Chapter 5

Imperative Construction System

5.1 Overview

The first module is an imperative system for performing geometry constructions. This is the typical input method for generating coordinate-backed, analytic instances of figures.

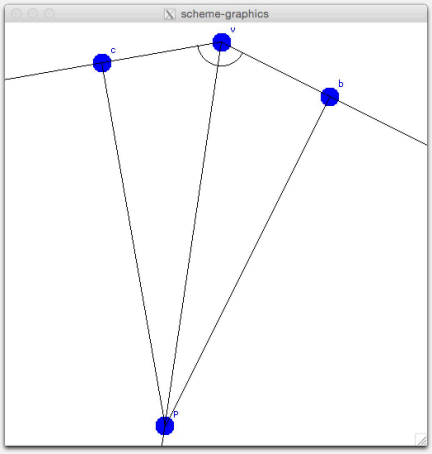
The construction system is comprised of a large, versatile library of useful utility and construction procedures for creating figures. To appropriately focus the discussion of this module, I will concentrate on the implementation of structures and procedures necessary for the sample construction seen in Example 5.1. Full code and more usage examples are provided in Appendix A.

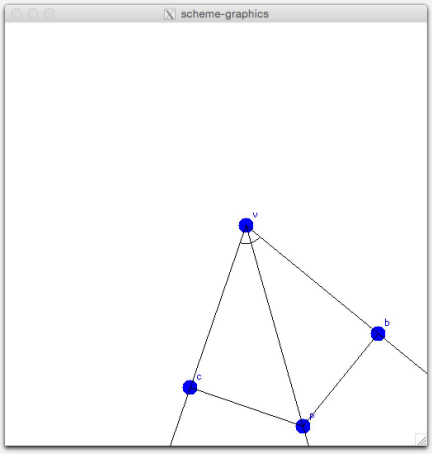
In doing so, I will first describe the basic structures and essential utility procedures before presenting some higher-level construction procedures, polygons, and figures. Then, I will Then, I will explore the use of randomness in the system and examine how construction language macros handle names, dependencies, and multiple assignment of components. Finally, I will briefly discuss the interface and implementation for animating and displaying figures.

Interaction Example 5.1: Sample Construction

```
(define (angle-bisector-distance)
  (let-geo* (((a (r-1 v r-2)) (random-angle))
             (ab (angle-bisector a))
             (p (random-point-on-ray ab))
             ((s-1 (p b)) (perpendicular-to r-1 p))
             ((s-2 (p c)) (perpendicular-to r-2 p)))
     (figure a r-1 r-2 ab p s-1 s-2)))

=> (show-figure (angle-bisector-distance))
```





5.2 Basic Structures

The basic structures in the imperative construction system are points, segments, rays, lines, angles, and circles. These structures, as with all structures in the system are implemented using Scheme record structures as seen in Listings 5.2 and 5.3. In the internal representations, lines and segments are directioned. Predicates exist to allow other procedures to work with or ignore these directions.

Code Listing 5.2: Basic Structures

```
1 (define-record-type <point>
2   (make-point x y)
3   point?
4   (x point-x)
5   (y point-y))
6
```

```
(define-record-type <segment>
  (%segment p1 p2)
  segment?
  (p1 segment-endpoint-1)
  (p2 segment-endpoint-2))

(define-record-type <line>
  (%make-line point dir)
  line?
  (point line-point) ;; Point on the line
  (dir line-direction))
```

Code Listing 5.3: Angle and Circle Structures

```
(define-record-type <angle>
  (make-angle dir1 vertex dir2)
  angle?
  (dir1 angle-arm-1)
  (vertex angle-vertex)
  (dir2 angle-arm-2))

(define-record-type <circle>
  (make-circle center radius)
  circle?
  (center circle-center)
  (radius circle-radius))
```

5.2.1 Creating Elements

Elements can be created explicitly using the underlying `make-*` constructors defined with the record types. However, several higher-order constructors are provided to simplify construction as shown in Listings 5.4 and 5.5. In `angle-from-lines`, we make use of the fact that lines are directioned to uniquely specify an angle. As with the angle construction case, in several instances, we use generic operations to handle mixed types of geometry elements.

