

BUILDING PARALLEL APPLICATIONS WITHOUT PROGRAMMING

J. Darlington, H.W. To

*Dept. of Computing, Imperial College, London SW7 2BZ
email: {jd,hwt}@doc.ic.ac.uk*

Abstract

To counter the problems of the parallel programming, we propose an approach based on using a fixed range of computation patterns, which we call skeletons. This approach provides a path to efficient but correct programs by separating the issues of behaviour and meaning, and through the exploitation of the knowledge we have about the fixed patterns of computation and communication of the skeletons.

The method can be applied to specific application domains by observing that recurring patterns of data and control structures appear in these domains. Often, for any one domain, there may be many different sets of patterns reflecting abstract and concrete structures. Equivalences can be derived between these sets, through transformations, to enable problem expression at an application's level, but efficient execution at the machine level. We demonstrate this through an example in solid modelling.

1.1 Introduction

Even though there has been a rise in the number of commercially available parallel machines, there are still too few real applications which use them. The main factor in this is software. Programming parallel machines is considered to be more difficult than is the case for sequential machines. The diversity of parallel machine architectures and their associated computational models make **predictability of performance** and **portability** of software difficult. These problems did not arise so critically for sequential machines because there exists a single computational model, the von-Neumann model, to which all sequential machines adhere.

The von-Neumann model of computation provides a simple relationship between language constructs and their implementation. Issues such as memory allocation are resolved by the compiler with no performance implications, allowing the programmer to concentrate on higher level aspects of programming. Thus the programmer can predict the performance of their program without worrying about the run-time resource allocation.

This is not true for parallel programming where mapping a program to a multiprocessor machine is a complex task involving decisions about task allocation, scheduling of competing processes' communication patterns, etc. Usually the only way to achieve the desired performance is through explicit control, which adds to the complexity and non portability of the program.

The universality of the von-Neumann model guarantees portability of sequential programs at the language level, with no danger of unforeseen degradation in performance. With the explicit task allocation of parallel programs there is rarely any portability. Even when portability is possible, through the use of high level programming languages, which can be compiled to other machines, there is no guarantee that performance will be maintained.

Given the success of the von-Neumann model for sequential computing the conventional solution to the parallel software dilemma is to attempt to produce a 'parallel von-Neumann' model. The search is then for a uniform abstract machine model for parallel computing. The rôle of such an abstract machine model is to provide an abstraction of the execution behaviour of the target machines. Programming languages are then defined on these models, with programs written to organise the machine's activities to produce correct answers and behave efficiently. The need to express both correctness and efficiency in the same framework requires that there needs to be a close correlation between the abstract and concrete machine models. So even if such a model is found, and there are several candidates (6, 14), programming will still remain a fairly low level and laborious activity. Critically, in the conventional programming language framework, the meaning and behaviour of an application is still required to be expressed via a single program. It is not possible to specify one without effecting the other.

Another approach is to exploit the **implicit parallelism** of functional languages. These languages provide a much higher degree of abstraction than their imperative cousins. Using this implicit parallelism would alleviate the need to explicitly decompose the problem into concurrent tasks, and to state the control synchronisation and communication between tasks. Unfortunately, automatic exploitation of this has not yet succeeded. This is because the problems of automatic resource allocation have still to be overcome. Compilers can generate too much fine grain parallelism which overwhelms resources and is too costly to spawn. It is difficult to automatically produce load balanced code given the general nature of the problem. Even though the parallelism is implicit, it can be difficult to detect and is dependent on the author of a program expressing it a way which reveals the natural parallelism.

However, functional languages do have some important properties:

- there is a clean separation of behaviour from meaning (Church-Rosser property)
- correctness preserving transformations can be applied to functional programs
- it is possible to write higher order functions, which provide a powerful abstraction mechanism

In the next section we consider another solution which exploits the advantages of functional languages, but in a new, hopefully more practical, manner.

