Improv - a language for explorations in movement design

Final Project Report - ME598 with Dr. Amy LaViers

Alexandra (Alli) Nilles

May 2017

Contents

"In choreography, as in all creative disciplines (including scientific experiment and discovery), error and mistake often play a crucial role in the creative process."
– Thecla Schiphorst [1]

Abstract

We introduce a new software tool, *Improv*, which allows for high-level description of robotic motion, along with immediate simulation of the resulting movement. To accomplish this, we have created a domain-specific language, embedded in Haskell, and inspired by Laban-Bartenieff Movement Studies. The DSL is compiled to a ROS node which publishes messages to any simulator compatible with ROS. The compilation step is accomplished by a program which monitors a file for changes made by the user, which allows users to create programs in any text editor and see the results in simulation without needing to perform any extra steps. We hope this tool will be of interest to roboticists designing high-level motion strategies who are frustrated with the current workflow in ROS, as well as newcomers to robotics looking for a relatively accessible tool for experimenting with creating robot motion.

1. Introduction

Within the robotics community, using ROS and its integrated simulators can be notoriously tricky. While a very powerful and widely-used tool, ROS requires users to specify motion commands at a relatively low level of abstraction, usually in Python or C++. Inspired by a few existing tools, such as *Dance*, a functional programming language inspired by Labanotation and augmented with a reactive layer which allows event-based control [2], *roshask*, a ROS client library written in Haskell [3], and *TidalCycles*, a language for live-coding electronic music [4], we imagine a tool where users can write code that is nearer to human language descriptions of robot motion, and immediately see the results of their code in a simulator. This would allow for faster development cycles, less frustration on the part of the user, and would generally make robotics programming more accessible to people not trained in robotics or computer science. This paper outlines the first steps that have been taken toward this tool, as well as an overview of related work.

LBMS Framework

Over the last century, the Laban Bartenieff Movement System (LBMS) has been developed by a community of movement researchers, led by Rudolf Laban, Irmgard Bartenieff, and Ann Hutchinson Guest. Labanotation has been developed as a movement notation system, used for precisely describing motion in time and space. The notation is specialized for human bodies, with symbols for specific body parts. Labanotation is used for a detailed, reproducible record of movement, somewhat similar to how sheet music can be used to reproduce a piece of music.

Motif is a reconception of Labanotation that shares many symbols and concepts, but denotes the most important elements of a dance without quite the same level of detail. In Motif, units of movement are combined into phrases. The phrasing or grouping of movement is important for expression: "two people might have the same Effort elements present, or the same body parts utilized, but if they have organized and combined the elements differently the message will be dissimilar" [5]. The LBMS paradigm also offers a categorization which is useful for movement analysis: Body, Effort, Shape, and Space. These terms will be capitalized through this document to differentiate them from the colloquial terms.

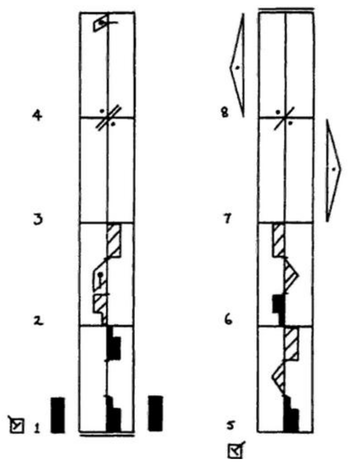
2. Computer Tools for Movement Design

> *"I think [computer technology] could affect choreographers' experience of movement in the same way electric light first altered the way visual artists saw the world"* – Merce Cunningham [6]

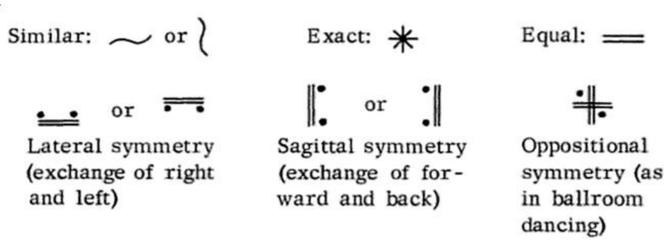
Adapting formalisms from LBMS to Computer Languages

As it was developed before computers were widespread, and encompasses many concepts that we do not know how to represent in code, Labanotation and Motif cannot be directly translated into executable specifications. Instead, in this project I have aimed to take inspiration from, and use some of the formalisms developed by the dance and movement analysis community, leveraging their expertise. At the same time, we need to change some of the mechanical aspects of the notation to improve the expressivity and compactness of representations in the language being developed.

For example, one of the most basic programming tools - iterating an instruction a certain number of times - has an interesting representation in dance notation. In Ann Hutchinson



(a) Examples of repeating with modification - the symbol at the beginning of measure four indicates that the first two measures are repeated to the other side (symmetric repeat), while the symbol with one line at measure 8 indicates an exact repeat, with some added arm gestures.


Similar: or
Exact:
Equal:
or
Lateral symmetry (exchange of right and left)
or
Sagittal symmetry (exchange of for-ward and back)
Oppositional symmetry (as in ballroom dancing)

(b) Abbreviations for repeated movements

Figure 1: Abstractions for repeated movements in Labanotation, from

Guest's book *Labanotation*, she details the expressive capabilities of the language with regards to repetition of motion. Measures may be repeated exactly or symmetrically across different body planes, they may be repeated with small differences noted, and they may be repeated an arbitrary number of times [7]. There are also abbreviations for exact repeats or "similar" repeats.

There is a notation for labelling a section of the score with a letter, and then later referencing that letter along with a repeat symbol to indicate which movement should be repeated. If there is no letter given, it is assumed that the most recent phrase is the one that should be repeated.

Modern programming practice has developed more compact and consistent representations of these concepts. However, the taxonomies and formalisms developed by the movement observation and analysis communities have proven to be very helpful in this project. The relevant concepts will be explored more in Section 5.

