SkateSKOOL: an app for skateboarders

Kinsey Ho, Owen Janssen, Jamie Lee, Jack Wheeler

Goals/motivations

We designed an app called SkateSKOOL (cuz skateboarding is cool and it is a fundamental human right to have access to skateboard schooling) that helps skateboarders progress and learn new tricks by providing personalized trick recommendations based on current experience levels, the tricks they have already mastered, and their desired level of difficulty.

Analyzing and reasoning on skateboarding histories helps our system provide a list of tricks within reach, suggesting the best order for learning them, ensuring skateboarders have a smooth and effective learning progression. This is applicable to all learners, whether they are a beginner or an experienced skateboarder. For beginner skaters, this system could explain what tricks are and provide a clear learning path to skate which builds off itself. Essentially, for both beginners and experienced skaters, it gives a gameplan of what new tricks to learn at the freedom and time management of the user.

Skateboarding is a dynamic and ever-evolving sport, but it can be challenging for both beginner and experienced skaters to navigate this vast landscape of tricks and progress in an efficient way. As a skater, it is easy to get stuck in a rut of tricks you already know and not learn new ones. So our app helps broaden their trick repertoire, overcoming mental blocks by providing a clear path to learning new tricks, allowing skaters to learn at their own pace and build off their progress, making it a flexible and personalized learning tool.

Moreover, our project demonstrates the potential of KRR to be applied in various fields beyond traditional domains such as robotics or medicine.

We hope to one day train and build up a legion of skateboard enthusiasts that will take over the world with their skateboarding prowess and expertise. Skateboards can be used as effective weaponry and are the ultimate mode of transportation.

Representing knowledge

The majority of our knowledge base is filled with skateboarding tricks. The user of our app is then linked with these tricks through predicates including *personKnowsTrick* and *personKnowsTrickComponent*. We should first review how the tricks themselves are defined.

A SkateboardingTrick is a genls of *(TransporterStuntFn Skateboard)*. This is our concept of a skateboard trick. The SkateboardingTrick is a TemporalObjectType.

Each SkateboardingTrick is composed of trick components. TrickComponent is a FirstOrderCollection, so each instance of it is a representation of individual trick components.

There are three trick components defined: BoardRotation (rotation of board), Body Rotation (rotation of body), and BoardFlip (how many degrees the board is flipped)[1].

Thus, each trick component is a rotation in a direction of *n* degrees. To represent rotation direction (clockwise or counter-clockwise), we have defined RotationDirection as a FirstOrderCollection. Clockwise and Counter-clockwise are both genls of RotationDirection. This is how RotationDirection is defined:

(isa RotationDirection FirstOrderCollection)

(isa Clockwise Collection)
(genls Clockwise RotationDirection)

(isa Counter-clockwise Collection)
(genls Counter-clockwise RotationDirection)

We can now use RotationDirection to create a trick component. Each trick component is a FunctionOrFunctionalPredicate, which takes in a RotationDirection and an Integer (representing degrees of rotation), and the result is an instance of the collection TrickComponent. For example, to represent the rotation of a board, or BoardRotation, this is what is performed:

[1] Boardflip is a kickflip or a heelflip. These are 360° rotations of the board in the clockwise or counterclockwise rotations.

```
(isa BoardRotation FunctionOrFunctionalPredicate)
(arity BoardRotation 2)
(arg1isa BoardRotation RotationDirection)
(arg2isa BoardRotation Integer)
(resultIsa BoardRotation TrickComponent)
```

We now have definitions of trick components. For tricks themselves, we have defined a predicate called trickContains. This relates a SkateboardingTrick to TrickComponent. We use trickContains three times to relate the three different trick component types to the trick. The below .krf excerpt shows the representation of the trick “Ollie”:

```
(genls Ollie SkateboardingTrick)
(isa Ollie FirstOrderCollection)
(trickContains Ollie (BoardRotation Clockwise 0))
(trickContains Ollie (BodyRotation Clockwise 0))
(trickContains Ollie (BoardFlip Clockwise 0))
```

As seen, Ollie is a genls of SkateboardingTrick, and we use the predicate trickContains to represent the three different trick components that Ollie is made up of: no rotation for BoardRotation, BodyRotation, and BoardFlip.

The next step is to relate a person to tricks. The predicate personKnowsTrick helps define that a person knows a trick. It simply takes a Person and a TrickComponent and relates them to each other (as it is a predicate). If we wanted to say that Jamie knows the trick “Ollie”, we would input into the KB:

```
(personKnowsTrick Jam Ollie)
```

On the converse, we have a predicate personNotKnowsTrick to define that a person does not know a trick. If a person has learned 5 tricks, then they have n-5 tricks that they do not know.

