

Categorical Data Types

D.B. Skillicorn*

Department of Computing and Information Science
Queen's University, Kingston, Canada
skill@qucis.queensu.ca

December 8, 1992

Abstract

An ideal abstract model for parallel computation must carefully balance requirements for effective software engineering with requirements for efficient implementation. Models based on sets of fixed communication/computation patterns satisfy these requirements but, in general, the sets of patterns are chosen arbitrarily. Categorical data types are a way of building such models while automatically generating operations, equations, and a guarantee of completeness. We illustrate this construction, and its usefulness for practical problems, by building the type of chemical molecules and showing how molecular properties can be computed in parallel.

Keywords: abstract models, parallel computation, program transformation, skeletons, data parallel programming, cost calculus, molecular modelling, category theory.

1 Introduction

An ideal abstract model of parallel computation is constrained by both its upward and downward relationships. Upwards, a model must allow programs to be constructed from specifications in a reasonable way and must hide unnecessary details of target architectures. Downwards, a model must allow effective implementations to be built. The central problem in choosing abstract models for parallel computation is to find the correct balance between these two sets of constraints.

The relationship between specification and model seems to require the following properties:

*This work was supported by the Natural Sciences and Engineering Research Council of Canada. Submitted to the Second Workshop on Abstract Machine Models for Highly Parallel Computers, Leeds, 1993.

- *Architecture Independence.* Since the range of possible target architectures changes frequently and there is no style of architecture that is likely to remain the best as technology changes, long lifetimes for software can only be achieved if they are insulated from architecture considerations. Thus an ideal model cannot include any assumptions about how abstract operations are achieved; for example, explicit message passing or assumptions about memory organisation must be ruled out. We can summarize architecture independence by saying that it means that program source does not need to be changed when a program is ported to a new architecture.
- *Development Method.* We have gotten away with being casual about the way sequential software is developed in all but the most safety-conscious and potentially litigious situations. In a parallel world, correctness is not an optional issue since incorrect programs will tend not to run at all. While it is just conceivable that a program development strategy based on verification could be used, it is much more natural to consider the calculational or derivational style of program development as the way to build parallel software. This can be based either on refinement or equational transformation. It requires that an abstract model have a sufficiently strong semantics to make derivations possible.
- *Intellectual simplicity.* Humans need to be able to keep in mind what a piece of software they are working on does, at least in some weak sense. It is hard to imagine a mental model of software with many thousands of threads, multiplexed onto some smaller number of processors, in which the detail of what is going on is important. Thus an abstract model must allow programmers not to think about substantial parts of the computation. While some of the management of this abstraction can doubtless be hidden by a compiler, it strongly suggests that regularity is important.

All of the properties listed above can be achieved by making the model sufficiently abstract. However, the requirements of interacting with an implementation exert a pull in the other direction, towards models that are very concrete. In particular, we can identify the following properties for this relationship:

- *Congruence.* If software is to be built in a calculational way, it must be possible to make decisions along the way with some understanding of their implications on the final cost of the computation. Thus the model must expose some of the details of the implementation, without violating either architecture independence or intellectual simplicity. Thus the model must permit some set of cost measures that can be computed from a small number of parameters. A cost calculus, that is measures that respect the structure of the transformation or refinement system are even better.
- *Efficient implementation.* An even stronger property than congruence is that the model should not itself introduce any non-intrinsic inefficiency into an implementation. Thus a comparison between the cost of a computation within the model and an abstract cost (using say Boolean circuits or the PRAM) should not reveal any hidden cost differences.

These requirements are challenging and it may not be possible to satisfy them all in general. Nevertheless, a clear picture of the ideal enables us to assess both goals and progress.

Considering these five criteria suggest two ways in which a good model might be found. Then first is to take a model that is very abstract and then try to make it efficiently implementable. This is the approach taken with Unity/PCN [6], higher order functional programming [13], and Concurrent Rewriting Logic (Maude) [12]. The other approach is to take a model that has a limited set of communication and computation patterns and try to make it an effective platform on which to build software. This is the approach taken by models that are called skeleton-based in a functional setting [7–9] and data-parallel in an imperative setting [5, 14]. It is also the approach that we advocate in this paper.

