Dancing Sticks

By

Sheela Surisetty

Rahul Gopinath

INTRODUCTION

This project specifies a language for animated stick figures. A stick figure is composed of lines some of which are joined together. Our language provides the following features. We can create figures that are composed of simpler figures, and while composing; we can make use of predefined figures. We can also specify a sequence of operations on the stick figures called moves. A move may be a simple movement of a figure or part of a figure or may be composed of smaller moves or can make use of predefined moves. The move also has the capability to add and delete figures and parts of figures. We also provide a type checker for this language. The type checker verifies that the sequence of operations is well typed. Finally we provide an implementation of the animated stick figures using the Graphics.Gloss API.

Specifics

A Stick figure is a collection of sticks. The simplest figure is a point that is a tuple of symbol and offset. Construction of complex figures is accomplished by adding more points and links between them. We call these links relations. A move can be applied to a stick figure (figure). A move may be a simple move, which specifies how a figure as a whole moves or composed of smaller moves that specify how each part moves. Different points in the stick figure are identified by the symbols (name) attached to them.

A stick figure: (output is shown in fig: 1)

```
{man {leg    {left_leg  {a  (0,0)}, {b (1,1)} [a b]},
             {right_leg {a  (2,0)}, {b (1,1)} [a b]}},
      {arm   {left_arm  {d  (0,2)}, {e  (1,2)},  [d e]},
             {right_arm {d  (2,2)},{e  (1,2)},  [d e]}},
             {body {g (1,3)}, {e (1,2)}, {b (1,1)} [g e] [e b]}}
```

Dance sequence: (output is shown in fig: 1,2)

```
[moves [move [man arm arm_left] (0,1),
        move [man arm arm_right] (0,1),
        move [man leg left_leg] (0,1),
        move [man leg right_leg] (0,1)]]
```

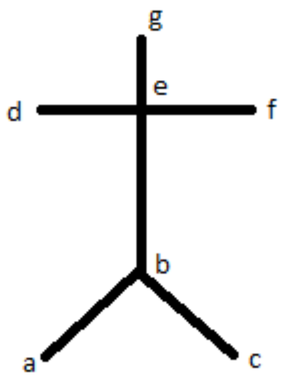

fig: 1

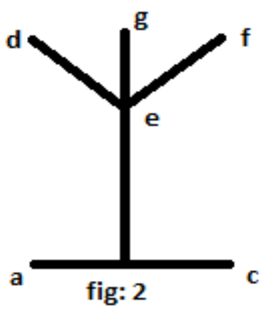

fig: 2

SYNTAX

Note that we represent all non-terminals in bold.

offset:

An *offset* is represented by two integers ie. x position and y position. It is identified by the *symbol* (name) attached to it. This specifies the coordinates of a point. From now on, we use point to denote the tuple containing symbol and offset. Also note that when we speak of moving the point, we mean that the symbol remains the same but the offset is changed to the new coordinates.

offset *::= (i,i)*

Ex: (1,2), (3,4), (5,6), etc.

A symbol is a string that represents the names of offsets, figures, moves, list of figures, list of moves, etc.

symbol = string

relation:

There is only one kind of relation. A link. A *link* is a connection between two points – drawn as a line.

relation *::= [symbol symbol]*

Ex: A *relation* connecting two points a (x,y), b (p,q) is represented as [a b]

figure:

A figure is represented as a collection of offsets and their symbols and/or figures, relations and their symbols.

```
figure ::= symbol   offset
         | symbol  figure* relation*
         | getfig symbol     -- this returns the figure that is associated to the symbol (name)
```

Ex: {left_leg {a (0,0)}, {b (1,1)} [a b]}

Given definition:
    def t1 = {triangle {a (0,0)}, {b (1,1)}, {c (1,0)} [a b] [b c] [c d]}

Using the definition:
    {t_tail {getfig t1} {e (10,10)} [d e] }

frame

A frame is a sequence of figures. Our animation is done using frame. The frame is a sequence of figures that is already captured. Executing the current move against a frame looks at the last figure in the frame, applies the move against it, and adds the newly generated figure at the end. This way after a sequence of moves is evaluated against a frame, the animation sequence is obtained.