1.2 Skeletons — A radical but practical solution

Both previous solutions result in the meaning and behaviour of a program being entwined. With explicit parallelism, through a uniform abstract machine, the two are expressed in the same language. With implicit parallelism, the behaviour is inferred from the programming language with no user control. Successful parallel programming requires that both meaning and behaviour are expressed. However, the two are orthogonal, programs should be written to be correct, then tuned to be efficient. This implies a separation of meaning and behaviour.

To achieve the separation we propose the use of a range of algorithmic forms, known as **skeletons**, which abstract the useful computational structures from applications. Applications are then constructed as instantiations of these algorithmic forms. The approach follows that taken by Cole (3), for imperative languages, and Backus's idea (1) of programming functional languages through a fixed set of operators.

The skeletons are expressed in a functional language as higher order functions. The skeleton's declarative meaning is then established by this function definition. This meaning is independent of any implementation issues and hence any behavioural constraints. It also provides a sequential prototype which is portable across different parallel machines. Behaviour is defined by the implementation of the skeleton on a particular architecture. So the behaviour of a skeleton can vary from machine to machine, as one would expect. The only parallelism in a program arises from the use of skeletons. All other functions are executed sequentially. Thus, the parallel behaviour of a program is given by the behaviour of its constituent skeletons. All aspects of a skeleton's behaviour, such as process placement and interconnectivity are documented with the skeleton's implementation, along with optional behaviours which can be specified at compile or run time. There is no reason why each skeleton could not be implemented onto every type of architecture. However, different skeletons are more naturally implemented on particular groups of architectures. It would seem wise to implement skeletons only on those architectures that were suitable. Thus the range of skeletons make up the programmer's target abstract machine

model, but one that is expressed at a much higher level and is capable of accommodating a much greater variety of machines and behaviours more easily than for a single abstract machine model.

Transformations are involved at all levels of building applications using skeletons. Using transformation it is possible to derive equivalences between skeletons. We have a fixed set of skeletons, so the possible equivalences can be computed once and stored for future use. These relationships between skeletons provide a route for portability. An application naturally expressed in one skeleton may find that there are no implementations on a particular machine. Choosing the right equivalences will map the application onto the desired machine. Once that particular skeleton-machine pair has been selected then further transformations can be used to optimise the skeleton for the particular machine. An example of a common transformation would be to vary the grain size through partial evaluation (4). Through transformation we have effectively preserved meaning while providing alternative implementations and behaviours.

For each skeleton-machine pair we can construct an associated performance model. The performance models are formulae which predict the performance and whose variables are machine and problem parameters. These models are used to predict the performance of instantiations of skeletons on particular machines, and once a particular machine has been selected to optimise the combination.

The ultimate goal of this work is to replace the inventive process of producing solutions by creating programs from scratch with the development of programs through selection and instantiation of a fixed range of alternatives. Thus there is an explicit identification of all the decisions that have to be addressed to produce an efficient mapping rather than having these decisions made implicitly by writing a program with the desired properties. By recording all the decisions for replay we have a facility for supporting retargeting and portability of applications.

1.3 Some Skeletons

1.3.1 *Initial Skeletons*

To highlight and explore some of the issues of skeletons, we introduce an initial, but incomplete, set of skeletons. The meaning of each skeleton is expressed in Haskell (8). For the definitions we use some standard primitive functions which can be found in (2).

Pipelining or linear process parallelism is captured by the PIPE skeleton. A list of functions are composed together and applied to a list of inputs. Parallelism is achieved by creating a different process for each function and streaming the problem list through the list of processes.

$$\begin{aligned} \text{PIPE} &:: [\alpha \rightarrow \alpha] \rightarrow (\alpha \rightarrow \alpha) \\ \text{PIPE} &= \text{foldr1 } (\cdot) \end{aligned}$$

Simple data parallelism is expressed by the **FARM** skeleton. A function is applied to each element of a list, each element of which is thought of as a task. The function also takes an environment, which represents data which is common to all tasks. Parallelism is achieved by distributing the tasks on different processors.

