J. Functional Programming 5 (3): 415–442, July 1995 © 1995 Cambridge University Press

A polymorphic library for constructive solid geometry

J. R. DAVY AND P. M. DEW

School of Computer Studies, University of Leeds, Leeds LS2 9JT, UK

Abstract

Solid modelling using constructive solid geometry (CSG) includes many examples of stylised divide-and-conquer algorithms. We identify the sources of these recurrent patterns and describe a Geometric Evaluation Library (GEL) which captures them as higher-order functions. This library then becomes the basis of developing CSG applications quickly and concisely. GEL is currently implemented as a set of separately compiled modules in the pure functional language Hope+. We evaluate our work in terms of performance and general applicability. We also assess the benefits of the functional paradigm in this domain and the merits of programming with a set of higher-order functions.

Capsule Review

This article is concerned with the implementation of a library for Constructive Solid Geometry (CSG). Divide-and-conquer algorithms are implemented as higher-order functions in HOPE+. The authors do not choose a functional language because of their background, but for very practical reasons: they need a prototyping tool, with a flexible type scheme, that naturally supports the divide and conquer paradigm. A functional language supports all of this, and leads to a sufficiently efficient implementation.

The discussion includes a detailed description of the data types and of the function signatures. The case study is realistic in that it is not just a toy example, it considers the empty and universal set, and input/output.

It is important to read through the positive experiences reported, and to dwell on the problems encountered. The authors perceived a lack of expressiveness in working with graph structures, the long compilation times are a nuisance and for large data sets the program needs a non linear time to run. There is more work to be done by the functional community to improve on these issues. It is from 'outsiders' who implement real applications that one can learn what improvements are really desired.

It would be interesting to continue research in this direction to evaluate which types of geometric modelling operations are well implementable through functional programming and which are not.

1 *Introduction*

We present a case study in functional programming, in the field of solid modelling with Constructive Solid Geometry (CSG). It is motivated by the observation that this application domain uses many stylised algorithms based on the divide-and-conquer (D&C) paradigm. Previous work on the Mistral series of parallel solid modellers (Holliman *et al.*, 1989; Wand and Dew, 1993) has demonstrated that these algorithms are highly parallel. Hence we pose the question: *is it possible to capture these common patterns of computation in a generic way?* If so, this would enable CSG applications to be developed more rapidly and also provide a route for implicit parallelism by mapping high-level descriptions onto known parallel solutions.

In this paper we attempt to answer the above question. Specifically, we present a polymorphic *Geometric Evaluation Library* (GEL) which captures the generic characteristics of these CSG algorithms, showing how this leads to rapid program development and concise programs. The operations in GEL are also able to capture a wider range of related algorithmic structures in computational geometry.

GEL was intended as a prototype to explore the systematic use of D&C in solid modelling, with a view to future parallel implementation. It is currently implemented as a set of parameterised data types and higher-order functions in the pure functional language Hope+, though other higher-order languages such as Haskell and Miranda would be equally appropriate. A functional language was indicated since higher-order functions provide an elegant way to model recurring algorithmic patterns. Furthermore, applications in this problem domain have a natural interpretation as function evaluators, mapping representations of geometric objects into properties, predicates or processes (Tilove, 1981).

We present our work as 'outsiders' to the functional programming community, and assess these experiences of the functional paradigm in our field. This contrasts with independent, related work at Imperial College (Darlington and To, 1995), where developers of functional programming technology used solid modelling as a case study.

Hope+ is a strongly typed, higher-order functional language. It supports only a limited form of currying, which we have not used. Explicit type declarations are required for all functions, though we will sometimes omit these. Our code fragments use standard Hope+, though where we are particularly concerned with the signatures of functions we make some minor changes in the interests of conciseness and clarity: type variables such as *alpha, beta* are shown as α, β, integer, character and boolean types as *Int, Char, Bool,* and list types by square brackets. Thus the signature of the *length* function for a list would, in our notation, be

$$[\alpha] \rightarrow Int$$

We use the term *support functions* for functions which are parameters of higher-order functions.

In view of its similarity to other higher-order functional languages we omit a description of Hope+ syntax. Details can be found in Perry (1989). We avoid low-level geometric details, since our concern is primarily with the structure of the algorithms. Issues of potential parallel implementation are also omitted: an outline of generic methods can be found in Davy (1992).

In section 2 we describe some basic principles of CSG, identifying two main sources of generic algorithmic patterns. Section 3 presents the main facilities of GEL with examples of their use. Sections 4, 5 and 6 discuss respectively the implementation, performance and applicability of GEL. In section 7 we assess the merits of using functional programming in this context. Section 8 notes related work before our final conclusions in section 9.

2 Solid Modelling

Solid modelling systems represent and manipulate descriptions of three-dimensional objects. Originally developed as an underpinning technology for computer-aided design in mechanical engineering, they have also been used in a wider range of applications, including computer vision (Brooks, 1981) and molecular modelling (Muuss, 1987). The distinguishing feature of solid models is *completeness*: a description of a solid contains sufficient information to compute any geometric property of that solid.

2.1 Constructive solid geometry (CSG)

Several schemes have been devised for complete representations of solids (Requicha, 1980). One of the most important is CSG, in which solids are represented by a set of *primitive solids* combined using *regularised boolean operations* such as union, intersection and difference. It is common for a small set of *bounded primitives*, such as spheres and cones, to be available to the users of CSG systems. Internally these are often represented as a combination of simpler unbounded primitives called *halfspaces*. These are defined by functions of the form $f(x, y, z) \leq 0$ which partition space into two halves, inside and outside the primitive.

CSG represents a solid as a tree structure with primitives at the leaves and boolean operators at interior nodes. Though usually concerned with 3-dimensional shapes, CSG is applicable in any number of dimensions; we include some examples with 2-dimensional 'solids' for simplicity.

The tree structure leads naturally to recursive D&C algorithms, which compute results for primitives, then combine the results using the boolean operators when returning up the tree. Tilove (1980) identified this as a generic paradigm for CSG, applying it to *set membership classification* problems. An example used later in this paper is point membership classification (PMC), which determines whether a point is inside, outside or on the boundary of a solid. Classifications are carried out on primitives and results of subtrees are determined using simple rewrite rules: for instance if a point is 'in' two solids it is 'in' their intersection.

2.2 Spatial subdivision

It is common to convert CSG trees into secondary data structures based on *octrees* (Meagher, 1982), using a process of *spatial subdivision*. Here the (usually cubical) space in which the solid object is situated is partitioned into eight quasi-disjoint subcells. Associated with each subcell is a localised CSG tree containing only the primitives which intersect the subcell. The subcells are obtained by a hierarchical approach which adapts the depth of subdivision to the local complexity of the model. For instance, subdivision may stop when the number of primitives in a localised tree falls below some threshold.

The benefits of spatial subdivision are exemplified in ray-tracing (Holliman, Wang and Dew, 1993). Crude CSG ray-tracing is very expensive, intersecting every ray with every primitive. When ray-tracing a spatially divided model, each ray is tracked through the octree and intersects only the primitives of the localised trees encountered, giving substantial performance improvements.

There are also many algorithms which use spatial subdivision but without explicitly creating a tree: the family of algorithms for computing integral properties of solids (Lee and Requicha, 1982) is a good example.

Spatial subdivision algorithms also typically follow a D&C approach. Thus CSG computations lead to D&C algorithms both because of the primary data structure and because of frequently used spatial subdivision techniques. The GEL library was developed to enable both these algorithmic patterns to be captured in a high-level fashion.

3 The geometric evaluation library (GEL)

GEL is currently written in Hope+ and provides a set of separately compiled modules which can be called from Hope+ programs. It is based on general D&C operations and two parameterised data types: a generic CSG tree and a generic *geometric decomposition tree* (GDT). The latter generalises the octree-based solid models noted in section 2.2. In this section we discuss the basic facilities of GEL, illustrate their use, and outline some additional features.

3.1 Divide and conquer

Variants of a higher-order D&C function have been defined by several writers, such as Cole (1989) and Kelly (1989). There are four support functions: *leaf* determines whether a problem is small enough to solve directly, *divide* splits a problem into a list of subproblems, *solve* computes the direct solution of a 'small' problem, *combine* combines the results of a list of subproblems. For instance, *mergesort* can easily be implemented, with *leaf* returning true for a singleton list, *divide* splitting the list into two equal-sized sublists (GEL provides this operation as a utility), *solve* as the identity and *combine* merging sorted sequences.

GEL has two versions of this function, distinguished by whether *combine* uses only the results of subproblems or needs the original divided data. In both cases, the function is invoked as

```
divacon(data, leaf, divide, solve, combine);
```

and returns the final result of the D&C operation. Both functions are called *divacon*, relying on the Hope+ overloading facility. Their signatures are

$$\alpha \times (\alpha \to \mathit{Bool}) \times (\alpha \to [\alpha]) \times (\alpha \to \beta) \times ([\beta] \to \beta) \to \beta$$
$$\alpha \times (\alpha \to \mathit{Bool}) \times (\alpha \to [\alpha]) \times (\alpha \to \beta) \times ([\beta] \times \alpha \to \beta) \to \beta$$

Note that the functions are disambiguated by the *combine* parameter. Their implementation is straightforward, as in Cole (1989) and Kelly (1989).