2 Categorical Data Types

The major drawback of models based on the fixed patterns is the choice of patterns. These have typically been chosen based on some perception of usefulness, but without any consideration to whether the resulting set is either complete or non-redundant. Building datatypes using the categorical data type (CDT) construction of Malcolm [11] avoids this drawback while producing models in which parallelism is perfectly natural.

Rather than give the details of the construction, we will illustrate by building a slightly unusual datatype – that of chemical molecules. The construction will automatically generate interesting parallel operations that can be applied to molecules, as well as a transformation system for parallel molecular computation programs. An elementary knowledge of category theory is assumed.

We begin with a category of sets and computable functions between them. One particular object, A , of this category will be our starting point – it is the set of chemical elements with which we wish to work. Other objects in the category are the set of integers, the set of reals, and so on. An example of a useful function is the function *atomic weight*, which is an arrow from A to the object *set_of_naturals*.

We define three constructors for the new type M (for Molecule) that we are going to build.

$$\begin{aligned}\alpha & : A \rightarrow M \\ \beta & : M \times M \times R^3 \rightarrow M \\ \gamma & : M \times R^3 \rightarrow M\end{aligned}$$

Intuitively, α takes an atom and makes it into a molecule, consisting of that single atom (essentially a type coercion). The constructor β takes two molecules and a vector giving the relative orientation of two of their component atoms and creates a new molecule, modelling a new bond between the two molecules. The constructor γ takes a molecule and creates a new bond between two of its component atoms, thus allowing cycles within the bond structure

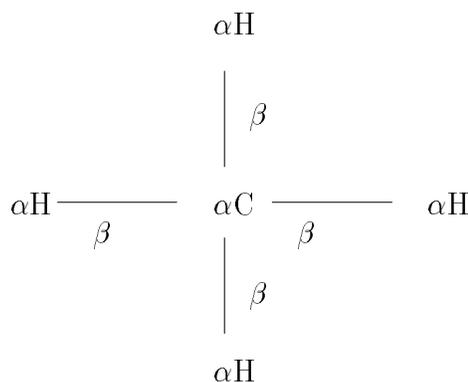


Figure 1: A Methane Molecule

of molecules. We require that β and γ are mutually associative and β is commutative in M – intuitively this means that it doesn’t make a difference in which order molecules are connected together, provided the final molecule is the same. This requirement can be expressed as a set of equations on β and γ and carry over into the algebras that we will shortly introduce. The atoms used for each of these constructions are “remembered” by the physical arrangement of the resulting molecule; that is we use a straight line to represent the application of the both the β and γ constructors.

Instances of the constructed type correspond closely to the stick diagrams that are normally used to draw molecules, except that they contain information about the orientation and length of each bond. A methane molecule is shown in Figure 1.

We can define a polynomial functor T_A by

$$T_A = K_A + Id \times Id \times K_{R^3} + Id \times K_{R^3}$$

such that

$$T_A M = A + M \times M \times R^3 + M \times R^3$$

that is, it maps the constructed type to the components from which it is built. Now define a T_A algebra to be the pair and arrow between them

$$T_A X \rightarrow X$$

(which we will write as $(T_A X, X)$ when the arrow is obvious and (TX, X) when the base type is also obvious) and consider the category $T_A\text{-Alg}$ whose objects are T_A -algebras, and whose arrows are T_A -algebra homomorphisms, that is pairs of arrows in our underlying category that respect T_A -algebra structure. This is shown in Figure 2. This category is where all of

