer assistance. Given such a tool, a knitter could check if a pattern is correct with minimal knowledge of how to manually perform the same computations. There is less cognitive overhead for an enterprising new knitter to experiment with modifying or designing patterns. For more experienced designers, it would help to eliminate uninteresting and error-prone tasks, allowing more focus and time to be spent thinking about the final product.

2 Linear Logic

Knitting is naturally resource-dependent; you can't make a stitch if there are no more loops on a needle. Similarly, once a loop on a needle is used, it shouldn't be possible to consume that resource again. We wish to articulate the rules for constructing knitting patterns in such a way that we are also properly considering "available loops" as our resources as we break down a pattern to analyze it. Here we begin to see that linear logic may be an ideal formalism for the rules of knitting patterns.

We will look at the focused form of the linear logic sequent calculus presented by Frank Pfenning in his course at Carnegie Mellon University [9]. Using a focused logic reduces nondeterminism in proof search because we choose to focus on a formula in the linear context. This reduces the search space, which is helpful on a path to implementing the work seen later. See Figure 2 for the focused linear logic rules that are used in this work. The reader is directed to [9] for a more thorough presentation of linear logic.

$$\frac{}{\Gamma; [P^-] \vdash P^-}\ id_{P^-}$$

$$\frac{\Gamma; \Delta \vdash A^-}{\Gamma; \Delta \vdash [A^-]}\ blur_R$$

$$\frac{\Gamma; \Delta \vdash [A] \quad \Gamma; \Delta', [B] \vdash C}{\Gamma; \Delta, \Delta', [A \multimap B] \vdash C}\ \multimap_L$$

$$\frac{\Psi \vdash M : \tau \quad \Psi; \Gamma; \Delta, [A\{M/x\}] \vdash C}{\Psi; \Gamma; \Delta, [\forall x : \tau.A] \vdash C}\ \forall_L$$

$$\frac{\Gamma; \Delta, [A^-] \vdash C}{\Gamma; \Delta, A^- \vdash C}\ focus_L$$

$$\frac{A \in \Gamma \quad \Gamma; \Delta, A \vdash C}{\Gamma; \Delta \vdash C}\ copy$$

$$\frac{\Psi \vdash M : \tau \quad \Psi; \Gamma; \Delta \vdash [A\{M/x\}]}{\Psi; \Gamma; \Delta \vdash [\exists x : \tau.A]}\ \exists_R$$

Figure 2: Focused linear logic rules relevant to this project

The polarity of all predicates seen later will be negative. This is because our proofs will use backchaining. Proofs will be explained in a bottom-up fashion and this direction should be the default reading in this work whenever it is not explicitly stated. This bias is also evident in the names of the rules in Figure 2.

Two other notes are necessary regarding our later use of this logic. We assume that we have natural numbers and polymorphic lists available to us with the standard operators, syntax, and notations for lists. Types are assumed by our use of the $\forall_L$ and $\exists_R$ rules. We do not discuss the type theory in depth here but assume the existence of natural numbers, base types (as required) and the type of lists.

3 Related Literature

In [5], the author makes a web application to compile a concise version of the language of knitting patterns. Resource use is managed using imperative programming techniques to check correctness. Here we wish to formalize a more sound theory for verifying pattern correctness.

In [4], the author makes a GUI editor for easier construction of knitting pattern objects. A tool serving this purpose is prescribed in the future work (see Section 6); such a tool should also support the logical foundations presented here.

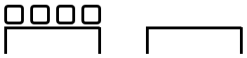
Our encoding of the language of knitting patterns is inspired by the object logics in case studies for the two-level logical frameworks Hybrid [8] and Abella [3]. Both systems can be used to encode logics or languages within a higher-order logic that can also be the foundation of a logic programming language. In [13], the authors show how to express a relational specification between two representations of λ-terms. In [8] we see an example object logic which is a fragment of MiniML.

4 Methods and Digression: A Blocks World Variant

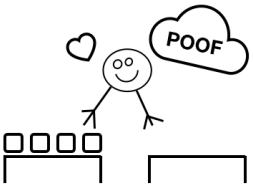
On our path to formalizing knitting patterns and showing that they are correct, we take a digression to explore a simplified yet conceptually similar problem.

Imagine two tables.

Initially only the table on the left has blocks on it.



There is a machine that processes the blocks on the left table and adds blocks to the right table.



> **Task 1:** Given a language of operations that the machine can perform and an initial count of blocks on the left table, determine if a given sequence of such operations is valid and clears the left table.

If we consider proof search, we can seek to solve an alternative task.

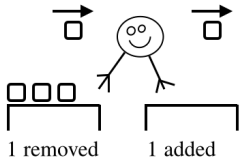
> **Task 2:** Given a sequence of operations, determine the initial and final block counts such that the left table is cleared.