Lifeforms

Lifeforms is a software system developed largely by Thecla Schiphorst and Thomas Calvert at Simon Fraser University, and used by the choreographer Merce Cunningham from 1989 onward. It has been under development since 1986 and is still commercially available. In the version described in her 1993 thesis, the software has three modes: a *movement sequence* editor, where the user can design small phrases or movement motifs on a single body; a *spatial sequence* editor, which allows the user to arrange groups of dancers in space; and a *timeline* editor, which allows rearrangement and transformation of sequences in time.

Users can design movement sequences in several ways - one way is that the user specifies *keyframes* - specific poses of a simulated human body - and the program generates smooth movements between the frames. Users can also click and drag parts of the body, with reactions calculated with inverse kinematics. Lifeforms can use skeletons other than the human body, but as it is specialized for dance choreography applications, it has a large library of predefined movement sequences, such as movements from ballet and modern dance.

Schiphorst's thesis [1] contains a detailed history of computer-assisted choreographic tools, starting in 1964 with a program developed at the University of Pittsburgh which generated random dance sequences. There has been a long history of using computer systems to record, interpret and generate notation and choreographic scores for dance, often based on Labanotation. As early as 1979, the importance of abstraction was recognized. A software tool developed at Simon Fraser University was perhaps the first exploration into parameterized abstractions for movement specifications: "the instruction sequence is not modified, but the important features, such as the direction the dancer is facing while performing the sequence, as well as the speed and the style, can be made parameters of the macro" [8]. The goals of *Improv* are not much different, though the software and visualization tools I have access to are nearly four decades further along.

These projects, and many that have followed, have been constrained by the available technology - programming languages with low-level instructions and unintuitive syntax, and graphical displays that are slow to update and without good physics simulators. These problems still plague tools for movement designers, including the tool described in this paper. But as Schiphorst argues, "it is not technological constraints that hold us back from using technology in new ways; technology changes at a tremendous rate. Our willingness to explore beyond the constraints of our imagination has the greatest effect" [1].

In Robotics

Dance

Dance is a domain-specific language built in Haskell, a strongly-typed functional programming language [2]. *Dance* was developed at Yale in 2003 by Liwen Huang and Paul Hudak. The project included a base DSL inspired by Labanotation, as well as a reactive layer that allowed the robot to react to sensor events. The project targeted humanoid robots and included a 3D simulator (the program would generate the necessary simulation code).

One feature of the language is the data type `Action`, defined as:

```
data Action = Move Direction Level
            | Turn Direction
            | Transpose Direction
            | Extend Length
            | Support
```

where the data type is able to represent limbs that can move in space, rotate around an axis, movements translating the robot in space, prismatic joints, and an action which makes sure the robot does not fall over, respectively.

The `Dance` data type is defined as:

```
Dance b = Prim Action Duration
        | Rest Duration
        | WithPart b (Dance b)
        | Dance b :+: Dance b
        | Dance b :=: Dance b
```

where `Dance`s are created as either motion primitives, defined as an action and a time duration, a rest for a duration, a `Dance` with an associated body part, or series or parallel `Dance`s. Other interesting functionality includes:

- `repeatn`: a function for repeating a sequence of motions n times
- `transform`: a function for transforming a dance, given a function that transforms actions (ex: a function which changes the level of all `Move` actions)

- `rewire`: a function which, given a mapping between two robot bodies, will map dances from one body to another
- reactivity: `Events` can be defined, and movement sequences will only happen if those events occur

All of this expressiveness is a consequence of the fact that movement sequences are treated as *first class* - they are data in the language that can be passed around and transformed the same way numbers or other values can be in most programming languages. *Dance* also has the full expressiveness and library support of Haskell. It obviously lacked widespread adoption - in my opinion, this was likely because of the relative obscurity of Haskell (something which is changing with time), and because of the lack of integration with a mainstream simulator. The examples given in the paper were also done on abstracted, simplified robots - it's possible that examples with more realistic robots would help motivate adoption of the software.

Automatic Phrase Generation

In [9], a movement description and generating framework is proposed. The method requires a transition system - a labelled NFA where states are poses, and transitions provide a way to move the robot between poses - as well as an LTL specification of requirements such as "never lift one leg up while the other is already raised," or "always eventually return to pose 3." The LTL specification is converted into a Büchi automaton, and the product automaton with the transition system is formed.

The resulting automaton will only accept and generate phrases that are consistent with both the physical dynamics and the high-level specifications.

The limitations of the expressivity of this system are bounded by the expressivity of the transition system - which is platform-specific - and LTL.

LTL has some expressive limitations that may be relevant for robotics, which is why Computational Tree Logic (CTL) has become popular for some verification tasks. Neither logic is strictly more expressive than the other. The models of LTL are infinite streams, and LTL only considers one stream at a time: so it can make statements about what will eventually or always happen in one run. However, CTL checks all possible runs of a system and has the capability to check all branches (**A** operator) or only one (**E** operator) [10]. Therefore, in a situation with concurrent events: such as multiple agents moving separately, or two body parts moving simultaneously, LTL may not be expressive enough. I am not suggesting that CTL is necessarily better; it suffers from syntax that is very bad at naturally expressing useful concepts. Additionally, clever representation in LTL may circumvent the problem of needing multiple streams.

Overall, the use of formal logics or automatic movement generation are outside the scope of this project - we are focused more on programming language design for accessibility and ease of use. However, the dream of a language which is easy to use *and* allows formal specification of movement properties is important to keep alive.

Hybrid automata

Hybrid automata uses discrete automata whose states describe different continuous dynamics that the system switches between. This approach allows researchers to leverage the insights of automata theory as well as traditional control theory.

A hybrid automata consists of [11]:

- a finite multidigraph (V, E). The vertices V are control modes
- (*atoms*), and the edges E are control switches (*composition operators*).

- A finite set $X = \{x_1, \ldots, x_n\}$ of real-numbered variables, along with their first derivatives $\dot{X}$, corresponding to the dynamical variables of the system being modelled
- *node labels:* initial conditions, invariants (constant variables), and dynamical variable labels for each $v \in V$.
- *edge labels:* A predicate condition on each edge that indicates when transitions occur, and an event label for each edge

Hybrid automata are used widely for modelling and control of systems with discrete and continuous dynamics, including motion of robots. There are several model-checking tools that are based on this model. However, there are some control structures that cannot be expressed as hybrid automata: for example, see Figure 2, from [12].

