Juggling Animation Language

Chris Bogart  12/5/2007
For CS589

Introduction

The Juggling Animation Language describes three-ball, two-hand juggling patterns. It is an experimental alternative to the Siteswap notation [1], widely known among jugglers. Siteswap represents a juggling pattern as a sequence of integers representing the height of each throw in chronological order. JAL differs from Siteswap in two important respects: 1) It encodes the approximate location of the hand when the ball is caught or thrown, and with what hand gesture (these are abstracted away in Siteswap), and 2) the path of each ball through a pattern is described separately, rather than interleaved sequentially.

While Siteswap can be performed directly from the notation by a skilled juggler, it is cryptic and inaccessible to novice or intermediate jugglers. Significant manipulations must be performed on the notation to understand what a pattern would look like. Furthermore, the subset of patterns readily represented by Siteswap, while enjoyed as a challenge by some jugglers, are relatively uninteresting in performance because changes in hand position are not represented.

Purpose

As a pure notation, JAL is aimed at beginning to intermediate jugglers. The description of a pattern is more or less legible to English speakers, and I hope that organizing pattern descriptions by ball instead of by throw will make the patterns easier to conceptualize.

As a computer-interpreted language, JAL serves the following functions:

- Patterns can be animated
- A single ball’s journey through the pattern can be animated separately
- Patterns can be checked to see that they are physically possible: (balls are not created or destroyed and may not jump from place to place without an explicit transition)
- Patterns can be checked for jugglability by different skill levels:
  - LIMIT5: No throw higher than a 5 may be thrown
  - NOMULTI: Two balls may never be caught at the same time
  - EXPERT: Only the basic physics are checked
- Patterns can be appended, and the physics are checked at the junction between the patterns
- Alternately, incompatible patterns can be appended, with a few throws inserted to make the transition possible.

Use Cases

**Cascade:** Cascade is the most common juggling pattern. The hands take turn throwing one ball each, and the ball is caught by the other hand after exactly two intervening throws.

**Multiplex pattern:** This is an unnamed pattern. It cannot be described in standard Siteswap because it includes a multiplex throw (two balls thrown by the same hand at the same time) and an elevated overhand catch. There is an extended Siteswap notation that can represent the multiplex throw, but the claw catch would not be represented. The pattern runs like this:

- Start with one ball (A) in the left hand and two in the right (B,C)
- Throw (A) towards the right hand
While (A) is in the air, throw (B) and (C) directly upward with the right hand
• Catch (A) in the right hand, and throw it back towards the left
• Claw (B) overhand in the left hand, hovering over the right hand
• Catch (C) in the right hand
• Catch (A) in the left hand back on the left side (it is still holding (B))

Basic Objects

**Ball**: Obviously could represent any juggled object. In Siteswap notation, balls are not represented explicitly; in fact, the number of balls in a pattern has to be calculated indirectly. In JAL there are always three balls, named Redball, Greenball, and Blueball.

**Hand**: Only two-handed patterns are allowed.

**Tick**: Time is defined in units of ticks, defined as the period of the fastest sequence of throws in the pattern.

**Areas**: We divide the juggle space into six areas: Upper Left, Lower Left, Upper Right, Lower Right, Upper Center, and Lower Center.

**Height**: JAL borrows from Siteswap the notion of height, represented as a small integer corresponding to the proportional time between a throw and a catch. Since higher throw takes longer, height and time are related. In Siteswap, even numbered heights mean throws to the same hand; in JAL this is not necessarily true. But typically: 1 is a fast transfer between hands; 2 could be a quick pop in one hand, but is more typically just a held ball; 3 is the typical cascade throw; 4 is the normal throw for two-in-one-hand juggling; and 5 is the throw needed for a 5-ball cascade.

**Grip**: we define grip as a hand (left or right), holding a ball, in one of the six areas, with a hand shape. This is not a standard concept per se, but a convenient grouping to simplify the syntax

**Hand Shape**: Cup (cupping the ball from underneath) or Claw (grabbing the ball from above). Others exist in the real world, and could be added.