3.2 CSG trees

CSG trees are binary trees with instances of primitive solids at the leaves and instances of boolean operators at interior nodes. We model them as algebraic types parameterised by primitive and operator types. Since CSG is a set-theoretic representation, there should be a means of denoting the empty set ($\emptyset$) and the universal set (Ω) in a CSG system. These considerations lead to the following type definition, in which *rho, chi* are types for primitives and operators respectively:

```
data CSG(rho, chi) ==
     Emptysolid  ++  Fullsolid  ++  Primitive (rho)  ++
     Compose (CSG(rho, chi) # chi # CSG(rho, chi));
```

Note that *chi* is simply an enumeration of the specific set of operators used, such as {*Intersection, Union, Difference*} or {*Union, Difference*}. Its inclusion as a parameter, rather than as a fixed enumeration, is motivated by the observation that CSG systems vary in both the number and specific kinds of boolean operations allowed. The actual implementation of these operations is carried out by combining functions, discussed below.

The primary higher-order operation on CSG trees is an *evaluation* of the tree. This is a D&C traversal, suitable for the family of set membership classification problems noted in section 2.1. Interpreting this in terms of the *divacon* function, we note that it needs a *solve* function to classify a primitive, and a function to *combine* the classifications. The latter will effectively implement the boolean operations for the evaluation in question. *Leaf* and *divide* can be incorporated into the schematic solution for this set of problems. A plausible higher-order function, *CSGtraverse* has the signature

$$\alpha \times CSG(\rho,\chi) \times (\rho \times \alpha \to \beta) \times (\beta \times \chi \times \beta \to \beta) \to \beta$$

where α, β are, respectively, the types of the object to be classified against the tree and the classification result. The equations implementing this function (using a characteristic pattern matching approach) are

```
--- CSGtraverse(_, Fullsolid, _, _) <= error[....];
--- CSGtraverse(_, Emptysolid ,_, _) <= error[....];
--- CSGtraverse(query, Primitive(prim), solve, _)
       <= solve(prim, query);
--- CSGtraverse(query, Compose(l,op,r), solve, combine)
       <= combine (CSGtraverse(query, l, solve, combine),
                   op,
                   CSGtraverse(query, r, solve, combine));
```

Note that the first two equations assume that CSG trees always have 'proper' primitives, that is, there are no instances of $\emptyset$ or Ω. This is not entirely realistic for at least two reasons; in a binary CSG tree a boolean complement operator must be modelled as $\overline{P} = \Omega - P$, and trees with empty primitives may exist as intermediate steps. Thus we must allow the result of carrying out a *solve* on $\emptyset$ or Ω to be specified.

Also, there are situations in which there is no 'query' data – a trivial example is an algorithm to count the number of primitives.

We again call on overloading to handle this situation, giving a family of *CSGtraverse* functions each with a slightly different signature. Invocations of two of these are

```
CSGtraverse(query, tree, solve, combine, fullsolve);
CSGtraverse(tree, solve, combine, fullsolve, emptysolve);
```

The complete set of signatures is shown in figure 1. Note that if a result for a primitive Ω is specified, a result for a primitive $\emptyset$ is a further option. The need for disambiguation precludes also allowing $\emptyset$ alone as an option; we chose to give greater importance to Ω because of its fundamental role in modelling complementation.

$$\begin{array}{l}
CSG(\rho,\chi) \times (\rho \rightarrow \beta) \times (\beta \times\chi\times \beta \rightarrow \beta) \rightarrow \beta \\
CSG(\rho,\chi) \times (\rho \rightarrow \beta) \times (\beta \times\chi\times \beta \rightarrow \beta) \times \beta \rightarrow \beta \\
CSG(\rho,\chi) \times (\rho \rightarrow \beta) \times (\beta \times\chi\times \beta \rightarrow \beta) \times \beta \times \beta \rightarrow \beta \\
\alpha \times CSG(\rho,\chi) \times (\rho \times \alpha \rightarrow \beta) \times (\beta \times\chi\times \beta \rightarrow \beta) \rightarrow \beta \\
\alpha \times CSG(\rho,\chi) \times (\rho \times \alpha \rightarrow \beta) \times (\beta \times\chi\times \beta \rightarrow \beta) \times \beta \rightarrow \beta \\
\alpha \times CSG(\rho,\chi) \times (\rho \times \alpha \rightarrow \beta) \times (\beta \times\chi\times \beta \rightarrow \beta) \times \beta \times \beta \rightarrow \beta
\end{array}$$

Fig. 1. Signatures of overloaded CSGtraverse functions

Example 1: Point membership classification

As a first example of the use of GEL we solve the Point Membership Classification problem introduced in section 2.1. (Recall that PMC determines whether a point lies on the inside, outside or boundary of a solid.) First the geometric domain must be defined, using planar, spherical, conical and cylindrical halfspaces, and a set of three boolean operators. We model a classification simply as an integer; this is an over-simplification in practice because it ignores problems of combining 'on' classifications (Tilove, 1980). Since the fuller classification structure would complicate the details without affecting the overall algorithmic structure, we use the simpler form for expository purposes.

```
type SP_HS  ==  real # POINT3;   ! radius # centre
type PL_HS  ==  real # VECTOR3;  ! dist. from Origin # normal to plane
! similarly for CY_HS, CO_HS

data HS ==  sphere(SP_HS) ++ plane(PL_HS) ++ cyl(CY_HS) ++ cone(CO_HS);
data ROP ==  Union ++ Inter ++ Diff;

type SOLID  ==  CSG(HS, ROP);    ! using generic CSG type

type CLASS  ==  num;             ! may be IN, OUT, ON.
```

PMC now requires only a single invocation of *CSGtraverse*:

```
CSGtraverse(pt, tree, classify_prim, combine_class);
```

All that remains is to implement the two support functions. *Classify_prim* contains the low-level geometric details for the set of primitives used. *Combine_class* implements the boolean operation for this problem. Its parameters are used to compute an index into a table, stored as a vector for efficiency, in which each entry gives the result of combining the relevant classifications. For instance, if a point is *IN* a subtree it is *IN* its union with any other subtree. Both these support functions use pattern matching, on the *HS* and *ROP* constructors respectively, in a simple, stylised fashion:

```
dec classify_prim : HS # POINT3 -> CLASS;
--- classify_prim(sphere(r, Point3(cx, cy, cz)), Point3(x, y, z))
  <= let (dx, dy, dz) == (x - cx, y - cy, z - cz) in
     let d == (dx * dx + dy * dy * dz * dz - r * r) in
         if d > 0.0         then   OUT
         else if d < 0.0    then   IN
         else                      ON;
! similarly for plane, cyl, cone

dec combine_classifications : CLASS # ROP # CLASS -> CLASS;
--- combine_classifications(c1, Union, c2)
  <= index(UNION_TABLE, 4 * c1 + c2 + 1);     ! Find result from table
! similarly for Inter and Diff
```

Thus the whole application has been implemented by a simple data modelling stage, invoking a single GEL operation, and supplying two support functions which implement respectively the primitive geometric evaluation and the boolean operation. This is characteristic of the way CSG operations are implemented in GEL.

3.3 Geometric decomposition trees

Section 2.2 described a second representation of solid models using octree-based spatial subdivision, in which leaves of the octree hold localised CSG trees. We generalise this structure to a *Geometric Decomposition Tree* (GDT). Noting that the interior nodes of the octree hold no geometric information, we parameterise GDTs by the type of geometric objects stored at the leaves, τ

```
data GDT(tau) == Terminal (tau) ++
                 Interior (list (GDT(tau)));
```

Clearly this captures the essence of a spatially divided octree, but its use of a list for subtrees avoids the restriction to eight-way subdivision and opens the way for more general application.