In task 2 these unknown values can be thought of as existentially quantified metavariables that can be solved during proof search.

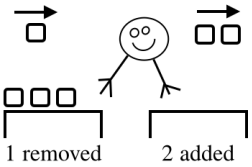
4.1 Blocks World Moves

Operations that move blocks from the left table to the right are called *moves*. A move that clears the left table is considered *valid*; we will see later that this subsumes sequences of moves. The base moves are `SAME`, `INC`, and `DEC`.

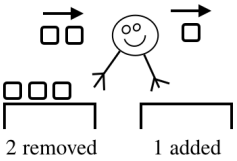
`SAME` is a move that takes one block from the left and turns it into one block on the right. The result is one fewer block on the left and one more on the right.



`INC` is a move that takes one block from the left table and turns it into two blocks on the right. The result is one fewer block on the left and two more on the right. `INC` means increase.



`DEC` is a move that takes two blocks from the left and turns them into one new block on the right. The result is two fewer blocks on the left and one more on the right. `DEC` means decrease.



We will write a predicate to contain a move and its update to the state of the tables.

> **Note:** *move*(m, x, y) means move m removes x blocks from the left table and adds y blocks to the right table.

So we have the following three facts:

move(`SAME`, 1, 1)
move(`INC`, 1, 2)
move(`DEC`, 2, 1)

There are also two compound moves. These are `SEQ` and `REP` meaning *sequence* and *repeat*, respectively. `SEQ` operates on two moves. `SEQ`(m_1, m_2) means complete move m_1 then move m_2. The state change caused by this sequence depends on the state change caused by each of m_1 and m_2. For example, `SEQ`(`INC`, `SAME`) means complete

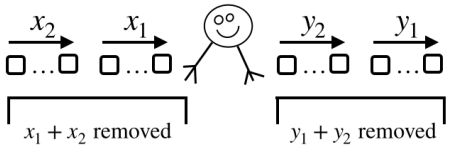
move INC then move SAME. Here INC removes one block from the left and adds two to the right, then SAME removes one block from the left and adds one to the right. The total removed from the left is two and the total added to the right is three. We can write this as a formula:

$$\begin{aligned} &\mathit{move}(\texttt{INC}, 1, 2) \multimap \\ &\mathit{move}(\texttt{SAME}, 1, 1) \multimap \\ &\mathit{move}(\texttt{SEQ}(\texttt{INC}, \texttt{SAME}), 2, 3) \end{aligned}$$

More generally, if we show $\mathit{move}(m_1, x_1, y_1)$ and $\mathit{move}(m_2, x_2, y_2)$, then we get $\mathit{move}(\texttt{SEQ}(m_1, m_2), x_1 + x_2, y_1 + y_2)$. As an inference rule, this says

$$\frac{\mathit{move}(m_1, x_1, y_1) \qquad \mathit{move}(m_2, x_2, y_2)}{\mathit{move}(\texttt{SEQ}(m_1, m_2), x_1 + x_2, y_1 + y_2)}\ \mathit{move\_seq}$$

This rule can be visualized as

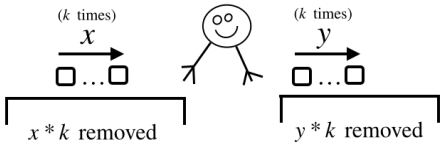


Note that we can clear any table with a single move that is a nesting of sequence moves, assuming the base moves have appropriate state change effects with respect to the initial count.

REP operates on one move and a natural number. $\texttt{REP}(m, k)$ means complete move m, repeated k times. Similar to SEQ, the state change caused by the repeat depends on the state change caused by m. If we can show $\mathit{move}(m, x, y)$, then we get $\mathit{move}(\texttt{REP}(m, k), x * k, y * k)$. We can write this as an inference rule As

$$\frac{\mathit{move}(m, x, y)}{\mathit{move}(\texttt{REP}(m, k), x * k, y * k)}\ \mathit{move\_rep}$$

A visualization of this rule is



From the discussion above, we can see that our language can be represented by the following grammar, where k is a natural number:

$$\begin{aligned} \mathit{RobotMove}\ m ::= \ & \texttt{SAME} \mid \\ & \texttt{INC} \mid \\ & \texttt{DEC} \mid \\ & \texttt{SEQ}(m_1, m_2) \mid \\ & \texttt{REP}(m, k) \end{aligned}$$