frame *::=* ***figure****

***move**:*

A move can be applied to the entire figure, a single symbol (name) or a list of symbols.

```
move ::= move: symbol* by: offset
           | moves: move*
           | getmove symbol          -- it returns a predefined move
           | add symbol* figure      -- adds figure at the symbol* path given
           | del symbol*             -- deletes figure at the symbol* path given
```

Ex:

```
 move [arm arm_left] (0,3)     -- single move

moves [move [arm arm_left] (0,3),
          move [leg, leg_right] (0,1)]   -- compound move
```

mseq:

An mseq is a sequence of moves. It is used along with frame to create animation. Each move in mseq generates a figure, which is captured (explained at frame).

mseq** ::= **move*

***def**:*

A def is used for defining/assigning names to figures and/or sequence of moves.

```
def ::= def symbol = figure
         | def symbol = move*
```

Ex:

Defining a name for the figure:

def t1 = {triangle {a (0,0)}, {b (1,1)}, {c (1,0)} [a b] [b c] [c d]}

Defining a set of moves:

def m1 = move [arm left_arm] (0,3)

***defs**:*

defs is a collection of def's

defs** ::= **def*

SEMANTICS

Judgment [DEF]

Given the definitions of figures/moves and a definition of a figure/move, they evaluate to definitions of figures/moves.

defs:def (def.eval)> defs ⊆ defs ×def ×defs

<u>Semantics of ***def***</u>

[DEFS-DEF]

defs:def (def.eval)> def ∪ defs

Judgment [MOVE]

Given the definitions of figures/moves and a move, they evaluate to a move.

defs: move (mov.eval)> move ⊆defs ×move ×move

<u>Semantics of ***move***</u>:

[MOVE-BY-OFFSET]

defs:move:symbol by:offset (mov.eval)>move:symbol by:offset

[MOVE-LIST]

defs: m1 (mov.eval) m1' .. defs:mn (mov.eval)> mn'
----
defs:moves:m1 ..mn (mov.eval)> moves:m1'..mn'

[MOVE-GETMOVE]

defs = {symbol = move'}
----
defs: getmove: symbol (mov.eval)>move'

Judgment [FIGURE]

Given the definitions of figures and a figure, they evaluate to a figure.

defs: figure (fig.eval)> figure ⊆defs ×figure ×figure

<u>Semantics of ***figure***</u>:

[FIG-OFFSET]

defs: (symbol offset) (fig.eval)> (symbol offset)

[FIG -FIGREL]

$$\frac{\text{defs: f1 (fig.eval)> f1'} \quad .. \quad \text{defs:fn (fig.eval)> fn'}}{\text{defs:(symbol [f1 .. fn] [r1 .. rn]) (fig.eval)> (symbol f1'..fn' [r1 .. rn])}}$$

where f1 .. fn corresponds to the list of figures and r1 .. rn corresponds to the list of relations (sticks).

[FIG -GETMOVE]

It returns the figure that is attached to the symbol.

$$\frac{\text{defs=\{symbol = figure'\}}}{\text{defs: getfig: symbol (fig.eval)> figure'}}$$

Judgment [MOVE FIGURE]

Given the definitions of figures/moves, figure and a move, they evaluate to a figure. That is, when a move is applied to a figure, it evaluates to a figure.

defs: (figure, move) (fig.move)> figure ⊆defs ×figure ×move ×figure

<u>Semantics of ***figure and move***:</u>

[FIG-OFFSET-MOVE-BY-OFFSET]

This moves the symbol offset to a new position. That is from position *offset* to a new position *offset"*, the amount of move is given by *offset'*.