$$\begin{aligned} \text{FARM} &:: (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \alpha \rightarrow ([\beta] \rightarrow [\gamma]) \\ \text{FARM } f \text{ env} &= \text{map } (f \text{ env}) \end{aligned}$$

So far the skeletons reflect computational patterns which are still quite close to the machine level. It is proposed that there are more abstract skeletons which are closer to the application level. One such skeleton is **RaMP**, reduce and map over pairs. It describes systems where we have two pools of objects. Each object from one pool can interact with every object in the other pool. We are interested in the combined results of these interactions for each object in the first pool. The pools are represented as lists. It is not intended that there should be a direct, efficient parallel implementation of **RaMP**, rather it provides a way to express certain problems naturally, but a naive implementation of **RaMP** would be inefficient. Efficient implementations of **RaMP** can be achieved through transformations to less abstract, but more efficient skeletons.

$$\begin{aligned} \text{RaMP} &:: (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow (\gamma \rightarrow \gamma \rightarrow \gamma) \rightarrow [\alpha] \rightarrow [\beta] \rightarrow [\gamma] \\ \text{RaMP } f \text{ g as bs} &= \text{map h as} \\ &\quad \text{where } h \ x = \text{foldr1 } g \ (\text{map } (f \ x) \text{ bs }) \end{aligned}$$

Other skeletons have been identified and defined. They reflect other common algorithmic forms and computation patterns. Examples of which are the divide and conquer paradigm, **DC**, and parallel map, **PMAP**. Others like dynamic message passing architecture, **DMPA**, express specific types of architecture. A fuller description and examples of their use can be found in (5).

1.3.2 Transformations Between Skeletons

The ability to apply transformations can be used to split the programming process into two phases. The program is first expressed at a natural level then transformations are applied to produce an efficient version targeted towards a particular architecture. Applying this process to skeletons means that we can, similarly, operate with a fixed repertoire of pre-derived equivalences rather than having to invent transformations on each occasion.

It is envisaged that application level skeletons will be transformed to lower level skeletons or compositions of them. This provides our route not only to efficiency, but portability. Porting from one architecture to another will be achieved by re-expressing the skeleton in terms of other skeletons.

As an example, reconsider **RaMP** which we know to be an inefficient, but application oriented skeleton. It can be expressed as a **PIPE**.

$$\begin{aligned}
\text{RaMP } f \text{ } g \text{ } \text{as } \text{bs} \\
&= (\text{map } \text{snd} . \text{PIPE} (\text{map } \text{map} (\text{map } g' \text{ } \text{bs})) . \\
&\quad \text{map } (\text{pair } \text{unit}g)) \text{ } \text{as} \\
&\quad \text{where } g' \text{ } b \text{ } (a, c) = (a, g \text{ } (f \text{ } a \text{ } b) \text{ } c)
\end{aligned}$$

The formal transformation of RaMP to PIPE can be found in (10). Informally, we can see that each stage of the pipe is constructed from an element of the second list, **bs**. The list passed through the pipe is the first list, **as**, with each element coupled with an initialised accumulator. Each stage of the pipe maps the interaction function, **f**, to its element of **bs** and the **as** part of the input list. This result is used to update the accumulator field of each item in the list using **g**. Each stage returns a list of tuples consisting of an element of **as** and an updated accumulator. There needs to be some pre and post processing to build the problem list and to eventually strip away the extra data to leave only a list of final accumulators.

Alternatively, there is a FARM version.

$$\begin{aligned}
\text{RaMP } f \text{ } g \text{ } \text{as } \text{bs} \\
&= \text{FARM } h \text{ } (f, g, \text{bs}) \text{ } \text{as} \\
&\quad \text{where } h \text{ } (f, g, \text{bs}) = \text{foldr1 } g \text{ } (\text{map } (f \text{ } x) \text{ } \text{bs})
\end{aligned}$$

The FARM derivation is relatively straight forward. Each slave is allocated an element of **as**. The processing for each slave involves comparing this with each element of **bs** and then doing the reduction.