MDLe

MDLe is a motion description language with atoms of the form (u, ξ), where u is a *control quark*, a control input which is a function of time and sensor readings, and ξ is a boolean function over the sensor readings which indicates when to end the current atom and move to the next (similar to mode switches in hybrid automata).

MDLe is defined by the following grammar, where E is a valid MDLe string:

$E \to \epsilon$	empty string
$E \to (u, \xi)$	atom
$E \to EE$	string concatenation
$E \to (E, \xi)$	encapsulation

Instead of being composed in a multidigraph, atoms are composed through *encapsulation* and *string concatenation.* Encapsulation means that an interrupt ξ can apply to an entire string, not just one control quark. Concatenation means that once the first string in the concatenation is interrupted, the next string begins.

While there are some similarities between MDLe and hybrid automata (switching between modes of of continuous dynamics depending on sensor readings), they do not have an exact equivalence. For example, the hybrid automata in Figure 2 (a) has a choice between two switching conditions, ξ_{12} and ξ_{13}, from mode s_1, which cannot be expressed in MDLe since encapsulation only associates one interrupt with each MDLe expression. In the other direction, Figure 2 (b) shows an MDLe expression that cannot be represented with a hybrid automata, since the control quark (node label) is repeated in two unique states. A hybrid automata could be created with a loop from u_2 to u_1, labelled ξ_2, but there is no way to encode that the only resulting allowed action is a transition ξ_3 to the end state. The hybrid automata would need to be augmented with more state to track the trajectory history.

We conclude that **different types of combinators** (composition operators), create languages with **different expressive capabilities** even when the atoms of the language are quite similar. Thus, analysis of combinators is important in language design - we will discuss this further in Section 5.

Modular, Compositional Actions

When considering how to compose movements, a natural scheme arises - movements can happen in series, one after the other, or they may happen in parallel, either with two actions occuring with two different parts of a body or by two disconnected bodies moving at the same time.

Series-parallel digraphs have been long known in computer science, but only recently applied to robotic motion languages, such as in [13]. The language developed has two composition

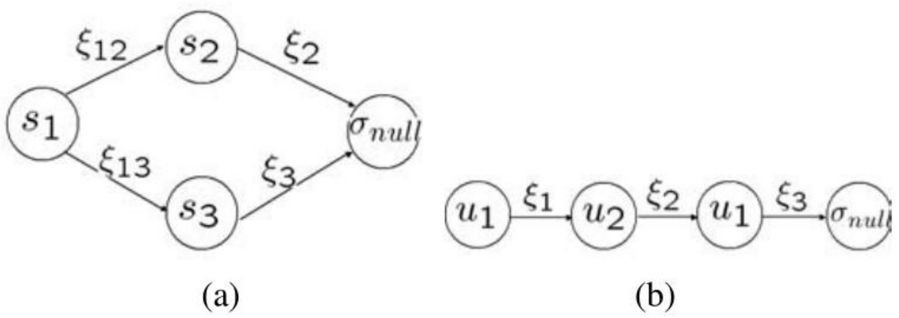


Fig. 2. (a) A hybrid automaton that has no MDLe equivalent. (b) An MDLe string that has no hybrid automata equivalent.

Figure 2: Though they share some characteristics, there is not a 1:1 correspondence between MDLe strings and hybrid automata.

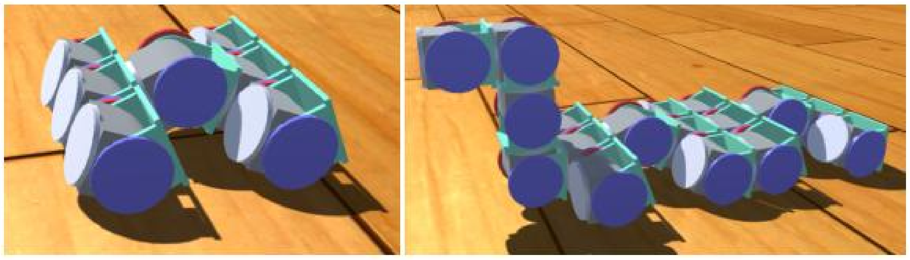


Fig. 4: Car design (left) and backhoe design (right)

Figure 3: Behaviors for relatively large and complicated robots can be easily composed from behaviors of smaller robots.

operators, that allow behaviors to be executed in series or in parallel. The language was designed for a modular, composable robot hardware platform, so when designs are physically composed, their behaviors can be composed as well as behaviors for the larger design. Using this language, the researchers were able to create "driving" and "turning" behaviors for a 28-degree-of-freedom backhoe by composing behaviors for smaller "car" designs (see Figure 3 for an example of a large robot design with modular actions).

Influences on Improv

From the tools and movement languages described so far, some lessons can be learned that I have applied to *Improv*:

- Labanotation, and other frameworks developed by the dance community, can be extremely helpful in formulating formal motion description and design languages
- Visualization and fast feedback is very useful in the creative process
- Formal logics, such as LTL, are useful for creating guarantees on behavior and allowing a high-level specification
- Simple composition semantics can be all that is needed to describe a wide range of motions

While formal specification and automatic synthesis are not incompatible with the live-coding paradigm, they do tend to require more training and create a higher barrier to the creation of code that "does something." This idea will be formalised in Section 4.

