Development of a Prototype Geometric Modelling System using a Functional Language

Iain Checkland and Colin Runciman*

Abstract

The world outside of academia lacks large scale real examples of applications written in functional programming languages. This technical report considers the development of one such application, a Geometric Modelling System. The core of the report describes the application in some depth. The report concludes with an appraisal of the results and considers the appropriateness of the functional programming style.

1 Introduction

The advantages claimed of the functional programming style over the conventional imperative style are now well rehearsed [Jon84] [Hug89] [Hud89]. Even so functional languages have only flourished in the laboratories of academia. This fact is currently being addressed by the FLARE project.

> *"The effectiveness of functional programming has been amply demonstrated on a laboratory scale and it is now appropriate to scale up the experiments with a view to encouraging the wider community of applications developers to embrace this technology."* — FLARE project proposal [Run89a]

We need to dispel the myth that functional languages are toys. To do this we need to build realistic examples of real applications so that the effectiveness of the functional paradigm can be demonstrated to the world outside of academia. To this end I have chosen the field of Geometric Modelling as a domain and have developed a Geometric Modelling system as an application.

Geometric modelling is about the representation and manipulation of solids. A geometric modelling system lies at the heart of any CAD or CAM system and is responsible for the creation, storage and modification of representations of solids. There are several reasons for choosing geometric modelling as an application domain:

*Iain Checkland has been funded by a SERC CASE award in conjunction with Geomatrix Ltd.

- Many imperative programmers would point to the domain of graphics as being one particularly dependent on state and side-effecting assignment. If we can demonstrate that a functional program can adequately handle graphics then a small battle will be won.
- Geometric modellers on the market currently consist of 100,000's of lines of FORTRAN code. Such systems are often a complex web of tricks and efficiency hacks being unintelligible to the uninvolved programmer. If we can produce a functional implementation of such a system, we can contrast the size and clarity of the two. The modeller developed within this report is only a prototype and contains only the core of the operations that full geometric modellers make available but the comparison with the core of an imperative modeller will still be of interest.
- Manipulation of geometric objects is computationally demanding: it essentially involves a considerable amount of number crunching. It is also an area where *reactive* response is essential. It will therefore provide an excellent test of the performance of functional languages.

A Geometric Modeller has been written in the language Haskell [HWe90] though initially development took place using Miranda[1]. The Geometric Modelling System (GMS) is built up from 16 modules. These modules fit nicely into the six groups shown in Figure 1. The diagram also indicates the dependency of groups: the bottom group of modules depends only on the `Prelude` functions; each other group depends on all those below it.

The next section outlines the informal specification for the GMS. Each module group is then discussed individually in its own section. We start by discussing, in detail, the core of the program, the modules that deal with creating and manipulating objects — the BSPT Group (see Section 3). We then discuss the supporting representation of Euclidean space (Section 4) and the implementation of an underlying number system (Section 5). We then briefly discuss the supporting libraries (Section 6). We then move up to the top of the hierarchy and discuss the definition of the program (Section 7) and finally the design of the interface to the core of the program (Section 8).

The report highlights the following issues that have arisen during development of the GMS:

- use of higher order functions;
- re-use of functions;
- memoisation of results in data structures;
- exploitation of Laziness;
- use of the Haskell Class system.

The report concludes with a discussion of the implementation of the GMS as a whole, a discussion of the results and consideration of the part played by Haskell. We also briefly

[1] Miranda is a trademark of Research Software Ltd.

Section	Group	Modules
7	Program	`Main, Prog`
8	Interface	`Interface, Interpret, Init, Params, Input`
3	BSPT	`BSPT, Merge, Render`
4	Euclidean	`Euclid, EuclidGMS`
5	Numbers	`GeomNum, Rationals`
6	Libraries	`Stdlib, MGRlib`

Table 1: Grouping of modules and dependency hierarchy

discuss the ramifications expected from executing the GMS on a *parallel* machine. While this has always been the long term aim, it is worth noting that the GMS has been designed to fit the sequential model, and no adjustment has been made to any aspect of the program to fit the future parallel implementation.

2 Specification of the Geometric Modelling System

The interface to the modeller developed is shown in Figure 1. The screen is split into five sections. Each is described below:

Object definition and display window This is the section of the screen that is used for defining and displaying objects.

Tree form window This portion of the screen displays a pretty-printed BSPT representation of the object in the screen described above.

Mouse Representation This diagram indicates the current functionality of the mouse buttons.

Text Window This window displays text information, i.e. area calculations and point classifications.

Buttons These form the menu of operations available to the user.

The GMS, via the Buttons, provides the following operations:

Polygon Allows the definition by the user of a closed polygon. This gets represented as a BSPT and displayed in the display window of the screen along with a pretty printed form of the representation.

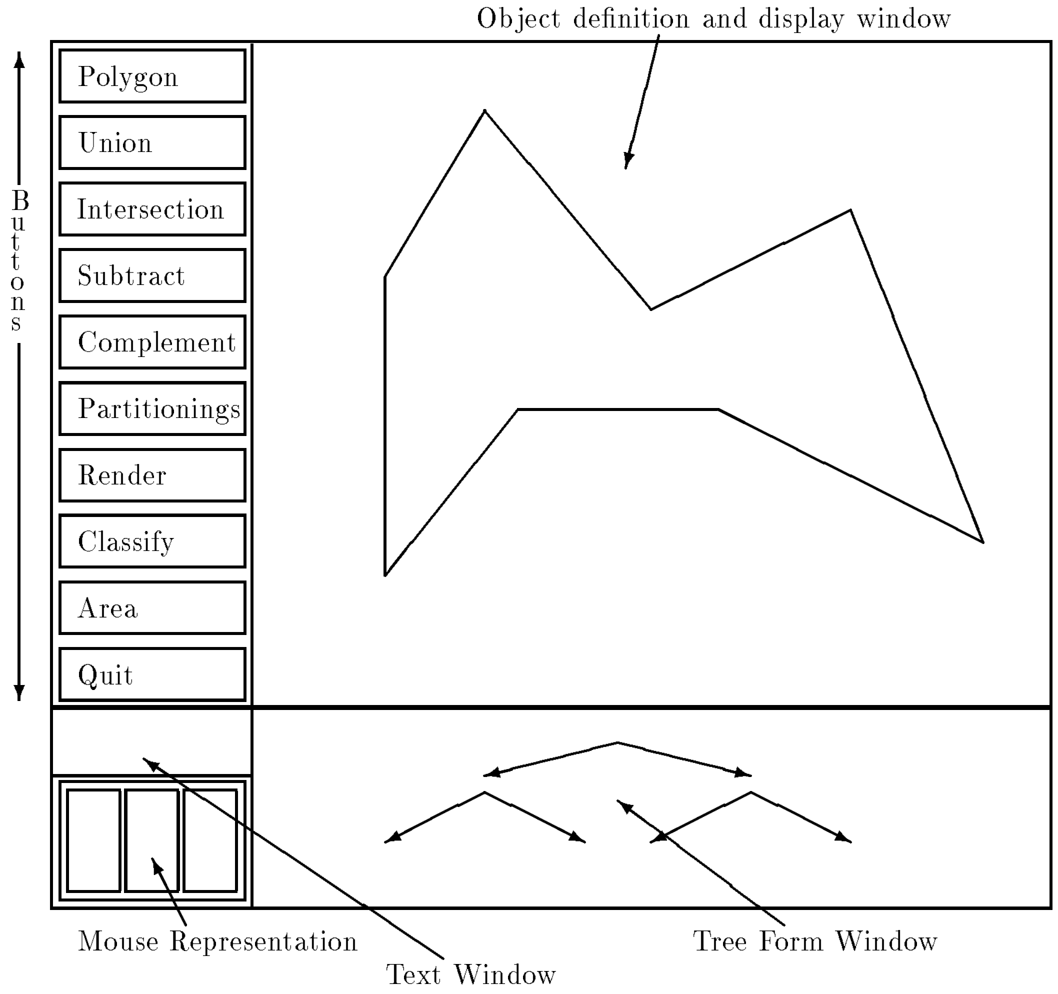


Figure 1: User interface to experimental modeller

Union Allows the Union of two objects to be formed and the resulting object displayed.

Intersect Allows the Intersection of two objects to be formed and the resulting object displayed.

Subtract Allows the Subtraction of one object from another. The resulting object is displayed on the screen.

Complement Allows an object to be inverted: inside becomes outside and *vice versa*.

Classify Allows determination of the classification (either *inside*, *outside* or *on* the boundary) of points selected by the user with respect to an object.

Render Allows shaded rendering of the object on the screen.

Area Allows the area of an object to calculated. Included as an example physical calculation.

The modeller currently works in 2 dimensions. This is for simplicity only — the techniques used apply equally to higher dimensions.

3 BSPT Module Group

All Geometric Modelling Systems are characterised by the representation of objects employed. We cannot go any further without first introducing the representation scheme that is used in our GMS — *Binary Space Partitioning Trees.*

3.1 The Binary Space Partitioning Tree (BSPT)

The BSPT is based on the partitioning of space into disjoint convex homogeneous subspaces. We refer to any sub-space bounded by a set of such partitions as a region. The BSPT is a dimension independent generalisation of the more common quadtree and octree. These representation schemes share the property of being representations of space rather than of a single object. When a global region contains more than one disjoint object they are represented within a single data structure. As a result *spatial locality* and hence *spatial relationships* between objects are implicitly stored within the representation. However, whereas the quadtree and octree schemes are based on space sub-division using static partitions, the BSPT employs partitions that are custom built for the objects modelled. This makes the BSPT scheme more concise and more accurate. For a full introduction and discussion of the relative merits of spatial sub-division representation schemes, see [Che91]. The BSPT representation of polyhedra was first presented by Thibault and Naylor [TN87].

The algorithm to construct a BSPT representation of a bounded region, R, where R contains objects represented by their boundaries, is as follows:

- If the region, R, is homogeneous (it does not contain any face of any object), classify it as either *inside* or *outside* (with respect to the object) and build a leaf node tagged with this classification.
- If the current region, R, is not homogeneous, select a face of an object in the region. Partition the space R into two by the line formed by the selected face. Generate a node for this partition and let the children be the results of applying this algorithm recursively to the generated halfspaces.