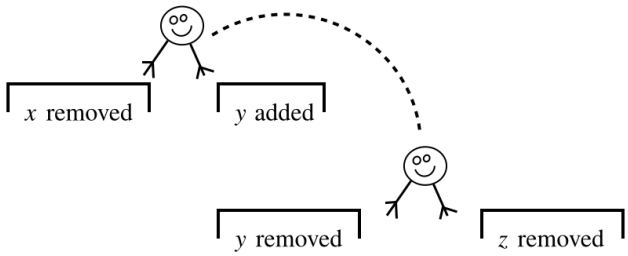
Notice that for any move we can always find values for the second and third arguments of the move predicate when they are unknown so that the move is valid (i.e. clears the left table). Recall that these arguments represent the change to the counts on the left and right tables, respectively, caused by the move. This is because we eventually break down to base moves SAME, INC, and DEC and we can propagate their second and third arguments to compute state change values in the rest of the proof. In this situation, the required needle state for the operations to be possible is implied by the second and third arguments to the move predicate.

4.2 Blocks World Program

We introduce a new predicate called *program* that is over a list of moves, a natural number representing the initial block count on the left table, and a final block count on the right table.

> **Note:** $program(p, x, y)$ means program p removes x blocks from the leftmost table and adds y blocks to the rightmost table, with all intermediate tables cleared in order by the moves in p.

A valid program is a list of moves such that the number of blocks on the right table after running a move in the program is the same as the number of blocks on the left table at the start of the next move in the program. We can visualize this as a (cascading) list of pairs of tables, where after completing a move, what was the right table for the current move now becomes the left table of the next move. A visualization of this process is:



As a rule, this says

$$\frac{move(head, x, y) \qquad program(tail, y, z)}{program(head :: tail, x, z)}\ prog\_check$$

where we use the standard syntax for lists, namely :: for concatenating an element to the left of a list and [ ] for the empty list. We also allow lists to be written between brackets and separated by commas (e.g. $[a, b, c]$ rather than $a :: b :: c :: [\,]$) whenever there is no ambiguity or overloading.

The program we will be breaking down in the goal is *head* :: *tail*, where *head* is a move and *tail* is a list of moves (i.e. a program). The remaining arguments x, y, and z represent the blocks on the leftmost table before the move *head*, the number of blocks on the table in the transition from *head* to *tail*, and the number of blocks on the final table after running the program, respectively.

We also need to consider how to prove $program([\,], x, z)$. An empty program is provable for any value such that x is equal to z. We can write this in a rule as:

$$\frac{}{program([\,], k, k)}\ prog\_empty$$

So far we have presented the rules of this application as inference rules but without assuming any other logical structure. What is desired is an encoding in linear logic.

4.3 Linear Logic Encoding

Toward encoding this application in linear logic, formulas in Figure 3 will be the initial contents of the persistent context, Γ. These are the linear logic formula representations of the rules introduced above. Note that wherever we have the universal quantifier $\forall$, we need to provide a type for the variable. The type system of this application is not explored in any detail but we assume we have natural numbers, lists, and base types representing moves, called `mv_tp`. Programs are lists of moves so their type is `list mv_tp`.

$$
\begin{array}{rl}
(\mathit{move\_same}) & \mathit{move}(\mathtt{SAME}, 1, 1) \\
(\mathit{move\_inc}) & \mathit{move}(\mathtt{INC}, 1, 2) \\
(\mathit{move\_dec}) & \mathit{move}(\mathtt{DEC}, 2, 1) \\
(\mathit{move\_seq}) & \forall m_1, m_2 : \mathit{mv\_tp}, \forall x_1, x_2, y_1, y_2 : \mathit{nat}\,. \\
 & \quad \mathit{move}(m_1, x_1, y_1) \multimap \mathit{move}(m_2, x_2, y_2) \multimap \mathit{move}(\mathtt{SEQ}(m_1, m_2), x_1 + x_2, y_1 + y_2) \\
(\mathit{move\_rep}) & \forall m : \mathit{mv\_tp}, \forall x, y, k : \mathit{nat}\,. \mathit{move}(m, x, y) \multimap \mathit{move}(\mathtt{REP}(m, k), x * k, y * k) \\
(\mathit{prog\_check}) & \forall \mathit{head} : \mathit{mv\_tp}, \forall \mathit{tail} : \mathit{list}\ \mathit{mv\_tp}, \forall x, y, z : \mathit{nat}\,. \\
 & \quad \mathit{move}(\mathit{head}, x, y) \multimap \mathit{program}(\mathit{tail}, y, z) \multimap \mathit{program}(\mathit{head} :: \mathit{tail}, x, z) \\
(\mathit{prog\_empty}) & \forall k : \mathit{nat}\,. \mathit{program}([\,], k, k)
\end{array}
$$

Figure 3: Blocks World persistent context, Γ

We will now look at proving sequents for each possible program construct as the goal. These proofs can then be used as derived rules in larger proofs. Note that we use the same name for a formula and its corresponding derived rule, understanding that it is clear from context of its use which we mean.