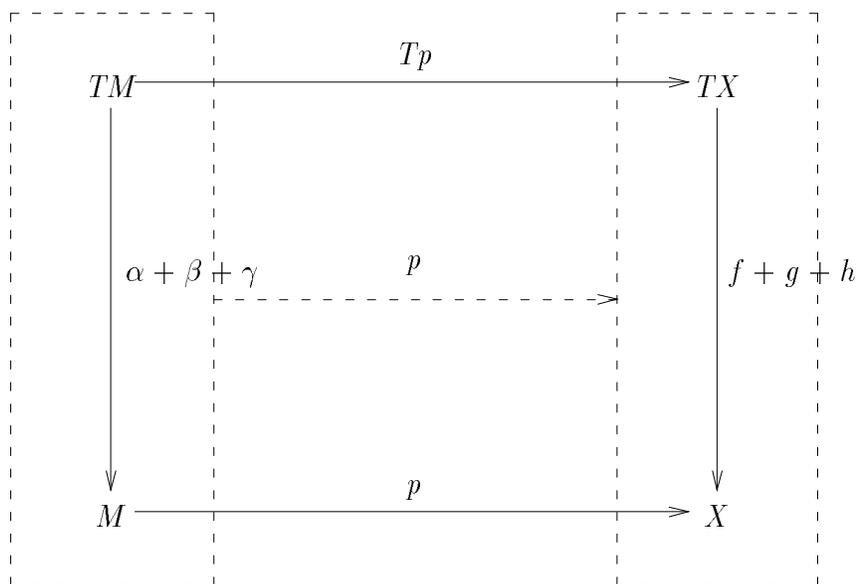


Figure 2: Objects and an Arrow in $T_A\text{-Alg}$

the interesting parallel computation takes place. If T is polynomial, then the T_A -algebra

$$TM \rightarrow M$$

is initial in the category $T_A\text{-Alg}$ and there is an isomorphism

$$TM \leftrightarrow M$$

or

$$A + M \times M \times R^3 + M \times R^3 \leftrightarrow TM$$

Intuitively, this means that a constructed type is isomorphic to the pieces from which it was built, in their proper relationships.

The initiality of $TM \rightarrow M$ in the category of T_A -algebras means that there is a unique arrow from it to any other T_A -algebra. We call these unique arrows *catamorphisms*; they are homomorphisms on the constructed type.

Let us now consider what T_A -algebras are for the molecule type we have constructed. A T_A -algebra is an arrow as shown in Figure 3. Such an algebra can only exist when there are three functions

$$f : A \rightarrow P$$

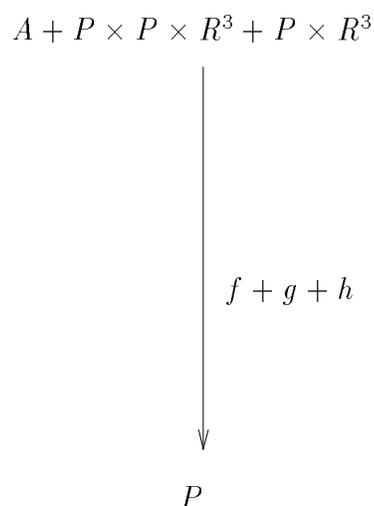


Figure 3: A T_A -algebra

$$\begin{array}{l}
 g : P \times P \times R^3 \rightarrow P \\
 h : P \times R^3 \rightarrow P
 \end{array}$$

that is when P is some *property* that is

1. computable for each individual atom (and hence there must have been an appropriate arrow in the underlying category with which we began);
2. recursively dependent on the properties of component molecules together with their relative orientation and distance;
3. dependent on the addition of new bonds in the molecule

Thus catamorphisms include all functions that compute molecular properties that are recursive and independent of the order in which the molecule was assembled.

Some examples of catamorphisms are:

- **Volume.** Given the volume of each individual atom it is possible to compute the volume of molecules from the volumes of the components and their relative positions.
- **Energy.** Given an energy value for each individual atom, the energy of a molecule depends on the energy of its component molecules and the orientation and distance of the bond that joins them. The energy also depends on any internal bonds that have been created using the γ constructor.

- **Convex Hull.** Given the convex hull of the individual atoms, the convex hull of the molecule can be computed from the convex hulls of the components and their orientation and distance.