Furthermore, a predicate personKnowsTrickComponent is defined, which relates a Person to a TrickComponent. This is an important predicate because trick components are used to reason which tricks a person can learn. In other words, this is the knowledge of the skateboarder. A horn clause is defined to break apart a trick such that if a person knows a trick, then the person knows all the trick components that compose of the trick. The horn clause is below:

```
(<== (personKnowsTrickComponent ?person ?trickComponent)
      (personKnowsTrick ?person ?trick)
      (trickContains ?trick ?trickComponent))
```

Each trick is rated by a difficulty rating from 1 to 10. The predicate trickDifficulty is used to relate this. We perform this on every trick. We now have all the information to perform reasoning for a person and what tricks they can learn. This will be explained in the reasoning section.

Reasoning with the represented knowledge

As mentioned previously, this is an app that helps skateboarders progress and learn new tricks by providing personalized trick recommendations based on current experience levels, the tricks they have already mastered, and their desired level of difficulty.

Our reasoning is based on utilizing every trick's trick components. We can reason that a skateboarder would find a good progression of tricks to learn by building on already-learned components. Since we have all the trick components a person already knows, we can reason what tricks a person can learn given that they know these trick components. We have a predicate called personCanLearnTrick and use a horn clause to reason what tricks can be learned. The horn clause takes the person and finds what trick components they know, and if a trick consists of these known trick components, then the trick can be learned. However, this could return tricks that we already know. Thus, we use our predicate personNotKnowsTrick, so that the trick that can be learned must be a trick that the person does not already know. The horn clause is defined below:

```
(<== (personCanLearnTrick ?person ?trick) (personNotKnowsTrick ?person ?trick)
(personKnowsTrickComponent ?person (BoardRotation ?direction ?boardRotation))
(personKnowsTrickComponent ?person (BodyRotation ?direction ?bodyRotation))
(personKnowsTrickComponent ?person (BoardFlip ?flip-direction ?boardFlip))
(trickContains ?trick (BoardRotation ?direction ?boardRotation))
```

(trickContains ?trick (BodyRotation ?direction ?bodyRotation))
(trickContains ?trick (BoardFlip ?flip-direction ?boardFlip)))

For example, if a person knows how to do a BS Shove-it, which is a 180 degree clockwise board rotation, and a Kickflip, which is a 360 degree clounter-clockwise board flip, they should be able to learn a Varial Flip, which builds off of *only* both those components and doesn't contain any new skills.

We can also reason that a skateboarder will have an easier time learning tricks at or below the difficulty level of the most difficult trick they know. Using the lessThanOrEqual, trickDifficulty, and personNotKnowsTrick predicates, we can make a query to find the tricks we don't know that are easier than a certain number. To help skaters know what difficulty level they should be targeting, we provide the user with the highest difficulty level of tricks they know on the frontend interface of our app. Using this number, we can construct the query for Owen, for example, whose highest difficulty rating they know is 4:

(and (personNotKnowsTrick Owen ?trick)
(trickDifficulty ?trick ?difficulty)
(lessThanOrEqualTo ?difficulty 4))

This returns the tricks that Owen does not know which are easier or equal in difficulty to the tricks they already know (4).

Finally, we can find the tricks that we can learn which are easier or equal in difficulty to a certain number. The highest difficulty rating to input is left for the user. This is the query for highest difficulty rating of 4:

(and (personCanLearnTrick Owen ?trick)
(trickDifficulty ?trick ?difficulty)
(lessThanOrEqualTo ?difficulty 4))

Demo

Our demo begins in our app where we upload a csv containing skateboarding tricks with all knowledge associated with them. We then parse the spreadsheet which reveals a list of tricks to select. We enter the name "Owen" then select "Ollie", "Fs 180", and "Kickflip" as the tricks. This reveals a highest difficulty level of 4, which we'll use later in queries. Then we downloaded the .krf files and loaded them into the interaction manager in companions. Once the knowledge is in companions we performed the following queries:

1. *(personCanLearnTrick Owen ?trick)*
   This reveals a list of tricks all of which we can click into to see are composed of skills needed for an ollie, fs 180, or kickflip.
2. *(and (personNotKnowsTrick Owen ?trick)*
   (trickDifficulty ?trick ?difficulty)
   (lessThanOrEqualTo ?difficulty 4))
   This shows all tricks that we don't know under a difficulty level of 4 which was our most difficult trick level.
3. *(and (personCanLearnTrick Owen ?trick)*
   (trickDifficulty ?trick ?difficulty)
   (lessThanOrEqualTo ?difficulty 4))
   Combining those two queries, we can see tricks we can learn that are easier than our most difficult trick.

How to run demo:

1. Extract folder and cd to "KRR-Skateboarding" in terminal
2. Run "npm install"
3. Run "npm run dev" and a web page should open with our app
4. Upload "Skateboarding Tricks.csv" from the folder
5. Parse
6. Enter name
7. Select tricks from list
8. Download KRF files – there are two
9. Upload both KRF files to companions
10. Query the knowledge base using the name you entered in the app