Chapter 6

Perception Module

6.1 Overview

The perception module focuses on "seeing" figures in our mind's eye. Given analytic figures represented using structures of the imperative construction module, the perception module is concerned with finding and reporting interesting relationships seen in the figure. In a generate-and-test-like fashion, it is rather liberal in the observations it returns. The module uses several technique to attempt to omit obvious properties, and combines with the learning module (Chapter 8) to filter already-learned discoveries and simplify results.

To explain the module, I will first describe the implementation of underlying relationship and observation structures before examining the full analyzer routine. I will conclude with a discussion of extensions to the module, including further ways to detect and remove obvious results and some attempted techniques used to extract auxiliary relationships from figures.

6.2 Relationships

Relationships are the primary structures defining what constitutes interesting properties in a figure. Relationships are represented as predicates over typed n-tuples and are checked against all such n-tuples found in the figure under analysis.

Code Listing 6.1: Relationships

```
(define-record-type <relationship>
  (make-relationship type arity predicate equivalence-predicate)
  relationship?
  (type relationship-type)
  (arity relationship-arity)
  (predicate relationship-predicate)
  (equivalence-predicate relationship-equivalence-predicate))

(define equal-length-relationship
  (%make-relationship 'equal-length 2 segment-equal-length?
                      (set-equivalent-procedure segment-equivalent?)))

(define concurrent-relationship
  (make-relationship 'concurrent 3 concurrent?
                      (set-equivalent-procedure linear-element-equivalent?)))

(define concentric-relationship
  (make-relationship 'concentric 4 concentric?
                      (set-equivalent-procedure point-equal?)))
```

Listing 6.1 displays some representative relationships. The relationship predicates can be arbitrary Scheme procedures and often use constructions and utilities from the underlying imperative system as seen in Listing 6.2. `concurrent?` is checked over all 3-tuples of linear elements (lines, rays, segments) and `concentric?` is checked against all 4-tuples of points.

Code Listing 6.2: Concurrent and Concentric Predicates

```
(define (concurrent? l1 l2 l3)
  (let ((i-point (intersect-linear-elements-no-endpoints l1 l2)))
    (and i-point
         (on-element? i-point l3)
         (not (element-endpoint? i-point l3)))))

(define (concentric? p1 p2 p3 p4)
  (and (distinct? p1 p2 p3 p4)
       (let ((pb-1 (perpendicular-bisector (make-segment p1 p2)))
             (pb-2 (perpendicular-bisector (make-segment p2 p3)))
             (pb-3 (perpendicular-bisector (make-segment p3 p4))))
         (concurrent? pb-1 pb-2 pb-3))))
```

In addition to the type, arity, and predicate checked against arguments, the relationship structures also include an equivalence predicate that is used in determining

whether two observations using the relationship are equivalent, as will be discussed after explaining the observation structure in Section 6.3.

6.2.1 What is Interesting?

The system currently checks for:

- concurrent, parallel, and perpendicular linear elements,
- segments of equal length,
- supplementary and complementary angles,
- angles of equal measure,
- coincident and concentric points, and
- sets of three concentric points with a fourth as its center.

These relationships covered most of the basic observations needed in my investigations, but further relationships can be easily added.

6.3 Observations

Observations are structures used to report the analyzer's findings. As seen in Listing 6.3, they combine the relevant relationship structure with a list of the actual element arguments from the figure that satisfy that relationship. Maintaining references to the actual figure elements allows helper procedures to print names or extract dependencies as needed.

Code Listing 6.3: Observations

```
(define-record-type <observation>
  (make-observation relationship args)
  observation?
  (relationship observation-relationship)
  (args observation-args))
```

It is important to know whether two arbitrary observations are equivalent. This enables the system to detect and avoid reporting redundant or uninteresting relationships. Listing 6.4 shows the implementation of `observation-equivalent?`. The

procedure checks the observations are the same and then applies that observation's equivalence predicate to the two tuples of observation arguments.

Code Listing 6.4: Equivalent Observations

```
(define (observation-equivalent? obs1 obs2)
  (and (relationship-equal?
        (observation-relationship obs1)
        (observation-relationship obs2))
       (let ((rel-eqv-test
              (relationship-equivalence-predicate
               (observation-relationship obs1)))
             (args1 (observation-args obs1))
             (args2 (observation-args obs2)))
         (rel-eqv-test args1 args2))))
```

These equivalence predicates handle the various patterns in which objects may appear in observations. For example, in an observation that two segments have equal length, it does not matter which segment comes first or which order the endpoints are listed within each segment. Thus, as shown in Listing 6.5, the equivalence procedure ignores these ordering differences by comparing set equalities:

Code Listing 6.5: Equivalence of Equal Segment Length Observations

```
(set-equivalent-procedure segment-equivalent?)

(define (set-equivalent-procedure equality-predicate)
  (lambda (set1 set2)
    (set-equivalent? set1 set2 equality-predicate)))

(define (set-equivalent? set1 set2 equality-predicate)
  (and (subset? set1 set2 equality-predicate)
       (subset? set2 set1 equality-predicate)))

(define (segment-equivalent? s1 s2)
  (set-equivalent?
   (segment-endpoints s1)
   (segment-endpoints s2)
   point-equal?))

(define (point-equal? p1 p2)
  (and (close-enuf? (point-x p1) (point-x p2))
       (close-enuf? (point-y p1) (point-y p2))))
```

6.3.1 Numerical Accuracy

Throughout the system, numerical accuracy issues and floating point errors arise when comparing objects. As a result, the system uses custom equality operators for each data type, such as `point-equal?` shown in Listing 6.5. These use an underlying floating-point predicate `close-enuf?` taken from the MIT Scheme Mathematics Library [15] that estimates and sets a tolerance based on current machine's precision and handles small magnitude values intelligently. With this floating point tolerance in comparisons, floating point errors have been significantly less prevalent.