What operations are appropriate for a GDT? Clearly we need to be able to create a GDT from a CSG model (or other geometric representation). Thus we need to determine the appropriate parameters for a *GDTcreate* function. Following the pattern of *divacon* we note that *leaf* determines whether the subdivision should continue further, *divide* carries out the subdivision process. These are application-dependent and must be specified as parameters. *Solve* creates the terminal nodes, a housekeeping task which can be absorbed into the body of *GDTcreate*. It may, however, first transform the geometric data, which is application-dependent; this transformation (possibly the identity function) must be specified as *solve. Combine* is a housekeeping task which forms the tree structure from a list of subtrees, so it can also be embedded in the body of *GDTcreate.* One further parameter often supplied is the maximum depth of the tree, which limits memory requirements and prevents pathological cases of non-termination. We allow for this by defining two overloaded versions, with and without this final parameter. Their signatures are:

$$(\alpha \rightarrow \mathit{Bool}) \times (\alpha \rightarrow [\alpha]) \times (\alpha \rightarrow \tau) \rightarrow \mathit{GDT}(\tau)$$
$$(\alpha \rightarrow \mathit{Bool}) \times (\alpha \rightarrow [\alpha]) \times (\alpha \rightarrow \tau) \times \mathit{Int} \rightarrow \mathit{GDT}(\tau)$$

The definition of the second version, counting the root as being at depth 0, is

```
--- GDTcreate(x, leaf, divide, solve, maxdepth)
   <= MakeGDT( x, leaf, divide, solve, maxdepth, 0);

--- MakeGDT( x, leaf, divide, solve, maxdepth, depth )
   <= if leaf(x) or depth = maxdepth
      then Terminal(solve(x))
      else Interior
          (mapGDT(leaf, divide, solve, divide(x),
                  maxdepth, depth+1)
          );
```

where *mapGDT* is a variant of *map*, applying *MakeGDT* to each element of *divide(x).*

Once a tree is created we must traverse it to evaluate useful information. Again a D&C method is possible, evaluating the geometric structure at each terminal node and combining results. Here the critical parameters, as for *CSGTraverse*, are the *solve* and *combine* functions. It is also possible that there may be some 'external data', corresponding to the point in the earlier PMC example, such as a point or line about which to find a moment of inertia. Thus we again have an optional parameter, implemented using two overloaded functions, with signatures

$$\mathit{GDT}(\tau) \times (\tau \rightarrow \beta) \times ([\beta] \rightarrow \beta) \rightarrow \beta$$
$$\mathit{GDT}(\tau) \times \alpha \times (\tau \times \alpha \rightarrow \beta) \times ([\beta] \rightarrow \beta) \rightarrow \beta$$

The definition of the second version is

```
--- GDTtraverse (Terminal(x), query, solve, combine)
    <= solve (x, query);
--- GDTtraverse (Interior(x), query, solve, combine)
    <= combine (mapTraverse(solve, combine, x, query));
```

where *mapTraverse* is a *map*-like function similar to *mapGDT.*

3.4 Example 2: Area of a 2-Dimensional CSG 'Solid'

We illustrate GDT operations by finding the area of a 'solid' defined in two dimensions with just two kinds of primitive halfspaces, namely lines and discs. We follow the previous pattern in modelling our *SOLID* type. In addition, we must also model the (square) cells of the spatially divided model; each leaf of the GDT will contain both its bounding cell and the localised CSG tree.

```
type CI_HS ==  real # POINT2;         ! radius and centre
type LI_HS ==  real # real # real;    ! halfspace is ax + by + c < 0

data HS2   ==  circle(CI_HS) ++ line(LI_HS);
data ROP   ==  Union ++ Inter ++ Diff;

type SOLID ==  CSG(HS2, ROP);         ! basic CSG tree

type INTERVAL ==  real # real;
type CELL     ==  INTERVAL # INTERVAL;

dec UNIVERSE : CELL
--- UNIVERSE <=   ((0.0, 1.0), (0.0, 1.0));
! modelling space is normalised to a unit square

type SDM      ==  GDT (CELL # SOLID);  ! spatially divided model
```

The algorithm to find the area creates a tree, stopping the subdivision when the localised tree in a cell is either $\emptyset$ or Ω. In addition we will specify a maximum termination depth *MAXDEPTH*. Finding the area sums the contributions from all terminal nodes, counting $\emptyset$ cells as zero and evaluating the complete area of Ω cells. A GEL implementation is:

```
GDTtraverse(GDTcreate((tree, UNIVERSE), leaf, divide, solve, MAXDEPTH),
            quad_area,
            add_areas);
```

It remains to define the various support functions. *Leaf* terminates the subdivision if the CSG tree is full or empty, using primitive functions supplied by GEL:

```
dec leaf : CELL #  SOLID -> truval;
--- leaf(_, t) <= is_empty(t) or is_full(t);
```

Divide requires two stages; first the cell is partitioned into quadrants using *split_cell*, then the CSG tree is 'pruned' to localise it to each quadrant, by invoking *CSGtraverse*:

```
dec make_subtrees: list(CELL) # SOLID -> list (CELL # SOLID);
--- make_subtrees (nil, _)
  <= nil;
--- make_subtrees (cell::rest, tree)
  <= (cell, CSGtraverse (cell, tree, prune_hs, merge_subtrees, Fullsolid))
      :: make_subtrees (rest, tree);

dec divide: CELL # SOLID -> list(CELL # SOLID);
--- divide (cll, tree) <= make_subtrees (split_cell(cll), tree);
```

The support functions of *CSGtraverse* (not shown) again use pattern matching on the *HS2* and *ROP* constructors. Of interest here is that *CSGtraverse* produces a modified tree, discarding primitives found to be full or empty in relation to an enclosing cell. However, it is possible, for Ω values to be retained in subtrees of form $\Omega - P$, as noted before, hence the need to specify a value to be returned when Ω is encountered.

Solve also has interesting features, since a terminal cell at MAXDEPTH need not contain Ω or $\emptyset$. To allow for this, we classify the mid-point of the cell against the local tree; if this returns OUT we count the cell as empty, otherwise as full. The PMC operation (using *CSGtraverse* again) returns IN or OUT trivially if the local tree is Ω or $\emptyset$. Note that this particular CSG traversal is unusual in that it requires both of the optional results for Ω and $\emptyset$. The following implementation makes use of the GEL function *ptmake* to construct a point.

```
dec solve: CELL # SOLID -> CELL # SOLID;
--- solve (cell & ((xmin, xmax), (ymin, ymax)), tree)
 <= let midpt == ptmake((xmin + xmax) * 0.5, (ymin + ymax) * 0.5) in
      (cell, if  CSGtraverse(midpt, tree, classify_prim,
                               combine_classifications, IN, OUT) = OUT
           then Emptysolid
           else Fullsolid);
```

Only *quad_area* and *add_areas* remain undefined; these are trivial and are omitted.

Creating a GDT is a substantial overhead, which is not justified when only a single traversal is required. A further inefficiency arises because there is the overhead of computing the midpoint and executing *CSGtraverse*, even when the cell was previously found empty or full by *leaf*. This is an instance of a general difficulty which we call the *leaf/solve* problem: *leaf* and *solve* may repeat some computation since there is no way to pass the results of subexpressions between them. Both these problems can be resolved by a more efficient algorithm which computes the desired results 'on-the-fly' instead of creating the tree. Since no GDT is created, this solution cannot be implemented by GDT operations, but it is still a D&C operation and can be solved using *divacon*:

```
divacon((UNIVERSE, tree, 0, MAXD),
        leaf, divide, quad_area, add_areas);
```

The support functions are similar to the previous example but there is some re-distribution of the computation between them. *Divide* again uses *CSGtraverse* to prune CSG trees to each cell. The termination condition and the PMC operation are both included in *quad_area*, removing the *leaf/solve* inefficiency noted earlier.

Both approaches involve the same geometric computations and produce the same results. The second method executes more quickly, but the first may be appropriate if the GDT can be reused for subsequent computations.

3.5 Input and output

GEL provides facilities enabling I/O of geometric objects to occur in a uniform way, with a single function call. Since Hope+ models I/O streams as lazily evaluated lists, input routines extract an item from the head of a list, returning a pair with the item and the modified list. Similarly, output routines append an item to a list, returning the modified list.

Simple routines for basic geometric types such as points and vectors are provided. CSG trees and GDTs are more interesting; higher-order functions allow these types also to be input or output by a single function call. This requires a standard file format, transparent to the user of GEL except for the one-character codes of the operators; for compatibility with other work at Leeds we use the same format as the Mistral-3 parallel solid modeller (Holliman *et al.*, 1993).

For *CSGget*, two support functions respectively extract a primitive from the head of the list and convert an operator code to its internal form; for *CSGput* they convert primitives and operators to their character form. The invocations of *CSGget* and *CSGput* are

```
CSGget(instream, get_prim, get_op);
CSGput(tree, outstream, put_prim, put_op);
```

and their respective signatures are

$$[Char] \times ([Char] \rightarrow \rho \times [Char]) \times (Char \rightarrow \chi) \rightarrow CSG(\rho,\chi) \times [Char]$$
$$CSG(\rho,\chi) \times [Char] \times (\rho \rightarrow [Char]) \times (\chi \rightarrow Char) \rightarrow [Char]$$

The support functions should be written when the geometric domain is defined. As in earlier examples, they use pattern matching in a stylised fashion, calling GEL routines to convert the individual components of primitives; indeed it would be a routine task to generate the support functions automatically from the primitive type definitions.