1.3.3 Implementation

Implementations of skeletons can take advantage of the extra knowledge one has about a skeleton's behaviour and optimising opportunities to produce efficient, specialised implementations on particular machines. This contrasts with the situation that pertains when using a general purpose programming language, here one has to extract such information by automatic analysis from arbitrary program. Using the skeletons defined we can consider some examples. For PIPE there is a logical locality between adjoining stages. This locality information can be used to allocate processes so that the locality is physically realised. FARM provides a more interesting example in that many machines do not have a broadcast network. To prevent the possible bottle-neck at the master, we may employ tree structures. The skeletons provide information that allows us to specialise their implementation. This is particularly important with regard to communication. For example, asynchronous communication is potentially dangerous, however it is also likely to be more efficient. From the structure of certain skeletons we can see when it is safe to perform asynchronous communication, for example in PIPE.

A prototype implementation of the skeletons has been developed. It uses Hope⁺ (13) as the source language and C as the target language. This initial implementation was carried out on a Meiko Transputer Sur-

face with the CS Tools library (11) providing the communications layer. The set of skeletons implemented to date are PIPE, DC, FARM, PMAP, and DMPA. The initial implementation had no compiler options. However, a preliminary study and implementation of such options has been carried out (9). For several of the skeletons alternative implementations were identified which exploit knowledge of the application to specialise the behaviour of the skeleton. These were implemented and initial investigations showed improvements over the general versions for selected problems.

1.3.4 An Example — Ray-Tracing

In this section we show the use of skeletons in a particular application. We take as our example a simple ray-tracing problem. It follows closely the example given by Kelly (10). Ray-tracing is a technique used to generate images of scenes. The visual attributes of each pixel are determined by tracing a ray from the viewpoint through the pixel and determining which object is struck first, if any. The intensity and colour of the pixel can then be determined from the properties of the object. Taking the rays and objects as two pools of items we can see how the problem can be naturally expressed using RaMP. The rays and objects interact by intersecting and we are interested in the earliest intersection for each ray.

```
RayTrace :: Int → Int → Point → [Object] → [Impact]
RayTrace wd ht viewpoint scene
  = RaMP TestForImpact Earlier (GenerateRays wd ht viewpoint)
```

```
TestForImpact :: Ray → Object → Impact
-- Returns the impact of a ray with an object
```

```
Earlier :: Impact → Impact → Impact
-- Given two impacts it returns the one closest to the view point
```

```
GenerateRays :: Int → Int → Point → [ Ray ]
-- Returns a list of rays for a given screen size and viewpoint
```

Typical parallel implementations of raytracers employ farm or pipeline parallelism. Using the equivalences between RaMP, FARM and PIPE we find that we arrive at the same parallel implementations. However, a ‘eureka’ step of expressing the problem in terms of its possible forms of parallelism was not necessary. The jump from natural specification to a parallel program was one of selection and instantiation.

```
RaMP wd ht viewpoint scene
  = (map snd . PIPE(map map (g' scene)) . map (pair NoImpact) )
  (GenerateRays wd ht viewpoint)
  where g' b (a, c) = (a, Earlier ( TestForImpact a b) c)
```

```

RaMP wd ht viewpoint scene
  = FARM h (TestForImpact, Earlier, scene)
    (GenerateRays wd ht viewpoint)
    where h (f, g, bs) x = foldr1 g (map (fx) bs)

```

1.4 Application Specific Skeletons

The skeletons previously discussed are general purpose. The skeletons methodology can be further refined by observing that many specific application domains have characteristic data and control structures. The concept of skeletons can be taken closer to the application level by abstracting these structures and capturing them as skeletons. Even from a sequential programming view point this has great benefits. Such a system would enable application specialists to develop applications quickly and efficiently without being burdened with low level programming. Once the structures have been identified and given a meaning via a definition then its parallel behaviour can be considered. To facilitate very high level domain specific operations, it may be necessary to contemplate unorthodox mappings to machines.