It is easy to see that catamorphisms can be evaluated in parallel, and we will make this intuition more precise in the next few paragraphs. Consider the diagram in Figure 4 which shows the arrows involved in a catamorphism in more detail. The lower horizontal arrow is the catamorphism that we wish to compute. One way to compute it is to go around the other three sides of the diagram. The first step is really pattern matching; that is, we examine the molecule to which we are applying the catamorphism and decide which constructor was used to build it. The second step is to follow the upper horizontal arrow, which computes the catamorphism on base cases or recursively on components of the molecule. The third step is to use the resulting values to compute the final result of the catamorphism.

We have built a new type based only on the original object A . These molecules should, strictly speaking, be written as M_A because we can use exactly the same construction for any of the other objects in our underlying category. For example, if we start with the object Int of integers, then we can define constructors:

$$\begin{aligned}\alpha & : Int \rightarrow M_{Int} \\ \beta & : M_{Int} \times M_{Int} \times R^3 \rightarrow M_{Int} \\ \gamma & : M_{Int} \times R^3 \rightarrow M_{Int}\end{aligned}$$

and a functor T_{Int} . As before, we can build a category $T_{Int}\text{-Alg}$ in which “molecules” whose nodes are integers, rather than atoms, are initial. In fact, all of the types constructed in this way have the same *structure* but different *content*, that is what is at the nodes of the “molecule” will differ, but the structure of an oriented graph remains the same.

All of the T_X -algebra categories are related in the following way. If the original underlying category has an initial object \emptyset , then the category $T_\emptyset\text{-Alg}$ has $(TM_\emptyset, M_\emptyset)$ as its initial object. All of the other $T_X\text{-Alg}$ categories are subcategories within it. Arrows in the underlying category lift to arrows in $T_\emptyset\text{-Alg}$ called *generalized maps*. Given an arrow $h : X \rightarrow Y$ in the underlying category, the lifted arrow $h* : (TM_X, M_X) \rightarrow (TM_Y, M_Y)$ is an algebra homomorphism that takes a “molecule” whose nodes are of type X and maps it to an identically structured “molecule” whose nodes are of type Y using function h . Clearly maps are parallel operations. This is shown in Figure 5 The maps “paste together” the initial objects of each subcategory in a way that reflects the arrow structure of the original underlying category. Thus we can define a functor from the underlying category to $T_\emptyset\text{-Alg}$ mapping object X to $(T_X M_X, M_X)$ and arrows h to $h*$.

The (polymorphic) type M we have built is a separable type, that is the functor T_A can be separated into two pieces, one of whose codomains is the original object A and the other of whose codomain does not contain an A . Separable types have the following property:

Property Every catamorphism on a separable type can be expressed as the composition of a generalized map and a generalized reduction.

$$\begin{array}{ccc}
& & id + p \times p \times id + p \\
A + M \times M \times R^3 + M \times R^3 & \xrightarrow{\quad} & A + P \times P \times R^3 + P \times R^3 \\
\uparrow & & \downarrow \\
& \alpha + \beta + \gamma & f + g + h \\
\downarrow & & \downarrow \\
\alpha^{-1} + \beta^{-1} + \gamma^{-1} & & \\
M & \xrightarrow{\quad p \quad} & P
\end{array}$$