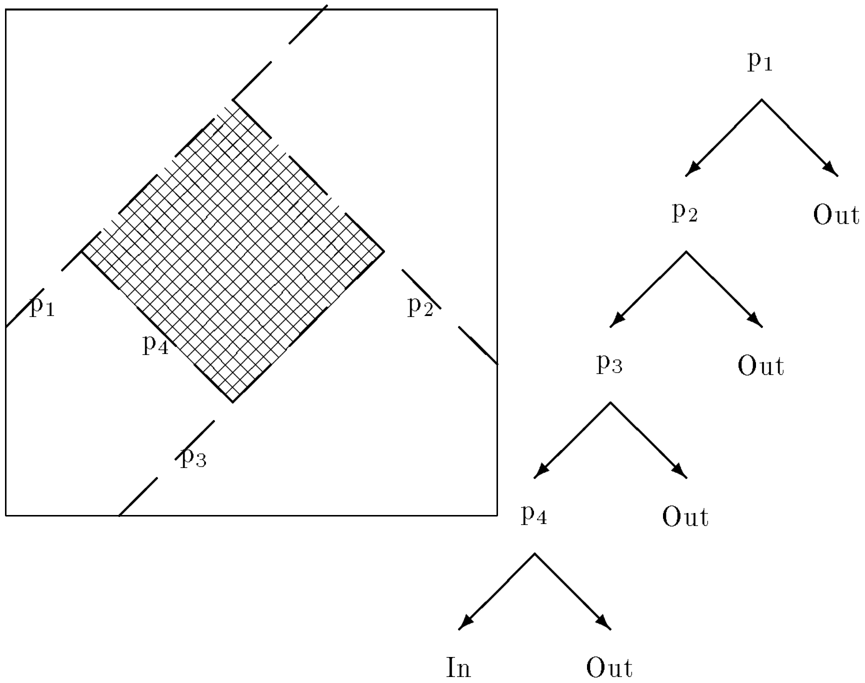


Figure 2: Binary partitioned square and representation

Note that:

- we introduce *normals* of lines formed from faces defining an object, pointing to the exterior;
- the *fore halfspace* of a line is the side to which the normal lies;
- the *right* child of a BSPT node represents the *fore* halfspace and the *left* child represents the *rear* halfspace

Figure 2 serves as an example. The bounded region shown is not homogeneous; the line p_1 is chosen (arbitrarily) to form the first partition. This creates two half spaces, the fore side is empty and is outside of the object so we create an Out leaf node. The rear side is not homogeneous, so the process is recursively repeated, this time the partition p_2 is used. The process continues until partition p_4 creates two homogeneous regions: the fore side being Out and the rear side In.

The left descending tree structure exhibited in Figure 3 is typical of all convex polygons. A second example using a concave polygon highlights an important issue. Figure 3 shows a concave polygon and the corresponding tree. When we make the first partition arbitrarily at a, the line d is found to cross this line, we need to split this line into two segments, d' and d'', passing each segment into the appropriate half-space recursion. If, however, we had made the first partition that of the line d then the splitting of d, or any other line, would not occur. Figure 4 shows the tree that would result from this course of action.

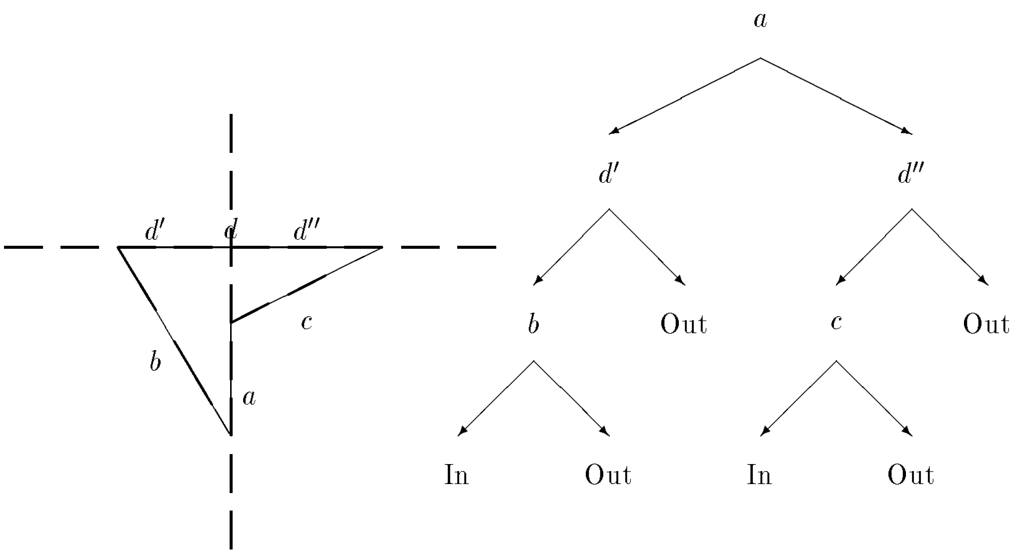


Figure 3: Concave polygon with partitioning beginning with face a and corresponding representation.

Note that the tree in Figure 3 is larger (by 2 nodes). The size of a BSPT is determined by the selection of partitioning planes.

3.2 The simple BSPT and its construction

Let us define a function that will produce the Binary Space Partitioning Tree representation of a polygon represented by its boundary. The boundary-representation of a polygon is a list of line segments representing the edges of the polygon. Each line segment is represented by its two end points. We define the types `Polygon` and `Segment` as follows:

```
> type Polygon = [Segment]
> data Segment = (Point,Point)
```

Now consider the following definition of the BSPT type:

```
> data BSPT = Cell Status | BSP Line BSPT BSPT
```

This is a much simplified version of that actually used in the GMS, but it serves our purpose for the time being. A BSPT is then either:

1. a leaf node tagged `Cell` with an argument of type `Status` where the type `Status` is defined below and describes the state of the convex region;
2. a branch node tagged `BSP` where the first argument is a `Line` equation describing the partitioning of space at this node. The remaining two arguments are `BSP` trees representing the rear and fore halfspaces formed by the partition at this node.

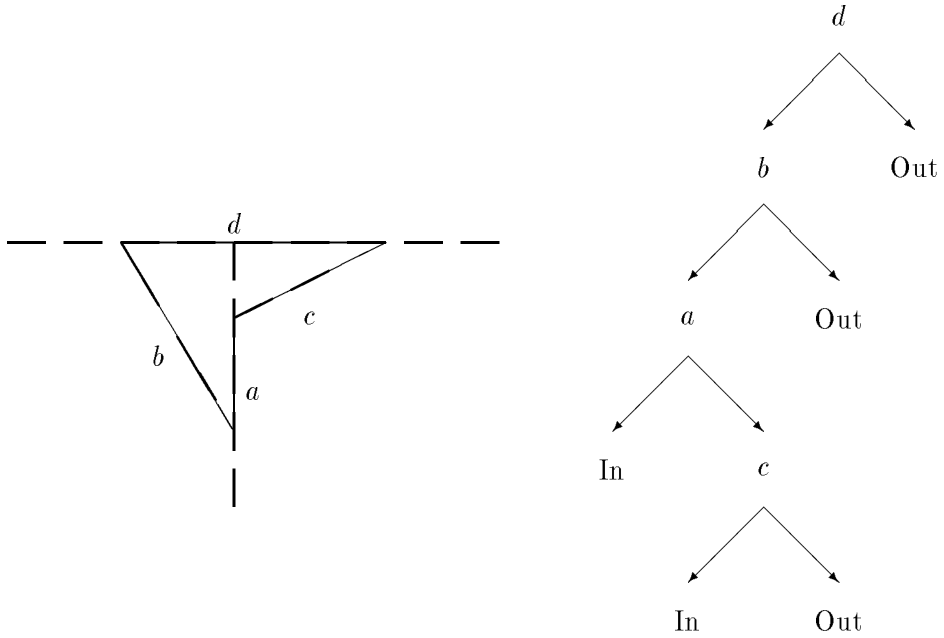


Figure 4: Concave polygon with partitioning beginning with face d and corresponding representation.

Status is an enumeration of the possible states of convex regions (i.e. either `In` or `Out`). It is also used for classifications of points with respect to objects hence the additional constructor `On`.

```
> data Status = In | Out | On deriving (Eq,Text)
```

Now, the code to build a BSPT from a polygon is

```
> buildBSPT :: Polygon -> BSPT
> buildBSPT = buildBSPTAux Out

> buildBSPTAux :: Status -> Polygon -> BSPT
> buildBSPTAux status [] = Cell status
> buildBSPTAux _ segments = BSP part left right
>                where
>                left = buildBSPT In rear
>                right = buildBSPT Out fore
>                (rear,fore) = partSegments part segments
>                part = convert.heuristic segments
```

The first line of the auxiliary function specifies that given an empty sub-space the result should be a cell. The status of this cell, either `In` or `Out`, is determined by the recursion. When recursing to the left, the default status for an empty sub-space is `In`. When recursing to the right the default status is `Out`. The original default value is `Out`.

The second clause considers what to do with a non-homogeneous region. We build a tree whose root node contains the partition and which has two children. The children are built by recursive calls to the `buildBSPT` function with the appropriate segments from original list — those in the rear halfspace and those in the fore.

The function `convert` returns the line equation that the given line segment lies on.

```
> convert :: Segment -> Line
```

The function `heuristic` performs the job of selecting one face from the list passed to it to form the partition.

```
> heuristic :: [Segment] -> Segment
```

We should like the partition chosen to yield:

1. a tree of minimal size;
2. a tree which is balanced.

Unfortunately these two interests conflict. A tree that is balanced may of necessity have some split segments and therefore may not be minimal. We need to determine a trade-off point. This is done by using a heuristic since it is not computationally feasible to test every face for partitioning candidacy. Thibault and Naylor [TN87] give brief analyses of three possible heuristics. These heuristics are functions of the size of the rear and fore partitions and of the number of segments split by the partition. A selected number of segments are chosen as candidate partitions and the segments in the current region are partitioned by each candidate in turn to produce these sizes. The program currently uses a very simple heuristic: it just takes the first face from the polygon. This gives some user control over the selection of partitioning faces which is useful for experimentation.

3.3 The augmented BSPT

The BSPT is a generic data type. The definition of a BSPT given in the previous section is its simplest form. In practice it is augmented with additional information; in the literature the most ubiquitous is the storage of face segments that form part of the boundary of the object. These are stored at the internal node of the partition which they lie on. In this GMS, however, we go further and add to every node a representation of the boundary of the region whose contents are described by the tree rooted at that node. We discuss the representation of a region later in this section. The storage of a region representation is performed for reasons of efficiency. Originally region data was unattached to nodes but it was noted that many functions kept an accumulation of the region in consideration as they recursed down the tree. It then seems sensible to add this information to nodes where it is created and stored once and for all. This also brings the added benefit of making the

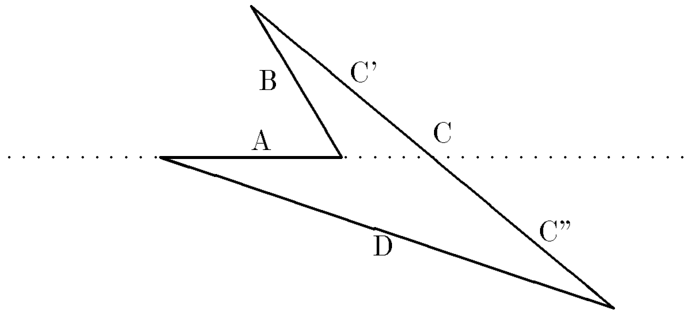


Figure 5: Partitioning of a segment into two

code more concise — there is no longer the need for regions to be accumulated. We also add a number field to leaf nodes which then stores the area of that cell. This, and the other additions to the `BSPT` data type are *memoisations.*

Memoisation of a value is the storage of that value in a data structure. It involves the addition of non-essential (i.e. derivable) information to data declarations. By memoising commonly calculated values we ensure that they are calculated once and because of lazy evaluation they may never be evaluated at all. We can use quasi-constructors to automatically derive memoised values (c.f. `mkFace`,`mkCell`)

This is the revised, augmented type declaration for BSPTs:

```
> data BSPT =  Cell Status Region Numb
>            | BSP Partition (Faces,Region) BSPT BSPT
```

We have introduced several new types. The type `Faces` is the most important. Consider the object in Figure 5 where partitioning is initially by line A. This causes line C to be split into two segments, C’ and C”. Eventually each will be selected as a partition and each will be converted into a `Line` equation describing the line the segment lies on. However C, C’ and C” will all produce the *same* line. Not only are we *storing* the same line several times but we are also having to *generate* the same line several times. The conversion function `convert` that produces a `Line` from a `Segment` (see Section 4) is quite expensive since it ensures the representation of a line is canonical. The solution to these problems is to memoise the line associated with a segment.

```
> data Face = Fc Segment Line
```

Now the initial polygon is described by a list of `Faces` each generated from a `Segment` by the function `mkFace`

```
> mkFace :: Segment -> Face
> mkFace (x,y) = Fc (x,y) (convert x y)
```

When a face is split into two, by the function `bisect`, each new face is constructed from one half of the split line segment and a *shared* copy of the line of the original face.