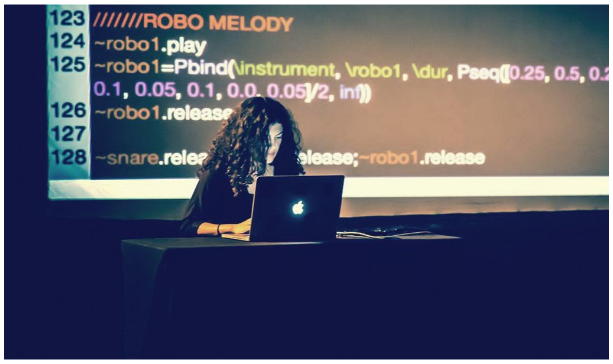
3. Music Improvisation with Digital Tools

Another artistic field with a long history of collaboration with digital technology is music. DJs and electronic music producers have experimented with a myriad of different designs for software and hardware tools for music design. Some of these tools are meant to be used by musicians during performances, while others are intended for "offline" use, though the line blurs easily.

In this section I will describe a few digital technologies, mostly used in the performance sphere, that have been influential in my thinking while designing *Improv.*

Algorave

"I think it's important to consider programming as exploration rather than implementation, because then we are using computer languages more like human languages. Any software interface can be thought of as a language, but the openness of programming allows us to set our own creative limits to explore, instead of working inside fixed, pre-defined limits. To me this is using computers on a deep level for what they are - language machines." – Alex McLean [14]



```
///////ROBO MELODY
~robo1.play
~robo1=Pbind(\instrument, \robo1, \dur, Pseq([0.25, 0.5, 0.2
0.1, 0.05, 0.1, 0.0, 0.05]/2, inf))
~robo1.release

~snare.relea                    elease;~robo1.release
```

Figure 4: In algoraves, performers often project code while using it to compose live music.

Algorave is a performance movement where "Using systems built for creating algorithmic music and visuals... musicians are able to compose and work live with their music as algorithms."[1] Specifically, I have focused on analyzing *TidalCycles*, a Haskell-based DSL for composing electronic music. Code written in this DSL looks like:

```
d1 $ sound "bd*4" # gain "0.5" # delay "0.5"

d2 $ sound "bd*4" # gain (every 3 (rev) $ "1 0.8 0.5 0.7")
```

where the two lines of code, when executed, play simultaneously. Music samples - such as "bd", bass drum - are imported and transformed with gains, delays, and patterns. In the second line, a pattern of gain transformations is applied, and every third time this pattern is applied, it is reversed.

[1]https://algorave.com/about/

Influences on Improv

The algorave community values tools which allow the user maximum freedom, to encourage creativity, but at the same time are trying to make their tools accessible to artists and people without much coding experience. These goals are shared by my project.

Many of the design decisions of *Improv* and *TidalCycles* are similar - how to most effectively use Haskell? How to manage timing of events, and what should be the semantics of composing events in parallel and series?

The algorave community also has a heavy emphasis on live-coding, or coding in front of people as part of a performance. This requires tools which have low latency between writing code and producing music. Similarly, I want to accomplish "on the fly" robotics programming, where users can get immediate feedback - visually, through a simulator - while they are writing code.

4. Evaluating Design Languages

Comparing Expressivity

When we ask *how expressive* a certain language is, we are always explicitly or implicitly comparing it with other languges. Formally, language A is **as expressive as** language B if for each formula ϕ of B there is a formula ψ of A such that $\models \psi \leftrightarrow \phi$ [15]. The $\models$ operator is the *semantic consequence* operator, which in this context means that the formulas ψ and ϕ implying each other is a tautology in the semantics of the languages: they are trivially equivalent, and the equivalence does not need to be derived syntactically.

The intuition for this formal definition is that A is as expressive as B if everything that can be said in B, can be said in A. If not everything in A can be said in B, then we say that A is strictly more expressive than B.

This definition examines the expressive *power* of languages - the space of all concepts which may be expressed. We may also informally use the word *expressive* to describe how easy it is to represent a certain concept in a language - for instance, while it may be possible to represent a binary tree data structrue in x86 assembly code, no one would argue that this is a clean representation! We may clarify this concept by using the phrase *naturally expressible*, since we are implicitly comparing the expressive power of a certain language to natural language or mathematics. [16] has an interesting discussion of expressiveness in the context of Maude.

Cognitive Dimensions

When discussing design principles for user interfaces and programming languages, the framework developed by Green and Petre - the *cognitive dimensions of notations* - can be quite useful [17]. There are fourteen "cognitive dimensions," or criteria for evaluating programming languages. In particular, the following have been ones I focused on while developing this project:

- *Progressive Evaluation:* Can a partially-complete program be executed to obtain feedback on 'How am I doing'?
- *Viscosity:* How much effort is required to perform a single change?
- *Abstraction gradient:* What are the minimum and maximum levels of abstraction?
- *Closeness of mapping*: What 'programming games' need to be learned?
- *Consistency:* When some of the language has been learnt, how much of the rest can be inferred?

- *Error-proneness:* Does the design of the notation induce 'careless mistakes'?

5. Improv

The main contribution of this project is the development of a domain specific language (DSL), embedded in the programming language Haskell, which allows the specification of different robotic platforms and motion sequences on those platforms. This DSL is integrated with the software platform ROS (robot operating system), a commonly used open-source tool for modelling and controlling robotic systems. Thus, instructions written in the DSL are converted into *ROS messages*, which are passed to a simulator to enable immediate feedback about the execution of the motion instruction on a given robotic platform. The necessary plumbing tools have been constructed to make Improv is a live-coding tool for the creative design of robot motion. See Figure 5 for an overflow of the dataflow of *Improv*.

All code referenced in this section of the paper is available at the author's github.[2]

Haskell syntax notes: `[Thing]` indicates a list of `Things`. Data structures are constructed as:

```
data Blah a = Foo a
            | Bar
```

indicating that the type `Blah` is parameterized over another type `a`, and consists either of types `Foo a` or types `Bar`.