The same approach is used for reading and writing GDTs, using an extension of the same file format. Since there is only a single type parameter, τ, only one support function is required for each operation. Outputting the tree requires the degree of the tree as a parameter. (The restriction to fixed degree trees is unimportant in practice). This parameter is not needed for input, since it is stored in the external file. The invocations of *GDTget* and *GDTput* are

```
GDTget(instream, get_geom);
GDTput(tree, degree, outstream, put_geom);
```

with signatures

$$[Char] \times ([Char] \rightarrow \tau \times [Char]) \rightarrow GDT(\tau) \times [Char]$$
$$GDT(\tau) \times Int \times [Char] \times (\tau \rightarrow [Char]) \rightarrow [Char]$$

As an example of the power and flexibility of these operators it is possible to input or output the spatially divided models of section 3.4 by a single call to *GDTget* or *GDTPut*. In these cases *get_geom* and *put_geom* will invoke *CSGget* and *CSGput* respectively.

3.6 Other GEL features

Two additional data types enlarge the scope of GEL. The first extends the range of CSG problems which can be handled, the second opens up a range of geometric problems outside the scope of solid modelling.

3.6.1 *Attributed CSG*

The CSG data type is parameterised by primitive and operator types, but in some situations other information may also be stored in the tree. If this information is only associated with primitives it can be included in the definition of the primitive type. In some cases, however, information is also associated with interior nodes. A simple example is the use of a hierarchy of bounding boxes, which speeds subsequent processing by allowing intersection tests to take place against the simple bounding boxes rather than the more complex primitives. Intersection of an object with a primitive need only occur when intersection with the box is indecisive. The benefits of such boxing are discussed in Cameron (1989). A rather different use of attributes is proposed in Alagar *et al.* (1990), where semantic information is stored at the nodes of CSG trees to guide finite element analysis.

These comments motivate the definition of an *attributed CSG tree*, in which a further parameter, *theta*, represents the type of such attributes:

```
data AttCSG(rho,chi,theta) ==
   Emptysolid  ++  Fullsolid  ++
   Primitive (rho # theta)  ++         ! attribute at leaf node
   Compose (AttCSG(rho, chi, theta)
            # (chi # theta) #          ! attribute at interior node
            AttCSG(rho, chi, theta));
```

Attributed CSG trees have a similar set of overloaded operators to simple CSG trees, including I/O functions. Their use is illustrated in the first author's PhD thesis (Davy, 1992).

3.7 Multi-resolution representations

A variant of the geometric decomposition paradigm outside solid modelling is the use of multi-level hierarchical approximations for digitised objects. For instance a *strip-tree* (Ballard, 1981) stores a digitised 2-D curve as a hierarchy of bounding strips. *Nested ternary triangulations* (Floriani *et al.*, 1984) and *nested quaternary triangulations* (Gomez and Guzman, 1979) are hierarchical representations for digitised surfaces. These structures are derived by a recursive technique similar to spatial subdivision. The most significant difference to the earlier GDTs is that similar geometric information is stored at varying levels of accuracy throughout the tree structure, with greater accuracy at greater depth. Thus an accuracy-time tradeoff is possible in subsequent processing of the tree.

We capture such structures using a *multi-resolution GDT* type:

```
data MGDT(tau) ==
      Terminal (tau) ++
      Interior (list (MGDT(tau)) # tau);  ! geometry at interior node
```

The MGDT type has a set of overloaded operations analogous to GDT, including input and output. One slight difference is in *MGDTtraverse.* Here we may wish to compute some property of the stored geometry with less accuracy than was used to create the tree, thus limiting the depth of the traversal. Hence there is an optional *leaf* parameter, which typically checks whether the desired accuracy has been reached.

Examples of striptree operations using the MGDT type are shown in Davy (1992).

4 Implementation

GEL is implemented as a set of separately compiled Hope+ modules, amounting to some 2095 lines (71 kbytes) of source code. Parts of GEL have also been translated to a range of other functional languages (including Haskell and Lazy ML) as part of a benchmarking exercise (Hartel and Langendoen, 1993; Hartel, 1994), using the 'area' derivation of section 3.4 (with a GDT) as the benchmark program.

The GEL modules fall into three categories:

- *Structural modules* provide the higher-order D&C functions on which GEL is based, including all support for the CSG and geometric decomposition types.
- *Geometric libraries* provide operations on basic entities such as points and vectors, from which more complex geometric domains can be constructed.
- A *utility* module provides a variety of utility functions relating to I/O and list processing.

To make effective use of these core modules *domain* modules must be constructed to define the geometric domains for specific applications. Four such modules are already included in GEL, for 2- and 3-dimensional halfspaces, bounded primitives, and striptrees. However, it is envisaged that users of GEL will commonly define their own domain modules.

5 Performance studies

In the context of a prototyping exercise, functionality was a higher priority than performance. However, a number of performance experiments were carried out to:

- measure the absolute performance of GEL on selected programs;
- compare its performance with C; and
- assess the overheads of using higher-order functions.

5.1 Execution times for PMC

Figure 2 shows the execution times (in seconds) for a program which inputs a CSG tree and carries out a single PMC operation. Times were measured on a lightly loaded Sun 3/60 workstation using the sum of system and user times returned by the Unix 'time' command; the best time for five successive runs was used.

For a tree with N_p halfspaces the execution time for both inputting the tree and carrying out PMC should be $O(N_p)$. Times for two versions of the program are shown in figure 2:

1. Using the GEL *CSGget* and *CSGtraverse* operations.

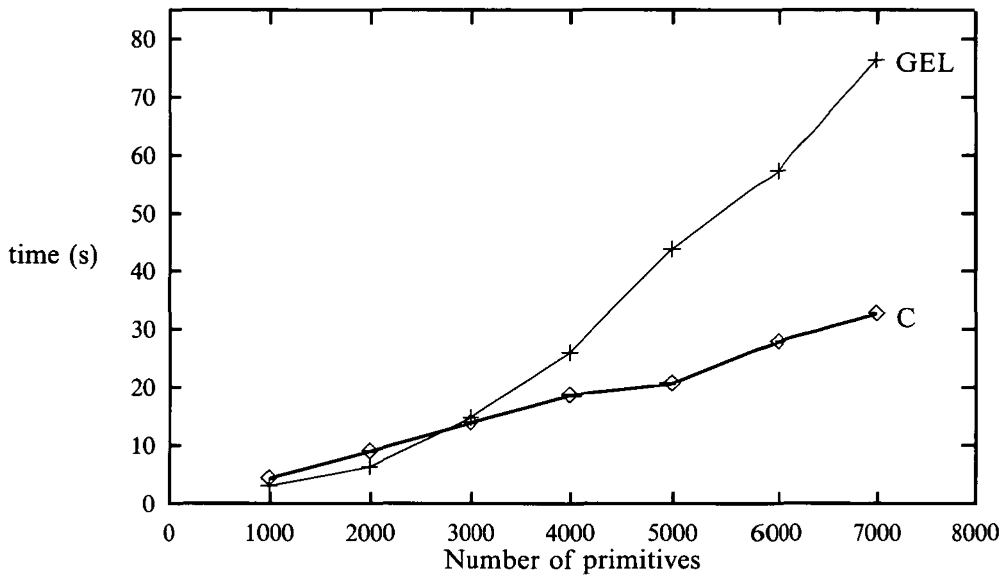

Fig. 2. Execution times for PMC

2. A comparable recursive algorithm in C, in which each tree node constructed required a call to *malloc()*, and no cells were deallocated.

In all cases the same models were used, including very large trees with up to 7000 halfspaces; the largest external file was around 258 Kbytes. Internally each halfspace requires 16 bytes of memory in addition to the space required for enumerated types at the interior nodes and links between nodes. In both cases the execution time is dominated by inputting and building the CSG tree.

It can be observed that GEL outperforms C for up to around 2500 halfspaces (a very substantial solid model). This is an encouraging confirmation of claims that modern functional language compilers can generate code comparable with established imperative languages. On the other hand, GEL is significantly slower than C beyond this point. Whereas the C times follow the expected linear behaviour, the Hope+ times are worse than linear. This is easily confirmed by curve-fitting; indeed a denser set of timings at intervals of 500 halfspaces up to 4000 halfspaces, showed a strong quadratic fit, indicating that the non-linear behaviour was not simply a feature of the largest data sets.

