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.

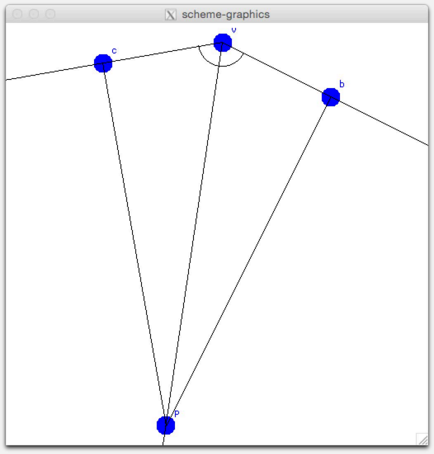
The construction system is comprised of a large, versatile library of useful utility and construction procedures for creating figures. To appropriately focus the discussion of this module, I will concentrate on the implementation of structures and procedures necessary for the sample construction seen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

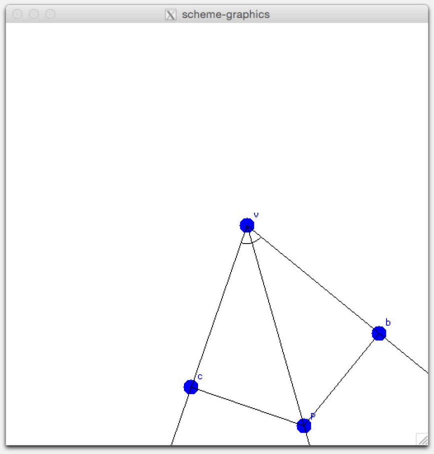
In doing so, I will first describe the basic structures and essential utility procedures before presenting some higher-level construction procedures, polygons, and figures. Then, I will explore the use of randomness in the system and examine how construction language macros handle names, dependencies, 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 Figure for Chapter

```
(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))
```





The sample construction in Example 5.1 constructs perpendiculars from an arbitrary point on an angle bisector to the ray extensions of the angle being bisected.

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, a segment is two ordered endpoints, a ray is an endpoint and a direction, and a line is a point and a direction from that point the line extends. Thus, lines and segments are directioned, and the same geometric line and segment can have several different internal representations. Predicates exist to allow other procedures to work with or ignore these distinctions.

Code Listing 5.2: Basic Structures

```scheme
(define-record-type <point>
  (make-point x y)
  point?
  (x point-x)
  (y point-y))

(define-record-type <segment>
  (make-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))
```

As shown in Listing 5.3, angles are represented as a vertex point and two arm directions, and circles have a center point and radius length.

Code Listing 5.3: Angle and Circle Structures

```scheme
(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.

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?)
```

Listing 5.5 also demonstrates one of many places in the system where I use generic operations to add flexibility. Here, `angle-from-lines` is defined as the handler for the generic operation `angle-from` when both arguments are lines. Similar handlers exist for other combinations of linear elements.

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 demonstrates some utilities using directions. Similar abstractions exist for working with vectors.

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 chapter's sample construction.

Code Listing 5.7: Perpendicular Constructions

```scheme
;; 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))))
```

Traditional constructions generally avoid using rulers and protractors. However, as shown in Listing 5.8, the internal implementation of the `angle-bisector` procedure uses measurements to simplify construction instead of repeatedly intersecting circle arcs to emulate compass sweeps. Although the internal implementations of some constructions use measured values, when providing the system with investigations, a user can still limit the construction steps used to ones that could be performed using only a compass and straight edge since the internal implementations of the constructions operations remain opaque to the learning module.

Code Listing 5.8: Angle Bisector Construction

```scheme
(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

Polygon 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

Given these underlying objects and operations, to allow figures to represent general spaces of diagrams, random choices needed when instantiating diagrams. The chapter's sample construction uses `random-angle` and `random-point-on-ray`, implementations of which are shown in listing 5.9. 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 that the figures stay reasonably legible for a human viewer, in `random-point-on-ray`, the procedure `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

```scheme
(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))))
```

Other random elements are created by combining these random choices, such as the random parallelogram in Listing 5.10. In `random-parallelogram`, a parallelogram is constructed by constructing two rays with an random angle between them, and selecting an arbitrary point on each. The final point is computed using vector arithmetic to "complete the parallelogram".

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 Listing 5.11 to avoid the edge cases of ranges, and some retry their local choices if the object they are returning has points too close to one another. However, further extensions could improve this system to periodically check for unintended relationships amongst all elements created previously in the figure and backtrack to select other values.

Code Listing 5.11: Safe Randomness