The operation <+> is vector addition given by (x,y) <+> (p,q) = ((x+p),(y+q))

$$\frac{\text{offset'' = offset <+> offset'}}{\text{defs:(symbol offset), (move: by: offset') (fig.move)>(symbol offset'')}}$$

[FIG-FIGREL-MOVE-BY-OFFSET]

When a move with empty list of symbols is applied to a figure it moves the complete figure (applies to all sub-figures.)

$$\frac{\begin{array}{c}\text{defs:f1 (fig.eval)> f1'} \quad \text{defs:f1' move: [] o (fig.move)> f1'' ...} \\ \text{defs:fn (fig.eval)> fn'} \quad \text{defs:fn' move: [] o (fig.move)> fn''}\end{array}}{\text{defs: (fig s f1..fn rel) (move: [] o) (fig.move)> fig s f1''..fn'' rel}}$$

[FIG-GETFIG-MOVE-BY-OFFSET]

The figure that is returned by *getfig* operation is moved by the *offset* (ie. *offset'*) that is specified in the move operation.

```
defs:(getfig s) (fig.eval)> f   defs:f move: [] o (fig.move)> f'
--------------------------------------------------------------------------------------------------
defs: (getfig s) (move: [] o) (fig.move)>  f'
```

[FIG-FIGREL-MOVE-SYMBOLS-BY-OFFSET]

The sub-figure indicated by move path is moved by offset. Note that we don't have to address the main figure (move path only begins from s2).

We define the operator symbolof this way

symbolof (symbol offset) = symbol

symbolof (symbol figures* relations*) = symbol

```
symbolof fi = s1    fi (fig.eval)> fi'   fi', move [s2..sn] (fig.move)> fi''
---------------------------------------------------------------------------------------------------------
defs: (sym [f1..fi..fn] rels) (move: [s1..sn] o) (fig.move)> (sym [f1..fi''..fn] rels)
```

[FIG-GETFIG-MOVE]

The figure looked up in definition by getfig is moved. The figure has to be predefined by using the definitions.

```
def: (getfig: sym) (fig.eval)> f'   f',m (fig.move)> f''
-------------------------------------------------------------------------------
defs: (getfig: sym ) m (fig.move)> f''
```

[FIG -MOVE-GETMOVE]

Here sym is used to look up a move to apply.

```
defs:(getmove: sym) (mov.eval)> m'   f (fig.eval)> f'  f',m' (fig.move)> f''
---------------------------------------------------------------------------------------------------
defs: f (getmove: s) (fig.move)> f''
```

[FIG -MOVE-ADD-EMPTY]

We can add a figure or a sub-figure. If the path is empty, it adds a figure that is a subfigure to the current figure.

```
defs: (sym figs rels) (add: [] fig) (fig.move)> (sym fig:figs rels)
```

[FIG -MOVE-ADD]

This adds the new figure at the indicated path.

symbolof fi = s1   fi (fig.eval)> fi'   fi' add: [s2..sn] (fig.move)> fi''
----
defs: (sym [f1..fi..fn] rels) (add: [s1..sn] fig) (fig.move)> (sym [f1..fi''..fn] rels)

[FIG -MOVE-DEL-EMPTY]

An empty delete does not do anything.

defs: fig (del: []) (fig.move)> fig

[FIG -MOVE-DEL]

But a non-empty delete deletes the specified figure. This is the base case with one single path element for delete.

symbolof fi = s1
----
defs: (sym [f1..fi..fn] rels) (del: [s1]) (fig.move)> (sym [f1..fi-1,fi+1..fn] rels)

[FIG -MOVE-DEL-LIST]

Drilling down with multiple path elements for delete.

symbolof fi = s1 fi (fig.eval)> fi'   fi' del: [s2..sn] (fig.move)> fi''
----
defs: (sym [f1..fi..fn] rels) (del: [s1..sn]) (fig.move)> (sym [f1..fi''..fn] rels)

Judgment [ANIMATION]

Given the definitions of figures/moves, frame and a move, they evaluate to a frame. As explained earlier, the move is applied to the last figure in the frame, which results in a new figure which is captured in the new frame.

defs: (frame, move) (frame.eval)>frame ⊆defs ×frame ×move ×frame

Semantics of ***frame and move***:

[FRAME-EMPTY-MOVE]

An empty list of moves does not add any new figures to a frame.

defs, frame: [f1..fp], [] (frame.eval)> frame: [f1..fp]

[FRAME-MOVE-EMPTY]

When an empty frame is acted upon by a list of moves, it fails to produce any new figures.