The type `Partition` still specifies a line but we will find in Section 3.5 that it is useful for a segment of the partitioning line to be maintained. This requirement means that a `Partition` shares the same representation as a `Face` and is manipulated by the same set of functions. We therefore define `Partition` as a synonym for `Face`.

```
> type Partition = Face
```

A `Region` represents a convex region as the intersection of the *rear* halfspaces of a list of lines. Again the line segment annotations will prove useful later so `Face` is used rather than `Line`. We define `Region` as a new data type to avoid abuse of its representation as a list. We also define two functions to construct and select `Region`s.

```
> data Region = Rg [Face]

> mkRegion :: [Face] -> Region
> mkRegion faces = Rg faces

> getRegion :: Region -> [Face]
> getRegion (Rg faces) = faces
```

We also define a function to produce a new region from an existing `Region` and a new `Face`.

```
> newRegion :: Region -> Face -> Region
> newRegion (Rg faces) face = Rg (face:faces)
```

Table 2 summarises the types employed in the GMS.

The actual BSPT building operation is:

```
> buildBSPT :: Faces -> BSPT
> buildBSPT = buildBSPTAux Out renderBorder

> buildBSPTAux :: Status -> Region -> Faces -> BSPT
> buildBSPTAux status region [] = mkCell status region
> buildBSPTAux _ region faces   = BSP partition (coin,region) left right
>       where
>       left = buildBSPTAux In (newRegion region partition) rear
>       right = buildBSPTAux Out (newRegion region (flip partition)) fore
>       (rear,coin,fore) = partFaces part faces
>       partition = mkPart region part
>       part = heuristic faces
```

```
> data BSPT       = Cell Status Region Numb
>                 | BSP Partition (Faces,Region) BSPT BSPT

> data Status     = In | Out | On

> type Partition  = Face

> data Region     = Rg [Face]

> type Faces      = [Face]

> data Face       = Fc Segment Line

> type Segment    = (Point,Point)

> data Point      = Pt Numb Numb

> data Line       = Ln Numb Numb Numb
```

Table 2: Summary of type declarations used in GMS

Notice:

1. that the initial `Region` that the BSPT is going to describe is the render screen. This is defined by the global defined value `renderBorder::Region`;
2. how `partFaces` now produces three groups of faces:
   (a) those in the `rear` halfspace;
   (b) those `coinc`ident to the partition;
   (c) those in the `fore` halfspace.

   The additional group, those coincident to the line, are used to augment the node.
3. how regions are accumulated by adding the partition to the current region when recursing to the left and by the inverse of the partition when recursing to the right. The inverse of a partition is generated by the function `flip` which swaps the rear and fore halfspaces by changing the orientation of the partitioning line.
4. that we can make more concise the building of a cell by use of the function `mkCell`.

   ```
   > mkCell :: Status -> Region -> BSPT
   > mkCell status region = Cell status region (areaRegion region)
   ```

   This replaces the constructor `Cell` by a function of only two arguments that automatically generates the third argument to the constructor according to the default method for calculating an area. Unfortunately this trick is not available when pattern matching against a cell; it is then that we pay the price of verbosity for augmenting the pure BSPT structure.

5. the function `heuristic` now has type `[Face] -> Face`. Since the `Face` chosen stores the partitioning `Line` with it there is no explicit need for the `convert` function. The function `mkPart` is used to create the partition from the partitioning, as well as the appropriate line segment (see Section 3.5).

Finally in this section let us consider the function `partFaces`.

```
> partFaces :: Line -> Faces -> (Faces,Faces,Faces)
> partFaces part [] = ([],[],[])
> partFaces part (face:faces) =
>       case (location part face) of
>               Intersects -> (rearHalf:rear,coin,foreHalf:fore)
>               Coincident -> (rear,face:coin,fore)
>               ToTheRear  -> (face:rear,coin,fore)
>               ToTheFore  -> (rear,coin,face:fore)
>       where
>       (rear,coin,fore) = partFaces part faces
>       (rearHalf,foreHalf) = bisect face part
```

The important auxiliary is `location` which determines the geometric relationship between each face and the current partition, `part`. This function is described in Section 4. Note that when the face is intersected by `part` we need to split this face in half and add the halves to the rear and fore lists appropriately. The function `bisect` performs the split and returns a pair such that the first item is the section of the face in the rear halfspace of the partition.

3.4 Rendering of BSPT objects

There are two kinds of object rendering that the GMS provides:

1. unfilled outline rendering;

2. filled object rendering.

We consider each in turn.

3.4.1 Unfilled rendering of BSPT objects

Unfilled rendering is a relatively easy task since we have augmented the BSPT with the lists of faces that make up the boundary of the object. We traverse the tree and print all the faces stored at the internal nodes. First let us look at a higher-order function that captures this common traversal type of operation — `foldBSPT`.

```
> foldBSPT :: (Status->Region->Numb->a)->
>                   (Partition->(Faces,Region)->a->a->a)->BSPT->a
> foldBSPT cellop nodeop (Cell x r a) = cellop x r a
> foldBSPT cellop nodeop (BSP part nodeinfo left right)
>               = nodeop part nodeinfo (f left) (f right)
>                 where f = foldBSPT cellop nodeop
```

This function folds up the tree into a single value. It may be thought of as the result of the expression tree formed from the BSPT when each partitioning node is replaced by the function `nodeop part nodeinfo` that further expects two arguments — the results of the fold on its children. Each leaf node produces a value determined by `cellop` applied to the items stored in the cell.

Now we can provide unfilled rendering with the function:

```
> drawBSPT :: BSPT -> String
> drawBSPT = foldBSPT (const3 []) drawEmbedded
>            where
>            drawEmbedded _ (faces,_) x y = drawFaces faces ++ x ++ y
```

The function `const3` takes four arguments and returns its first, ignoring the others.

As another example of the use of `foldBSPT`, consider the function `countleaves`. The function `countleaves` is used when a sub-tree is too big to be displayed in the tree-form window. Instead an indication of the size of the sub-tree is printed. The indication used is the number of `Cell` leaves in that tree. In terms of `foldBSPT`, `countleaves` is:

```
> countleaves :: BSPT -> Int
> countleaves = foldBSPT (const3 1) plus
>               where
>               plus _ _ = (+)
```

The point classification algorithm is a further application of `foldBSPT`. Point classification, with respect to a BSPT, is accomplished as follows:

- If the BSPT is a cell, classify the point according to the status of the cell. See Figure 6, points I and II.
- If the BSPT is partitioned, determine whether the point lies in the rear halfspace, the fore halfspace or is coincident with the partition.

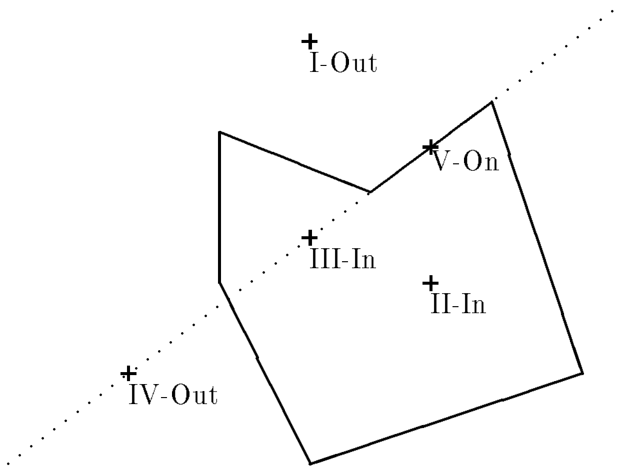


Figure 6: Object and classifications of three points.

- If it is to the rear, classify the point with respect to the left child of the partitioning node.
- If it is to the fore, classify the point with respect to the right child of the partitioning node.
- If it is coincident with respect to the partition, classify the point with respect to both sub-trees.
    * If the classification for each is the same then this is the classification of the point. See Figure 6, points III and IV
    * If the classifications differ then the point must be on the edge of the object, return `On`. See Figure 6, point V.

Although classification requires only a partial traversal of the tree, because we employ lazy evaluation, we can still implement `classifyPoint` efficiently with `foldBSPT` as follows.

```
> classifyPoint :: Point -> BSPT -> Status
> classifyPoint pt = foldBSPT status (deter pt)
>                       where
>                       status s _ _ = s
>                       deter pt (Fc _ part) _ = deter' (space part pt)
>                             where
>                             deter' Fore _ x = x
>                             deter' Rear x _ = x
>                             deter' At x y | x==y = x
>                                           | otherwise = On
```

3.4.2 Filled rendering of BSPT objects

There are a number of algorithms in common use for filled rendering of objects. A naive algorithm is to use `classifyPoint` on every pixel to determine its status and fill in those pixels that are either `On` or `In`. This is far too inefficient. A second method, the *seed fill* algorithm[NS81], involves finding one pixel that is inside the object, and recursing in each direction, filling in pixels until the edge is reached. There are complications — finding the inside pixels in the first place for instance. Instead we take the *scan-line*[NS81] filling approach to filled object rendering. This involves casting a line across each pixel line of the screen and filling in sections of that line that are within the object. We find that another higher-order function plays a part here. Let us define a function, `segments` whose purpose is to split up a given face into a list of sections where each section runs through a different cell of the BSPT tree. These pieces then have the function `cellop`, partially applied to the status of the cell in question, applied to them.

```
> segments :: (Status->Face->[a]) -> BSPT -> Face -> [a]
> segments cellop (Cell status _ _) face = cellop status face
> segments cellop (BSP part@(Fc _ p) _ left right) face@(Fc fs _)
>               = case (location p fs) of
>                     Coincident -> cellop In face
>                     Intersects -> segments cellop left leftside ++
>                                     segments cellop right rightside
>                     ToTheRear  -> segments cellop left face
>                     ToTheFore  -> segments cellop right face
>               where
>               (leftside,rightside) = bisect face p
```

To produce one scan line, we use `segment` with `cellop` ignoring the segments that pass through `Out` cells and only returning those that cut `In` cells.

```
> scanLine :: BSPT -> Face -> Faces
> scanLine = segments filterInside
>               where
>               filterInside In face = [face]
>               filterInside Out _ = []
```

Now the function to provide filled rendering, `render`, simply has to apply `scanLine tree` to each of the scan lines of the rendering window, concatenate the result to produce a list of faces that need drawing and pass this as an argument to `drawFaces`.

```
> render :: BSPT -> String
> render btree = drawFaces (mapcat (scanLine btree) scanLines)
```

3.5 Merging BSPT represented objects

Once we can build BSPT representations of objects and display them, it is natural to want to manipulate them. A common form of manipulation is a boolean operation, forming the union, intersection or difference of two objects A good algorithm has recently been developed [NAT90] that is symmetric: it expects both its arguments to be BSPTs. Previously boolean operations had been based on the asymmetric notion of inserting an object represented by its boundary into a BSPT tree. The new algorithm, in addition to being more aesthetically satisfying, is more efficient because the additional information yielded by the hierarchical ordering of the second operand is exploited.