There could be several possible causes of this performance degradation, including page faults, memory leaks, overheads of lazy evaluation, and garbage collection. Page faults could be discounted since 'time' usually reported none. After discussion with one of the Hope+ implementors we were unable to discover any memory leaks. Although Hope+ normally constructs data lazily, strictness can be enforced using a compiler option. When this was done, execution times were almost identical, sometimes slightly greater, more often slightly smaller; subsequent experiments

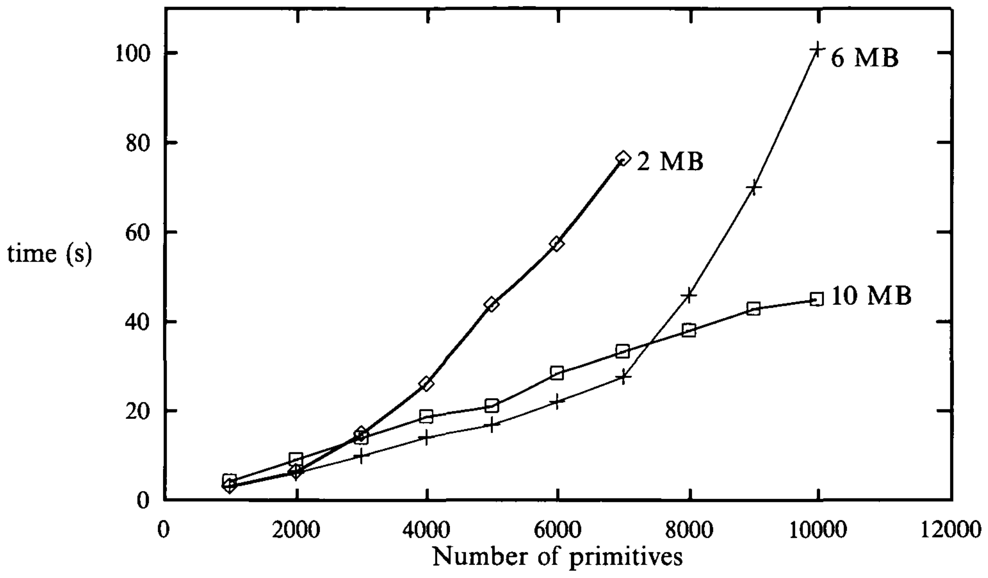

Fig. 3. Effects of varying the initial heap size

therefore used strict compilation. This left garbage collection as the primary suspect, particularly as the default initial heap size for Hope+ is only 2 Mbytes. Increasing the initial heap size might be expected to reduce garbage collection, possibly at the cost of some page faults.

Figure 3 shows the effect on execution time of using initial heaps of sizes 6 Mbytes and 10 Mbytes as well as the default 2 Mbytes. It confirms that garbage collection was indeed the primary cause of the non-linear behaviour; for the 10 Mbytes initial heap, the graph is almost linear, though the data fit slightly better to $a + bN_p \log N_p$ than to $a + bN_p$. It is interesting to note that the larger initial heap led to higher execution times on the smaller data sets, so optimal setting of the initial heap size is data-dependent. Despite the increased heap size, normally no page faults were reported by 'time'. When the program was compiled with a 16 Mbytes heap, however, a run time 'memory exhausted' error was obtained.

As shown in figure 3, the much improved times made it feasible to increase the tree sizes, up to 10000 primitives. The C program was also run on the larger data sets. Results were very similar to the GEL results with a 10 Mbytes initial heap; C was marginally faster than GEL from 2000 primitives upwards.

5.2 Effects of input/output

The PMC program contains three main components; input of the data from an external file, building the CSG tree, and traversing the tree to classify the point. The first two are integrated within the *CSGget* function and cannot be separated, but their combined effect can be assessed by removing the *CSGtraverse* function which

Table 1. *I/O and computation times for PMC, with varying initial heap size*

	2 Mbytes		6 Mbytes		10 Mbytes		C	
N_p	T_I[a]	T_C[b]	T_I	T_C	T_I	T_C	T_I	T_C
1000	2.7	0.3	2.7	0.3	2.8	1.4	4.0	0.3
2000	5.6	0.8	5.7	0.4	5.7	3.4	8.6	0.4
3000	14.2	0.7	9.2	0.7	9.1	4.8	13.2	0.7
4000	25.1	0.9	13.2	0.9	13.1	5.6	17.9	0.8
5000	42.6	1.1	15.6	1.2	15.6	5.4	19.9	0.8
6000	55.1	2.2	20.6	1.4	18.3	10.1	27.0	0.9
7000	73.6	2.8	25.9	1.7	23.4	9.8	31.6	1.1
8000	-	-	44.2	1.8	26.1	11.9	36.2	1.2
9000	-	-	68.0	2.1	28.7	14.2	40.8	1.3
10000	-	-	98.4	2.5	35.1	9.9	42.8	1.7

[a] T_I is the time (in seconds) to build the CSG tree including input from the external file
[b] T_C is the time (in seconds) to carry out point membership classification

implements the PMC. Table 1 shows the execution times which can be attributed to input/tree-building and PMC components.

With the largest initial heap size GEL consistently outperforms C for input and tree-building. This because input is faster in Hope+ than C; tree-building is also likely to be faster because heap allocation is done with inline code using reserved registers to store important heap information (Perry, 1995). For all but the smallest data sets the input and tree-building time substantially decreases with increasing initial heap size. On the other hand the PMC computation is consistently slower in Hope+ than C, and seriously degrades in a non-monotonic way with increasing initial heap size. Clearly different parts of the same program behave differently with respect to varying initial heap sizes and data sizes; this has adverse implications for performance optimisation.

5.3 *Overheads of using higher order functions*

GEL's simplicity for programming arises from its use of higher-order functions. It is reasonable to question whether this is achieved at the cost of performance. To investigate this, the PMC program was 'hand-coded' in Hope, using a similar recursive structure but no higher-order functions. Predictably the execution times were somewhat lower. The overheads of using GEL (expressed by the increase in execution time as a percentage of the raw Hope+ time) are shown in table 2.

For the lower heap sizes the overhead is pleasingly small (apart from the largest data sets with 6 Mbytes initial heap); it appears an acceptable price to pay for the generality and simplification in programming. On the other hand, for the 10 Mbytes initial heap, the overhead is consistently much more significant; its acceptability is more questionable.

Table 2. *Percentage overheads of using higher order functions, with varying initial heap size*

N_p	2 Mbytes	6 Mbytes	10 Mbytes
1000	3.4	3.4	44.8
2000	6.7	1.7	51.7
3000	11.2	5.3	44.8
4000	2.4	2.9	36.5
5000	4.0	-0.6	28.0
6000	3.4	3.8	32.1
7000	2.2	3.8	36.1
8000	-	27.4	39.2
9000	-	17.4	42.5
10000	-	24.6	23.3

5.4 Execution times for area by recursive subdivision

The computation of area by spatial subdivision, described in section 3.4, provides a more compute-intensive benchmark program. Recall that two versions of the program were developed, the first creating then traversing a GDT, the second computing the area 'on-the-fly' without building a GDT. Table 3 shows results obtained for the 'on-the-fly' version, in both GEL and C, with maximum depth of subdivision up to 13. Identical floating point computations were carried out in each case. All times were obtained using 'time' with multiple runs on a Sun 3/60, as with PMC.

Table 3. *Execution times for area by spatial subdivision 'on the fly'*

	Area computed		Execution times			
depth	C	GEL	C	GEL_2[a]	GEL_6	GEL_{10}
2	1.000000	1.0000000	4.2	1.8	1.8	1.7
3	0.968750	0.9687500	5.0	2.2	2.2	2.2
4	0.960938	0.9609375	5.5	2.6	2.6	2.6
5	0.957031	0.9570312	6.1	2.9	2.9	2.9
6	0.956787	0.9567871	6.7	3.3	3.3	3.3
7	0.955261	0.9552612	7.4	3.7	3.7	3.8
8	0.955551	0.9555664	8.2	4.5	4.4	4.4
9	0.955616	0.9556236	9.6	5.4	5.5	5.4
10	0.955612	0.9556236	12.2	7.3	7.4	7.4
11	0.955612	0.9556201	17.4	11.2	11.2	11.1
12	0.955614	0.9556208	27.5	19.0	18.9	19.0
13	0.955614	0.9556212	46.9	34.4	34.4	34.3

[a] GEL_N is the execution time with initial heap N Mbytes

Here the worst case execution time for depth d is $O(N_p 4^d)$, but this is of little practical use since typical performance is much better than this and highly data-dependent. Hence no asymptotic complexity result can be predicted. The measured times are much better than the worst case, and the computed areas show a clear convergence by around depth 9; it seems unlikely that the costs of processing to a greater depth are justified. The original results for mass properties in (Lee and Requicha, 1982) were reported only to depth 6. GEL consistently outperforms C; since there is extensive tree-building when CSG trees are pruned during spatial subdivision, this is not surprising. Of more interest is that, unlike PMC, the GEL times do not appear to be sensitive to initial heap size. The slight discrepancy between the C and GEL results is because the version of Hope+ we used appears to have a minor error in real input conversion. We understand that this is now corrected.

Results for the GDT version (GEL only) are shown in table 4. For each heap size the time for the full computation is followed by the separate times for creating the GDT and traversing it to find the area.