**Events**: The following events are defined

- **Throw**: a throw represented as a Grip object (a hand, a ball, an area, and a hand shape). The direction and speed of the throw are not specified, but are determined by the time and location of the later catch of the same ball. A throw takes up one time unit.
- **Catch**: Has the same parameters as a throw. A catch in a pattern does *not* advance the Tick clock. It happens half a tick before the time when it appears in the pattern.
- **Rethrow**: Semantically identical to a Catch followed by a Throw of the same ball and the same hand, but possibly in different areas or with different hand shapes. Takes up one time unit.
- **Tick**: Represents the passing of one Tick.
- **Hold**: Represents a hand holding a ball for one tick
- **HandAt**: Represents a hand being physically present at an area at a certain time. Hold and HandAt are generated automatically in certain circumstances to indicate to the animation engine where the objects are, when starting, ending, or connecting patterns.
- **Simul**: Indicates that two events happen simultaneously. The time that passes is the maximum of the two events. *It is deprecated as an end-user construct*: use `||` instead (see Combinators below)
- **Seq**: Indicates that two events happen sequentially. *Deprecated*: use a list of events instead.

**Pattern**: a list of events `[(Event)]`.

**Syntax**

JAL is a Haskell DSEL, with the following type definitions and smart constructors:
type Pattern = [Event]
data Ball = Redball | Greenball | Blueball deriving (Eq, Show)
data Shape = Cup | Claw deriving (Eq, Show)
data Grip = Grip Hand Area Ball Shape deriving (Eq, Show)
data Then = Then deriving (Eq, Show)
data Event = Throw Grip | Catch Grip | Rethrow Grip Then Grip | Tick Integer | Simul Event Event | Seq Event Event | Hold Grip | HandAt Hand Area Shape deriving (Eq, Show)
type Height = Integer
data Hand = LeftHand | RightHand deriving (Eq, Show)
data Direction = Up | Down | Leftward | Rightward deriving (Eq, Show)
data Area = HiLeft | HiRight | HiCenter | LoLeft | LoRight | LoCenter deriving (Eq, Show)

allballs = [Redball, Greenball, Blueball]

-- Smart constructors
leftgrip b = Grip LeftHand LoLeft b Cup
rightgrip b = Grip RightHand LoRight b Cup
centergrip b = Grip RightHand LoCenter b Cup
tick = ticks 1
ticks n = [Tick n]

**Fundamental Combinators**

A few basic combinators are necessary to begin building and animating patterns:

(+++) :: Pattern -> Pattern -> Pattern
Patterns can be appended using the normal list concatenation operator. However the user is responsible for ensuring that they form a valid pattern.

(|||) :: Pattern -> Pattern -> Pattern
The parallel combinator allows two patterns to happen in parallel.

run :: Pattern -> IO ()
This function checks a pattern for validity, then animates it.

**Examples**

With these combinators we can build any juggling pattern the system allows. Some examples follow:

**Cascade pattern:** We first define a single ball arc, with a two tick delay between the throw and the catch. Then we derive the cascade pattern recursively by successively overlaying the same pattern, delayed by one tick, mirrored left to right, and rotating appropriately through the three balls.
arc :: Ball -> Pattern
arc b = [Throw (leftgrip b),
        Tick 2,
        Catch (rightgrip b)]

mk_cascade a b c = arc a ||| (tick ++ mirror (mk_cascade b c a))
cascade = mk_cascade Redball Greenball Blueball

**Multiplex pattern:** To describe this pattern, we lay out each ball's journey separately, then compose them as simultaneous streams of events.

ball1 b = [Throw (leftgrip b),
           Tick 2,
           Catch (rightgrip b),
           Throw (rightgrip b),
           Tick 2,
           Catch (leftgrip b)]
ball2 b = [Throw (Grip RightHand LoCenter b Cup),
           Catch (Grip LeftHand HiRight b Claw)]
ball3 b = [Throw (Grip RightHand LoCenter b Cup),
           Tick 1,
           Catch (Grip RightHand LoCenter b Cup)]
multi = ball1 Redball ||| ( [Tick 2] ++ (ball2 Greenball ||| ball3 Blueball))

**Mills' Mess:** This is essentially the cascade pattern, but the hands move into different parts of the juggle space, and return to their starting position after two cascade cycles.