```
(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)))))
```

5.5 Construction Language Support

To simplify specification of figures, the module provides the `let-geo*` macro which allows for a multiple-assignment-like extraction of components from elements and automatically tags resulting elements with their variable names and dependencies. These dependencies are both symbolic for display and procedural so the system can generalize observations into conjectures that can be applied in other situations.

5.5.1 Multiple Component Assignment

Listing 5.12 shows the multiple component assignment expansion of a simple usage of `let-geo*`. In this case, `((a (r-1 v r-2)) (random-angle))` will assign to the variable `a` the resulting random angle, and to the variables `r-1`, `v`, and `r-2` the resulting angle's ray-1, vertex, and ray-2, respectively. If the specification were for a random quadrilateral, such as `((s (a b c d)) (random-square))`, the macro would assign to the variable `s` the resulting random square, and to the variables `a`, `b`, `c` and `d` the resulting square's vertices.

Interaction Example 5.12: Expansion of let-geo* macro

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

=> macro expands to:
(let* ((a (random-angle))
       (r-1 (element-component a 0))
       (v   (element-component a 1))
       (r-2 (element-component a 2)))
  (figure a r-1 r-2 ...))
```

To handle these varied cases, the macro expands to use the generic operation `element-component` to determine what components are extracted from an object during multiple component assignment. As shown in Listing 5.13, for polygons, the components are the point references directly, whereas angles and segments construct their handlers from a given list of getters.

Code Listing 5.13: Generic Element Component Handlers

```scheme
(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?)

(declare-element-component-handler
 (component-procedure-from-getters
   segment-endpoint-1 segment-endpoint-2)
 segment?)

(define (component-procedure-from-getters . getters)
  (let ((num-getters (length getters)))
    (lambda (el i)
      (if (not (<= 0 i (- num-getters 1)))
          (error "Index out of range for component procedure: " i))
      ((list-ref getters i) el))))
```

Listing 5.14 demonstrates the multiple assignment portion of the `let-geo*` macro in which the user's specifications are expanded into the element-component expressions.

Code Listing 5.14: Multiple and Component Assignment Implementation

```scheme
(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))))
```

5.5.2 Names and Dependencies

The other task the `let-geo*` macro handles is assigning names and dependencies to objects. As shown in Listing 5.15, these properties are attached to elements using the `eq-properties` method. In this approach, an hash table is used to store mappings of elements to property values. Similar interfaces are provided for element dependencies and element sources.

Code Listing 5.15: Element Names

```
(define (element-name element)
  (or (eq-get element 'name)
      *unnamed*))

(define (set-element-name! element name)
  (eq-put! element 'name name))
```

When an assignment is made in the `let-geo*` macro, three pieces of information are associated with the assigned object: (1) its `name`, taken from the variable used for the object in the let statement, (2) its symbolic `dependency` that stores the procedure name and arguments used to obtain the object primarily for display purposes, and (3) a `source` procedure that allows the object to be recreated from a different starting premise. Example 5.16 shows the expansion of these dependencies in a very simple construction. These dependencies are added after the multiple component assignments are expanded so will apply to all variables created in the form.

Interaction Example 5.16: Dependency and Name Assignment

```
(let-geo*
    ((s (make-segment a b)))
  (figure s))

=> macro expands to:
(let* ((s (make-segment a b)))
  (set-element-name! s 's)
  (set-source! s
   (lambda (p)
     (make-segment (from-new-premise p a) (from-new-premise p b))))
  (set-dependency-if-unknown! s (list 'make-segment a b))
  (figure s))
```

The decision to attach a procedure of a premise argument to an element as its `source` allows other starting premises to be injected during later explorations in the learning module. `from-new-premise` allows the system to recreate the corresponding object for a specified element given a different premise. Example 5.17 shows the implementation of `from-new-premise` and the interface for specifying an explicit premise dependency via `set-as-premise!`. To allow for multiple premises to be injected, the premise structure is represented as a list.

Code Example 5.17: Using sources with new premises

```scheme
(define (from-new-premise new-premise element)
  ((element-source element) new-premise))

(define (set-as-premise! element i)
  (set-dependency! element (symbol '<premise- i '>))
  (set-source! element (lambda (p) (list-ref p i))))
```

These source and premise structures will be used more later in learning new terms, but Example 5.18 provides a concrete example of its use. The first definition creates a random square and obtains the intersection point of its two diagonals. `let-geo*` sets up the names and dependencies, and the square is marked as the initial premise. However, the intersection point is returned rather than a figure. The print statements (continued on the next page) show that while `diag-intersection-point` is a point structure with explicit coordinates it can produce information about how it was created.