Deriving sequents with any of the first three formulas as a goal follow a very similar process. Proving bottom-up, we copy the formula into the linear context, then focus it and finish with an identity rule. For *move_same* this proof is:

$$
\dfrac{\dfrac{\dfrac{}{\Gamma; [\mathit{move\_same}] \vdash \mathit{move}(\mathtt{SAME}, 1, 1)}\, id_{P^-}}{\Gamma; \mathit{move\_same} \vdash \mathit{move}(\mathtt{SAME}, 1, 1)}\, \mathit{focus}_L}{\Gamma \vdash \mathit{move}(\mathtt{SAME}, 1, 1)}\, \mathit{copy}
$$

So the derived rules for SAME, INC, and DEC are

$$
\dfrac{}{\Gamma \vdash \mathit{move}(\mathtt{SAME}, 1, 1)}\, \mathit{move\_same}
$$

$$
\dfrac{}{\Gamma \vdash \mathit{move}(\mathtt{INC}, 1, 2)}\, \mathit{move\_inc}
$$

$$
\dfrac{}{\Gamma \vdash \mathit{move}(\mathtt{DEC}, 2, 1)}\, \mathit{move\_dec}
$$

where we use the dashed horizontal line to indicate that the rule is derived.

Next we will consider the derived rules for the formulas for the move connectives SEQ and REP. To save space in the derivations of these rules, we elide the typing judgments and silently instantiate universally quantified variables. So it is not shown explicitly that variables must have the correct type, but we understand that this must be the case.

To derive a rule for the *move_seq* formula, we begin by copying the formula from the persistent context and focusing. We don't show the five applications of the $\forall_L$ rule but move on to show we need to use $\multimap_L$ twice. We blur any negative atoms on the right of sequents to get premises of the derived rule and complete any branches that we can with the *id* rule. This results in the derivation below:

$$
\dfrac{\dfrac{\dfrac{\dfrac{\Gamma; \Delta_1 \vdash \mathit{move}(m_1, x_1, y_1)}{\Gamma; \Delta_1 \vdash [\mathit{move}(m_1, x_1, y_1)]}\, \mathit{blur}_R \quad \dfrac{\Gamma; \Delta_2 \vdash \mathit{move}(m_2, x_2, y_2)}{\Gamma; \Delta_2 \vdash [\mathit{move}(m_2, x_2, y_2)]}\, \mathit{blur}_R \quad \vdots}{\Gamma; \Delta_1, \Delta_2, [\mathit{move\_seq}] \vdash \mathit{move}(\mathtt{SEQ}(m_1, m_2), x_1 + x_2, y_1 + y_2)}\, \multimap_L \text{ (twice)}}{\Gamma; \Delta_1, \Delta_2, \mathit{move\_seq} \vdash \mathit{move}(\mathtt{SEQ}(m_1, m_2), x_1 + x_2, y_1 + y_2)}\, \mathit{focus}_L}{\Gamma; \Delta_1, \Delta_2 \vdash \mathit{move}(\mathtt{SEQ}(m_1, m_2), x_1 + x_2, y_1 + y_2)}\, \mathit{copy}
$$

where the missing right branch is completed with

$$\frac{}{\Gamma; [move(\texttt{SEQ}(m_1, m_2), x_1 + x_2, y_1 + y_2)] \vdash move(\texttt{SEQ}(m_1, m_2), x_1 + x_2, y_1 + y_2)}\ id_{P^-}$$

So the derived rule for *move_seq* is

$$\frac{\Gamma; \Delta_1 \vdash move(m_1, x_1, y_1) \qquad \Gamma; \Delta_2 \vdash move(m_2, x_2, y_2)}{\Gamma; \Delta_1, \Delta_2 \vdash move(\texttt{SEQ}(m_1, m_2), x_1 + x_2, y_1 + y_2)}\ move\_seq$$

To derive a rule for the *move_rep* formula, we again silently instantiate universally quantified variables and begin by copying and focusing the desired formula. Next $\multimap_L$ is applied once, followed by $blur_R$ on the first premise to get a premise of the derived rule and id_{P^-} on the second to complete that branch. We get the derivation below:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma; \Delta \vdash move(m, x, y)}{\Gamma; \Delta \vdash [move(m, x, y)]}\ blur_R \qquad \dfrac{}{\Gamma; [move(\texttt{REP}(m, k), x * k, y * k)] \vdash move(\texttt{REP}(m, k), x * k, y * k)}\ id_{P^-}}{\Gamma; \Delta, [move\_rep] \vdash move(\texttt{REP}(m, k), x * k, y * k)}\ \multimap_L}{\Gamma; \Delta, move\_rep \vdash move(\texttt{REP}(m, k), x * k, y * k)}\ focus_L}{\Gamma; \Delta \vdash move(\texttt{REP}(m, k), x * k, y * k)}\ copy$$