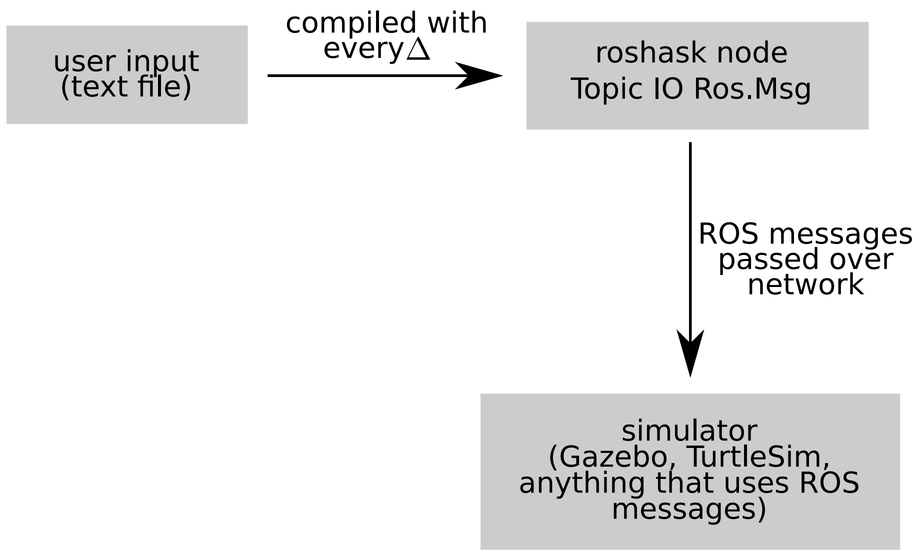


Figure 5: An illustration of how user input, written to a text file, is converted into a ROS node which publishes messages to a simulator.

Language Design

Motion Composition

In this section, we will overview some of the syntax and semantics of the DSL, and explain some of the different composition operators.

[2]https://github.com/alexandroid000/improv

We take some inspiration from the formalisms developed by LBMS - for example, using directions defined relative to a kinesphere centered at a origin on the body. We define origins by their unique integer ID. We also allow five different `Length`s or Spatial extents.

Natural `Direction`s such as left, right, forward, etc are defined and we introduce a composition operator, `:*:`, of the directions. Right now, there are no reductions done over this composition operator until the robot-specific compilation step - this leaves maximum flexibility in how to interpret compositions of directions. In LBMS, there are 27 possible directions, 9 at each level. However, we may imagine a robot with fine motor control that is able to interpret a composition of three `Lef`s and one `Forward` as a weighted average of vector representations of those directions, instead of binning this instruction as `Forward-Left`.

Algebraically, we say that we use the *free monoid* over directions to avoid losing any information in the composition process. For an interesting discussion of the usefulness of monoids in library design, see [18].

```haskell
data Direction = Lef | Righ | Forward
               | Backward | Center
               | Low | Mid | High
               | Direction :*: Direction


data Length = Zero | Quarter | Half | ThreeFourths | Full

data Origin = O Int -- unique id for each origin
```

The next important data structure is `Action`s, which can either be a single action - defined by an origin, direction, and spatial extent, or a list of actions (to be performed sequentially). I plan to refactor this data type to be independent of `Origin` - it makes more sense to me to specify the `Origin` and body part involved in the motion together, at a later stage, and leave `Action`s independent of the robot body.

I also think I am going to remove lists of actions from the definition, and have sequential composition only at the `Dance` level.

```haskell
data Action = A Origin Direction Length
            | As [Action]

data Dance b = Prim Action b
             | Rest
             | Dance b :+: Dance b -- in series
             | Dance b :||: Dance b -- in parallel
```

The `Dance` datatype is heavily inspired by the *Dance* library described earlier. This data type is parameterized over the type of robot body, which could be a kinematic chain or some other type. `Dance`s can be `Action`s paired with body parts, a `Rest`, or series or parallel compositions. Here it is important to note the difference between sequential movements defined by the `:+:` operator, and sequential movements defined by a list of `Dance`s. Similarly to the semantics of *TidalCycles*, a list of `Dance`s will compile down to a sequence of motions where each element in the list is performed in the same amount of time (a "beat"). A series composition *inside* one of those elements implies that each of those motions will be "sped up" to fit within the same beat.

Consider the example:

```
danceLeft = Prim (A 0 Left Full) arm1
center = Prim (A 0 Center Zero) arm1
danceRight = Prim (A 0 Right Full) arm2

dance = [danceLeft, Rest, center, danceRight :+: danceLeft]
```

In this example, `danceLeft` is a primitive that extends `arm1` fully to the `Left`. `danceRight` does the same with a different arm to the `Right`. `center` brings `arm1` back to the center. The function `dance`, which has type `[Dance b]`, specifies a dance where `arm1` is extended left for one beat, the whole robot rests for one beat, the arm returns to center for one beat, and then `arm2` is extended to the right in half a beat after which `arm1` extends to the left for the remaining half a beat. This requires the robot to move twice as fast, placing physical limitations on how many actions can be performed in one beat. These limits are not described by the language - we allow the user to discover them through experimentation.

The final important data type to discuss is the description of robots themselves. So far, I have implemented a kinematic chain data type, which has the usual tree structure, except I allow more than two chains to be connected by a single joint. Also, I allow "forests" of disconnected kinematic trees.

I also define a type class `Parts` which requires the functionality `contains`, which takes a body part and returns all its sub-parts (parts moved if the input part is moved). It also requires a function `origin` which returns a default `Origin` for each body part.

```
data KineChain a = Joint Origin [KineChains a]
                 | Link Origin a
                 | Collection [KineChains a]

class Parts b where
    contains :: b -> [b]
    origin :: b -> Origin
```

Often, robots and motion patterns have many natural symmetries that can be useful for compactly representing specifications and instructions. Often in dance classes, dancers will learn a routine on one "side" and then be instructed to perform it on the "other side". Haskell type classes provide a way to make a general abstraction of symmetry that can be applied to any data type, whether that type is `Directions`, `Actions`, robot body data types, etc. To be an instance of the `Symmetric` class, data type `a` must have a function defined for it that takes a `Plane` (cartesian or body-centric), and computes the reflection of type `a` over that plane.

```
class Symmetric a where
    refl :: Plane -> a -> a

instance Symmetric Direction where
    refl plane (dir1 :*: dir2) = (refl plane dir1) :*: (refl plane dir2)
    refl YZ Lef = Righ
    refl YZ Righ = Lef
    etc
```

Timing

Some robot motion description languages, such as MDL [12] or hybrid automata [11], require "triggers" which signal when to end a specific action. These triggers can take the form of explicit timers or can be conditional sensor-driven events. However, natural language description of movement often use only implicit timing commands: for example, "step forward, turn right, and do that again twice as fast." Thus, to minimize the amount of low-level details that must be specified by the user, we adopt the idea from TidalCycles of a global "cycle" or beat. In TidalCycles, the default cycle plays at 1 Hz (one cycle per second). You can compress multiple samples or patterns inside of one cycle, or stretch one pattern over multiple cycles, but everything is specified relative to the global clock.