$$p \cdot \alpha = f \cdot id$$

$$p \cdot \beta = g \cdot (p \times p \times id)$$

$$p \cdot \gamma = h \cdot p$$

Figure 4: Catamorphism Recursion Structure

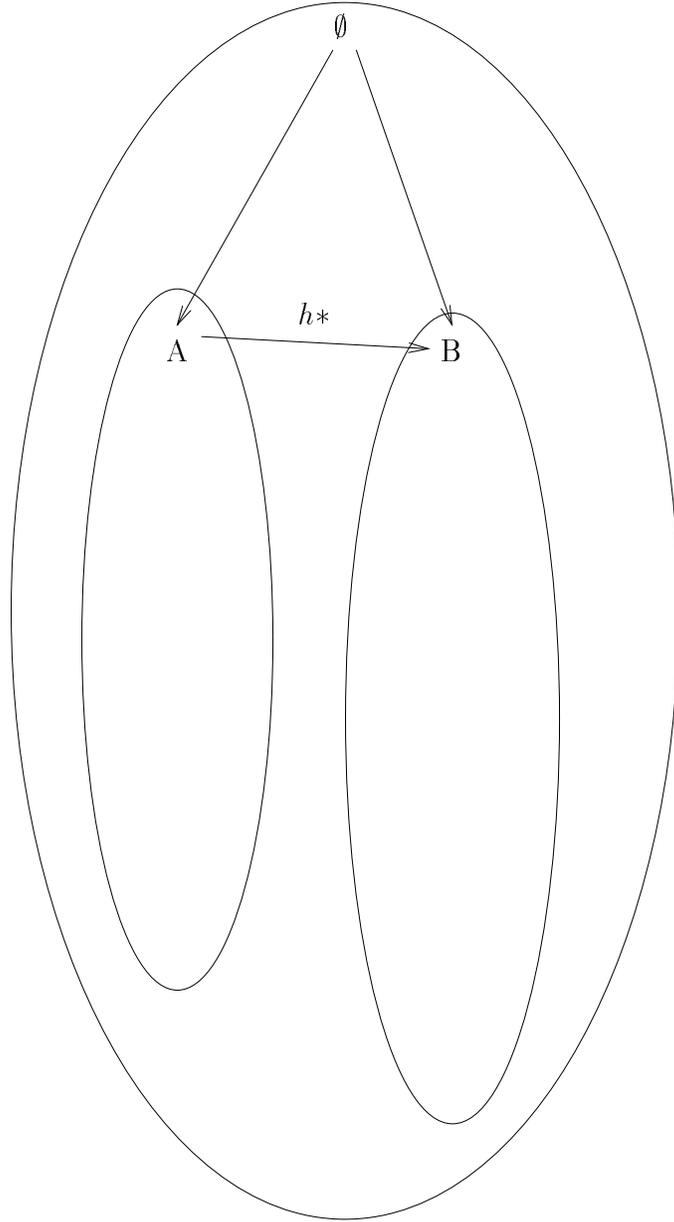


Figure 5: Structure of the category $T_\emptyset\text{-Alg}$ (where \emptyset is the algebra $(TM_\emptyset, M_\emptyset)$, A is the algebra (TM_A, M_A) , B is the algebra (TM_B, M_B) , and h is an arrow from A to B in the underlying category)

A generalized reduction is the catamorphism from an initial algebra to a T_A -algebra (TA, A) in which the function $f : A \rightarrow A$ is an identity. Its effect is to alter the structure without changing the content, and it generalizes the associative reduction on lists. Reductions are also parallel operations, because a reduction is some function of reductions of its component parts, and these can be evaluated in parallel.

The property above can be seen clearly in some of the catamorphisms given as examples above. The energy of the molecule can be computed more or less directly as we described above. However, it can also be computed by mapping the molecule to a “molecule” whose nodes are energy values; and then applying a recursive function to this “molecule” to compute the final value of the energy. The first step, the generalized map, is solely concerned with manipulating the content of the structure; the second step manipulates only the structure.

Another example is the function from molecules to natural numbers that computes the molecular weight (mw). It is an extension of the function aw that computes atomic weights, an arrow in the underlying category from atoms to natural numbers. Figure 6 shows the way in which mw decomposes into a generalized map (the lifting of aw) composed with a generalized reduction.

Another useful benefit of the categorical construction is the property known as *promotion*. Promotion means that there is a canonical form for every catamorphism and that compositions of catamorphisms with homomorphisms can be reduced to a single catamorphism. Suppose that there is a catamorphism

$$p : (TM, M) \rightarrow (TX, X)$$

and a T_A -algebra homomorphism (that is a function that respects the algebra structure)

$$q : (TX, X) \rightarrow (TY, Y)$$

Then there is a catamorphism

$$r : (TM, M) \rightarrow (TY, Y)$$

by the initiality of (TM, M) , and the following equation holds

$$r = q \cdot p$$

Many other equations arise as the result of diagrams in the category $T_A\text{-Alg}$. These equations can be used to do transformational software development of programs that compute with the constructed type.