The next section presents a case study of application level skeletons.

1.4.1 Solid Modelling

Solid modelling is the term used to describe the methods used for representing 3-dimensional objects within a computer. Often these representation are only approximations, hence the term modelling. Associated with these modelling techniques are functions used for manipulating the models and for gathering information from these models. Typically users wish to view the models at a high level of abstraction. One representation of models at such a level is Constructive Solid Geometry, CSG, trees. A scene is constructed from the composition of primitive solids. They can be composed using standard set operations. Each primitive solid is defined not only by its size, but also its orientation and position in space. An example is given in figure 1.1, but without the orientation and position information.

This can be expressed functionally in Haskell as:

```

data PrimitiveSolid = Block Int Int | Sphere Int | Cylinder Int Int |
                    Torus Int Int

type PrimitiveInstance = (PrimitiveSolid, CoordinateSystem3D, Material)

type CoordinateSystem3D = (Position3D, Orientation3D)

type Position3D = (Int, Int, Int)

type Orientation3D = (Int, Int, Int)

```

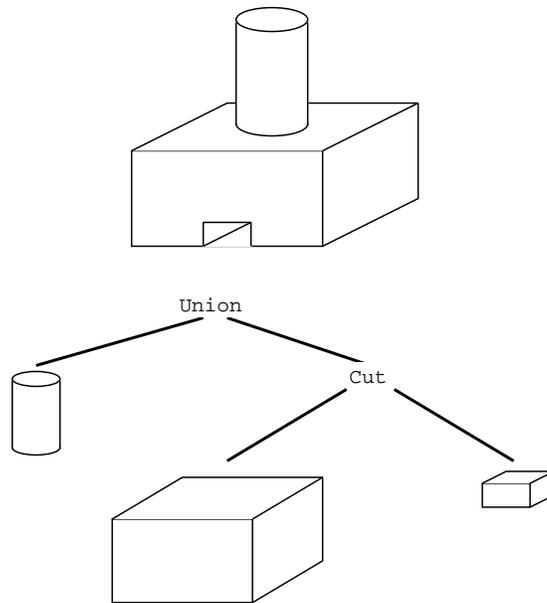


FIG. 1.1. Example of a CSG tree.

```
type Material = MATERIAL
```

```
data CsgTREE = NullSolid | Primitive PrimitiveInstance |
              Composite CsgTREE SetOp CsgTREE
```

```
data SetOp = Join | Cut | Intersect
```

Two classes of operations are performed over `CsgTREE`s; transformations and reductions. These can be expressed as higher order functions, `transformCSG` and `reduceCSG`. Intuitively, `transformCSG` allows us to change the primitives in the tree whilst keeping the structure of the tree unchanged. This provides the ability to make global changes to the model in a succinct way. To extract information from the tree `reduceCSG` is provided. Thus questions can be posed at a model rather than primitive level.

```
transformCSG :: (PrimitiveInstance → PrimitiveInstance)
              → CsgTREE → CsgTREE
transformCSG primF NullSolid
            = NullSolid
transformCSG primF (Primitive primInst)
            = Primitive (primF primInst)
```

```
transformCSG primF (Composite s1 op s2)
  = Composite (transformCSG primF s1) op
    (transformCSG primF s2)
```

```
reduceCSG ::  $\alpha \rightarrow (\text{PrimitiveInstance} \rightarrow \alpha) \rightarrow (\text{SetOp} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha)$ 
   $\rightarrow \text{CsgTREE} \rightarrow \alpha$ 
```

```
reduceCSG base primF compF NullSolid = base
```

```
reduceCSG base primF compF (Primitive primInst) = primF primInst
```

```
reduceCSG base primF compF (Composite s1 op s2)
```

```
  = compF op r1 r2
```

```
    where r1 = reduceCSG base primF compF s1
```

```
          r2 = reduceCSG base primF compF s2
```