defs, frame: [], m (frame.eval)> frame: []

[FRAME-LIST-MOVE]

Given a frame with last figure fp, and it evaluates to fp' and move m1 is evaluated to m1' and fp' is acted on by m1' to produce fp+1 and so on for all the sequence of moves from m1 .. mq to produce all the figures upto fp+q, then all these figures are added to the end of the frame after fp.

```
        defs:fp (fig.eval)> fp'   defs:m1 (mov.eval)> m1'
                   defs:fp',m1' (fig.move)> fp+1
     defs: frame: f1..fp+1 [m2..mq] (frame.eval)> frame [f1..fp+q]
------------------------------------------------------------------------------
     defs, frame: [f1..fp], [m1..mq] (frame.eval)>  frame [f1..fp+q]
```

TYPE SYSTEM

As can be seen from our semantics, our language has the ability to add and delete figures midway through execution. Also moves are specific to figures. What our type system does is to make sure that the sequence of moves is valid on the figures.
For example, if move M is valid on figure F, and moves part f' in figure F, then after deleting f' from F, the same move M is no longer valid on the new figure. What our type system does is to ignore all the offsets, but makes sure that the symbols that each move refers to are valid on the figure that the move will apply to.

Types [FIGURE]

t_figure ::= point_ symbol
  | figure_ symbol **t_figure***

Types [Move]

t_move ::= move_ symbol*
  | add_ symbol* **t_figure**
  | del_ symbol*
  | moves_ **t_move***

Types [Frame]

t_frame ::= frame_ t_figure*

Types [Seq]

t_seq ::= **t_move***

Typing Judgment [FIGURE]

defs ⊢ figure: figure_ symbol t_figure*⊆defs × figure × symbol × t_figure*

[TFIG-OFFSET]

The type of a point is just the symbol

defs ⊢point symbol offset : point_ symbol

[TFIG-RELS]

The type of a figure is its symbol and the list of types of its subtypes.

```
defs |-  f1:t_f1 .. defs|- fn:t_fn
--------------------------------------------
defs |- (s [f1 .. fn] rel) : figure_ s [t_f1 .. t_fn]
```

[TFIG-GETFIG]

If we are looking up a symbol in def, then its type is whatever type the figure that symbol is bound to in def has.

```
{(s=fig:t_f)..}: getfig s : t_figure
-----------------------------------------------
defs |- getfig: s : t_figure
```

Typing Judgment [MOVE]

The type of a move is just the symbol path it has. Or it is the list of types of all the sub-moves.

defs ⊢ move: move_ symbol* ⊆defs ×move × t_move ×symbol

[TMOVE-OFFSET]

The type of a simple move with a list of symbols is move_ with list of symbols

defs ⊢move: symbol* by: offset : move_ symbol*

[TMOVE-LIST]

The type of a compound move is composed of the type of its sub-moves.

```
defs ⊢m1 : t_m1 .. defs ⊢ mn : t_mn
--------------------------------------------
defs ⊢moves: [m1 .. mn] : moves_ [t_m1 .. t_mn]
```

Typing Judgment [MOVE FIGURE]

Now the main part is combining both figures and moves. A move is valid on a figure if the path described by the move describes some element in the figure the move applies to. More over the type of a move applied to a figure is another figure the type of which can be found from original figure and move.

defs ⊢ figure: t_figure, move: t_move : t_figure ⊆ defs ×figure×move ×t_figure

[TFIG-TMOVE]

```
figure:t_figure   move:t_move t_figure,t_move = t_figure'
           ----------------------------------------------
                 defs |- figure,move : t_figure'
```

PROTOTYPE

The DSL describes how a figure can be moved given a list of moves. This produces a list of figures each of which shows one particular instance during movement. Our prototype takes these figures and converts the coordinates and relations to lines and points and draws them on the screen.
Installation:
Installation requires the Haskell gloss API. It can be installed through cabal.
http://hackage.haskell.org/package/gloss-1.0.0.1