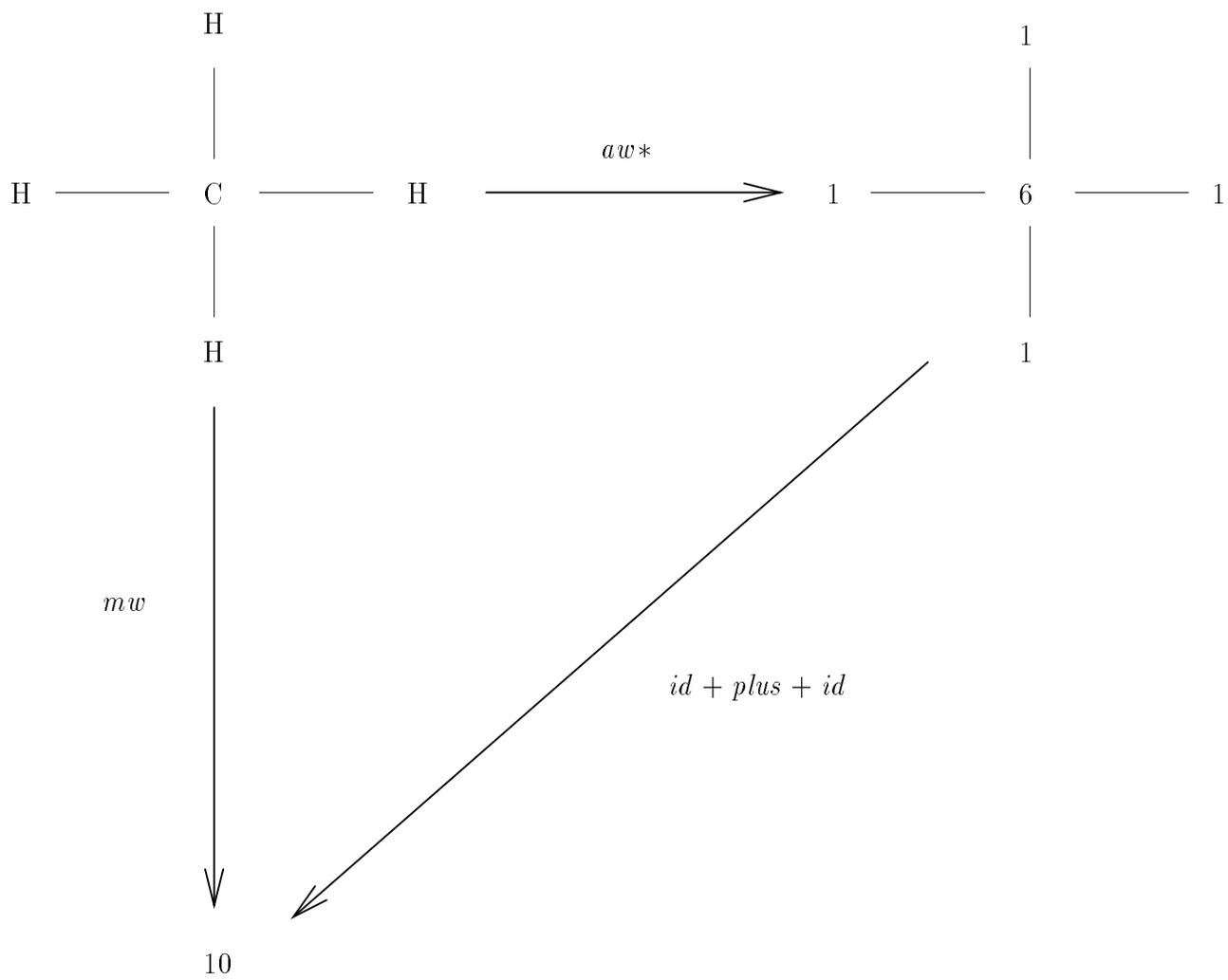


Figure 6: Computing the Molecular Weight of a Molecule

3 Properties of the Construction

We can summarize the properties of the categorical data type construction as follows. Many of these properties should now be obvious from the type of chemical molecule that we have constructed.