millsl r = [Throw (Grip LeftHand LoRight r Cup),
            Tick 2,
            Catch (Grip RightHand HiCenter r Cup),
            Throw (Grip RightHand LoLeft r Cup),
            Tick 2,
            Catch (Grip LeftHand HiCenter r Cup)]
mills2 b = [Throw (Grip RightHand LoRight b Cup),
            Tick 2,
            Catch (Grip LeftHand HiCenter b Cup),
            Throw (Grip LeftHand LoLeft b Cup),
            Tick 2,
            Catch (Grip RightHand HiCenter b Cup)]
mills3 g = [Throw (Grip LeftHand LoRight g Cup),
            Tick 2,
            Catch (Grip RightHand LoRight g Cup),
            Throw (Grip RightHand LoLeft g Cup),
            Tick 2,
            Catch (Grip LeftHand LoLeft g Cup)]
mills1r = mills1 Redball ++ mills1r
mills2r = mills2 Blueball ++ mills2r
mills3r = mills3 Greenball ++ mills3r
mills = mills1r ||| (Tick 1: mills2r) ||| (Tick 2: mills3r)
Advanced Combinators

The other available combinators are built on (or could in principle be built on) the basic combinators:

`permute :: (Ball -> Ball) -> Pattern -> Pattern`
A permutation function, mapping each ball to a different ball, is applied to an entire pattern.

`mirror :: Pattern -> Pattern`
All Hands and Areas in a pattern are swapped left to right.

`(+++ :: Pattern -> Pattern -> Pattern)
This combinator checks to see which balls are in the left and right hands, and generates extra throws and catches to synchronize the two patterns, before concatenating them. Unlike ++, this operator guarantees that the resulting pattern will be valid, if its arguments are valid.

`(+|+) :: Pattern -> Pattern -> Pattern`
This third way to append patterns, uses not only extra throws and catches, but can also mirror or permute the second pattern in its efforts to synchronize the two

`oneBall :: Ball -> Pattern -> Pattern`
Creates a new pattern consisting only of those events relating to one of the balls; the timing and positions are not changed

`add_intro :: Pattern -> Pattern`
Starts the hands in a standard position, holding all three balls, before a pattern, using the +|+ operator to synchronize it.

`add_finale :: Pattern -> Pattern`
Ends a pattern by adding catches into a standard position, using the +|+ operator

`bracket :: Pattern -> Pattern`
Adds a start and end position using add_intro and add_finale

`claw :: Pattern -> Pattern`
Turns all catches in a pattern into high claws.

`siteswap :: [Integer] -> Pattern`
Builds a pattern from siteswap notation: for example the cascade pattern is just siteswap [3]. Some other interesting siteswaps are [4,4,1], [4,2,3], and [5,1]. This function only handles three-ball siteswaps (the average of the listed numbers must be three; and there are other constraints as well).

Examples

`multi ++ multi`
Append two "multi" patterns one after the other. When executed this will throw an error; the balls are not in the right position at the end of the first to begin the second.

`multi +++ multi`
Multi's only effect is to move the green ball from the right to left hands, so this will perform the multi pattern, throw the green ball from left to right, then perform multi again.

`multi +|+ multi`
Multi starts with two balls in the right hand and one in the left; and ends with two in the left hand and one in the right. This means that after the trick is performed once, the hands hold the right number of balls to do a mirror image of the trick, ignoring ball color. This is what +|+ can do, and the result is a
pretty smooth transition.

add_finale $ (take 5 cascade) +|+ claw (take 15 mills)

This sequence will start with the first few throws of a cascade pattern, then segue into 15 throws of a clawed Mills' Mess, and finally catch all the balls before terminating.

**Type Checking**

When combining patterns, it is useful to have a notion of type, that lets us know if we can concatenate two matching patterns, or if they do not match and we need to coerce them. We represent the left and right hands' contents at a point in time as a pair of lists of balls.

```
type HandContents = ([Ball],[Ball])
```

with the functions `preHold` and `postHold` returning the `HandContents` before and after the pattern, respectively. The function `signature` returns the states before and after a pattern:

```
*Main> signature multi
([Redball],[Greenball,Blueball]) --> ([Redball,Greenball],[Blueball])
```

The `preHold` and `postHold` functions are used internally by combinators such as `+|+`, `+++`, and `bracket`.