Table 4. *Execution times for area by spatial subdivision with a GDT*

	2 Mbytes heap			6 Mbytes heap			10 Mbytes heap		
depth	Time	Tree	Area	Time	Tree	Area	Time	Tree	Area
2	1.7	1.7	0.0	1.8	1.8	0.0	1.8	1.8	0.0
3	2.2	2.2	0.0	2.2	2.2	0.0	2.2	2.2	0.0
4	2.6	2.6	0.0	2.6	2.6	0.0	2.6	2.6	0.0
5	3.0	3.0	0.0	3.0	3.0	0.0	3.0	2.9	0.1
6	3.3	3.3	0.0	3.3	3.3	0.0	3.4	3.3	0.1
7	3.9	3.8	0.1	3.9	3.8	0.1	3.9	3.8	0.1
8	4.8	4.6	0.2	4.7	4.6	0.1	4.8	4.7	0.1
9	6.4	5.9	0.5	6.3	5.9	0.4	6.3	5.8	0.5
10	9.2	8.6	0.6	9.3	8.5	0.8	9.3	8.6	0.7
11	16.4	15.1	1.3	15.7	14.4	1.3	15.7	14.4	1.3
12	49.4	46.2	3.2	29.8	26.4	3.4	29.8	26.4	3.4
13	166.7	151.0	15.7	73.8	65.6	8.2	57.7	52.1	5.6

Predictably, all times are worse than for the corresponding 'on-the-fly' version but here they improve with increasing initial heap size, as for PMC. There is, however, an interesting contrast with PMC: the cost of the final tree traversal is greatest for the smallest heap size. Despite the additional cost of creating the GDT, a comparison with Table 3 shows that this is justified if only one further traversal is carried out, up to depth 11 (further for the larger heap sizes). This confirms the value of the distinct GDT structure within GEL.

5.5 Conclusions from performance experiments

Clearly, results from a small number of specialised programs should be treated with some caution. We did not set out to compare GEL and C performance comprehensively, and the test algorithms might be considered favourable to GEL, accounting for the better absolute performance. By contrast, C was much faster than GEL for a simple mergesort using *divacon*, because of the extensive use of list processing.

Nevertheless, the performance experiments cover the main features of GEL: input and traversal of CSG trees, creation and traversal of GDTs, and a general divide-and-conquer operation. In all cases the best absolute performance of GEL was comparable to or better than C, a very positive result given the folklore concerning the inefficiency of functional language implementations. This can largely be attributed to the relative efficiency of heap management in Hope+, aiding fast tree-building.

By contrast, the experiments show an unpredictable aspect of Hope+ performance, making optimisation problematic. Execution times may depend in a problem-dependent way on initial heap size, and the nature of this dependence may vary with data sizes. Also, the cost of different parts of the same program may change in different ways as the initial heap size changes. Perhaps more seriously, the match between asymptotic complexity analysis and the observed behaviour on realistic data sets may also be affected by the initial heap size.

We also found in practice that small changes in a program could have unexpected consequences for execution times. This coincides with the observations of S Peyton Jones (1987) that

> seemingly innocuous (and meaning preserving) changes to a functional program may have dramatic effects on its runtime behaviour.

Peyton Jones goes on to suggest that

> we have no good means of reasoning about runtime behaviour so as to understand how good or bad our programs are. In order to reassure himself that his program does not have undesirable runtime behaviour the programmer may have to know a lot about the particular implementation.

Certainly this matches our experience; one of the Hope+ implementors made suggestions which improved the performance of a crucial input function, and which were non-intuitive to us. During this dialogue, however, one suggestion unexpectedly degraded the performance, provoking an inspection of some implementation details. It appears that even implementors are not infallible!

Peyton Jones' remarks were made several years ago, but our experience suggests that they are still valid. It appears that performance optimisation is both application- and data-dependent in ways which are difficult to predict, and may require non-trivial implementation knowledge; thus the provision of a general optimised GEL library in Hope+ is problematic. While we have no direct evidence to indicate that this would be true for other functional languages, it suggests that a more conventional language would be appropriate for moving beyond the prototyping stage.

6 Applicability of GEL

The facilities supplied by GEL are of little benefit unless they are of reasonably general use. We now address this issue, specifically considering the range of useful operations captured and the ease of extending geometric domains.

6.1 Range of useful applications

The CSG principle is applicable, in principle, in an arbitrary number of dimensions. CSG-based systems vary in the formal properties of primitives, which have included *r-sets* (Requicha, 1980), *s-sets* (Arbab, 1984), and primitives bounded by B-spline patches (Saia *et al.*, 1987). Specific sets of available primitives may differ, even between two systems with the same theoretical basis. Furthermore different sets of boolean operators have been used, varying from difference only (Perng *et al.*, 1990) to union, difference, intersection and complement (Arbab, 1984). All these variations can be modelled using our CSG type. The attributed CSG type models both bounding boxes (Cameron, 1989) and semantic attributes (Alagar *et al.*, 1990).

CSG traversals directly model the family of Set Membership Classification algorithms identified by Tilove (1980). They provide a general utility operation which we have seen in each CSG example.

GDT operations directly model spatial subdivision operations, such as the family of integral property algorithms identified by Lee and Requicha (1982). Spatial subdivision is a widely used technique which has also been used on other solid representations than CSG, including boundary representations (Kela, 1989), polyhedral models (Carlblom, 1987) and sweeps (Brunet and Navazo, 1990). Typically, trees are generated in which terminal nodes contain an exact or approximate representation of the model localised to the relevant subspace. Variants of this type have been called *octrees* (Meagher, 1982) (with the equivalent *quadtrees* (Samet, 1984) in two dimensions), *polytrees* (Carlblom, 1987) and *extended octrees* (Brunet and Navazo, 1990). A similar *bintree* structure has been proposed (Samet and Tamminen, 1985) with a binary subdivision at each stage, cycling between the dimensions. All these structures can be modelled by the GDT type.

In addition to applications previously noted, spatial subdivision techniques have been used for wire-frame edge evaluation (Woodwark, 1986), NC program verification (Wallis and Woodwark, 1984), collision detection (Samet and Tamminen, 1985), finite element mesh generation (Shepard *et al.*, 1988) and boolean operations in polyhedral modellers (Carlblom, 1987). In view of this broad applicability, spatial subdivision can be seen as a fundamental approach to solid modelling applications.

Many other geometric problems can be solved by D&C methods, especially in discrete computational geometry. Examples include the construction of Voronoi diagrams and Delaunay triangulation (Shamos, 1977), and the determination of convex hulls (Preparata and Hong, 1977). While these cannot be expressed using the specialised CSG and GDT operations, they can, in principle, be solved using the general *divacon* function.

It thus appears that the generic nature of D&C, and of geometric decomposition trees in particular, makes GEL applicable far beyond its initial aims.

On the other hand, the limitation to D&C means that not all possible algorithms on the types provided are currently included in GEL; for instance, ray-casting involves following a single path through an octree, which is not a D&C operation. Such algorithms can of course be programmed directly using the constructors of the types concerned. Similarly, operations between two tree structures (for instance, boolean operations between striptrees (Ballard, 1981), or addition and subtraction of octrees (Wyvill and Kunii, 1985)) cannot currently be directly described in GEL. Further work is needed to determine whether these and other computations could usefully be captured as higher-order functions. (Note that the initial focus on D&C arose from the wish to exploit known methods of parallelism.)

6.2 Geometric extensibility

One of the practical difficulties of solid modelling systems is extending the domain of solids which can be represented. This issue is discussed in depth by Dunnington (1989), where a generic recursive subdivision approach is proposed to solve the various aspects of this problem. GEL captures this method, and effectively isolates the domain-specific parts of geometric computations in type definitions and well-defined support routines. Since these aspects of computational geometry are well-known for difficulties with floating point accuracy, this isolation helps to localise such problems. In principle, it should be possible to import code from other geometric libraries, taking advantage of the extensive efforts on this area.

The main algorithmic structures of GEL, however, do not depend on the specific geometric domain; in this sense GEL is 'geometry-independent'. Indeed, the core of GEL contains no geometric routines, other than support for basic entities such as points and vectors which are likely to be of use to all applications.

The first task for a user of GEL is therefore to customise it for the application's requirements by defining a suitable geometric domain. Examples of two such domains were modelled earlier. In each case primitive I/O functions must be specified, and support functions provided for the needed geometric facilities, such as classifying primitives. Adding an additional primitive to an existing domain is also straightforward; an extra constructor is needed in the relevant type definition and an extra equation to match that constructor in each support function. Thus adding new geometry can be done in a simple, systematic fashion. Effectively, GEL provides syntactic support for extensibility. It is intended as a framework for developing geometric applications which can be tailored to the user's requirements.

7 Evaluation

The GEL library was a prototype intended to explore both the potential for programming with higher-order functions and the benefits of using functional languages in this context. This section discusses these issues in relation to both the functionality of the current library and our experiences in developing and using it. Performance issues have already been considered.