In Improv, this global clock is governed by the rate at which our node publishes messages to ROS. In the current implementation, we are sending velocity commands to the simulator, so if we publish a command to "move forward at 1 meter/second" at 1 Hz, we will not be able to specify movements which only translate the robot by 0.5 meters. The only way to accomplish this would be to tell the robot to travel half as fast for the same amount of time, which would not necessarily be the desired movement.

To solve this problem, I am proposing to allow the user to vary the global publishing rate as a parameter in their robot configuration file. This serves essentially as the "movement resolution" - the fastest rate at which we can switch between movement primitives.

For example, say the user sets the global publishing rate to be 100Hz. Then we are faced with the question, how long should a single movement primitive - say, "turn core left 90 degrees" - take? The user will be able to speed up and slow down movements - by composing them in series or in parallel with other movements or rests - but it seems that we should have a "default" duration of movement. This duration should be invariant of the global publishing rate, which the user may change according to their resolution needs.

This is an area where I have not yet worked out the desired semantics and how to implement them. This is the area of the project under most active development currently.

`Dance` Transformations

Using features of Haskell such as higher-order functions and pattern matching, we can implement functions to transform sequences of motions. For example, we can take a list of `Dance`s and combine them sequentially or in parallel:

```haskell
seqL, parL :: (Parts a) => [Dance a] -> Dance a
seqL = foldr (:+:) (Rest 0)
parL = foldr (:||:) (Rest 0)
```

We can also take a single `Dance` and repeat it `n` times in sequence:

```haskell
repeatn :: (Parts a) => Int -> Dance a -> Dance a
repeatn n dance = seqL $ take n $ repeat dance
```

And given a function that transforms `Action`s (say, one which composes all the directions with `High`, "lifting" them), we can apply this function to all primitives in a `Dance`:

```haskell
-- map over all actions in a dance
transform :: (Parts a) => (Action -> Action) -> Dance a -> Dance a
transform f (x :+: y) = (transform f x) :+: (transform f y)
transform f (x :||: y) = (transform f x) :||: (transform f y)
transform f (Rest) = Rest
transform f (Prim act dur) = Prim (f act) dur
```

Many more transformation are possible, and in particular the LBMS framework can be used to inspire transformations. We can imagine transformations which change the dynamic qualities of a movement, such as its Effort quality or expanding or contracting it in Space.

Editor-agnostic live-coding

One challenge that commonly ails interactive text-based tools is being tied to a specific editor. For example, the algorave live-coding tool TidalCycles was originally developed for emacs, a powerful editor which has a notoriously steep learning curve. Many people prefer to use simpler editors, so new live-coding plug-ins have been developed for editors such as Atom and Sublime Text. The downside is that for every new editor that people want to use, significant programming and interfacing effort must be expended.

Often, developers of new programming languages and software tools have the time and motivation to learn how to use complex text editors, but the desired user base may not. In our case, we are developing a tool that should be usable by artists, children, and programming novices, as well as experienced roboticists. Thus, we wish to allow users flexibility to choose their editor, so they may use more intuitive and graphical editors such as Sublime text or more specialized editors such as vim.

To accomplish this, instead of creating a plugin interface for each desired editor, we run a shell script which simply monitors the file that the user is editing for changes. Every time the user saves the file, the program detects a change, compiles the program to a ROS node, resets the simulator, and restarts the ROS node. This design choice circumvents the need to interface with specific editors.

ROS interface (roshask)

Roshask is a client library for ROS, written in Haskell. It was developed by Anthony Cowley at the University of Pennsylvania GRASP Lab [3]. It is capable of hooking into a ROS server and publishing to and subscribing from topics. It treats topics as first class entities, which allows for topic *combinators*, such as an `everyNew` function which takes two topics as arguments and publishes a tuple with a message from each every time *either* of them publishes a new messages. Likewise, it has a `bothNew` function which takes two topics and publishes a tuple with their data only when *both* topics have new messages. This infrastructure makes it much easier to reason about timing of events in a ROS system with topics which publish at different rates (such as sensors with different update frequencies). The types of topic combinators that can be created are limited only by our imagination.

Roshask treats topics as *streams* - infinite sequences of events, processed one at a time. Haskell has natural data structures for handling streams, such as lists and monads, and uses *lazy evaluation* to process computation - only computing values when they are needed, which allows it to handle infinite lists. This is a natural framework for robotics, where we have a robot operating for an indefinite amount of time in the world, observing streams of sensor values and requiring a stream of commands to follow. This is also the model used by ROS - nodes run forever, always knowing how to publish the next value in the stream they produce.

To integrate a robot with *Improv* for the first time, one must provide a function which converts `Dance` data types to ROS message types.