**Physics Checking**

JAL has a flexible physics checking tool, which validates not only the basic linear logical sanity of a pattern, but beyond that allowing us to define and check physical constraints that may be relevant to different jugglers. A sane pattern is one in which:

1. throws and catches of a ball alternate
2. a ball caught in one hand is thrown from that same hand
3. a hand does not throw two balls in different areas of the space at the same time
4. a hand does not catch two balls in different areas of the space at the same time

Note that it is implicit in the notation that a throw and a catch cannot happen at the same time: the catch is assumed to happen half a Tick before any "simultaneous" throw. In the real world a simultaneous throw and catch is possible, but this is not expressible in JAL, except globally by setting the animator's dwell time to zero.

Additionally, juggler-specific constraints may be defined:

- `minHandMove`: a hand may not move from one area to another in less than this many milliseconds. This is mostly useful for guaranteeing rules 3 and 4 above.
- `minBallArc`: a ball may not fly for less than this many milliseconds (prevents very fast throws)
- `maxBallArc`: a ball may not fly for more than this many milliseconds (prevents very high throws)
- `multiCatch`: if False, then two balls may not be caught by the same hand at the same time (this is a difficult maneuver for a beginning juggler)

These constraints are specified for checking with the following data structure:
data Criterion = Criterion {  
    minHandMove :: Time,  
    minBallArc  :: Time,  
    maxBallArc  :: Time,  
    multiCatch  :: Bool }  

...and the following criteria are predefined:

expert = Criterion (dwell-5) (ticklen-dwell) ticklen*20 True
nomulti = Criterion (dwell-5) (ticklen-dwell) ticklen*20 False
limit5  = Criterion (dwell-5) (ticklen-dwell) (5*ticklen-dwell) False

All constraints are checked with the check function:

check :: Criterion -> Pattern -> IO ()

and implicitly by the run function, which does not start the animation if the pattern does not at least validate to the “expert” level.

Internal Functions and Datatypes

The run function is defined in terms of three internal functions: check, sem1, and sem2. See above for a description of check. The sem1 function converts the pattern into a list of Transit objects. A Transit object represents a hand or ball transitioning from one type of fluid motion to another, at a particular area of the juggle space, at a particular point in time. Hands always move linearly; balls may move linearly when carried by a hand, or in parabolic arcs when thrown. (When and Where fields are misspelled to avoid clashing with Haskell keywords).

data Offset = Offset Direction | NoOffset deriving (Eq, Show)
data Transit = Transit {  
    what   :: Object,  
    whan   :: Time,  
    whare  :: Area,  
    offset :: Offset,  
    from   :: MoType,  
    to     :: MoType,  
    shape  :: Shape } deriving (Eq, Show)
data MoType = Arc | Glide | Carry Hand deriving (Eq, Show)
data Object = ObHand Hand | ObBall Ball deriving (Show, Eq)

The sem2 function, in turn, takes this list of Transit objects, separates them into matching sequential pairs (describing the start and end of a motion), and from those pairs creates a list of Trace objects. Trace objects are defined in the Animator module, as described in the next section.

The check and postHold functions described above are actually implemented in terms of Transit lists.

Animator Module

The animation runs off a list of objects of type Trace:

data Trace = Trace {  
    fromT :: Time,  
    toT  :: Time,  
    anim :: (Time -> Graphic),  
    desc :: String }  

where each item contains a time span and a function drawing a graphic as a function of a time value. The list of Traces is assumed to be sorted by start times (fromT). The animation engine, pausing to wait for a clock pulse, takes the items off the top of the list whose start time has passed, draws (for the current time) all the graphics of these items if their end time (toT) has NOT passed, and then
recursively passes itself a new list of items discarding those whose end time has passed.

Three prefabricated animations are provided: a hand, represented by a capital 'E' shape, moving at a constant velocity, and possibly rotating; a ball moving at a constant velocity, and a ball following a parabolic path with a fixed downward acceleration. These are constructed with parameters stating the length of the animation segment and the desired start and end coordinates and rotation.