So the derived rules for *move_rep* is

$$\frac{\Gamma; \Delta \vdash move(m, x, y)}{\Gamma; \Delta \vdash move(\texttt{REP}(m, k), x * k, y * k)}\ move\_rep$$

We wrote our formulas using linear connectives, but so far the application has not required linear logic. We could have used intuitionistic implication and intuitionistic logic and would have derived very similar rules, other than new facts in the context in some cases. This is because all of the *move* predicates are facts that could be allowed to persist. When we begin to consider table state or traverse a program, then we need linear logic.

The final two formulas we need to consider are the ones we use to show that a program is correct with respect to initial and final block counts. Now it is important to only have one move predicate available in the linear context at any point in the proof, because we need to know the most recent table state according to the *move* predicate.

For the nonempty program case, we begin as in the derived rule proofs above by copying the formula of interest from the persistent context into the linear context and focusing on it. Then apply $\multimap_L$ and finish branches either with $blur_L$, leaving a premise of the derived rule, or use the identity rule to finish the branch. We see the derivation below:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma; \Delta_1 \vdash move(head, x, y)}{\Gamma; \Delta_1 \vdash [move(head, x, y)]}\ blur_R \quad \dfrac{\Gamma; \Delta_2 \vdash program(tail, y, z)}{\Gamma; \Delta_2 \vdash [program(tail, y, z)]}\ blur_R \quad \vdots}{\Gamma; \Delta_1, \Delta_2, [prog\_check] \vdash program(head :: tail, x, z)}\ \multimap_L \text{ (twice)}}{\Gamma; \Delta_1, \Delta_2, prog\_check \vdash program(head :: tail, x, z)}\ focus_L}{\Gamma; \Delta_1, \Delta_2 \vdash program(head :: tail, x, z)}\ copy$$

where the rightmost branch is completed with

$$\frac{}{\Gamma; [program(head :: tail, x, z)] \vdash program(head :: tail, x, z)}\ id_{P^-}$$

If we instead derived this rule in intuitionistic logic with intuitionistic implication, then we would end up with multiple *program* formulas in the context, possibly being able to complete the proof with the incorrect formula choice, suggesting an erroneous program is valid.

The derived rule is then

$$\frac{\Gamma; \Delta_1 \vdash move(head, x, y) \qquad \Gamma; \Delta_2 \vdash program(tail, y, z)}{\Gamma; \Delta_1, \Delta_2 \vdash program(head :: tail, x, y)}\ prog\_check$$

For the empty program case, we again silently instantiate the universal quantifier in *prog_empty* and get the following proof:

$$\dfrac{\dfrac{\dfrac{}{\Gamma; \Delta, [prog\_empty] \vdash program([\,], k, k)}\, id_{P^-}}{\Gamma; \Delta, prog\_empty \vdash program([\,], k, k)}\, focus_L}{\Gamma; \Delta \vdash program([\,], k, k)}\, copy$$

which suggests derived rule

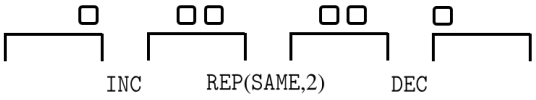
$$\dfrac{}{\Gamma; \Delta \vdash program([\,], k, k)}\, prog\_empty$$

4.4 Example

Let $p = \texttt{INC} :: \texttt{REP}(\texttt{SAME}, 2) :: \texttt{DEC} :: [\,]$. To illustrate an example, we will show how to prove that there are natural numbers X and Y representing the initial and final table states, respectively, such that the program p is valid. That is, we wish to prove the following:

$$\dfrac{\vdots}{\Gamma; \cdot \vdash \exists X, Y : nat\,.\, program(p, X, Y)}$$

Before working through the proof, the program is small enough that we can build a visualization of it:



Working bottom up, we begin with two applications of $\exists_R$ to instantiate new variables for the unknown values. Then the only rule we can apply is *prog_check*, which allows us to separately consider the head and the tail of the program.