Code Listing 5.4: Higher-order Constructors

```
(define (line-from-points p1 p2)
  (make-line p1 (direction-from-points p1 p2)))
```

Code Listing 5.5: Generic Constructors for Creating Angles

```
(define angle-from (make-generic-operation 2 'angle-from))

(define (angle-from-lines l1 l2)
  (let ((d1 (line->direction l1))
        (d2 (line->direction l2))
        (p (intersect-lines l1 l2)))
    (make-angle d1 p d2)))
(defhandler angle-from angle-from-lines line? line?)
```

5.2.2 Essential Math Utilities

Several math utility structures support these constructors and other geometry procedures. One particularly useful abstraction is a `direction` that fixes a direction in the interval $[0, 2\pi]$. Listing 5.6 provides a taste of some operations using such abstractions.

Code Listing 5.6: Directions

```
(define (subtract-directions d2 d1)
  (if (direction-equal? d1 d2)
      0
      (fix-angle-0-2pi (- (direction-theta d2)
                          (direction-theta d1)))))

(define (direction-perpendicular? d1 d2)
  (let ((difference (subtract-directions d1 d2)))
    (or (close-enuf? difference (/ pi 2))
        (close-enuf? difference (* 3 (/ pi 2))))))
```

5.3 Higher-order Procedures and Structures

Higher-order construction procedures and structures are built upon these basic elements and utilities. Listing 5.7 shows the implementation of the perpendicular constructions used in the sample figure.

Code Listing 5.7: Perpendicular Constructions

```
;; Constructs line through point perpendicular to linear-element
(define (perpendicular linear-element point)
  (let* ((direction (->direction linear-element))
         (rotated-direction (rotate-direction-90 direction)))
    (make-line point rotated-direction)))

;;; Constructs perpendicular segment from point to linear-element
(define (perpendicular-to linear-element point)
  (let ((pl (perpendicular linear-element point)))
    (let ((i (intersect-linear-elements pl (->line linear-element))))
      (make-segment point i))))
```

Although traditional constructions generally avoid using rulers and protractors, Listing 5.8 shows the implementation of the `angle-bisector` procedure from our sample figure that uses measurements to simplify construction.

Code Listing 5.8: Angle Bisector Construction

```
(define (angle-bisector a)
  (let* ((d2 (angle-arm-2 a))
         (vertex (angle-vertex a))
         (radians (angle-measure a))
         (half-angle (/ radians 2))
         (new-direction (add-to-direction d2 half-angle)))
    (make-ray vertex new-direction)))
```

5.3.1 Polygons and Figures

Polygons record structures contain an ordered list of points in counter-clockwise order, and provide procedures such as `polygon-point-ref` or `polygon-segment` to obtain particular points, segments, and angles specified by indices.

Figures are simple groupings of geometry elements and provide procedures for extracting all points, segments, angles, and lines contained in the figure, including ones extracted from within polygons or subfigures.

5.4 Random Choices

To allow figures to represent general spaces of diagrams, random choices are commonly used to instantiate diagrams. In our sample figure, we use `random-angle` and `random-point-on-ray`, implementations of which are shown in listing **??**. Underlying these procedures are calls to Scheme's random function over a specified range ($[0, 2\pi]$ for `random-angle-measure`, for instance). Since infinite ranges are not well supported and to ensure the figures stay reasonable legible for a human viewer, `extend-ray-to-max-segment` clips a ray at the current working canvas so a point on the ray can be selected within the working canvas.

Code Listing 5.9: Random Constructors

```
(define (random-angle)
  (let* ((v (random-point))
         (d1 (random-direction))
         (d2 (add-to-direction
              d1
              (rand-angle-measure))))
    (make-angle d1 v d2)))

(define (random-point-on-ray r)
  (random-point-on-segment
   (extend-ray-to-max-segment r)))

(define (random-point-on-segment seg)
  (let* ((p1 (segment-endpoint-1 seg))
         (p2 (segment-endpoint-2 seg))
         (t (safe-rand-range 0 1.0))
         (v (sub-points p2 p1)))
    (add-to-point p1 (scale-vec v t))))

(define (safe-rand-range min-v max-v)
  (let ((interval-size (max 0 (- max-v min-v))))
    (rand-range
     (+ min-v (* 0.1 interval-size))
     (+ min-v (* 0.9 interval-size)))))
```

Other random elements are created by combining these random choices, such as the random parallelogram in Listing 5.10.