Gravity and animation granularity are currently fixed parameters in the Animator.hs module that need to be changed in the source code. Because the list of Traces is built from absolute system clock time values, the smoothness of the animation can be adjusted to allow it to keep up on a slower machine.

**Considerations during development**

The original design for the syntax followed the Siteswap tradition of constructing a pattern sequentially by describing each throw in turn, not distinguishing the balls. I decided to change that after getting a comment in class that "you should have made the balls different colors on the slide", and I realized that tracing balls through a pattern was a more natural way to understand a pattern.

Describing the ball paths separately required the addition of Ticks: these represents points in time when something is happening to another ball, but the current ball is either in the air or being held. Ticks were originally redundant, since the height of each throw used to be specified, and determined the number of necessary ticks. I think a description with Ticks is easier to read, like rests in music. The throw height was later dropped with the Velocity object, as explained below.

Another change along the way was the invention of Grip and Veloc objects. Originally I had Throw and Catch statements with a lot of parameters, and the "Introduce Parameter Object" refactoring suggested itself [2]. Grouping related parameters allows reuse of common objects, as well as data definitions that are easier to understand. The Veloc object encompassed the speed (i.e. time until next catch) and direction of a throw. This was later eliminated.

The Rethrow, HandAt, and Hold objects are used internally, and while they are available to language users, they are more useful when generated automatically by the combinators.

Ball permutation was added to meet the need of the synchronized append operator +|+.

The Transit type changed significantly as I worked out exactly what information was needed to do the animation and physics checking. Originally I tried to only track time between events when linearizing and timestamping the event sequence; I deduced which balls were in the air by examining that list during the animation process. The biggest problem with this was that if a hand moved around, throwing and catching one ball, while holding another one at the same time, that ball would not follow the hand. I had no way to generate traces of that ball's motion; it was not reflected explicitly in the throw/catch events I was processing. So I had to add another bit of state between processing of events, namely the contents of both hands. This became messy enough that I needed to learn to use a state monad, which cleaned up my code significantly.

The velocity indicator on throws was removed as redundant information that limited combination. The original idea was that you would know from the throw what direction it was thrown in and how hard, and that that was important to a user. However in the end I think that information is specified more clearly, and less redundantly, by the language’s recommended (but not required) style of specifying ball paths separately, then combining them as parallel operations. We have

```
Throw (leftgrip Redball), Tick2, Catch (rightgrip Redball)
```

as a fairly terse string of events, that I think conveys the information just as well as the previous notation of

```
Throw (leftgrip Redball) (Veloc 3 Rightward), Tick 2, Catch (rightgrip Redball).
```
My intuition is that a person really internally represents a throw in terms of where it ends up, rather than how hard and in what direction they throw it to get it there. We can throw a ball to hit a target fairly accurately, but without a lot of practice we probably cannot choose to throw it at a particular speed.

My animation module is fairly limited to the needs of the task at hand. I was inspired by FRAN and would have liked to use it, but it does not compile under GHC, and fixing it or reimplementing it was beyond the scope of the project. My design is influenced by FRAN [3], but much simpler; FRAN code involves a significant amount of infrastructure to make the combinators work so cleanly, and I simply didn’t need that much flexibility.

Future Work

The animation library I used (SOE, aka HGL) is buggy on the Mac OS X version of GHC, and is not even included in GHC 6.8. It could be that moving to OpenGL would be more congruent with the current direction of the GHC compiler, but there was not time to migrate to OpenGL by the time I realized it would be a portability problem.

Given more time, I would also like to make ||| into a true constructor :|||, so that patterns would be stored exactly as they are specified, and patterns could be pretty printed. As it is, the ||| smart constructor adds a lot of simultaneous Tick 1 events, decomposes Tick N events, and packages up Catch and Throw events so as to make each Pattern list element one Tick long. This is convenient internally but makes pretty printing problematic.

Infinite patterns are not currently handled; the code currently chops off patterns after 100 Ticks to prevent infinite loops; this could probably be fixed fairly easily, and combinators like add_finale would have to truncate the pattern.

An Offset parameter is in place but not implemented: this could allow finer placement of hands for certain tricks (though I do not have a use case for this), or slight movement of hands to simulate throwing and catching motions.

References

