

Questions and Answers About Categorical Data Types

D.B. Skillicorn*

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

May 1994

What are categorical data types?

The categorical data type approach is an extension of the abstract data type approach in a way that seems particularly useful for parallel computation. It is more than a programming language, however. It is an approach to software development and implementation that relies on mathematical underpinnings for useful structure.

There are several ways of looking at categorical data types. From one perspective, they capture a style of second-order functional programming, in which programs are built from restricted forms of second-order functions that are polymorphic over patterns of computation and communication.

From another perspective they are an extension of objects without state, or abstract data types. Categorical data types merge these two concepts by encapsulating control flow as well as data representation.

Many of the ideas behind categorical data types have been arrived at directly for common data types, sometimes from the perspective of data-parallelism [16], sometimes from the perspective of objects containing parallelism, such as Mentat [15]. The benefits of the categorical data type approach are not primarily in the particular program operations that it provides, but in the framework it provides for building and reasoning about parallel programs.

Why is encapsulating control flow a good thing?

The problem with programming parallel machines is that they vary so much. The best way to compute something on one machine won't necessarily be the best way on another. Encapsulating control flow means that the programmer doesn't have to worry about how

*©D.B. Skillicorn, 1994. Permission is given to copy this document without fee provided it is copied in its entirety and this notice remains attached.

the computation and communication of an operation on the data type are arranged. That's a job for the implementer and compiler writer. So there's a separation of concerns at just the right level — programmers think about monolithic operations on data types, while implementers worry about how to make them happen.

This provides architecture independence. If the target machine is replaced during the night by some new machine, even a completely different architecture, there is no need to alter the software. The differences between machines can be hidden by the compiler.

That means I won't be able to use my knowledge of Machine X to optimise my program to get high performance?

Yes, that's right. But there are several reasons why that might be a good thing. First, unless you're working on a grand challenge problem, the cost of some inefficiency in executing your program is vastly cancelled out by the improved ease of building software, and keeping it running as machines change, which an abstraction provides. Unless you're very unusual, you're spending a lot of time and money keeping your parallel applications running, much more than the cost of software that doesn't extract the last ounce of performance from your hardware. Many companies have not even begun to use parallel computing because they can't see how to get back their initial investment in developing parallel solutions to their existing problems.

Second, it isn't at all clear that a structured implementation will do worse than your hand-optimisations anyway. In the late Sixties, people argued that using structured programming would lead to inefficient programming. The opposite turned out to be the case — structured programs can be compiled to faster code than unstructured ones. In the Seventies, people argued that assembly language coding skills were important for high performance applications — not any more. Compiler generated assembly code will outperform hand code every time. There's no reason to suppose that it will be different for parallel code.

What's good about categorical data types?

Categorical data type programs are compositions of monolithic operations that compute homomorphisms on objects of the data type. These operations hide internal details of how they are computed and how the objects of the data type are arranged and represented. In a parallel setting, this means that they can hide how the computation is decomposed into threads, how these threads are mapped to processors, and how the processors communicate and synchronise. Thus categorical data types provide an abstraction from the complexity of parallel computation. Because they also abstract from the target architecture, they allow software to be *architecture independent*.

The approach also has a natural software development methodology. This has two parts. The first is that the computation of all homomorphisms has exactly the same structure, so that programmers are freed to concentrate on those parts specific to each individual homomorphism. The second is that the construction of a data type automatically provides

a set of equations that can be used for program transformation. The approach encourages derivational software development, in which specifications are transformed into programs.

The underlying regularities that come from a mathematical construction technique can also be exploited in implementations. The communication needed to evaluate a homomorphism can be inferred from each data type's constructors. This communication requirement defines a standard topology for each data type. Any architecture into which the standard topology can be embedded without dilation can implement homomorphisms without any communication cost surprises. Thus efficient implementation of categorical data types is usually possible, although it does not come for free like the software-related properties.

What's a homomorphism?

A homomorphism is a function that respects the structure of its arguments. Suppose A is a type with a constructor \bowtie that makes objects of type A into bigger objects. Then h is a homomorphism if there exists an operation \otimes such that

$$h(a \bowtie b) = h(a) \otimes h(b)$$

We say that h respects the structure induced by \bowtie . If a and b are themselves built from smaller objects of the type A , then the computation of $h(a)$ and $h(b)$ can also be expressed in terms of \otimes . This recursion continues until h is applied to objects of built-in or base types.