7.1 Design and functionality

GEL was developed in incremental, exploratory fashion without a formal design phase, for which Hope+ proved an excellent tool in several respects. Polymorphic types matched the requirement to provide different geometric domains within the same generic structure. Higher-order functions successfully captured the computational structures initially identified and isolated low-level geometric details in support functions. The clean separation of geometric processing and boolean operation implementation functions was particularly pleasing from an application perspective. These comments are equally applicable to other higher-order languages such as ML, Miranda and Haskell, and confirm the well-advertised claims made for the expressiveness of functional languages.

An apparent drawback of functional programming has also been perceived. Many of the above benefits can be attributed to the hierarchical nature of the key data structures, which is not true of the other major solid representation scheme, boundary representation. This uses a complex graph structure, showing the topological relationships between the faces, edges and vertices which constitute the solid's boundary. It is much harder than CSG to model in Hope+; a possible way forward is shown in Burton and Yang (1990), where similar structures may be built on an array type. In general, however, it seems that the modelling of multilinked structures is generally more difficult in functional than in imperative languages. For instance, we were unable to implement a *father-of* function to move back up a tree, which would preclude some of the efficient octree and quadtree traversal algorithms in the literature (Samet, 1990).

7.2 Implementation

The implementation of GEL was found to be mostly straightforward and quick, though compilation was slow; subjectively, we found pattern matching a helpful and systematic way to develop code. Overall, we consider the claim that functional languages enable rapid code development to be substantiated, with the *caveat* that we were dealing with data structures to which functional languages are particularly well-suited.

A flaw with the current implementation is its heavy reliance on overloading. The reason for this was the desire to emphasise the similarity of related operations without resorting to large numbers of almost identical function names. Clearly this solution is language-dependent; with hindsight it is not all necessary. Consider the signatures of overloaded operators in figure 1. The heavy overloading was occasioned by the need to specify a result for the *solve* function when a special case of $\emptyset$ or Ω is encountered at the leaves. This could be better handled by declaring a new type of the form

```
data SpecialSolve(alpha)
     == NoSolve ++ Full(alpha) ++ FullEmpty(alpha, alpha);
```

A value of this type could then be supplied as a final parameter to all *CSGtraverse* functions, removing most of the need for overloading. A possible drawback is that the parameter must always be included even if no special case is involved; on the other hand, drawing attention to the possibility of special cases is perhaps desirable.

The use of *currying* would further reduce the number of overloaded functions. Consider again the signatures of the two *GDTtraverse* functions:

$$GDT(\tau) \times (\tau \rightarrow \beta) \times ([\beta] \rightarrow \beta) \rightarrow \beta$$
$$GDT(\tau) \times \alpha \times (\tau \times \alpha \rightarrow \beta) \times ([\beta] \rightarrow \beta) \rightarrow \beta$$

Rewriting these in curried form, with some re-ordering of arguments, allows the surplus argument to the second operation to be omitted:

$$GDT(\tau) \rightarrow (\tau \rightarrow \beta) \rightarrow ([\beta] \rightarrow \beta) \rightarrow \beta$$
$$GDT(\tau) \rightarrow (\alpha \rightarrow \tau \rightarrow \beta) \rightarrow ([\beta] \rightarrow \beta) \rightarrow \beta$$

The second operation can now be carried out in terms of the first, simpler function. For instance, the 'area' example of section 3.4 can be modified to compute a moment of inertia about a point using

```
GDTtraverse sdm (quad_moment pt) add_moments;
```

where *sdm* is the spatially divided model, and *quad_moment* computes the moment of inertia of a quadrant about a point. This removes a further needless overload and is clearly a better solution for a language which supports currying; indeed it could be implemented using the limited form of currying available in Hope+. The remaining overload of *CSGtraverse* can also be removed in this way.

The overloadings of *GDTcreate* and *MGDTcreate* cannot be avoided by currying, because the extra parameter is actually used in the body of the function, rather than simply being passed to support functions. Thus some overloading appears inevitable, unless different names are adopted for the variant functions, which would make GEL more language-independent.

7.3 Use of GEL

Programming using a small number of higher-order functions proved to have both benefits and drawbacks. There is a helpful discipline to facilitate program development, code is concise, and the low-level, error-prone geometric computations are isolated in a few support functions. On the other hand, the limited set of operations may lead to more imaginative and appropriate solutions being missed, or incur inefficiencies such as the *leaf/solve* problem noted in section 3.4. Moreover, 'short-cut' solutions, such as 'early-outs' (Mudur and Koparkar, 1984), may not fit into the higher order function framework; Tilove's work on generic CSG algorithms also pointed to this conflict between generality and efficiency (Tilove, 1980). Adding extra CSG and GDT operations for special cases may partly resolve this, but would conflict with the simplicity of using a small set of generic operations, and does not guarantee that no more special cases will occur. Of course it should still be possible to code such special cases directly in Hope+ or any other base language used.

GEL has been used by several undergraduate students to aid writing CSG and striptree programs in Hope+ as part of their final year projects. The largest code was a CSG ray-tracer, translated from C. This was interesting because GEL was not able to model casting a ray through an octree, as noted earlier, and this operation had to be hand-coded in Hope+. All the students concerned were novices in both computational geometry and Hope+. Their verdict on GEL was uniformly positive in terms of the programming process itself, criticisms being related mainly to other features such as long compile times.

8 Related work

Programming with a small number of higher-order functions was promoted by Meertens (1986) and Bird (1987). Theories are developed for various classes of data structures, allowing a transformational approach to program development. In GEL, we observe that most CSG and GDT operations are specialisations of one of the basic *divacon* operations, in which support functions have been embedded in the body of higher-order function. For instance, we have derived the *GDTcreate* function of section 3.3 from *divacon*, using standard 'fold-unfold' techniques (Burstall and Darlington, 1977), though we have made no further use of this insight.

The notion of using higher-order functions as 'skeletons' for parallel programs has been developed by Cole (1989) and Darlington and To (1995). The latter, independently of us, show application-specific skeletons for solid modelling, similar to our CSG and GDT operations. The data types used are not polymorphic, hence much less general than in GEL, and they do not allow for Ω primitives in CSG, a necessary feature for realistic modelling. Their work differs from ours in stressing transformations between CSG and octree structures, with a view to carrying out all parallel operations on octrees. Thus a programmer specifies an operation on CSG but it is carried out on the corresponding octree. While this is an interesting example of transformation techniques, it has two practical weaknesses from the application perspective. Firstly, some standard algorithms (such as mass property derivations) are defined on the octree model rather than on CSG directly. Secondly, traversal of a CSG tree does not necessarily map directly to a corresponding D&C traversal of the octree – ray-tracing is a good example. By contrast, GEL encourages D&C operations to be defined in terms of the most suitable data structure.

Recent work on categorical data types (Skillicorn, 1995) shows a way in which new data types may be defined so that a set of (parallel) higher-order operations can automatically be generated from the constructors, specifically a generalised map and a generalised reduction. Such formal treatment, based on category theory, seems appropriate for our CSG and GDT types; traversal operations are reductions and it is straightforward to define a corresponding map.

Cameron (1989) notes that CSG trees are purely a syntactic structure – the same is true of all our specialised data types and also for categorical data types. He defines a denotational semantics for CSG trees, in which an interpretation function maps leaf nodes of a CSG tree into subsets of $\mathscr{R}^n$. Combining this semantic approach with categorical data types has the potential to give a complete formal foundation for GEL.

9 Conclusions and future work

We began by asking whether common patterns of computation in solid modelling could be captured in a systematic way. The answer to this is clearly yes. GEL implements a set of higher-order functions which successfully exploit widely applicable algorithmic patterns using CSG and spatial subdivision. They provide an effective library for the systematic development of CSG applications, and their polymorphism aids geometric extensibility.

There are, however, some CSG and spatial subdivision applications whose main algorithmic structures cannot currently be described by GEL, such as tracing a ray through an octree. This is not an insuperable obstacle, since they can still be implemented directly in the base language, but it indicates that GEL is not yet a comprehensive package. Further work may establish whether other general algorithmic patterns could be included in later versions of GEL. We conjecture that special cases will continue to arise which cannot be handled within a small set of higher-order operations. Thus there is likely to be some limitation to the use of this approach.

The use of a functional language had both positive and negative aspects. Some elegant and very useful features are offset by difficulties in handling multi-linked structures. Good absolute performance for typical CSG problems is marred by difficulties in performance tuning and prediction. There is no doubt that for this prototyping exercise Hope+ was invaluable. Data modelling was extremely straightforward; provision of an equivalent level of polymorphism would have been much more difficult in, say, C. For a 'production' version, however, our view is that in the current state-of-the-art an imperative or object-oriented base language would be more appropriate, even at the cost of more complex implementation. This verdict is reinforced by the inevitable pragmatic consideration of 'legacy' imperative code. Thus we have no short-term plans to proceed further with the use of a functional language in this area, though we await further developments in functional programming with interest.