Example input and output

```
{man {leg    {left_leg  {a  (0,0)}, {b (1,1)} [a b]},
             {right_leg {a  (2,0)}, {b (1,1)} [a b]}},
      {arm    {left_arm  {d  (0,2)}, {e  (1,2)},  [d e]},
             {right_arm {d  (2,2)},{e  (1,2)},  [d e]}},
             {body {g (1,3)}, {e (1,2)}, {b (1,1)} [g e] [e b]}}
[moves [move [man arm arm_left] (0,1),
   move [man arm arm_right] (0,1),
      move [man leg left_leg] (0,1),
        move [man leg right_leg] (0,1)]]
```

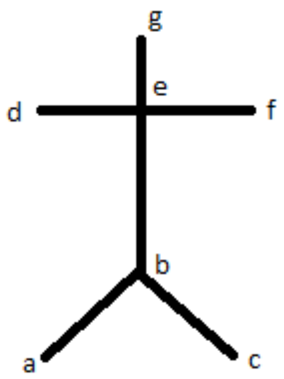

fig: 1

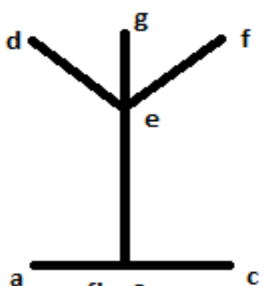

fig: 2

Limitations:

We did not address the relations by paths; instead we named them using points. This will lead to problems when two points similarly named need to be linked together (like the end points of two arms).

A second limitation is that our def syntax does not allow creating templates where the offsets of each point could be given as a parameter. Without templates, getfig is of limited utility.

A third limitation is that we ignore any restrictions on the length of sticks. Thus a stick may elongate or shorten after a move.

Code:

```
> module Dance where
> import Graphics.Gloss
> type Offset = (Integer,Integer)
> type Symbol = String
> data Relation = Link Symbol Symbol
>               deriving (Show, Eq)
> data SFigure = SPoint Symbol Offset
>             | SFigure Symbol [SFigure] [Relation]
>             | GetFigure Symbol
>               deriving (Show, Eq)
> data Move = Move [Symbol] Offset
>          | Moves [Move]
>          | GetMove Symbol
>          | Add [Symbol] SFigure
>          | Del [Symbol]
>               deriving (Show, Eq)
> type MSeq = [Move]
> data SDef = SDefMove Symbol MSeq
>          |  SDefFig Symbol SFigure
>            deriving (Show, Eq)
> type SDefs = [SDef]
```

SEMANTICS

===

Showing the definition semantics (part).
```
> defEval :: SDefs -> SDef -> SDefs
> defEval defs def = (def:defs)
```
Showing the move eval semantics for getmove.
```
> movEval (defs, GetMove s) = Moves m'
>        where m' = defs <^-> s
```
Showing the fig eval semantics for getfig.
```
> figEval (defs, GetFigure s) = f'
>       where f' = defs <^#> s
```
Applying move to figure. empty move list means the entire object moves.
```
> figMov ::(SDefs, SFigure, Move) -> SFigure
> figMov (defs, SFigure s figs rels, Move [] o) = SFigure s figs'' rels
>      where figs' = map (\f -> figEval (defs, f) ) figs
>           figs'' = map (\f -> figMov (defs, f, Move [] o) ) figs'
```
Or we can move only a part of the object.
```
> figMov (defs, SFigure s_ fs rel, Move (m:ms) o) = SFigure s_ fs' rel
>      where fs' = map (\f -> case (sym (defs,f)) == m of
>                           True -> figMov (defs, fe f,Move ms o)
>                           False -> f
>                  ) fs
>           fe f = figEval (defs,f)
```
Showing the application of a compound move.
```
> figMov (defs, fig, Moves (m:ms)) = f''
>      where f' = figMov (defs, fig, m)
>           f'' = figMov (defs, f', Moves ms)
```
Add a figure
```
> figMov (defs, SFigure s_ figs rels, Add [] fig) = SFigure s_ (fig:figs) rels
> figMov (defs, SFigure s_ fs rel, Add (m:ms) fig) = SFigure s_ fs' rel
>      where fs' = map (\f -> case (sym (defs,f)) == m of
>                           True -> figMov (defs, fe f,Add ms fig)
>                           False -> f
>                  ) fs
>           fe f = figEval (defs,f)
```
Delete a figure
```
> figMov (defs, SFigure s_ fs rel, Del [m]) = SFigure s_ fs' rel
>      where fs' = filter (\f -> sym (defs,f) /= m) fs
> figMov (defs, SFigure s_ fs rel, Del (m:ms)) = SFigure s_ fs' rel
>      where fs' = map (\f -> case (sym (defs,f)) == m of
>                           True -> figMov (defs, fe f,Del ms)
>                           False -> f
>                  ) fs
>           fe f = figEval (defs,f)
```