6.4 Analysis Procedure

Given these relationship and observation structures, Listing 6.6 presents the main analyzer routine in this module. After extracting various types of elements from the figure, it examines the relationships relevant for each set of elements and gathers all resulting observations.

Code Listing 6.6: Analyzer Routine

```
(define (analyze figure)
  (let* ((points (figure-points figure))
         (angles (figure-angles figure))
         (linear-elements (figure-linear-elements figure))
         (segments (figure-segments figure)))
    (append
     (extract-relationships points
                            (list concurrent-points-relationship
                                  concentric-relationship
                                  concentric-with-center-relationship))
     (extract-relationships segments
                            (list equal-length-relationship))
     (extract-relationships angles
                            (list equal-angle-relationship
                                  supplementary-angles-relationship
                                  complementary-angles-relationship))
     (extract-relationships linear-elements
                            (list parallel-relationship
                                  concurrent-relationship
                                  perpendicular-relationship)))))
```

The workhorses of `extract-relationships` and `report-n-wise` shown in Listing 6.7 generate the relevant n-tuples and report observations for those that satisfy the relationship under consideration. For these homogeneous cases, `all-n-tuples` returns all (unordered) subsets of size n as lists.

Code Listing 6.7: Extracting Relationships

```
(define (extract-relationship elements relationship)
  (map (lambda (tuple)
         (make-observation relationship tuple))
       (report-n-wise
        (relationship-arity relationship)
        (relationship-predicate relationship)
        elements)))

(define (report-n-wise n predicate elements)
  (let ((tuples (all-n-tuples n elements)))
    (filter (nary-predicate n predicate) tuples)))
```

6.5 Focusing on Interesting Observations

The final aspect of the perception module involves filtering out obvious and previously-discovered observations. Several techniques will be discussed but Listing 6.8 shows the module's current state of accomplishing this task via the `interesting-observations` procedure. The procedure first extracts all observations from the figure and aggregates a list of obvious relations specified during the construction. It then uses the learning module to examine all polygons found in the figure and determine the most specific definitions each satisfies. The procedure obtains in `polygon-implied-observations` all previously-discovered facts about such shapes to remove from the final result and adds new polygon observations in their place. Example 6.10 shows a concrete example of this.

Code Listing 6.8: Obtaining Interesting Observations

```
(define (interesting-observations figure-proc)
  (set! *obvious-observations* '())
  (let* ((fig (figure-proc))
         (all-obs (analyze-figure fig))
         (polygons (figure-polygons fig))
         (polygon-observations (polygon-observations polygons))
         (polygon-implied-observations
           (polygon-implied-observations polygons))
    (set-difference (append all-obs polygon-observations)
                    (append *obvious-observations*
                        polygon-implied-observations)
                    observation-equivalent?)))
```

Listing 6.9 shows the implementation of the `save-obvious-observation!` procedure

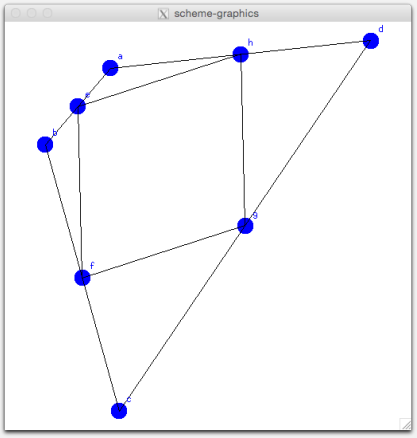
Code Listing 6.9: Marking Obvious Observations

```
(define (save-obvious-observation! obs)
  (if *obvious-observations*
      (set! *obvious-observations*
            (cons obs *obvious-observations*))))
```

Example 3.7 in the demonstration chapter demonstrated a simple example of a construction procedure that marked obvious properties of its results.

Interaction Example 6.10: Substituting Polygon Observations

```
Observations
```



6.6 Discussion and Extensions

Perfectly determining is a large task, particularly since filtering out obvious relations often requires relationship-specific information:

As one example, imagine implementing and a `collinear?` predicate that only reports non-obvious relations. Testing whether three arbitrary coordinate-based points are collinear is straightforward. However, it is not interesting that a random point on a line is collinear with the two points from which the line was defined. In order to accurately know whether or not it is interesting that three points are collinear, the system would need to have access to a graph-like representation of which points were specified to be on which lines.

The analysis routine was initially one large, arbitrarily complicated procedure in which individual checks were added. This reformulation to use relationships and observations has simplified the complexity and enabled better interactions with the learning module, but limited the ability for adding many relationship-specific optimizations.

Despite these limitations, the perception system has been sufficient to discover

several relations via the learning model and use basic filtering of obvious relations to present intelligible results.

The examples below describe further efforts explored for improving the perception module. This involves extracting relationships for elements not explicitly specified in a figure, such as auxiliary segments between all pairs of points in the figure, treating all intersections as points, extracting angles, or merging results. These are areas for future work.

6.6.1 Auxiliary Segments

In some circumstances, the system can insert and consider segments between all pairs of points. Although this can sometimes produce interesting results, it can often lead to too many elements being considered. This option is off by default but could be extended and enabled in a self-exploration mode.

6.6.2 Extracting Angles

In addition, I briefly explored a system in which the construction module also maintains a graph-like representation of the connectedness and adjacencies in the figure. In addition to the complexity of determining which angles to keep, keeping track of "obvious" relationships between such extracted angles due to parallel lines, for instance, is quite a challenge.

6.6.3 Merging Related Observations

A final process I explored involved merging related observations into larger, combined results. For instance, when reporting segment length equality for a square, it is excessive to report all possible pairs of equal sides.