- The construction is polymorphic over the types in the underlying category (although this was only used in a peripheral way in the chemical example, it is crucial for more conventional types);
- Based only on the chosen set of constructors, a set of (second order) functions on the constructed type is generated automatically by the construction. These operations are a generalized map and a generalized reduction, both of which are inherently parallel;
- Many interesting functions on the constructed type can be expressed as catamorphisms; examining catamorphisms is therefore a good way to find new and interesting functions;
- A set of equations is automatically generated; these can be used for equational transformational software development.

Two further properties hold that are not obvious from the example above:

- The set of equations forms a *complete* transformation system in the sense that the syntactic form of any homomorphism on the constructed type can be transformed into any other equivalent syntactic form within the system;
- The communication properties of the constructors are directly reflected in the locality of the communication properties of the second order operations; so that for some types, implementations that require only local communication on a range of parallel architectures are possible; when this occurs, cost calculi that apply to a wide range a target architectures can be constructed [15, 16].

The construction of fixed pattern models of parallel computation using categorical data types has all the advantages of such models with respect to the five criteria with which we began; but it also removes any *ad hoc* choices of patterns to use, while at the same time providing a framework for software development.

The type of chemical molecule is directly useful for parallel computation in molecular applications. It also generalizes naturally to a data type of *graph* (the objects become anonymous and the distance and orientation can be dispensed with). Categorical data types for lists (join, cons, and snoc) [2], trees [10], and arrays [1] have also been built. In each case, we get a similar fixed set of parallel operations with known communication and computation patterns. Most of the work has been done with lists, for which many derivations are known [2–4], efficient architecture independent implementations have been built [15, 17], and there is a cost calculus [16].

The categorical data type construction is an important general technique for building fixed pattern abstract models for parallel computation. We have suggested why, in both abstract and practical terms, this should be so.

References

- [1] C.R. Banger and D.B. Skillicorn. Flat arrays as a categorical data type. submitted.
- [2] R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer-Verlag, 1987.
- [3] R.S. Bird. Algebraic identities for program calculation. *The Computer Journal*, 32(2):122–126, February 1989.
- [4] R.S. Bird, J. Gibbons, and G. Jones. Formal derivation of a pattern matching algorithm. *Science of Computer Programming*, 12:93–104, 1989.
- [5] G.E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.
- [6] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
- [7] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.
- [8] M. Danelutto, R. di Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi. A methodology for the development and the support of massively parallel programs. *Future Generation Computer Systems*, 1992. Also appears as “The P^3L language: an introduction”, Hewlett-Packard Report HPL-PSC-91-29, December 1991.
- [9] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, R.L. While, and Q. Wu. Parallel programming using skeleton functions. May 1992.
- [10] J. Gibbons. *Algebras for Tree Algorithms*. D.Phil. thesis, Programming Research Group, University of Oxford, 1991.
- [11] G. Malcolm. *Algebraic Data Types and Program Transformation*. PhD thesis, Rijksuniversiteit Groningen, September 1990.
- [12] J. Meseguer and T. Winkler. Parallel programming in Maude. In J.P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, pages 253–293. Springer Lecture Notes in Computer Science 574, June 1991.
- [13] S. Peyton-Jones. *Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

- [14] G. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming*. MIT Press, 1989.
- [15] D.B. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–51, December 1990.
- [16] D.B. Skillicorn and W. Cai. A cost calculus for parallel functional programming. *J. Functional Programming*, submitted 1992. Also appears as Department of Computer Science Technical Report 92-329.
- [17] D.B. Skillicorn and W. Cai. Equational code generation. *Int. J. Parallel Programming*, submitted.