Main engine - frame capture.

```
> frameEval::(SDefs,SFrame, MSeq) -> SFrame
> frameEval (defs, SFrame fr@(f:fs), (m:ms)) = fr'
>       where f' = figEval (defs,f)
>           m' = movEval (defs,m)
>           f_nxt = figMov (defs,f',m')
>           fr' = frameEval (defs, SFrame (f_nxt:fr),ms)
```

Graphics.

```
> draw :: SFigure -> Picture
> draw (SPoint sym (x,y)) = blank
> draw f@(SFigure sym figs rels) = Pictures $ (map (\fig -> draw fig ) figs) ++ (convertRel f)
> mydefs = [SDefMove "ab" [Move ["a"] (1,1)]]
> fig_leg1 = SFigure "leg.left" [SPoint "a" (0,0), SPoint "b" (1,1)] [Link "a" "b"]
> fig_leg2 = SFigure "leg.right" [SPoint "b" (1,1), SPoint "a" (2,0)]  [Link "a" "b"]
> fig_legs = SFigure "legs" [fig_leg1, fig_leg2] []
> fig_arm1 = SFigure "arm.left" [SPoint "d" (0,2), SPoint "e" (1,2)] [Link "d" "e"]
> fig_arm2 = SFigure "arm.right" [SPoint "e" (1,2), SPoint "d" (2,2)]  [Link "d" "e"]
> fig_arms = SFigure "arms" [fig_arm1, fig_arm2] []
> fig_body = SFigure "body" [SPoint "g" (1,3), SPoint "e" (1,2), SPoint "b" (1,1)]  [Link "g" "e", Link "e"
"b"]
> fig_man = SFigure "man" [fig_body, fig_arms, fig_legs] []
> myfig = SFigure "ab" [SPoint "a" (1,1), SPoint "b" (2,2), SPoint "c" (0,0)]  [Link "a" "b", Link "b" "c"]
> myseq = [GetMove "ab", GetMove "ab" , Move ["ab","a"] (1,0),  GetMove "ab" ,Move ["ab","b"] (0,1)]

> dance :: MSeq
> dance1 = [Moves [Move ["arm.left"] (0,-1)]]
> dance = [Moves [Move ["man","arms", "arm.left", "d"] (0,1), Move ["man", "arms", "arm.right", "d"]
(0,1)],
>        Moves [Move ["man", "arms", "arm.left", "d"] (0,-1),
>            Move ["man", "arms", "arm.right", "d"] (0,-1)],Move ["man", "arms", "arm.left", "d"] (0,-1),
>        Add []  fig_triangle,Move ["tri"] (0,-1), Del ["tri"],Add []  fig_sq,Move ["sq"] (0,1), Del ["sq"],
>        Moves [Move ["man", "legs", "leg.left", "a"] (1,0),Move ["man", "legs", "leg.right", "a"] (0,1)],
>        Move ["man"] (1,0)]
> preresult = (mydefs, SFrame [SFigure "main" [fig_man] []], dance)
> result = frameEval preresult
> toPicture:: SFigure -> Picture

> toPicture f = Pictures $ [draw f]
> getFigure::SFrame->Int->SFigure
> getFigure (SFrame f) i = f!!i
> main::IO ()
> main = animateInWindow "Sticks" (450,150) (10,10) white frame
> frame time = do Scale 20 20 $ toPicture $ (getFigure result index)
>           where index = ((truncate time) `mod` 11)
```