Example Programs

Here is a minimal working example, as would be written by a user:

```
module Demo where

import RobotSpec
import Improv
```

```
left :: Action
left = A (origin core) Lef Quarter

moveTopic = moveCommands $ move core left
```

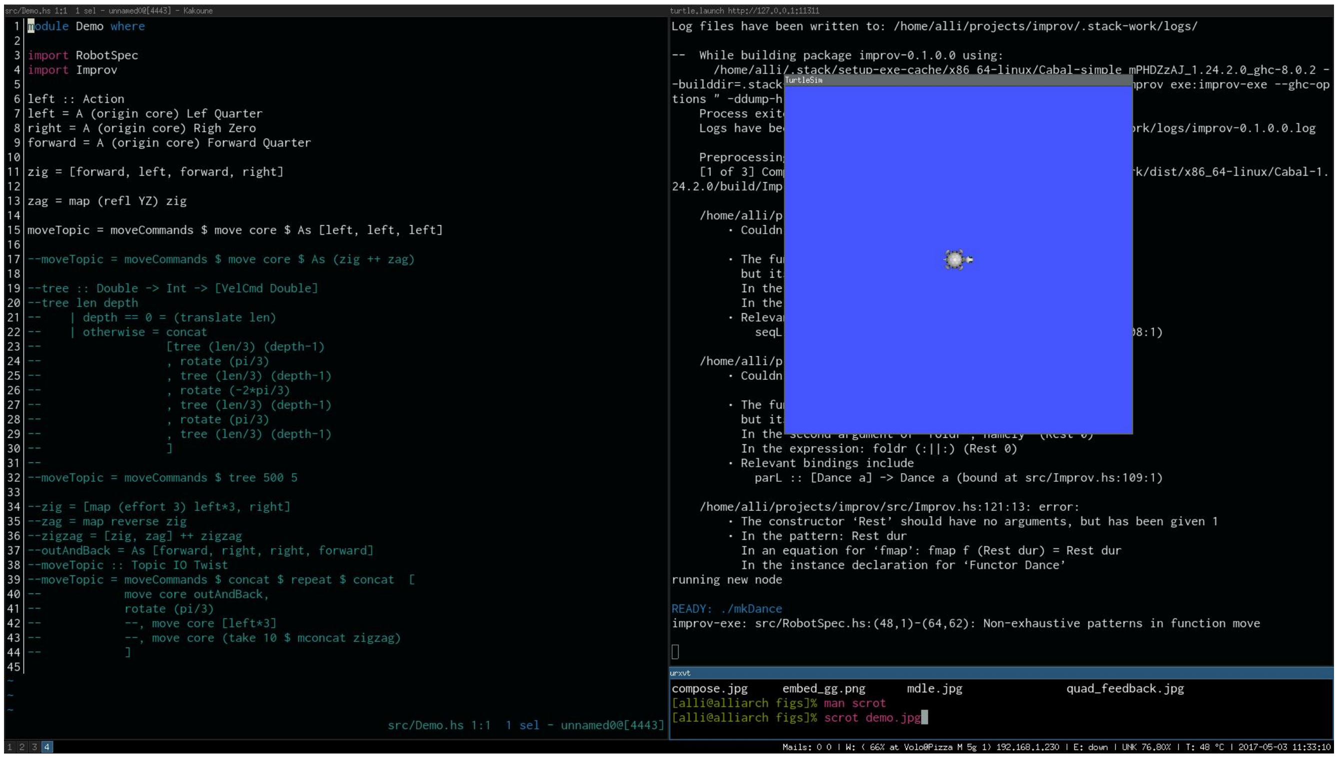
where `core` and `moveCommands` are functions defined for a specific robot (in this case, a simple differential-drive robot).

A more complicated program may involve patterns, such as:

```
zig = [forward, left]
zag = map (refl YZ) zig
zigzag = [zig, zag] ++ zigzag
```

which would create an infinitely long sequence of commands that cause a zig-zagging motion.

Below is a screenshot of what the author's screen looks like while using *Improv*, using TurtleSim as the simulator. The author uses a vim-like terminal editor, but it should be emphasized that any editor can be used to write *Improv* programs.



```
src/Demo.hs 1:1  1 sel - unnamed0@[4443] - Kakoune
module Demo where

import RobotSpec
import Improv

left :: Action
left = A (origin core) Lef Quarter
right = A (origin core) Righ Zero
forward = A (origin core) Forward Quarter

zig = [forward, left, forward, right]

zag = map (refl YZ) zig

moveTopic = moveCommands $ move core $ As [left, left, left]

--moveTopic = moveCommands $ move core $ As (zig ++ zag)

--tree :: Double -> Int -> [VelCmd Double]
--tree len depth
--     | depth == 0 = (translate len)
--     | otherwise = concat
--                    [tree (len/3) (depth-1)
--                    , rotate (pi/3)
--                    , tree (len/3) (depth-1)
--                    , rotate (-2*pi/3)
--                    , tree (len/3) (depth-1)
--                    , rotate (pi/3)
--                    , tree (len/3) (depth-1)
--                    ]
--
--moveTopic = moveCommands $ tree 500 5

--zig = [map (effort 3) left*3, right]
--zag = map reverse zig
--zigzag = [zig, zag] ++ zigzag
--outAndBack = As [forward, right, right, forward]
--moveTopic :: Topic IO Twist
--moveTopic = moveCommands $ concat $ repeat $ concat  [
--             move core outAndBack,
--             rotate (pi/3)
--             --, move core [left*3]
--             --, move core (take 10 $ mconcat zigzag)
--             ]

~
~
~
                                          src/Demo.hs 1:1  1 sel - unnamed0@[4443]
```

```
turtle.launch http://127.0.0.1:11311
Log files have been written to: /home/alli/projects/improv/.stack-work/logs/

--  While building package improv-0.1.0.0 using:
      /home/alli/.stack/setup-exe-cache/x86_64-linux/Cabal-simple_mPHDZzAJ_1.24.2.0_ghc-8.0.2 -
-builddir=.stack                                                  nprov exe:improv-exe --ghc-op
tions " -ddump-h
    Process exit
    Logs have be                                                  ork/logs/improv-0.1.0.0.log

    Preprocessin
    [1 of 3] Com                                                  rk/dist/x86_64-linux/Cabal-1.
24.2.0/build/Imp

    /home/alli/p
        • Couldn

        • The fu
          but it
          In the
          In the
        • Releva
            seqL                                                  08:1)

    /home/alli/p
        • Couldn

        • The fu
          but it
          In the second argument of 'foldr', namely '(Rest 0)'
          In the expression: foldr (:||:) (Rest 0)
        • Relevant bindings include
            parL :: [Dance a] -> Dance a (bound at src/Improv.hs:109:1)

    /home/alli/projects/improv/src/Improv.hs:121:13: error:
        • The constructor ‘Rest’ should have no arguments, but has been given 1
        • In the pattern: Rest dur
          In an equation for ‘fmap’: fmap f (Rest dur) = Rest dur
          In the instance declaration for ‘Functor Dance’
running new node

READY: ./mkDance
improv-exe: src/RobotSpec.hs:(48,1)-(64,62): Non-exhaustive patterns in function move

▯
```

```
urxvt
compose.jpg     embed_gg.png      mdle.jpg              quad_feedback.jpg
[alli@alliarch figs]% man scrot
[alli@alliarch figs]% scrot demo.jpg
```

```
1 2 3 4        Mails: 0 0 | W: ( 66% at Volo@Pizza M 5g 1) 192.168.1.230 | E: down | UNK 76.80% | T: 48 °C | 2017-05-03 11:33:10
```