Union and intersection are each defined in terms of a higher order function which *merges* two BSPT together. It does this by effectively superimposing one representation onto the other and then splitting one tree up into the cell regions of the other. The way the algorithm works, not only does it not matter which tree is partitioned but further we can switch this decision during the execution of the algorithm. The semantics of the merge are given by a set of rules for merging a cell with an arbitrary tree, and with appropriate rules we yield either `union` as

```
> union = merge rules
>         where
>         rules :: BSPT -> BSPT -> BSPT
>         rules cell@(Cell In _ _) tree = cell
>         rules cell@(Cell Out _ _) tree = tree
>         rules tree cell@(Cell In _ _) = cell
>         rules tree cell@(Cell Out _ _) = tree
```

or `intersection` as

```
> intersection = merge rules
>         where
>         rules :: BSPT -> BSPT -> BSPT
>         rules cell@(Cell In _ _) tree = tree
>         rules cell@(Cell Out _ _) tree = cell
>         rules tree cell@(Cell In _ _) = tree
>         rules tree cell@(Cell Out _ _) = cell
```

The merge algorithm is based on the invariant that we will always be merging trees which represent a common region. We start with two BSPT representations of the same space. Unless one of the BSPTs represents a `Cell`, we split each tree with the same partition. This splits the space that each represents into the *same* two sub-spaces. The respective sub-spaces can then be merged in turn.

```
> merge :: (BSPT -> BSPT -> BSPT) -> BSPT -> BSPT -> BSPT
> merge op (Cell x r a) tree = op (Cell x r a) tree
> merge op tree (Cell x r a) = op tree (Cell x r a)
> merge op (BSP p nodeinfo left right) tree
>             = bsp'' p nodeinfo left' right'
>               where
>               left'= merge op left rear
>               right'= merge op right fore
>               (rear,fore) = partTree p tree
```

The first two clauses deal with the cases when a cell is superimposed on a tree or vice versa, they merely apply the semantic function `op` to the two trees. In order to merge non-trivial trees, *tree1* and *tree2*, we need to split each into trees representing objects in the regions:

1. Partition *tree2* by the root partition of *tree1*. This forms two trees — the BSPT representations of the objects in the rear and fore halfspaces of the root partition of *tree1*.[2]
2. The generated trees representing the rear and fore halfspaces are then merged with the left and right tree of *tree1* respectively.
3. The respective results of the sub-merges above are the left and right children of the resulting tree with the partition and embedded faces of *tree1* and representing the same region as *tree1*.

The hard work has been delegated to the function `partTree`.

```
> partTree :: Partition -> BSPT -> (BSPT,BSPT)
> partTree part (Cell x region a)
>               = (mkCell x (newRegion region part),
>                  mkCell x (newRegion region (flip part)))
> partTree part@(Fc sp p) tree@(BSP (Fc st t) (_,region) _ _)
>       = case (location p st, location t sp) of
>                 (Coincident,_)        -> if p==t
>                                          then onParallel part tree
>                                          else onAntiparallel part tree
>                 (ToTheFore,ToTheFore) -> pinPostinPos part tree
>                 (ToTheFore,ToTheRear) -> pinNegtinPos part tree
>                 (ToTheRear,ToTheFore) -> pinPostinNeg part tree
>                 (ToTheRear,ToTheRear) -> pinNegtinNeg part tree
>                 (_,_)                 -> inBoth part tree
```

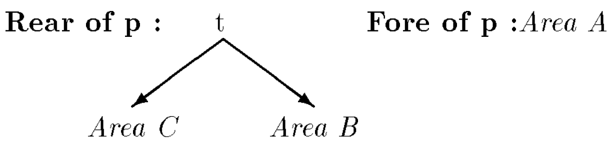
[2] The opposite arrangement could be taken. Some function could be used to decide which split would be most beneficial since it is desirable to perform the split on the smaller tree. Unfortunately we do not have the tree sizes immediately available. Future work should decide whether it pays to determine the tree sizes here or whether an indication of tree size should also be added to other information stored at the node.

Partitioning a cell is easy, we just generate two cells with the same status as the original but with a reduced region and area. Partitioning an already partition tree is more involved: `partTree` first needs to determine the relationship between the partition, `p`, and the partition of the root of the tree, `t`. It is important to note that this relationship is determined with respect to their common region. The 7 possible relationships are shown in Figure 7:

1. partitions `p` and `t` are coincident and parallel, see Figure 7.1;
2. partitions `p` and `t` are coincident and antiparallel, see Figure 7.2;
3. partition `p` is in the fore halfspace of `t` and partition `t` is in the fore halfspace of `p`, see Figure 7.3;
4. partition `p` is in the fore halfspace of `t` and partition `t` is in the rear halfspace of `p`, see Figure 7.4;
5. partition `p` is in the rear halfspace of `t` and partition `t` is in the fore halfspace of `p`, see Figure 7.5;
6. partition `p` is in the rear halfspace of `t` and partition `t` is in the rear halfspace of `p`, see Figure 7.6;
7. partition `p` is in the both halfspace of `t` and partition `t` is in the both halfspace of `p`, see Figure 7.7.

To determine which of these cases applies we must have access to the segment of each line that runs though the current region. This is where we find a use for the segment stored with the partition. We maintain this segment to be the section of the partitioning line that runs through the region defined at the node. The function `mkPart` generates the section that runs through the region when the tree is built. This is done using the function `section`. Each stored segment is compared against the opposite partition using `location`; the results of both operations are used to select a partitioning function which will bisect `tree`. The functions `section` and `location` are discussed in Section 4.

Let us consider one partitioning function, say `pinPostinNeg` (case 4 above), to see how it goes about producing the partitioning of the tree `(BSP t (face,region) tRear tFore)` into two using the partition `p`. Consider the situation, as shown in figure 8. What should we return as the trees representing the rear and fore halfspaces of `p`? Clearly the following is required:



But what are *Areas A,B* and *C*? *Area C* is simply the left child of the tree being partitioned, `tRear`. The *Areas A* and *B* can be generated by recursively partitioning the right child, `tFore` by the partition `p`. That is (*Area A*,*Area B*) = `partTree p tFore`.

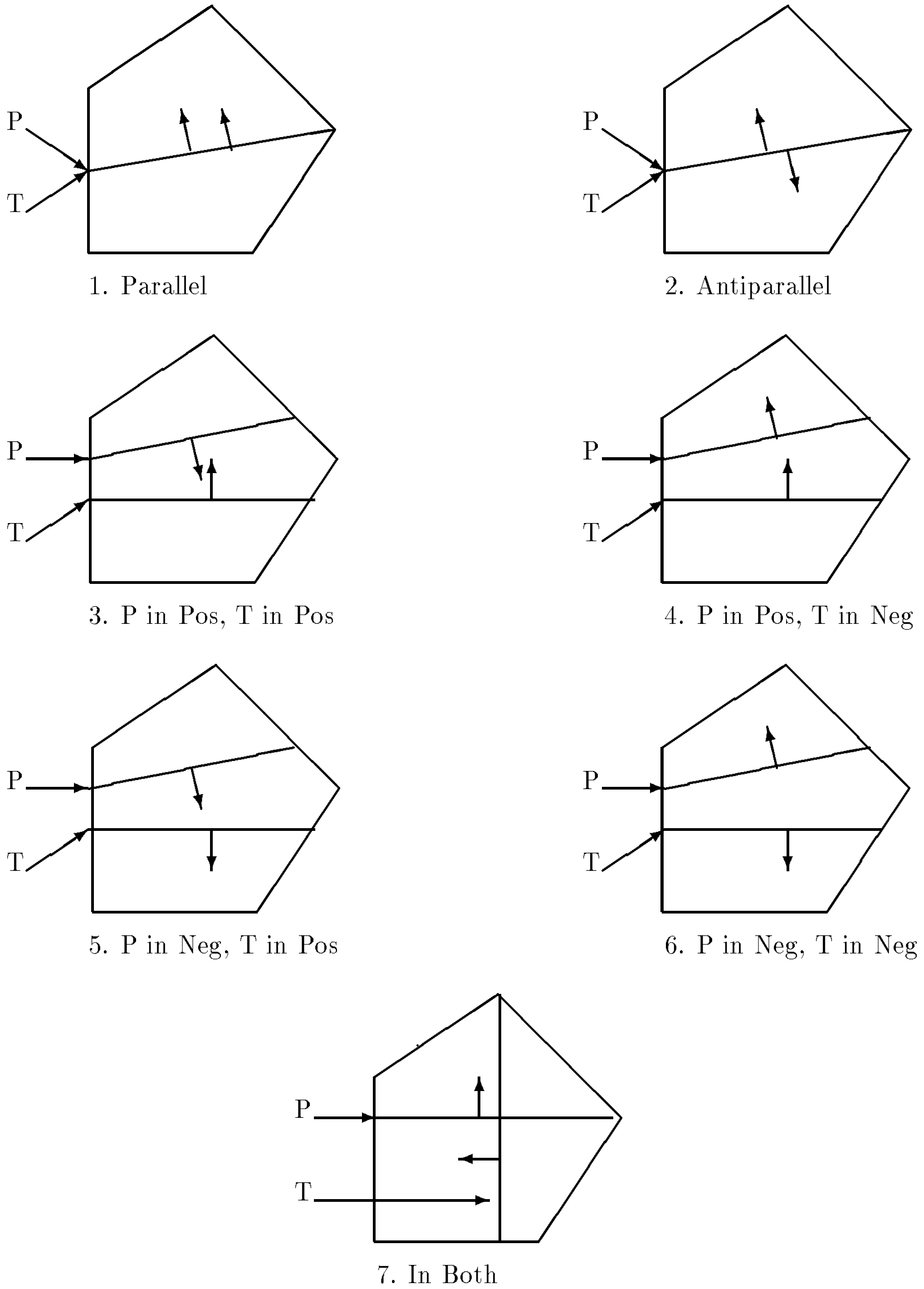


Figure 7: Possible relationships between two partitions in a region.

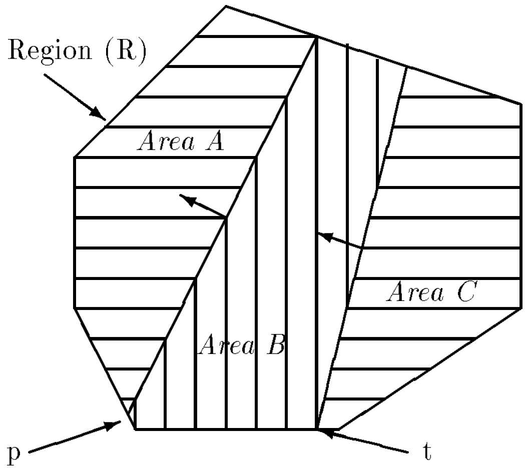


Figure 8: Primary partitions (p and t) of objects

```
> pinPostinNeg :: Partition -> BSPT -> (BSPT,BSPT)
> pinPostinNeg p (BSP t (faces,region) tRear tFore) =
>       (bsp' t (faces,newRegion region p) tRear tForepRear,
>        tForepFore)
>       where
>       (tForepRear,tForepFore) = partTree p tFore
```

The partitioning functions `pinPostinPos`, `pinNegtinPos`, `pinNegtinNeg` follow a similar pattern.