$$\dfrac{\dfrac{\dfrac{}{\Gamma; \cdot \vdash move(\texttt{INC}, 1, 2)}\, move\_inc \qquad \dfrac{\vdots}{\Gamma; \cdot \vdash program(\texttt{REP}(\texttt{SAME}, 2) :: \texttt{DEC} :: [\,], 2, 1)}}{\Gamma; \cdot \vdash program(\texttt{INC} :: \texttt{REP}(\texttt{SAME}, 2) :: \texttt{DEC} :: [\,], 1, 1)}\, prog\_check}{\Gamma; \cdot \vdash \exists X, Y : nat\,.\, program(\texttt{INC} :: \texttt{REP}(\texttt{SAME}, 2) :: \texttt{DEC} :: [\,], X, Y)}\, \exists_R \text{ (twice)}$$

Since the program is built by appending three moves to the empty list, we apply *prog_check* two more times. Our choice of rules for the remainder of the proof is also deterministic. After breaking down the program with *prog_check*, the appropriate *move* rule (e.g. *move_same*, *move_rep*, etc) is chosen according to the first argument to *move*.

$$\dfrac{\dfrac{\dfrac{}{\Gamma; \cdot \vdash move(\texttt{SAME}, 1, 1)}\, move\_same}{\Gamma; \cdot \vdash move(\texttt{REP}(\texttt{SAME}, 2), 2, 2)}\, move\_rep \qquad \dfrac{\dfrac{}{\Gamma; \cdot \vdash move(\texttt{DEC}, 2, 1)}\, move\_dec \qquad \dfrac{}{\Gamma; \cdot \vdash program([\,], 1, 1)}\, prog\_empty}{\Gamma; \cdot \vdash program(\texttt{DEC} :: [\,], 2, 1)}\, prog\_check}{\Gamma; \cdot \vdash program(\texttt{REP}(\texttt{SAME}, 2) :: \texttt{DEC} :: [\,], 2, 1)}\, prog\_check$$

Above we have seen how to prove that programs in this Blocks World example are valid. Notice that by using derived rules, we have almost entirely removed the need to use linear logic rules for this application, as they are embedded in the derivations of the derived rules. These derived rules act like modules in the larger proofs. The example program is quite small but is sufficient to illustrate the proof search process. We will now move to the objective application area of this work.

5 Formalizing Knitting Patterns

At this point we can bring our focus back to the knitting application. We will map the language and terminology of the Blocks World problem of Section 4 to the language of knitting patterns.

Recall from Section 1 that the process of knitting works as follows: there are some loops on the left needle which are processed to make loops on the right needle. Once all loops on the left needle have been processed, the needles switch hands and the next row begins; what was added to the right needle in the last row is the count on the new left needle. This is reminiscent of the construction we saw in the last section. What were moves in Section 4 are now called *stitches* representing the content of *rows* and the programs there are now called *patterns*.

The basic stitches of knitting can be broken up into three groups: increases, decreases, and stitches that don't change the count from left to right. Most of these stitches match up to one of `INC`, `DEC`, or `SAME` from the previous section, but there are a few more available in knitting. Many of these basic stitches are shown in Figure 4, but we acknowledge that there are other stitches that we have knowingly excluded and this language can be modified by the addition of other new stitches.

Stitches that remove one from the left and add one to the right include *knit*, *purl*, and *slip*, notated `K`, `P`, and S, respectively. Increases that add more to the right than they remove from the left include *make-one*, *yarnover*, and *knit-front-back*, notated `M1`, `YO`, and `KFB`, respectively. Decreases that add fewer to the right than they remove from the left include *knit-slip-pass*, *knit-together*, and *purl-together*, notated `KSP`, `K2T`, and `P2T`, respectively. Both *knit-together* and *purl-together* can be functions of a natural number, where any number of stitches can be knit or purled together (when physically possible). We constrain ourselves to 2 here to avoid extra consideration of function arguments to *move*. The *move* formulas for each of these new stitches are listed in Figure 4.

Same count:	*move*(`K`, 1, 1)	*move*(`P`, 1, 1)	*move*(`S`, 1, 1)
Increases:	*move*(`M1`, 0, 1)	*move*(`YO`, 0, 1)	*move*(`KFB`, 1, 2)
Decreases:	*move*(`KSP`, 2, 1)	*move*(`K2T`, 2, 1)	*move*(`P2T`, 2, 1)

Figure 4: Knitting base stitch *move* predicates (incomplete list)

Notice that there are increases and decreases with different values than indicated in the blocks world example. This is fine because these cases are at the leaves of a proof and the only difference will be in the final values computed for the counts on the tables in the rest of the proof. Proof search dynamics are unchanged. It is still the case that for increases more is added to the right than taken from the left and for decreases more are taken from the left than added to the right.

The choice of which increase or decrease to use can depend on how many stitches still need to be processed. For example, `YO` creates a stitch on the right without using any from the left, while `KFB` uses one on the left to create two on the right. Yet it is most often the case that the choice of increase or decrease is an aesthetic decision and has less to do with the resource consumption of the operation. For example, `YO` creates a hole and is useful in making lace, and `KFB` maintains a full closed fabric while still allowing dynamic shape changes in the cloth.