Figure 6

7. Conclusions

So far in the project, work has mostly focused on the "plumbing" aspects: how to monitor the input file, and compile user input to a ROS node. Now my focus is shifting to the language design side of the project, as well as optimizing for the cognitive criteria described earlier. To conclude, I will check in on how well my project meets these criteria:

- *Progressive Evaluation:* As seen by the minimal working example, it is fairly simple to get a program that does *something*. Every time the input file is saved, the user will get feedback on their program, either through a compiler error if they have incorrect syntax or from seeing the robot move. One problem that must be fixed is the somewhat large delay - about five seconds - between saving the file and seeing the updated simulation.

- *Viscosity:* It is relatively easy to make a simple change, especially at the level of `Actions` - one can change the direction, origin, or size of an action easily.
- *Abstraction gradient:* Since this is a DSL embedded in Haskell, we have very powerful abstraction capabilities. We can store motion sequences in variables and pass them around as data.
- *Closeness of mapping*: Some syntax requirements of Haskell can cause confusion. For instance, many errors can be caused by incorrect order of operations, which must be fixed by using parenthesis or the `$` operator to enforce the correct order of computation.
- *Consistency:* I have made an effort to make patterns consistent across different data structures, but this area also requires more work.
- *Error-proneness:* The tool is currently very error-prone, especially for people who are not familiar with Haskell and ROS. It can be debated whether it is more or less error-prone than traditional ROS client libraries.

Future Work

One way to solve some of the above issues is to create a *deep* embedding of the DSL by writing a parser which can interpret the text file, instead of compiling it as native Haskell. This will speed up the update process dramatically, reducing latency for the user. It will also allow more flexibility with the syntax - for instance, we could allow case insensitivity, and create more intuitive syntax for defining actions.

I also will continue work on the timing semantics, add more example robots with more articulation than a differential drive robot, and add examples of using Gazebo to simulate instead of just TurtleSim.

I will also continue work on creating useful high-level transformers for motion sequences, especially ones using concepts from LBMS.

8. References

[1] T. Schiphorst, “A case study of Merce Cunningham’s use of the lifeforms computer choreographic system in the making of trackers,” Master’s thesis, Simon Fraser University, 1986.

[2] L. Huang and P. Hudak, “Dance: A declarative language for the control of humanoid robots,” Yale University, YALEU/DCS/RR-1253, August 2003.

[3] A. Cowley and C. J. Taylor, “Stream-oriented robotics programming: The design of roshask,” in *Intelligent robots and systems (iROS), 2011 iEEE/RSJ international conference on*, 2011, pp. 1048–1054.

[4] A. McLean and G. Wiggins, “Tidal–pattern language for the live coding of music,” in *Proceedings of the 7th sound and music computing conference*, 2010.

[5] P. Hackney, *Making connections: Total body integration through bartenieff fundamentals*. Taylor & Francis, 2003.

[6] “Feature: Merce cunningham and lifeforms.” [Online]. Available: http://londondance.com/articles/features/merce-cunningham-and-lifeforms/.

[7] A. Guest, *Labanotation: Or, kinetography laban : The system of analyzing and recording movement*. Theatre Arts Books, 1977.

[8] T. W. Calvert, J. A. Landis, and J. Chapman, “Notation of dance with computer assistance,” in *New directions in dance: Collected writings from the seventh dance in canada*

conference, 1979.

[9] A. LaViers, Y. Chen, C. Belta, and M. Egerstedt, “Automatic sequencing of ballet poses,” *IEEE Robotics & Automation Magazine*, vol. 18, no. 3, pp. 87–95, 2011.

[10] M. Y. Vardi, “Branching vs. linear time: Final showdown,” in *International conference on tools and algorithms for the construction and analysis of systems*, 2001, pp. 1–22.

[11] “Hybrid automaton.” Wikipedia, February-2017.

[12] D. Hristu-Varsakelis, M. Egerstedt, and P. Krishnaprasad, “On the structural complexity of the motion description language mDLe,” Dec 2003.

[13] T. Tuson, G. Jing, H. Kress-Gazit, and M. Yim, “Computer-aided compositional design and verification for modular robots.” The International Symposium on Robotics Research, September-2015.

[14] S. Fortune, “What on earth is livecoding?” 2014. [Online]. Available: http://www.dazeddigital.com/artsandculture/article/16150/1/what-on-earth-is-livecoding.

[15] D. Harel, D. Kozen, and J. Tiuryn, *Dynamic logic.* 2000.

[16] J. Meseguer and et. al., “Maude manual,” 2016. [Online]. Available: http://maude.lcc.uma.es/manual271/maude-manualch1.html#x4-40001.1.

[17] T. R. G. Green and M. Petre, “Usability analysis of visual programming environments: A ‘cognitive dimensions’ framework,” *Journal of Visual Languages & Computing*, vol. 7, no. 2, pp. 131–174, 1996.

[18] B. A. Yorgey, “Monoids: Theme and variations (functional pearl),” in *Haskell*, 2012.