Code Listing 5.10: Random Parallelogram

```
(define (random-parallelogram)
  (let* ((r1 (random-ray))
         (p1 (ray-endpoint r1))
         (r2 (rotate-about (ray-endpoint r1)
                           (rand-angle-measure)
                           r1))
         (p2 (random-point-on-ray r1))
         (p4 (random-point-on-ray r2))
         (p3 (add-to-point
              p2
              (sub-points p4 p1))))
    (polygon-from-points p1 p2 p3 p4)))
```

5.4.1 Backtracking

The module currently only provides limited support for avoiding degenerate cases, or cases where randomly selected points happen to be very nearly on top of existing points. Several random choices use `safe-rand-range` (seen in Listing 5.9) to avoid the edge cases of ranges, but further extensions could improve this system to periodically check for unintended relationships and backtrack to choose other values.

5.5 Construction Language Support

To simplify specification of figures, the module provides the `leg-geo*` macro which allows for a multiple-assignment-like extraction of components from elements and automatically tags resulting elements with their variable names for future reference. Listing 5.11 shows the expansion of a simple usage of `let-geo*` and listing 5.12 shows some of the macros' implementation.

Code Example 5.11: Expansion of let-geo* macro

```
(let-geo* (((a (r-1 v r-2)) (random-angle)))
  (figure a r-1 r-2 ...))

(let* ((a (random-angle))
       (r-1 (element-component a 0))
```

```
      (v   (element-component a 1))
      (r-2 (element-component a 2)))
  (set-element-name! a   'a)
  (set-element-name! r-1 'r-1)
  (set-element-name! v   'v)
  (set-element-name! r-2 'r-2)
  (figure a r-1 r-2 ...))
```

Code Listing 5.12: Multiple and Component Assignment

```
(define (expand-compound-assignment lhs rhs)
  (if (not (= 2 (length lhs)))
      (error "Malformed compound assignment LHS (needs 2 elements): " lhs))
  (let ((key-name (car lhs))
        (component-names (cadr lhs)))
    (if (not (list? component-names))
        (error "Component names must be a list:" component-names))
    (let ((main-assignment (list key-name rhs))
          (component-assignments (make-component-assignments
                                  key-name
                                  component-names)))
      (cons main-assignment
            component-assignments))))

(define (make-component-assignments key-name component-names)
  (map (lambda (name i)
         (list name `(element-component ,key-name ,i)))
       component-names
       (iota (length component-names))))
```

Once expanded, a generic `element-component` operator shown in Listing 5.13 defines what components are extracted from what elements (endpoints for segments, vertices for polygons, (ray, angle, ray) for angles).

Code Listing 5.13: Generic Element Component Handlers

```
(declare-element-component-handler polygon-point-ref polygon?)

(declare-element-component-handler
 (component-procedure-from-getters
  ray-from-arm-1
  angle-vertex
  ray-from-arm-2)
 angle?)
```

5.6 Graphics and Animation

The system integrates with Scheme's graphics system for the X Window System to display the figures for the users. The graphical viewer can include labels and highlight specific elements, as well as display animations representing the "wiggling" of the diagram. Implementations of core procedures of these components are shown in Listings 5.14 and 5.15.

Code Listing 5.14: Drawing Figures

```
(define (draw-figure figure canvas)
  (set-coordinates-for-figure figure canvas)
  (clear-canvas canvas)
  (for-each
   (lambda (element)
     (canvas-set-color canvas (element-color element))
     ((draw-element element) canvas))
   (all-figure-elements figure))
  (for-each
   (lambda (element)
     (canvas-set-color canvas (element-color element))
     ((draw-label element) canvas))
   (all-figure-elements figure))
  (graphics-flush (canvas-g canvas)))
```

To animate a figure, constructions can call `animate` with a procedure f that takes an argument in $[0, 1]$. When the animation is run, the system will use fluid variables to iteratively wiggle each successive random choice through its range of $[0, 1]$. `animate-range` provides an example where a user can specify a range to wiggle over.

Code Listing 5.15: Animation

```
(define (animate f)
  (let ((my-index *next-animation-index*))
    (set! *next-animation-index* (+ *next-animation-index* 1))
    (f (cond ((< *animating-index* my-index) 0)
             ((= *animating-index* my-index) *animation-value*)
             ((> *animating-index* my-index) 1)))))

(define (animate-range min max)
  (animate (lambda (v)
             (+ min
                (* v (- max min))))))
```