One would expect functions of these shapes, since they correspond to map and fold over trees. The next two examples show the use of these skeletons in expressing typical manipulations upon models. In the first example we write a function which translates a model by a given vector.

```
translateCGS :: Vector3D  $\rightarrow \text{CsgTREE} \rightarrow \text{CsgTREE}$ 
```

```
translateCSG v = transformCSG ( translatePrim v )
```

```
translatePrim :: Vector3D  $\rightarrow \text{PrimitiveInstance} \rightarrow \text{primitiveInstance}$ 
```

```
translatePrim v (prim, cs, mat) = (prim, cs', mat)
```

```
    where ((x,y,z), or) = cs
```

```
          (dx, dy,dz) = v
```

```
          cs' = ((x + dx, y + dy, z + dz), or)
```

A more interesting and computationally intensive problem is to decide if a given point lies inside the model. It would also be desirable to know the type of material the point lies in.

```
data Class = Out | In MATERIAL
```

```
testPointCSG :: Position3D  $\rightarrow \text{Class}$ 
```

```
testPointCSG p = reduceCSG Out (testPointPrim p) combinePointClass
```

```
combinePointClass :: SetOp  $\rightarrow \text{Class} \rightarrow \text{Class} \rightarrow \text{Class}$ 
```

```
combinePointClass Join class1 class2
```

```
  = class1 , if (class1 = In mat)
```

```
  = class2 , otherwise
```

```
combinePointClass Cut class1 class2
```

```
  = class1 , if (class1 = In mat) and (class2 = Out)
```

```
  = Out , otherwise
```

```
combinePointClass Intersect class1 class2
```

```
  = class1 ,if (class1 = In mat1) and (class2 = In mat2)
```

```
  = Out , otherwise
```

```

testpointPrim :: Position3D → CsgTREE → Class
testpointPrim p (prim, cs, mat)
  = In mat , if ( inPrimitive p' prim)
  = Out   , otherwise
  where p' = pointToGivenCS3D cs p

```

CSG trees are a very natural way of representing objects. Unfortunately they are not very efficient to process. Other representations are more efficient to process, but are less natural to use. One such method is octrees. It is one of a class of methods known as spatial decomposition. Here the bounding volume of the scene is divided into regular blocks and an object is constructed by marking a block when it contains part of the object. With octrees the space of interest must be a rectangular box. This box is divided into eight regular boxes, octants. An octant is marked white if it contains only space, black if it contains only material. If it is mixed then it is marked grey, and the octant is recursively subdivided to some resolution. This information is held in a tree with white and black leaves and grey nodes. Each node has eight children and the depth indicates the resolution. A two dimensional example is shown in figure 1.2.

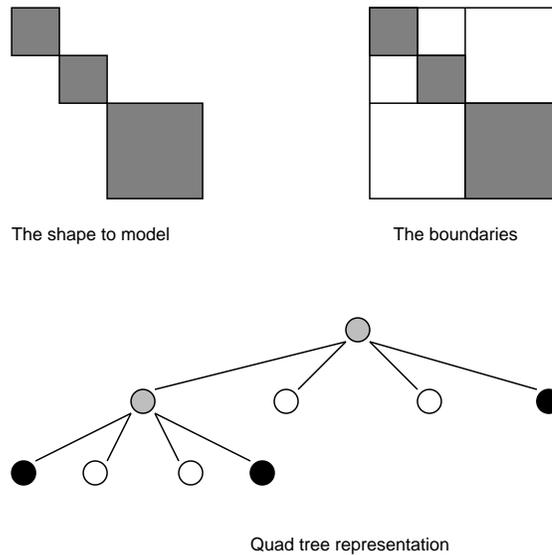


FIG. 1.2. Example of a Quadtree; a 2-D Octree.