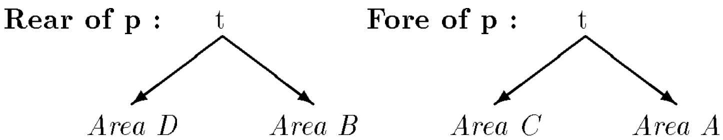
Consider the case for the partitioning function `parallel`. See Figure 9. Clearly this situation is much simpler since the area in the rear halfspace of `p` is just `tRear` and the fore halfspace is just `tFore`.

```
> onParallel p (BSP t _ rear fore) = (rear,fore)
```

The antiparallel function just reverses this situation.

```
> onAntiparallel p (BSP t _ rear fore) = (fore,rear)
```

This leaves us with just the intersecting case to consider. For the partitioner `inBoth`, the situation is shown in Figure 10. Now the fore and rear halfspace of `p` are:



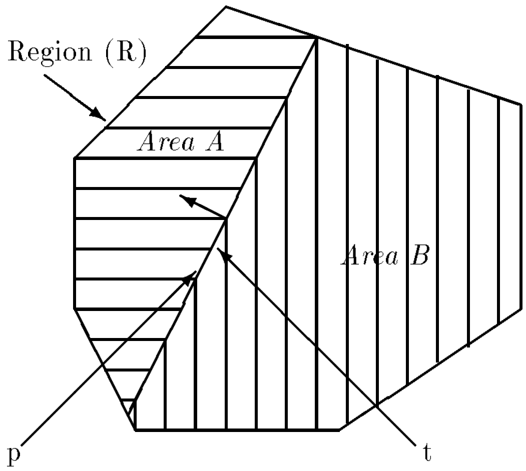


Figure 9: Situation for case `p` and `t` parallel

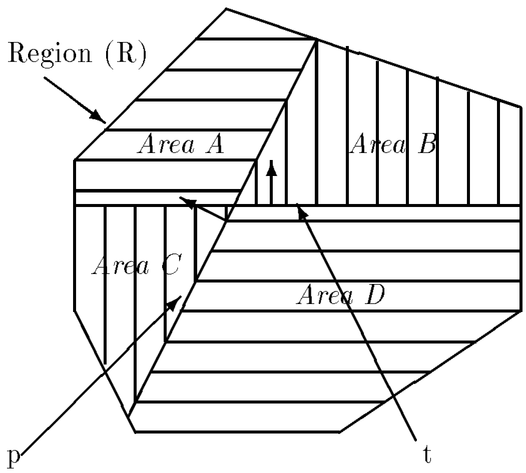


Figure 10: The situation for partitioning function `inBoth`

Where (*Area A*,*Area B*) = `partTree p tFore` and (*Area C*,*Area D*) = `partTree p tRear`.

```
> inBoth :: Partition -> BSPT -> (BSPT,BSPT)
> inBoth p (BSP t (faces,region) tRear tFore) =
>       (bsp' tLeft (rearFaces,leftReg) tRearpRear tForepRear,
>        bsp' tRight (foreFaces,rightReg) tRearpFore tForepFore)
>       where
>       (tRearpRear,tRearpFore) = partTree pLeft tRear
>       (tForepRear,tForepFore) = partTree pRight tFore
>       (rearFaces,_,foreFaces) = partFaces p' faces
>       (leftReg,rightReg) = mappair (newRegion region) (pLeft,pRight)
>       (tLeft,tRight) = bisect t p'
>       (pLeft,pRight) = bisect p (getPart t)
>       p' = getPart p
```

The faces embedded in the partition `t` need to be split by the partition `p`; this is a reuse of the function `partFaces`.

We now have the code in place for `union` and `intersection`. Subtraction of objects is by definition:

```
subtract x y = intersect x (complement y)
```

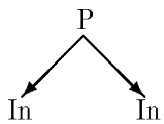
with `complement` being defined using `foldBSPT`.

```
> complement :: BSPT -> BSPT
> complement = foldBSPT compCell BSP
>              where
>              compCell In  = Cell Out
>              compCell Out = Cell In
```

3.6 BSPT minimality and maintaining embedded faces.

In the last section the functions `bsp'` and `bsp''` were frequently used in place of the `BSP` constructor. This section explains why they should be needed and how they work.

Consider the union of the regions shown in Figure 11. If we take the union of both left children we get `(Cell In)` and if we take the union of both right children we also get `(Cell In)`. Now, if `merge` used `BSP` we would create the following tree:



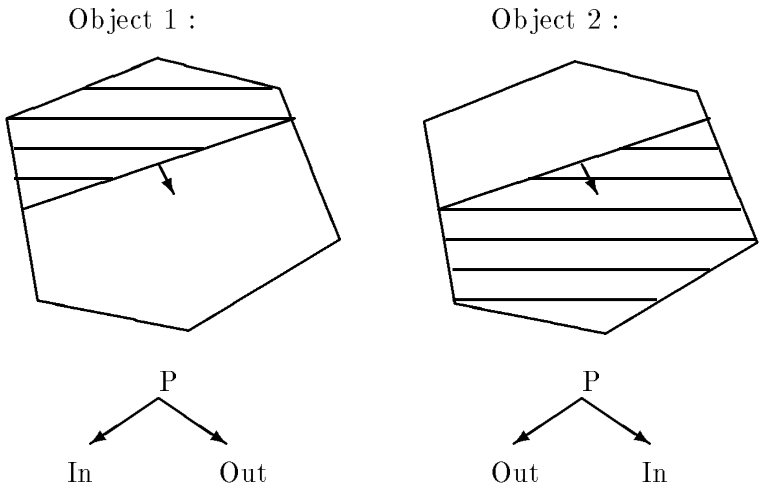


Figure 11: Two regions and the BSPT representation of their contents.

Clearly the partition `P` is redundant and the region should be represented simply as `(Cell In)`. This is the purpose of `bsp'`. It checks that the two prospective children are not cells of the same status. If they are it condenses them into one cell, otherwise it just constructs the tree as normal using `BSP`.

```
> bsp' :: Partition -> (Faces,Region) -> BSPT -> BSPT -> BSPT
> bsp' part (faces,region) (Cell x _ a) (Cell y _ b) | x==y
>                                              = Cell x region (a+b)
> bsp' part nodeInfo left right = BSP part nodeInfo left right
```

Notice that we can simply form the area of the new condensed cell by adding the areas of the previous two cells.

Consider the union of the two objects in Figure 12. If we do not update the faces embedded at nodes during a merge operation we will retain faces that belonged to one of the constituent objects even if they do not lie on the boundary of the resulting object.

It is the job of `bsp''` to maintain the appropriate faces stored in nodes. We find the higher-order function `segment` useful again. The algorithm for maintaining faces is based on the observation that for the face (or a part of it) to be on the border it must have different classifications with respect to each of its children. If the classification of a face with respect to both sub-trees is the same then it cannot be on the border. The algorithm goes like this:

- Classify the face with respect to the left sub-tree, splitting the face into smaller sections as necessary. Produce a list of line sections tagged with their classification.
- Classify each section with respect to the right sub-tree, splitting each section further if necessary. Erase any section whose classification remains the same.

The first step is implemented by `classifyFace`.

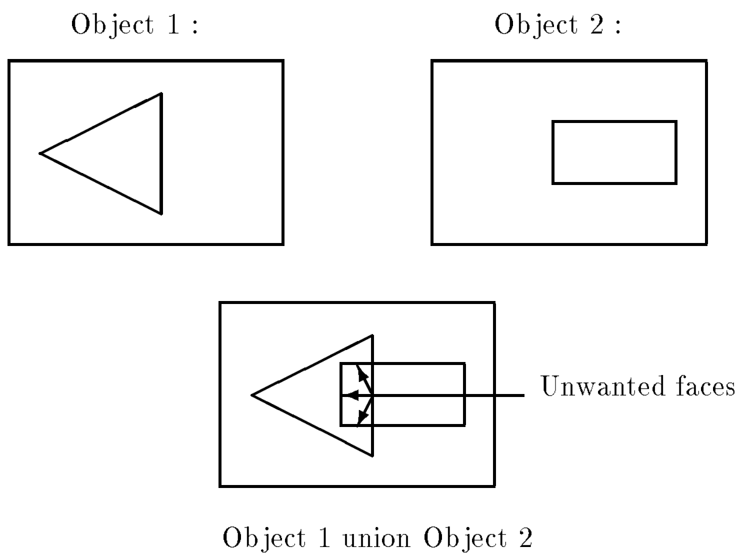


Figure 12: Two objects and their union.

```
> classifyFace :: BSPT -> Faces -> [(Face,Status)]
> classifyFace tree = mapcat (segments tagStatus tree)
>                       where
>                       tagStatus x face = [(face,x)]
```

The second step is implemented by `rubout`.

```
> rubout :: BSPT -> (Face,Status) -> Faces
> rubout tree (face,x) = segments (erase x) tree face
>       where erase x y face | x==y = []
>                            | otherwise = [face]
```

Putting these together, we define `updateFaces`:

```
> updateFaces :: BSPT -> BSPT -> Faces -> Faces
> updateFaces left right = mapcat (rubout right).classifyFace left
```

Now `bsp''` is written as follows:

```
> bsp'' :: Partition -> (Faces, Region) -> BSPT -> BSPT -> BSPT
> bsp'' part (faces,region) left right =
>       if newfaces==[]
>       then simplify region part left right
>       else BSP part (newfaces,region) left right
>       where
>       newfaces = updateFaces left right faces
```

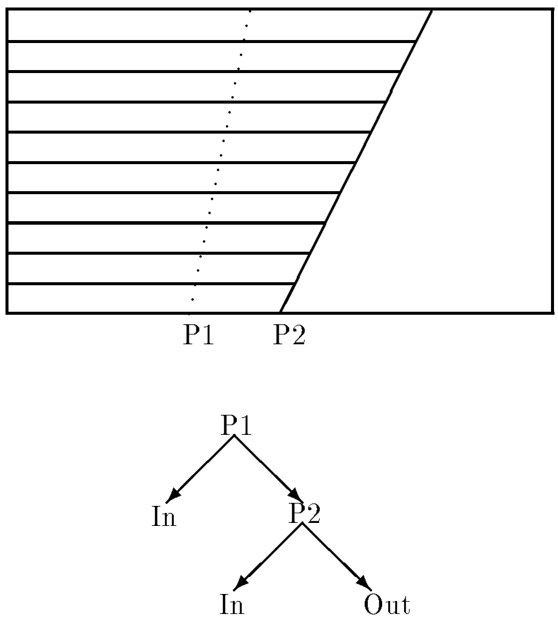
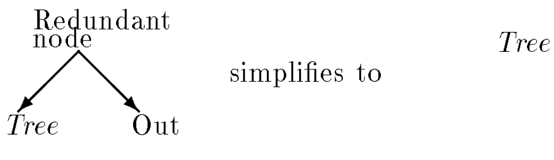


Figure 13: Object with redundant partition (P1)

Sometimes a merge can produce a situation as in Figure 13 where a redundant node exists. A redundant node is a node whose partitition no longer has any face of the object embedded within it. The function `simplify` is used to remove some, but not all, such nodes and rearrange the tree. It is only feasible to simplify trees when one child of the redundant node is a `Cell`. There is no published algorithm for rearranging two partitioned trees, chiefly because such an operation will be too expensive. To remove a redundant node `simplify` merely promotes the non-trivial child up the tree to take its place.