Moving to an object-oriented base language would give the opportunity to exploit inheritance. For instance, an attributed CSG tree is essentially a CSG tree with an additional attribute instance at each node. It seems natural to express this using subclassing, but there is no facility to do so in Hope+. We have recently begun porting GEL to C++ in order to explore this possibility. The initial experience with Hope+ has been invaluable in identifying the issues of importance.

Our original motivation for this study was to simplify parallel programming by making use of known packaged solutions tailored to the application domain. This remains a focus of our research. We are continuing to investigate the use of application-specific parallel algorithmic skeletons, with particular emphasis on modelling their performance (Deldarie *et al.*, 1995).

Availability of GEL

Source code for GEL and the demonstration programs described in this paper is available by anonymous ftp from *agora.leeds.ac.uk* in file *scs/GEL/gel.tar.Z.*

Acknowledgements

Thanks are due to Professor Alan de Pennington and the Geometric Modelling Project at Leeds for their support, and to Professor John Darlington of Imperial College for his interest and supplying the Hope+ compiler used. Nigel Perry gave much useful advice on the use and performance of Hope+, and Hossain Deldarie helped with curve-fitting the experimental results. Several discussions with David Dunnington stimulated the development of the work. Don Goodeve gave helpful comments on an early version of the paper. Further improvements came as a result of insights from the anonymous referees.

The first author was partly supported by a studentship from the Science and Engineering Research Council.

References

Alagar, V. S., Bui, T. D. and Periasamy, K. (1990) Semantic CSG Trees for Finite Element Analysis. *Computer Aided Design* **22**(4): 194–198.

Arbab, F. (1984) RSC: A Calculus of Shapes. In: *Proc. CAD 84*, Brighton, UK, pp. 244–251.

Ballard, D. H. (1981) Strip Trees: A Hierarchical Representation for Curves. *Comm. ACM* **24**(5): 310–321.

Bird, R. S. (1987) An introduction to the theory of lists. In: M. Broy, ed., *Logic of Programming and Calculi of Design*, Springer-Verlag, pp. 5–42.

Brooks, R. A. (1981) Symbolic Reasoning among 3-D Models and 2-D Images. *Artif. Intell.* **17**: 285–348.

Brunet, P. and Navazo, I. (1990) Solid Representation and Operation using Extended Octrees. *ACM Trans. Graphics* **9**(2): 170–197.

Burstall, R. M. and Darlington, J. (1977) A Transformation System for Developing Recursive Programs. *J. ACM* **24**(1): 44–67.

Burton, F. W. and Yang, H.-K. (1990) Manipulating Multilinked Data Structures in a Pure Functional Language. *Software - Practice and Experience* **20**(11): 1167– 1185.

Cameron, S. (1989) Efficient Intersection Tests for Objects Defined Constructively. *Int. J. Robotics Res.* **8**(1): 3–25.

Carlblom, I. (1987) An Algorithm for Geometric Set Operations using Cellular Subdivision Techniques. *IEEE Computer Graphics and Applications* **7**(5): 44–55.

Cole, M. (1989) *Algorithmic Skeletons: Structured Management of Parallel Computation.* Pitman/MIT Press.

Darlington, J. and To, H. W. (1995) Building parallel applications without programming. In: J. R. Davy and P. M. Dew, eds, *Abstract Machine Models for Highly Parallel Computers.* Oxford University Press.

Davy, J. R. (1992) Using Divide-and-Conquer for Parallel Geometric Evaluation. PhD thesis, University of Leeds.

Deldarie, H., Davy, J. R. and Dew, P. M. (1995) The Performance of Parallel Algorithmic Skeletons. Research Report, School of Computer Studies, University of Leeds.

Dunnington, D. R. (1989) A Recursive Subdivision Strategy for Solid Modelling with Sculptured Surfaces. PhD thesis, Leeds University.

Floriani, L. D., Falcidieno, B., Nagy, G. and Pienovi, C. (1984) A Hierarchical Structure for Surface Approximation. *Computer and Graphics* **8**(2): 183–193.

Gomez, D. and Guzman, A. (1979) Digital Model for Three-dimensional Surface Representation. *Geo-processing* **1**: 53–70.

Hartel, P. H. and Langendoen, K. G. (1993) Benchmarking implementations of Lazy Functional Languages. In: *6th Functional Programming Languages and Computer Architecture*, pp. 341-349, Copenhagen, Denmark.

Hartel, P. H. (1994) Benchmarking implementations of Lazy Functional Languages II – Two Years Later. Technical Report CS-94-21, Department of Computer Systems, University of Amsterdam.

Holliman, N. S., Morris, D. T. and Dew, P. M. (1989) An Evaluation of the Processor Farm Model for Visualising Constructive Solid Geometry. In: P. M. Dew, R. A. Earnshaw and T. R. Heywood, eds, *Parallel Processing for Computer Vision and Display*, Addison Wesley, pp. 452–460.

Holliman, N. S., Wang, C. M. and Dew, P. M. (1993) Mistral-3: Parallel Solid Modelling. *The Visual Computer* **9**(7): 356–370.

Kela, A. (1989) Hierarchical Octree Approximations for Boundary Representation-based Geometric Models. *Computer Aided Design* **21**(6): 335–362.

Kelly, P. H. J. (1989) *Functional Programming for Loosely-coupled Multiprocessors*, Pitman/MIT Press.

Lee, Y. T. and Requicha, A. A. G. (1982) Algorithms for Computing the Volume and Other Integral Properties of Solids. II. A Family of Algorithms Based on Representation Conversion and Cellular Approximation. *Comm. of the ACM* **25**(9): 642–650.

Meagher, D. (1982) Geometric Modeling using Octree Encoding. *Computer Graphics and Image Processing* **19**: 129–147.

Meertens, L. G. T. (1986) Algorithmics – Towards Programming as a Mathematical Activity. In *Proc. CWI Symposium on Mathematics and Computer Science*, North Holland, pp. 289–234.

Mudur, S. P. and Koparkar, P. A. (1984) Interval Methods for Processing Geometric Objects. *IEEE Computer Graphics and Applications* **4**(2): 7–17.

Muuss, M. J. (1987) RT and REMRT: Shared Memory Parallel and Network Distributed Ray Tracing Programs. In *USENIX Association, 4th Computer Graphics Workshop*, pp. 86–97.

Perng, D.-B., Chen, Z. and Li, R.-K. (1990) Automatic 3D Machining Feature Extraction from 3D CSG Solid Input. *Computer Aided Design* **22**(5): 285–295.

Perry, N. (1989) *Hope+*. Technical documentation, Department of Computing, Imperial College London.

Perry, N. (1995) Private communication.

Peyton Jones, S. (1987) *The Implementation of Functional Programming Languages*, Prentice-Hall.

Preparata, F. P. and Hong, S. J. (1977) Convex Hulls of a Finite Set of Points in Two and Three Dimensions. *Comm. ACM* **20**(2): 87–93.

Requicha, A. A. G. (1980), Representations for Rigid Solids: Theory, Methods and Systems. *ACM Comput. Surv.* **12**(4): 437–464.

Saia, A., Bloor, M. S. and de Pennington, A. (1987), Sculptured Solids in a CSG Based Geometric Modelling System. In: *The Mathematics of Surfaces II*, IMA, Oxford University Press.

Samet, H. (1984) The Quadtree and Related Hierarchical Data Structures. *ACM Comput. Surv.* **16**(2): 187–260.

Samet, H. (1990) *Applications of Spatial Data Structures*, Addison Wesley.

Samet, H. and Tamminen, M. (1985) Bintrees, CSG Trees and Time. *Computer Graphics* **19**(3): 121–130.

Shamos, M. I. (1977) Computational Geometry, PhD thesis, Yale University.

Shephard, M. S., Baehmann, P. L. and Grice, K. R. (1988) The Versatility of Automatic Mesh Generators Based on Tree Structures and Advanced Geometric Constructs. *Comm. Applied Numerical Methods* **4**: 379–392.

Skillicorn, D. B. (1995) Categorical data types. In" J. R. Davy and P. M. Dew, eds., *Abstract Machine Models for Highly Parallel Computers*, Oxford University Press.

Tilove, R. B. (1980) Set Membership Classification: A Unified Approach to Geometric Intersection Problems. *IEEE Trans. Computers* **29**(10): 874–883.

Tilove, R. B. (1981) Exploiting Spatial and Structural Locality in Geometric Modelling. PhD thesis, University of Rochester.

Wallis, A. F. and Woodwark, J. R. (1984) Creating Large Solid Models for NC Toolpath Verification. In" *Proc. CAD 84*, Brighton, UK, pp. 455–460.

Woodwark, J. R. (1986) Generating Wireframes from Set-Theoretic Solid Models by Spatial Subdivision. *Computer-Aided Design* **18**(6): 307–315.

Wyvill, G. and Kunii, T. L. (1985) A Functional Model for Constructive Solid Geometry. *The Visual Computer* **1**: 3–14.