It is a simple matter to define octrees in Haskell.

```
type Octree = (Position3D, Size, CubeSubdivisionTree)
```

```
data CubeSubdivisionTree = Cube Class | Subcubes [ CubeSubdivisionTree ]
```

```
data Class = Out | In material
```

Again, we need to process over the model so we must provide functions to manipulate octrees, viz.

```
reduceOctree :: (position3D → size → α) → ([α] → α) → Octree → α
reduceOctree solve combine (p, s, Cube Class)
    = solve ps Class
reduceOctree solve combine (p, s, SubCubes octants)
    = ( foldr1 combine . map F ) octants'
    where F = reduceOctree solve combine
          octants' = zip (generateSubPositionSize p s) octants
```

We have presented two methods of solid modelling. CSG trees provide a natural way of representing objects, but are inefficient to process. Octrees are low level and unnatural to use, but are efficient to process. Functional languages allows us to combine the best of both approaches. Transformation allow us to derive equivalences between CSG tree functions and equivalent octree functions. The user could therefore work entirely at the conceptual level of CSG trees, while all the processing is performed at the octree level.

To generate equivalent function we need to assume that a function `csgToOct`, which transforms any `CsgTREE` to an `Octree`, exists. One is defined in (12). Then for any function `f` over `CsgTREE`'s we require a function `g` over `Octree`'s which conforms to the transitive diagram of figure 1.3. Usually, either α is equal to β or α is equal to `Octree` and β is equal to `CsgTREE`. Given `csgToOct` and a function `f` then `g` can be derived through

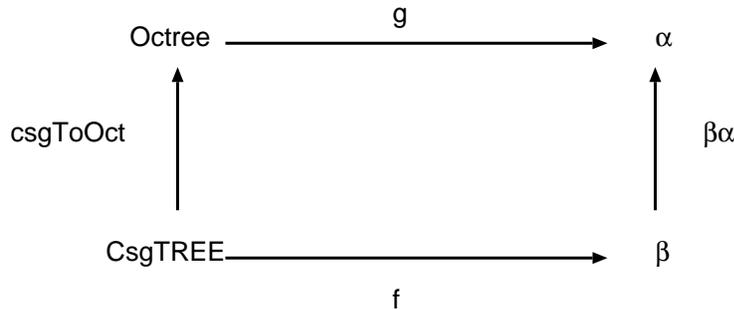


FIG. 1.3. Relationship between `CsgTREE` and `Octree`.

transformations. As an example consider `reduceCSG`. If `combineF` obeys certain properties then the following equivalence holds.

```
reduceCSG nullVal primitiveF combineF csgtree
    = reduceOctree cubeF combineFF (csgToOct csgtree)
```

```

where cubeF p s Out = nullVal
      cubeF p s (In mat) = primitiveF (makePrimitiveBlock p s mat)
      combineFF = combineF Join

```

This example is explored in greater detail in (12) and the technique for transformation is described in (7). Since both the abstract and the concrete types are known in advance, the equivalences can be developed once and then stored for future use.

Given these equivalences the intention is that although the user would specify his computation in terms of CSG trees all execution would take place on the octree representation.

Possible implementations of the octree skeletons are guided by its tree structure. For message passing architectures we may consider dynamic divide and conquer to traverse the tree, if the tree is balanced or nearly balanced. For sparse trees we may consider a master-slave implementation with the master walking the tree and handing leaves to slaves; this may also be the case if the tree is large so to reduce communication only the leaves of the tree need to be passed. A method for processing octrees with SIMD architectures is studied in (12).

1.5 Conclusions

Writing parallel programs is difficult. There are many issues; resource allocation, communications protocol, etc.; that require explicit machine dependent programming to ensure efficiency. This, of course, has an adverse effect on portability, predictability of performance for any ports, and complexity. The skeletons approach addresses these problems by identifying the common computation patterns in problems. By isolating these as the building blocks of parallel programming, we can build efficient implementations for each block. Portability is provided by various implementation on different machines and by transformations between skeletons. Performance can be predicted through the use of performance models attached to each skeleton. The use of skeletons allows for the separation of behaviour and meaning. Programming for complexity is reduced by allowing the programmer to first address the correctness or meaning of a program, and then to tackle the behaviour to ensure efficiency.