We must be careful to preserve the region data, hence the region argument to `simplify`.

```
> simplify :: Partition -> Region -> BSPT -> BSPT -> BSPT
> simplify _ region (Cell _ _ _) (BSP part (faces,_) left right) =
>       BSP (mkPart region (getPart part)) (faces,region) left right
> simplify _ region (BSP part (faces,_) left right) (Cell _ _ _) =
>       BSP (mkPart region (getPart part)) (faces,region) left right
> simplify part region tree1 tree2 = bsp' part ([],region) tree1 tree2
```

3.7 Calculation of the Area of BSPT represented object

Let us now consider the calculation of the area of a BSPT. Finding the area of an object that is represented as a BSPT is simply a matter of summing the areas of each of the `In` cells — recall that areas are memoised at cells. We perform this summation once again using the function `foldBSPT`.

```
> area :: BSPT -> Numb
> area = foldBSPT sumInRegions plus
>               where
>               sumInRegions In _ a = a
>               sumInRegions _ _ _ = 0
>               plus _ _ = (+)
```

The auxiliary `sumInRegion` returns `0` when an `Out` cell is visited and returns the area of an `In` cell.

Producing the area of a cell in the first place is quite a complex task. This is because we do not have available an ordered set of vertices for the region (which is convex) stored at the leaf. However a `Region` a made up of a list of `Face`s each of which contains segments that bound that region. The set of end points of these segments is a superset of the vertices requiredregion. The function `findVertices` (see Section 4) yields the vertices of the region. Because the points are unordered we appeal once again to the notion of space partitioning to convert the convex polygon into a set of disjoint triangles. This is done by the function `triangles` which takes a list of points representing vertices of the convex polygon (in no particular order).

```
> triangles :: [Point] -> [[Point]]
> triangles [p1,p2,p3] = [[p1,p2,p3]]
> triangles (p1:p2:ps) = if left/=[] && right /=[] then
>                           triangles (p1:p2:left)
>                           ++ triangles (p1:p2:right)
>                        else triangles (p1:ps++[p2])
>                        where
>                        (left,right) = partPoints (convert p1 p2) ps
```

If the region is defined by three points, it forms a triangle. If there are four or more points these points are split into two, smaller, groups. This is done by selecting two points to form a partition and by splitting the remaining points according to the halfspace of the partition they lie in. A partition that produces an empty group for one side of the partition or the other is discarded and a new partition is tried. The smaller groups will either form triangles or will be further partitioned until they do.

The function `partPoints` is given by:

```
> partPoints :: Line -> [Point] -> ([Point],[Point])
> partPoints eqn [] = ([],[])
> partPoints eqn (p:pts) = if toBack p eqn
>                          then (p:left,right)
>                          else (left,p:right)
>                          where
>                          (left,right) = partPoints eqn pts
```

Now we define a function `areaRegion` that takes a convex region and produces the area of that region.

```
> areaRegion :: Region -> Numb
> areaRegion = sum.map triangleArea.triangles.findVertices
```

This converts the region into a list of its vertices, splits the region into a list of triangles, uses `map` to get the area of each and then sums the resulting list.

4 Euclidean Module Group

The modules in this group are responsible for the representation of Euclidean objects. These include `Points`, `Lines` and `Segments`. Let us start by stating our definition of these objects.

```
> data Point = Pt Numb Numb deriving Eq
```

A point is just a pair of numbers tagged `Pt`. The point `(Pt` x y`)` represents the point (x,y). The type Numb is a type synonym and refers to the current number system in use. We defer discussion of the number system type, `Numb`, until Section 5.

```
> data Line = Ln Numb Numb Numb deriving Eq
```

A line is a triple of numbers tagged `Ln`. The line `(Ln` a b c`)` represents the line described by the equation $ax + by + c = 0$.

```
> type Segment = (Point,Point)
```

A segment (a section of a line) is described by its end points. Thus `(Pt` $x1$ $y1$`, Pt` $x2$ $y2$`)` represents the line from $(x1,y1)$ to $(x2,y2)$.

4.1 Classification of Segments and Points

As we have seen, both the building and merging of BSPTs has involved use of the function `location`. This returns a constructor from the type `Location`.

```
> data Location = Coincident | Intersects |
>                 ToTheRear | ToTheFore deriving Eq

> location :: Line -> Segment -> Location
> location line (p1,p2) = case (locale p1,locale p2) of
>                         (Coin,Coin)     -> Coincident
>                         (Fore,Rear)     -> Intersects
>                         (Rear,Fore)     -> Intersects
>                         (Rear,_)        -> ToTheRear
>                         (_,Rear)        -> ToTheRear
>                         (_,_)           -> ToTheFore
>                         where
>                         locale = space line
```

The location of the face with respect to the line given is in turn determined by the halfspace classification of the two end points of the face with respect to the line, determined by `space`.

Now that we have reduced face classification to classification of its end points, we should consider how the latter operation is done. Given a point `Pt x y` and a line `Ln a b c`, the point is

- in the **fore** halfspace if $ax + by + c > 0$
- **coincident** to the partitioning line if $ax + by + c = 0$
- in the **rear** halfspace if $ax + by + c < 0$

We define the function `eqn` that evaluates $ax + by + c$:

```
> eqn :: Line -> Point -> Numb
> eqn (Ln a b c) (Pt x y) = (a*x) + (b*y) + c
```

We then define the function, `space` on top of `eqn` that produces the appropriate constructor of type `Halfspace` where `Halfspace` and `space` are defined as follows:

```
> data Halfspace = Fore | Coin | Rear deriving Eq
```

```
> space :: Line -> Point -> Halfspace
> space line pt = if (zero val) then Coin else
>                      if (positive val) then Fore
>                      else Rear
>                 where val = eqn line pt
```

Note the use of predicates `zero` and `positive`, we will explain why we have chosen to introduce these functions in Section 5. The function `space`, and hence `eqn`, lies at the base of most computations the modeller performs; they are probably the most frequently called functions.

4.2 Operations on Lines

In this section we briefly discuss some of the operations on lines. A line is generated from two points when a `Face` is created. The function `convert` performs this task.

```
> convert :: Point -> Point -> Line
> convert (Pt x1 y1) (Pt x2 y2) = Ln (diffy) (-diffx) (diffx*y1-diffy*x1)
>                                  where
>                                        dy=y2-y1
>                                        dx=x2-x1
>                                        (diffx,diffy) = ratio dx dy
```

Notice that the function keeps the co-efficients `diffx,diffy` whole by use of the function `ratio`. This provides the ratio of its arguments in the lowest whole number terms (see Section 5).

A second operation on lines is to alter their orientation. Recall that by convention, the line `(Ln a b c)` has the normal (`a`,`-b`) and that the direction of the normal specifies the fore halfspace of the line. To alter the orientation of the line and hence flip the fore and rear halfspaces we use the function `invert`

```
> invert :: Line -> Line
> invert (Ln a b c) = Ln (negate a) (negate b) (negate c)
```

The point where two lines intersect is often required, (c.f. `bisect, section`). This is produced by a function `solve` which uses a simple method of solving simultaneous equations to provide the point of intersection.

```
> solve :: Line -> Line -> Point
> solve (Ln a b c) (Ln d e f) | zero (a*e-b*d)
>                               = Pt 0 0
>                             | not (zero a)
>                               = solveAux (Ln a b (-c)) (Ln d e (-f))
>                             | otherwise
>                               = solveAux (Ln d e (-f)) (Ln a b (-c))

> solveAux :: Line -> Line -> Point
> solveAux (Ln a b c) (Ln 0 e f) = Pt x y
>                                 where
>                                 y = f/e
>                                 x = (c-b*y)/a
> solveAux (Ln a b c) (Ln d e f) =
>                       solveAux (Ln a b c) (Ln 0 (e-b*g) (f-c*g))
>                       where g = d/a
```

The function bisect is defined as follows:

```
> bisect :: Face -> Line -> (Face,Face)
> bisect (Fc (pt1,pt2) line1) line2 =
>      if toBack pt1 line2 then (face1,face2) else (face2,face1)
>      where
>      face1 = Fc (pt1,pti) line1
>      face2 = Fc (pti,pt2) line1
>      pti = solve line1 line2
```

The line stored in the face and the partitioning line are passed to solve to generate the point of intersection. Two new segments are generated each upon the original line and hence tagged as such. The faces are returned such that the first element of the pair is the face in the rear halfspace of the partition. The simple predicate toBack is used to arrange this.

```
> toBack :: Point -> Line -> Bool
> toBack pt line = space line pt /= Fore
```

The function section produces the section of a line that runs through a given region.

```
> section :: Region -> Line -> Segment
> section region line = f points
>       where
>       points = [x| x <- intersections, inRegion region x]
>       intersections = map (solve line.getPart) (getRegion region)
>       f [pta,ptb] = (pta,ptb)
>       f a = f (mkset a)
```

Recall that a Region is defined as a tagged list of Faces and represents the region defined by the intersection of the rear halfspaces of those lines stored in the list. The function above produces the intersection of the line whose section is required with each of the lines forming the region. From this list are filtered all the points not in the required region (using inRegion see below). This will usually leave two points that are the end points of the section required. Sometimes, a line may intersect with a vertex of the region in this case both lines which come to form this vertex will produce a point. It is thus possible for the list points to contain 2,3 or even 4 points. If 3 or 4 points are generated there will only be two distinct values among them so in these cases we apply the function mkset to points to remove duplicate points from the list. This will finally leave two points.

inRegion is defined as follows

```
> inRegion :: Region -> Point -> Bool
> inRegion region pt = all (map (toBack pt.getPart) (getRegion region))
```

A point is in a region if it is in the rear halfspace of all the lines forming the region. inRegion maps the predicate toBack over the lines of the region and uses the all predicate to check every result was true.

Finally, we need a function to produce the vertices of a region.

```
> findVertices :: Region -> [Point]
> findVertices region = [pts | pts <- xs ++ ys, inRegion region pts]
>       where
>       xs = [x | (x,_) <- segments]
>       ys = [y | (_,y) <- segments]
>       segments = map getSegment (getRegion region)
```

This just creates a list of the possible vertices and then filters away those not in the required region.

5 Numbers Module Group

Geometric operations in Section 4 were parametrised over a type Numb. This section explains why this is done and why numbers are of interest for the GMS.

In constructing the prototype, the type num supplied by Miranda was used. Unfortunately this number system hides a few problems. In Miranda the type num is an *ad hoc* amalgam of arbitrarily sized integers and floating point numbers. Floating point numbers come into calculations when points are generated from line intersections, all other points can be expressed as integers because they are generated by the mouse from points on the screen.