Interaction Example 5.18: Using from-new-premise

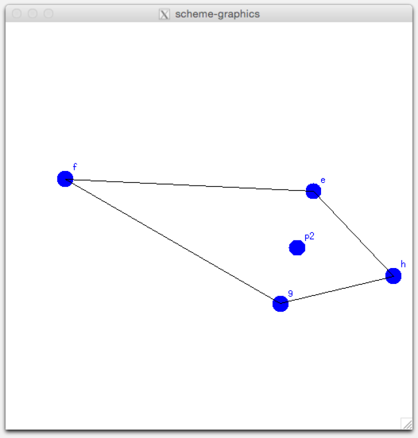
```scheme
(define diag-intersection-point
  (let-geo*
      (((sq (a b c d)) (random-square))
       (diag-1 (make-segment a c))
       (diag-2 (make-segment b d))
       (p (intersect-linear-elements diag-1 diag-2)))
    (set-as-premise! sq 0)
    p))

=> (pp diag-intersection-point)
#[point 26] (x -.1071) (y -0.4464)
```

```
=> (print-dependencies (element-dependency diag-intersection-point))
(intersect-linear-elements
 (make-segment (element-component <premise-0> 0)
               (element-component <premise-0> 2))
 (make-segment (element-component <premise-0> 1)
               (element-component <premise-0> 3)))

(define new-figure
  (let-geo* (((k (e f g h)) (random-kite))
             (p2 (from-new-premise (list k) diag-intersection-point)))
    (figure k p2)))

=> (show-figure new-figure)
```



The second definition at the end of Example 5.18, `new-figure`, demonstrates using `from-new-premise` to apply source information from an existing object to a new premise. The specification of `new-figure` constructs a random kite and uses that object `k` as the new premise in creating point `p2` using the source information of the `diag-intersection-point`. As seen in the image, `from-new-premise` was able to correctly extract the construction steps about how `diag-intersection-point` was constructed and apply it to the new kite to set `p2` at the intersection of the kite's diagonals.

A similar process to this example will appear in an abstracted form later in the learning module as the system tests whether conjectures apply to new situations.

Listing 5.19 presents the implementation of the manipulations used to add dependency information to figures, and Listing 5.20 presents the top-level definition for the `let-geo*` form.

Code Listing 5.19: Implementation of Dependency Expressions

```
(define (args-from-premise args)
  (map (lambda (arg)
         `(from-new-premise p ,arg))
       args))

(define (set-dependency-expressions assignments)
  (append-map
   (lambda (a)
     (let ((name (car a))
           (value (cadr a)))
       (if (list? value)
           (let ((proc (car value))
                 (args (cdr value)))
             `((set-source!
                ,name (lambda (p) (,proc ,@(args-from-premise args))))
               (set-dependency!
                ,name (list (quote ,proc) ,@args))))
           `((set-source! ,name (element-source ,value))
             (set-dependency! ,name (element-dependency ,value))))))
   assignments))
```

Code Listing 5.20: Full let-geo* Implementation

```
(define-syntax let-geo*
  (sc-macro-transformer
   (lambda (exp env)
     (let ((assignments (cadr exp))
           (body (cddr exp)))
       (let ((new-assignments (expand-assignments assignments))
             (variable-names (variables-from-assignments assignments)))
         (let ((result `(let*
                          ,new-assignments
                        ,@(set-name-expressions variable-names)
                        ,@(set-dependency-expressions new-assignments)
                        ,@body)))
           ;; (pp result) ;; To debug macro expansion
           (close-syntax result env)))))))
```

5.6 Graphics and Animation

Given the primitive objects and a language for specifying constructions, the final task of the imperative system is to display and animate figures. To do so, the system integrates with Scheme's graphics system for the X Window System. It 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 5.21 and 5.22.

Code Listing 5.21: Drawing Figures

```scheme
(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)
     ((draw-label element) canvas))
   (all-figure-elements figure))
  (graphics-flush (canvas-g canvas)))
```

To support animation, 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 5.22: Animation

```scheme
(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))))))
```

5.7 Discussion

In creating the imperative construction module, the main challenges involved settling on appropriate representations for geometry objects and properly yet effortlessly tracking dependencies. Initial efforts used over-specified representations such as an angle consisting of three points and a line consisting of two points. Reducing these to nearly-minimal representations using directions helped simplify other operations. In addition, the module initially had each individual construction procedure set the dependencies of the elements it produced. Automating this in the `let-geo*` macro helped simplify the annotation process and make the setting of source procedures feasible.

Future extensions could provide additional construction procedures, particularly with added support for circle and arc-related operations, or improve the resilience of random choices. However, I believe the imperative module provides a sufficiently versatile library of components and procedures to enable users to specify interesting investigations. With the ability to construct and represent figures, the following chapters explain details of how the system is able to make and generalize observations to learn from user-specified constructions.