This approach can be applied at a higher level to application domains. The recurring data and control structures for applications can be abstracted and incorporated into skeletons. The applications builder can then work at a familiar level, and need not address the unfamiliar low level details of parallel machines. We have seen this applied to solid modelling and investigations are continuing into other fields, such as numerical problems and databases.

The aim of the work is to provide programming through the selection and instantiation of skeletons aimed at the desired application area.

Further work needs to be done to investigate the information needed to specify behaviour of a skeleton completely, and a notation for expressing it. So far we have not discussed the composition of skeletons. For, efficiency the behaviour of interacting skeletons must be compatible. We must consider how these behaviours can be optimised and whether this process can be automated. These research areas are being actively pursued. Work is also continuing with the specification of performance models for skeletons.

1.6 Acknowledgements

We would like to thank our colleagues in the Advanced Languages and Architectures Section at Imperial College for their assistance and ideas. Much of the work in Subsection 1.4.1 is based on (12). The work reported here was initially developed in the UK SERC/DTI funded project 'The Exploitation of Parallel Hardware using Functional Languages and Program Transformation' and used equipment funded under the SERC's Parallel Equipment Initiative. This work is being continued under a SERC Research Scholarship to the second author.

REFERENCES

1. Backus, J. (1978). Can Programming Be Liberated from the von-Neumann Style? A Functional Style and its Algebra of Programs. *CACM*, **21(8)**, 613-41.
2. Bird, R. and Wadler, P. (1988). *Introduction to Functional Programming*. Prentice Hall.
3. Cole, M. (1989). *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman/MIT Press.
4. Darlington, J. and Pull, H. M. (1988). A Program Development Methodology Based on a Unified Approach to Execution and Transformation. *Partial Evaluation and Mixed Computation*. North-Holland.
5. Darlington, J. , Field, A.J. , Harrison, P. G. , Kelly P. H. J. , Sharp, D. W. N. , Wu, Q. (1993). Parallel Programming Using Skeleton Functions. *Parallel Languages And Architectures, Europe:Parle '93*. To appear.
6. Fortune, S. and Willey, J. (1978). Parallelism In Random Access Machines. *10th ACM Symposium on Theory of Computing STOC*.
7. Harrison, P. G. and Khoshnevisan, H. (1992). The Mechanical Transformation Of Data Types. *The Computer Journal*, **35(2)**.
8. Hudak, P. , Peyton Jones S. L. , Wadler, P. I. , Boutel, B. , Fairburn, J. , Fasel, J. , Guzmán, M. , Hammond, K. , Hughes, J. , Johnsson, T. , Kieburtz, R. , Nikhil, R. S. , Partain W. and Peterson J. (May 1992). Report on the Functional Programming Language Haskell. *SIGPLAN Notices*, **27(5)**.
9. Isaac, C. A. (1992). *Structured Implementations of Functional Skeletons*. MSc Project Report, Dept. of Computing, Imperial College.
10. Kelly, P. H. J. (1989). *Functional Programming for Loosely-coupled Microprocessors*. Pitman/MIT Press.
11. Meiko Ltd. (1990). *CS Tools for SunOS*. 83-009A00-02.02.
12. Papachrysantou, G. (1992). *High Level Forms for Computation in Solid Modelling*. MSc Project Report, Dept. of Computing, Imperial College.
13. Perry, N. (1989). *Hope⁺*. Internal document IC/FPR/LANG/2.5.1/7, Dept. of Computing, Imperial College.
14. Valiant, L. G. (1990). General Purpose Parallel Architectures. *Handbook of Theoretical Computer Science*. North-Holland.