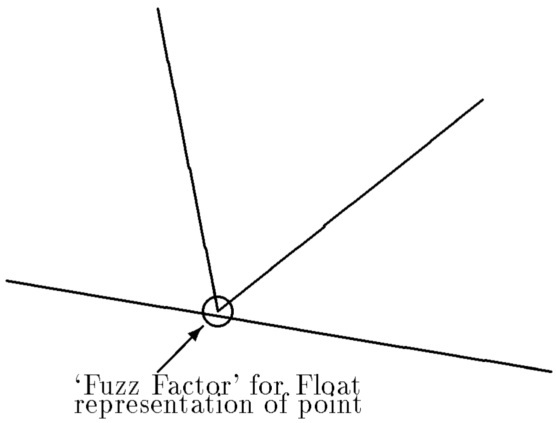


Figure 14: Floats incur errors

Unfortunately, floating point numbers are inherently inexact. While much numerical analysis work goes into quantifying and reducing the inaccuracies, Knuth sums up the situation as follows:

> *Many a serious mathematician has attempted to give rigorous analysis of a sequence of floating-point operations, but has found the task to be so formidable he has tried to content himself with plausibility arguments instead.* — [Knu69]

In geometric modelling, we cannot content ourselves with plausibility. Working with numbers of unknown accuracy could lead to great problems. Suppose that we are using `Floats` in Haskell as a representation of numbers. Further consider a point that lies very close to a line as in Figure 14. Its approximate representation using floats, somewhere within the circle indicated, may well take it onto the line or even to the other side of the line. This will have a dramatic effect on the object being modelled. Programmers can get around this problem using ad hoc means. For instance, the point in question could be tagged stating its relationship to the line. However, such measures are unacceptable in that they make for unclear programs and incur unnecessary overheads. We dismiss floating point numbers. An alternative numbering system is required.

The GMS is parametrised over its number system, so that changing the system becomes a simple task. We define a type synonym `Numb` that is used throughout the program. The number system can be instantiated by a declaration:

`> type Numb =` *current_number_system*

Further we define a class, `GeomNumb`, that specifies a number of operations that are essential to the GMS but whose methods may differ between number systems.

Here is the definition of the GeomNumb class:

```
> class (Fractional a) => GeomNumb a where
>       rnd :: a->Int
>       zero,positive,negative :: a->Bool
>       ratio :: a -> a -> (a,a)
>       xCoordInRange :: a->Bool
>       yCoordInRange :: a->Bool
```

The class requires methods for

- rounding a number to the nearest Int. This is needed to provide the nearest pixel when points are displayed.
- predicates to check a number of the class for being zero, positive and negative.
- produces the ratio of two numbers in the class. When the instance is for Rationals, we can return the ratio as a canonical form by dividing the first argument by the second and simplifying the result. This is not possible for the Float instance and so ratio will just return its two arguments as a pair.
- determining whether x or y co-ordinates are on the screen. Needed for filtering away those points that are not. The screen is defined in the module Params.

A partial instance declaration of GeomNum for Rationals is as follows:

```
> instance GeomNumb Rational where
>       rnd = rndRationals
>       zero x = x==0
>       positive = x>0
>       negative = x<0
>       ratio = ratRationals
>       xCoordInRange x = (fromInt renderLeft) <= x &&
>                             x <= (fromInt renderRight)
>       yCoordInRange y = (fromInt renderTop) <= y &&
>                             y <= (fromInt renderBottom)
```

Now consider declaring Float as an instance of GeomNumb. The rather ugly definition of zero would be

```
> zero :: Float -> Bool
> zero x = abs x <= eps
```

where the value eps is defined to be something small with respect to the context! This suggests that eps should be an argument to zero, however the Haskell Class system will not allow such overloading of functions.

Given that we have dismissed `Float`s as a possible number system due to the inaccuracies involved, what are the alternatives?

5.1 Ratios of Ints

In addition to the space and time benefits, the memoising of line equations for segments also solves our number problems. With memoised lines it is possible to work with ratios of `Int`s. This is the case because all lines are derived from pixels points on the screen originally and ratios of `Int`s are powerful enough to represent all such lines. If lines were not memoised with segments, when a face, say (p_1, p_2), is split into two parts, $(p_1, p_i), (p_i, p_2)$, the point p_i is not necessarily a simple pixel point. We would eventually need to derive the line for, say, (p_1, p_i), and although the canonical form of this line is representable, because p_i is complex it is possible that `Int`eger overflow will take place before this canonical form is reached.

This solution to our number problem is somewhat domain dependent. A larger screen, for instance, could render `Int`s insufficiently powerful, as could additional functionality in the GMS which required lines to be generated from non-pixel points. It is therefore worth looking at some options explored before the ramifications of memoing line data became apparent.

5.2 Rationals (Ratios of Integer)

While Rationals are powerful enough to represent any line required by the GMS application, the implementation of `Rational`s provided in the prototype Glasgow Haskell compiler does not perform well enough for this application. This is a fault of the *Integer* package rather than the *Ratio* package. The problem with Integers is that they are implemented in Haskell itself. We really need Integers to be added to the run time system.

5.3 LazyRationals/LazyIntegers

Arithmetic operations on Integers are expensive; if we can avoid any such calculations it would be beneficial to do so. This is the motivation for introducing LazyRationals. We observe that to locate the halfspace a point lies in, we need only know whether $ax + by + c$ is *positive*, *negative* or *zero*. That is, we do not care about the actual value of $ax + by + c$. Using tagged numbers and implementing the sign rules as below we can get at this result lazily.

```
> Pos a * Pos b = Pos (a*b)
> Pos a * Neg b = Neg (a*b)
> Neg a * Pos b = Neg (a*b)
> Neg a * Neg b = Pos (a*b)
> Zero * _ = Zero
> _ * Zero = Zero

> Pos a + Pos b = Pos (a+b)
> Pos a + Neg b = ?????
> Neg a + Pos b = ?????
> Neg a + Pos b = Neg (a+b)
> Zero + a = a
> a + Zero = a
```

Although the rules for multiplication are straightforward, addition runs into problems when numbers of differing signs are involved. We cannot produce a sign from such an addition without at least determining which of the operands is bigger. This motivates the idea that the numbers themselves should be lazy. We should not necessarily have to have to calculate the values of a and b to determine whether $a > b$. Consider, for instance, the example given by [Run89b]

$$complicated + 1 > 0$$

where *complicated* is an expensive operation producing a natural number. We can clearly deduce from the information given that the expression is `True`.

During certain operations on BSP trees, intersections of lines generate points that lie off of the screen. Such points are of no interest at all — they are immediately filtered away. Suppose *expr* represents the intersection of two lines, an expensive operation. Without lazy numbers, filtering involves evaluating *expr* fully to the form (`Point` x y) and comparing x and y with the parameters of the screen.

Consider the following definitions:

```
data LazyInteger = Pos Numlist | Zero | Neg Numlist
type Numlist = [Int]
```

The `Numlist` represents an integer where each element is a digit to an arbitrary base (but greater than the screen size), and the least significant digit is at the head with the most significant being at the tail. Say 1000 is chosen as the base. The point $(1444, 300)$ is thus represented as `Pt (Pos [444,1]) (Pos [300])`. Now filtering can be performed with the predicate below applied to both co-ordinates in a point:

```
onScreen (Pos [x]) = coordOnScreen x
onScreen _ = False
```

That is, any co-ordinate that does not match the pattern `Pos [x]` is filtered away immediately. This has the potential to save a lot of work.

An implementation of `LazyRationals` (module `LazyRat`) was built on top of an implementation of `LazyIntegers` (module `LazyInt`). Both were declared as instances of the `Num` class. The class system makes the definitions succinct. `Lazyintegers` are defined in terms of `Numlists` (module `Numlist`). `Numlists` cannot be defined as members of the class `Num` unfortunately. This is because they are unsigned magnitudes and therefore no sensible definition of `negate` can be given.

Having implemented `LazyIntegers` along with `LazyRationals` and experimented with them, the results were very disappointing. For instance, the union of two simple objects would take several minutes which is clearly unsatisfactory. The inefficiency comes partly because there is not that much laziness to be exploited in general, though we would expect the sign tagging to be beneficial. Another source of inefficiency comes from the implementation of the Haskell class system. As an experiment, the Numbers modules were re-written with LazyIntegers defined as an independent entity rather than as an instance of the class Num. The results on some large calculations gave a 13% increase in speed. However, it must be said that the resulting code was far less elegant and notably longer than the Class-based code.

5.4 Intervals

Duncan Sinclair [Sin90] uses an Interval numbers system in an implementation of a CSG based solid Modeller, *Fulsom*, in Haskell. Intervals are a range of values which serve to approximate a unique value within that range. For example, the interval (3,4) represents a single value between 3 and 4. Sinclair defines the type

```
data Interval  = Point Int | Int :#: Int
```

which a allows a single exact value (tagged `Point`) to be used if available. The normal arithmetic operations are defined on Intervals. For instance consider the definition of addition.

```
(+) :: Interval -> Interval -> Interval
(+) (Point a) (Point b) = Point (a+b)
(+) (Point a) (c :#: d) = (a+c) :#: (a+d)
(+) (a :#: b) (Point c) = (a+c) :#: (b+c)
(+) (a :#: b) (c :#: d) = (a+c) :#: (b+d)
```

Intervals look promising because we rarely want to know the value of number beyond the nearest whole integer. Since we cannot plot pixels to any greater accuracy, the ability to determine a single integer range is adequate. However, they would not be good for deciding whether two objects are disjoint, say.

$$V[x_0, x_1, \cdots, x_{n-1}]r(n) = x_0 + \cfrac{1}{x_1 + \cfrac{1}{\ddots + \cfrac{1}{x_{n-1} + \cfrac{1}{V[r(n)]}}}}$$

Figure 15: Representation of a real as a continued fraction

5.5 Exact Reals

There has been recent interest in the use of exact reals [Vui88] [BCOR86]. Vuillemin reports promise from systems based on continued fractions. A real is represented by a stream $< x_0, x_1, ..x_{n-1}, r(n) >$ where the first n terms are integers and the last term is a function that computes a new closure with the added integer term x_n. The value of such a stream is given by the function V as shown in Figure 15. Lester is working on another implementation of such a system [Les91].

6 Libraries Module Group

These modules form the libraries for the other modules. There are two such libraries: `Stdlib` and `MGRlib`.

The first, `Stdlib`, contains all the general auxiliaries, some of which perhaps should be in the preludes, notably `pair`, `const` and `mapcat`.

The second is a library package for interfacing with the window managing package MGR[3]. MGR is a simple window manager that picks up commands, indicated by a leading ESCAPE character, off the standard output stream. For instance the command to draw a line from $(x1, y1)$ to $(x2, y2)$ is `'\ESC':"x1,y1,x2,y2"`.

It is also possible to set up the effect of pressing the mouse button via the `setevent` function. A string is specified as the result of an action. This string is placed onto the standard input when that action takes place. Special codes can be placed in this string, one such code is `"%p"`, equivalent to the string "*x y*" where (x,y) is the position on the screen of the mouse at the time of the event taking place. Therefore, arranging that the current position of the mouse should be read when the right button is pressed is done using the call: `setevent` *right-button* `"%p"`. See `initialisemouse` in the module `Init`.

The MGR library here only contains the functions that are used in the GMS application, the MGR window manager does provide a lot more adventurous facilities.

[3] MGR is a copyright of Bellcore

7 Program Module Group

This is the smallest group, it consists of two simple modules: `Main` and `Prog`. These modules, between them, serve to define the GMS as an expression. The program is defined as a function from `String -> String`. The function of the module Main is merely to wrap this up in Haskell input/output terms, converting the program to type `[Response] -> [Request]`.

```
> main :: [Response] -> [Request]
> main ~((Str str):_) = [ReadChan stdin, AppendChan stdout (prog str)]
```

The module `Prog` contains the definition of the program, `prog`.

```
> prog ::  String -> String
> prog rawinp =
>       initialise ++ modeller (buildBSPT []) operations
>       where
>       initialise = initialiseMouse ++ initialiseScreen
>       operations = (interpret (lines rawinp))
```

The important auxiliary here is `modeller`. It takes two arguments: the first is essentially a state, but consists solely of one BSPT object (known as the *current* object). The second argument is a list of operations interactively selected by the user. The function `interpret` (see Section 8) lazily generates the operations on demand — this is essential for correct interaction. If `interpret` were strict in its argument we would require the user to specify all the operations desired before any results could be shown!