Cables could also be considered base stitches because they have codes which can be associated with a count removed from the left needle and a count added to the right needle. These counts are usually the same in the case of cables, although the positions of the stitches are permuted.

Knitting patterns also allow the `SEQ` and `REP` operators as they are defined in Section 4. A reader with knowledge of this application area might observe that there is a very useful construct that is very commonly used in knitting patterns but is missing here. This construct might be called something like an *undetermined repeat* and instructs the knitter to knit a stitch as many times as possible to the last x stitches in the row. For example, applying this operator to `K`, the knit stitch, *to the last two stitches*, means make a knit stitch as many times as possible but leave the last two stitches on the left needle. Patterns using this construct can be written far more efficiently than without it. Unfortunately it requires extra reasoning to figure out when it is used correctly. Another important point is that its nested use must be restricted to very specific conditions. For these reasons its inclusion and analysis is relegated to future work.

Now we can show an example of a knitting pattern and know how to read it. The example below is the start of the classic "Grandma's Favourite Dishcloth" [1] (designer unknown). Figure 5 shows the pattern in the syntax of

Section 4. Patterns in the standard syntax suggested by the Craft Yarn Council [7] have some extra annotations to make them easier to read, rather than only including the operations. These include "row" at the start of every row and a period at the end to indicate the end of the row. Some patterns also write the row number and loop counts for each row. Figure 6 shows the same pattern in this standard pattern notation.

```
REP(K, 4) ::
SEQ(REP(K, 2), SEQ(YO, REP(K, 2)) ::
SEQ(K, 5) ::
SEQ(REP(K, 2), SEQ(YO, REP(K, 3)) ::
SEQ(K, 6) ::
SEQ(REP(K, 2), SEQ(YO, REP(K, 4)) ::
SEQ(K, 7) ::
...
```

Figure 5: First rows of a cloth knitting pattern

Notice that the syntax of the pattern in Figure 6 also uses easier to read notation for sequences and repeats: elements of a sequence are separated by a comma and repeats are written with the repeat count appended to the stitch code. When appending the repeat count to a stitch a pattern author must be careful and put the stitch code in parentheses if necessary to eliminate ambiguity. For example, it is clear that $K2$ means K twice, but to repeat K, P twice we must write $(K, P)\,2$.

row: K4.
row: K2, YO, K2.
row: K5.
row: K2, YO, K3.
row: K6.
row: K2, YO, K4.
row: K7.
...

Figure 6: First rows of a cloth knitting pattern: standard syntax

The pattern in Figures 5 and 6 show the first 7 of nearly 200 rows. A version of the pattern using the *undetermined repeat* construct can be seen in Figure 7. Notice that we can easily encapsulate the first 81 rows of the pattern in three rows. This is encouragement to attempt to include this construct in a knitting formalization project in the future.

row: K4.
row: K2, YO, *K; to end.
row: *K; to end.
(repeat last two rows until 44 stitches on needle)

Figure 7: First rows of a cloth knitting pattern: standard syntax condensed

We can see that writing the concrete syntax and a proof of correctness for the entire pattern would be a very laborious task. From the work in Section 4, we can see how it would be done, but also how large the proof of even a tiny program or pattern can become. This suggests that an implementation in a linear logic programming language and the availability of a GUI for easily building patterns would be an essential component of any tool using this work if it is to be a useful productivity tool for knitters.

6 Discussion and Future Work

The knitting language presented here is a subset of the full language. Adding more base stitches does not change the dynamics of the language but rather adds more axiomatic rules, These translate to formulas in the persistent context that do not have any linear logic connectives.

A more challenging construct to include is what we may call *undetermined repeats*. Recall from Section 5 that this construct tells the knitter to repeat a stitch as many times as possible to the last x stitches. This operation requires knowledge of the number of loops available on the left needle at the time that the stitch is reached. We would need to also keep track of state that would be updated as we traverse a pattern. If we were to add a predicate representing the state of the needles, then any formula encoding a rule about the dynamics of an undetermined repeat requires linear logic. This is because the predicate holding the needle state must be ephemeral. As discussed in Section 5, this is best left as future work.

There are other knitting techniques that have a different notion of correctness than we have presented here. An example of such a technique is short rows, where some loops can be left on the left needle before beginning the next row. We have excluded short rows from this work because they require additional constructs in the language and it would be far more complicated to manage and explain the changes to the pattern state. Exploring this would be more appropriate in future work.