Notice that the computation of h follows the structure of its argument. The structure is always the same for each homomorphism. For example, if two homomorphisms, h and h' are applied to an argument $a \bowtie (b \bowtie c)$ then the result will be, in each case

$$\begin{aligned} h(a \bowtie (b \bowtie c)) &= h(a) \otimes (h(b) \otimes h(c)) \\ h'(a \bowtie (b \bowtie c)) &= h'(a) \odot (h'(b) \odot h'(c)) \end{aligned}$$

Only the operations involved are different; the recursive structure and communication of the computation are identical.

Notice also that the the right hand side shows how to compute h in terms of two recursive calls to h , each of which is independent of the other and can therefore be computed in parallel. This is the source of much interesting parallelism in CDT computations.

The operation \otimes depends on the homomorphism h ; in fact, there is a one-to-one correspondence between h and the algebra to which it maps (the target set and operation \otimes). If we wish to find a homomorphism h , one way to do it is to look for an appropriate algebraic structure for it to map to. This amounts to ignoring the common part of the homomorphism, the part that involves the *structure* of the data type, and concentrating on the variable part, the component functions that parameterise the recursive evaluation of the homomorphism. This is an important simplification, and one that becomes more important for programs involving data types with complex structure.

Isn't computing with homomorphisms restrictive?

Yes, it is restrictive, but probably not as much as you think. For a start, all injective functions are homomorphisms, so that includes many useful functions right away. Second, there's a theorem which says that every function can be expressed as a homomorphism followed by a projection. This result is mainly of theoretical interest, because the expression it produces is just a disguised way of computing the original function. However, sometimes it does give a real alternate way of the computing the function; such functions have been called almost homomorphisms [10].

So why are the properties of monolithic operations and equational transformation good ones for CDTs to have?

The big problem with parallel computation today is finding the right level of abstraction to talk about parallel computation. Machines and architectures change frequently, and will continue to do so. The tremendous investment in developing parallel software can only pay off if the software can use these new architectures more or less without change.

We want to be able to write programs that are independent of particular architectures. Parallel programs are significantly more complex than sequential ones, because of all the extra decisions about decomposition, placement, arrangement of communication, and so on, so we need a way to organise the building of such programs. We also want to be sure that the programs we build are correct, that is that they satisfy their specification. This is particularly important because debugging parallel programs is much, much harder than debugging sequential programs, perhaps too difficult to be practical.

This suggests that a good level of abstraction (or *model*) should be mathematically based, so that reasoning and formal development are possible, abstract enough not to be concerned about the detailed arrangement of computations, and architecture-independent.

On the other hand, we want our programs to run efficiently on existing machines, and on those that haven't yet been built or even thought of. This suggests that a good model should be low-level enough that efficient implementations can be found.

Models like higher order functional programming [20], UNITY [9], and Maude [19] have the right kind of properties for programmers, but it has proven hard to build efficient implementations. Models like the Message Passing Interface [11], and Occam make it easy to build efficient implementations, but do not help much with properties that programmers want.

Categorical data types have both kinds of properties: they form a model that is architecture independent, abstract, and equipped with a software development methodology (equational transformation plus decomposition of concerns). The internal communication and computation structure of each homomorphism depends on the constructors of the type, and there is always a standard way to implement the evaluation of any homomorphism (although it may not be the best way). This helps with finding efficient implementations.

Maybe an example would make it clearer?

Let's take full homogeneous binary trees [13]. The objects of this type are binary trees with a value from some underlying type A at each node (leaves and internal nodes) — so the trees are homogeneous — and a node has either two descendants or none — so the trees are full.

This type has two constructors, one that makes a single A value into a tree consisting of a single leaf, and a second that takes two trees and a value of type A and joins them together into a larger tree.

$$\begin{aligned} \text{Leaf} & : A \rightarrow A \text{ } \text{!} \\ \text{Join} & : A \text{ } \text{!} \times A \times A \text{ } \text{!} \rightarrow A \text{ } \text{!} \end{aligned}$$

Homomorphisms on these trees are functions from the set $A \text{ } \text{!}$ with the two operations, *Leaf* and *Join*, to any other set P with operations

$$\begin{aligned} p_1 & : A \rightarrow P \\ p_2 & : P \times A \times P \rightarrow P \end{aligned}$$

In fact, there is a one-to-one correspondence between such structures (set + operations) and homomorphisms.