8 Interface Module Group

The interface to the geometric modeller is described by the modules in this group. The most important module is `Interface`. This defines a function, `modeller`, which implements a loop (via recursion) which invokes the operations selected by the user, performing the necessary actions and updating the state appropriately. The loop is terminated by the selection of the Quit button.

```
> modeller :: BSPT -> Operations -> String
> modeller current ((Quit,_):_) = reset
> modeller current (operation@(op,_):more) =
>       indicate op actions ++ modeller newstate more
>       where
>       (actions,newstate) = perform current operation
```

For each operation the following occurs:

- A call is made to the function `indicate`, this takes two arguments. The first indicates the button on the screen that should be highlighted for the duration of this operation. The second argument is a list of the actual actions required for the operation. These are performed and upon completion the highlighted button is returned to normal.
- A recursive call to modeller is made with a revised state and the remaining operations required by the user.

The actions and revised state are determined from the current operation by the function `perform`. This returns a pair with the first item of the pair being the list of actions appropriate to the operation and the second item being the appropriate revised state. There is a `perform` clause for each command. We shall consider representative examples only. Consider the definition of `perform` for `Area`.

```
> perform :: BSPT -> Operation -> ([String],BSPT)
> perform current (Area,_) =
>       ([toTextRegion, "Area (pixels)\n",
>         show (rnd objArea), toNoTextRegion],
>         current)
>       where
>       objArea = area current
```

The actions for the `Area` operation do the following:

- set up the text region of the screen ready for text;
- display the message "Area (pixels)";
- display the calculation of the area of the current object;
- remove the text region.

An unchanged state is passed back as second element of the pair.

Now consider the clause for defining a primitive (`Polygon`):

```
> perform :: BSPT -> Operation -> ([String],BSPT)
> perform current (Polygon,operand)
>       = ([clearRender, labelDefinePoly, drawFaces polygon,
>           "\n", clearTree, prettyPrintBSPT btree, clearRender,
>           drawBSPT btree, unlabelButtons],
>          btree)
>       where
>       btree = buildBSPT (validate polygon)
>       polygon = mkPolygon (transform operand)
```

The actions here are:

- clear the tree region;
- clear the render region;
- relabel the mouse representation with the instructions for defining a polygon on the render screen;
- draw the polygon formed from the points supplied with the operation; (Because this is done lazily as the user defines the polygon, he sees the faces on the screen.)
- clear the render screen again;
- display the BSPT;
- display polygon defined;
- unlabel the buttons on the mouse representation.

Note that `modeller` is then called with the representation of the new object as the state.

The module `Init` provides all the code required to set-up and effect changes to the user interface. It makes extensive use of the MGR library. There is very little of programming interest in the module. The screen itself is completely parametrised by the module `Params`, which contains definitions of every aspect of the screen; screen size, button positions, button sizes, window sizes *etc.*

The module `Interpret` is responsible for the interpretation of the user's input. The input to the GMS is in the form of one long string. This is immediately split into a list of strings each delimited by a newline on the input (see the definition of `prog`, Section 7). Interpretation of these strings is dependent on the context.

The `modeller` function takes a list of `Operations` as arguments.

```
> type Operations = [Operation]

> type Operation = (Command,Points)

> type Points = [Point]

> data Command = Polygon | Union | Intersect
>              | Subtract | Complement | Partition
>              | Render | Classify | Area
>              | Quit | Null deriving Eq
```

Module Group	Lines of code
Program	17
Interface	354
BSPT	264
Euclidean	128
Numbers	71
Libraries	77
Total	911

Table 3: Modules groups and code size in lines

`Operations` are produced from the input by the function `interpret`

```
> interpret :: [String] -> Operations
> interpret (head:residue) = (operation:interpret residue' )
>       where
>       (operation,residue') = toOperation command residue
>       command = (toCommand.mkPoint) head
```

which takes the next `String` off the input and treats it as a point that specifies a command by virtue of the fact that it should lie within the area of one of the buttons on the screen. The function `mkPoint` takes a string of the form "x y" and produces as a result the point `Pt` x y. If the string does not take the required form the null point is returned, `(Pt 0 0)`. The point is converted to a command by `toCommand` if the point does not refer to a button, the `Null` command is returned. Some operations are only specified fully with an associated list of points. The function `toOperation` is used to spot such operations and to extract the list of points from the input stream.

9 Conclusions

Conciseness and clarity of the GMS code are the major benefits of the functional style. The figures given in Table 3 are evidence of the conciseness. The table gives a break-down of code size by module group. The entire GMS consists of only 911 lines with the core of the program (groups BSPT and Euclidean) forming less than half this figure. The user-interface to the modeller is responsible for the majority of the rest of the program. Compare this with the approximate 16,000 lines of C code quoted by Naylor for his BSPT based system which has a only a slightly greater functionality[4].

Such figures are partly achieved by the re-use of functions: it was notable during development how often one could start coding an algorithm and then realise that what was required was already partially implemented — what was required could be obtained by

[4] Naylor's system facilitates set operations, affine transformations, calculation of metric proprerties and rendering of polyhedra

composing some existing functions, possibly with some new ones. As a result, there is a considerable amount of software re-use. The other contributing factor is the frequent use of higher order functions, notably `foldBSPT` and `segments`. Common computation patterns were frequently spotted during development and abstracted to form higher order functions, decreasing code size significantly, making programming easier and, once the reader understands the operation of such functions, making for a more readable program. To a certain extent clarity is also derived from conciseness.

This report has highlighted the benefits of memoising results. In the context of the GMS this has yielded an added benefit, since it has served to solve our numerical problems. There are two potential disadvantages of memoisation. The first is that memoising results will cost space; we should be sure that a memoised result is important enough to warrant this expense. The second is the extra complexity to the code that memoised fields in data structures impose. One method to combat such complexities is to introduce Views [BC91] [Wad87] into the language. Views can be used to complement a declared structured data type with a set of alternative structures. A View on the BSPT type could hide away the appearance of the area of a cell when this is not relevant.

While some qualms have been expressed about the applicability of the Class system [Run91] in other applications, the class system was very useful in providing the lazy arithmetic packages. This was precisely the kind of application of Classes that was considered in their design.

Our appraisal has also to consider the efficiency of the functional GMS. The current system yields adequate response times. This is the system based on ratios of Ints. Experiments with sign tagging these Ints on several sample computations produced an average slowdown by a factor of approximately 3. This is because the overhead of tagging numbers, in terms of storage and time, outweighs the savings gained by avoiding some computations, since computations with Ints have only a negligible cost. In the long term we hope that Integers will be built into the Haskell run time system and so expand the possible functionality of the GMS. It may be the case then that computations will be costly enough to make them worth avoiding. An alternative lies in the *exact reals* domain[Les91].

Even though our attempt to exploit the lazy properties of numbers in the program have not been very successful, laziness does play a part in the GMS. Notably, laziness allows us to abstract the interpretation of the user's input away from the definition of the modeller. This is a very clean method of handling the user's requirements.

In addition to being satisfied with the current response times, we can expect these to improve by going parallel. This is the next stage in the work. The sequential version of the GMS will be converted to one designed to run on the GRIP[HJ90] machine. This will involve the addition of `par` annotations and the possible re-writing of sections of code to make them more amenable to parallel execution. But there lies the real beauty of the BSPT representation scheme — it has a natural fit to the parallel execution model. The `buildBSPT` operation involves taking an object domain as a problem and, if it is not trivial, splitting the object domain into two halfspaces, the two halfspaces are independent subproblems and hence we should hope to execute them in parallel. It is not all as glorious as it sounds: in order to produce sub-problems that are not themselves trivial we will have to employ good heuristics to select a partition. Some objects (convex polygons for instance) are not going to yield anything from parallel execution because one sub-problem is always trivial.

In conclusion, the functional style yields a Geometric Modelling system that supports the claims that functional programming yields concise, clear programs. The system performs satisfactorily when running sequentially. We now look forward to the benefits that parallelism will bring.

References

[BC91] F.W. Burton and R.D. Cameron. Pattern matching with abstract data types. School of Computer Science, Simon Fraser University, British Columbia, 1991.

[BCOR86] H.J. Boehm, R.S. Cartwright, M.J. O'Donnell, and M. Riggle. Exact real arithmetic: A case study in higher order programming. In *Proccedings of the 1986 ACM conference on LISP and funtional languages*, pages 162–172, 1986.

[Che91] I.G. Checkland. Space partitioning models in computational geometry. York Functional Programming Group Memo 64, 1991.

[HJ90] K. Hammond and S. Peyton Jones. Some early experiments on the GRIP parallel reducer. GRASP group internal memo, Department of Computer Science, Glasgow, September 1990.

[Hud89] P. Hudak. Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys*, 21(3):359–411, 1989.

[Hug89] J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[HWe90] P. Hudak and P. Wadler (editors). Report on the programming language Haskell — a non-strict, purely functional language. Department of Computer Science, Glasgow, 1990.

[Jon84] S. Peyton Jones. Directions in functional programming research. In D.A. Duce, editor, *Distributed Computing Systems Programme*, chapter 14. Peter Peregrinus, 1984.

[Knu69] D.E. Knuth. *The art of computer programming. Vol 2 – Seminumerical Algorithms.* Addison-Wesley, 1969.

[Les91] D.R. Lester. Some computable numbers: An Orwell implementation of Vuillemin's continued fractions. In *Fourth Annual Workshop on Functional Programming – Draft proceedings*, pages 344–378. Department of Computer Science, Glasgow University, 1991.

[NAT90] B.F. Naylor, J. Amanatides, and W.C. Thibault. Merging BSP trees yields polyhedral set operation. In *Proceedings of the SIGGRAPH*, pages 115–124, 1990.

[NS81] W.M. Newman and R.F. Sproull. *Principles of interactive computer graphics*. Computer science series. McGraw-Hill, second edition, 1981.

[Run89a] C. Runciman. Functional languages applied to real exemplars. Detailed proposal for research submitted to Information Engineering Advanced Technology Programme, 1989.

[Run89b] C. Runciman. What about the *natural* numbers ? *Computer Languages*, 14(3):181–191, 1989.

[Run91] C. Runciman. TIP in Haskell – another exercise in functional programming. In *Fourth Annual Workshop on Functional Programming – Draft proceedings*, pages 525–538. Department of Computer Science, Glasgow University, 1991.

[Sin90] D. Sinclair. Solid modelling in Haskell. In *Workshops in Computing: Functional Programming, Glasgow*, pages 246—263. Springer Verlag, 1990.

[TN87] W.C. Thibault and B.F. Naylor. Set operations on polyhedra using binary space partitioning trees. In *Proceedings of the SIGGRAPH*, pages 153–162, 1987.

[Vui88] J.E. Vuillemin. Exact real computer arithmetic and continued fractions. In *Proccedings of the 1988 ACM conference on LISP and funtional languages*, pages 14–27, 1988.

[Wad87] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *14th Annual Symposium on the Principles of Programming Languages*, pages 307—313, 1987.