There is quite a bit of work that can be done to increase the usability and benefits of this project. Implementing the logic in a linear logic programming language would be required to elevate this work to a productivity tool that could improve the lives of knitters rather than purely an object of theoretical interest. This is because writing proofs of pattern correctness by hand using this work is similarly laborious compared to other ways of manually checking pattern correctness, although the approach here is more formal and organized. An implementation realizes the benefits made available by a computer for checking pattern correctness.

By embedding this logic in a linear logic programming language, proof search can be used as a tool to figure out if a pattern is correct, to fill in segments of a pattern, or to tell the knitter how many stitches they require to be able to complete some sequence of stitches. All of these tasks are important parts of pattern design.

Using such an implementation would require specialized knowledge. A top layer would be necessary to parse standard text knitting patterns (as described by the Craft Yarn Council [7]). Another option is a GUI allowing restricted selection of pattern elements, thus reducing opportunity for errors in pattern syntax. This "pattern element selection" could either take the form of a drag-and-drop interface or buttons that add selected elements to the pattern. A screenshot of a rudimentary version of this layer can be seen in Figure 8.

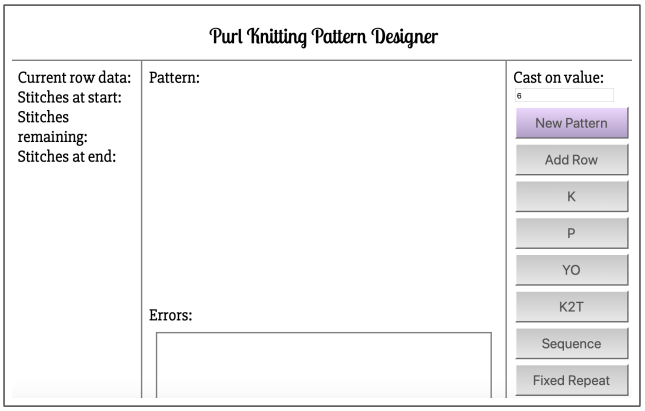


Figure 8: Purl Knitting Pattern Designer screen shot from [4]

This work hopes to provide a system that can lead to a concrete tool that would reduce opportunities for frustration and improve efficiency when hand knitting. This tool would also provide a resource for pattern designers to check the correctness of their patterns and improve accuracy in the definition of knitting patterns. Another implication is pattern design can become more accessible to all knitters.

References

[1] Grandma's favourite dishcloth. https://app.box.com/s/abmmg1xufttgptywmmamuif9bahws2j7. Accessed: 2019-04-28.

[2] Lyndsay W Anderson and Christina U Gustavson. The impact of a knitting intervention on compassion fatigue in oncology nurses. *Clinical Journal of Oncology Nursing*, 20(1):102–104, Feb 2016.

[3] David Baelde et al. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning*, 7(2), 2014.

[4] Chelsea Battell. Purl knitting pattern designer. http://chelsea.lol/purl/. Accessed: 2019-04-28.

[5] Chelsea Battell. Domain specific language for modular knitting pattern definitions: Purl. Compilers course project, April 2014.

[6] M. Clave-Brule, A. Mazloum, R.J. Park, E.J. Harbottle, and C. Laird Birmingham. Managing anxiety in eating disorders with knitting. *Eating and Weight Disorders - Studies on Anorexia, Bulimia and Obesity*, 14(1):e1–e5, 2009.

[7] Craft Yarn Council. Standards & guidelines for crochet and knitting. PDF document hosted by CYC, 2015. http://media.craftyarncouncil.com/files/CYC_YS_s_and_g_rev2015_6.pdf.

[8] Amy Felty and Alberto Momigliano. Hybrid: a definitional two-layer approach to reasoning with higher-order syntax. *Journal of Automated Reasoning*, 48(1):43–105, 2012.

[9] Frank Pfenning. Linear logic. https://www.cs.cmu.edu/~fp/courses/15816-s12/, 2012. Course: 15-816, spring 2012. Accessed: 2019-04-01.

[10] Jill Riley, Betsan Corkhill, and Clare Morris. The benefits of knitting for personal and social wellbeing in adulthood: Findings from an international survey. *British Journal of Occupational Therapy*, 76(2):50–57, 2013.

[11] David J. Spencer. *Knitting Technology: A comprehensive handbook and practical guide*. Woodhead Publishing Limited, third edition, 2001. First edition 1983.

[12] Heike Utsch. *Knitting and Stress Reduction*. PhD thesis, Antioch University New England, 2007.

[13] Yuting Wang, Kaustuv Chaudhuri, Andrew Gacek, and Gopalan Nadathur. Reasoning about higher-order relational specifications. In *PPDPÍ3 Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, pages 157–168, 2013.