So if I want to find a homomorphism from trees of A s to some type P , I can either think about what h should do directly, or I can think about what operations on the set P would correspond to the action of h . This is usually an easier problem because h involves two different sets, $A \text{ } \text{!}$ and P , whereas p_1 and p_2 are operations on a single set.

All homomorphisms on trees can be computed using the following single recursion schema, which is parameterised by functions p_1 and p_2 . Using the fact that tree homomorphisms are in one-to-one correspondence with such sets of functions can be a useful way to find interesting tree algorithms.

The recursion schema for trees is shown in the following pseudocode:

```
eval_hom (p1, p2, t)
case t of
  Leaf(a) :: p1 (a)
  Join(t1, a, t2) :: p2 ( eval_hom (p1, p2, t1,
                                a,
                                eval_hom (p1, p2, t2))
```

Let's look at some examples of homomorphisms. Each one is completely defined by giving its component functions.

Example 1 Number of nodes:

$$p_1 = K_1 : A \rightarrow \mathbb{N}$$

$$p_2 = \oplus : \mathbb{N} \times A \times \mathbb{N} \rightarrow \mathbb{N}$$

where

$$\oplus(n, a, m) = n + m + 1$$

and K_1 is the constant 1 function. To see how this works, follow the evaluation of the recursion schema above. Each leaf of the tree is replaced by the value 1; then each internal node computes the sum of the number of nodes in its left and right subtrees, incremented by 1.

Example 2 Height:

$$\begin{aligned} p_1 &= K_0 : A \rightarrow \mathbb{N} \\ p_2 &= \oplus : \mathbb{N} \times A \times \mathbb{N} \rightarrow \mathbb{N} \end{aligned}$$

where

$$\oplus(n, a, m) = \uparrow(n, m) + 1$$

Once again, exactly the same control flow is used to compute this function.

So the recursion schema gives the implementation for all homomorphisms?

Well, it gives *an* implementation for all homomorphisms; so it's at least a starting place. You can see that the sequential time complexity of evaluating tree homomorphisms is $O(n)$ for a tree with n nodes; and the parallel time complexity is of the order of the height of the tree.

In general we can't do better — but there are some common special cases that deserve special treatment.

And they are?

The first kind are *maps*. These are homomorphisms that take some function f that maps A s to B s and lifts it into a function that maps trees of A s to trees of B s by applying f pointwise to every node of the tree.

Formally, $map(f)$ is

$$\begin{aligned} p_1 &= (Leaf \cdot f) : A \rightarrow B \text{‡} \\ p_2 &= (Join \cdot id \times f \times id) : B \text{‡} \times A \times B \text{‡} \rightarrow B \text{‡} \end{aligned}$$

If this is implemented using the recursion schema, we must recurse down the tree, dividing it into its subtrees, apply f to the leaves, then join the tree back together exactly as it was, applying f to the values at the internal nodes as we do so. The time complexity will be proportional to the depth of the recursion, which is the height of the tree. Any decent parallel representation of the tree will allow f to be applied to each node concurrently,

giving a constant time implementation. Clearly, this operation is worth treating as a special case with a better implementation.

The second special kind of homomorphism are the *reductions*. These are catamorphisms in which p_1 is the identity. As an example, consider the homomorphism that computes the sum of a tree of natural numbers.

Example 3 Sum is

$$\begin{aligned} p_1 &= id : \mathbb{N} \rightarrow \mathbb{N} \\ p_2 &= \oplus : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \end{aligned}$$

where \oplus is ternary addition.

Reductions leave the nodes alone, but collapse them to compute a single value rather than a tree structure.

Under mild conditions on the function p_2 , reductions can be computed in parallel using an algorithm called *tree contraction* [1, 12]. This algorithm has a parallel time complexity $O(\log n)$ where n is the number of nodes in the tree. This is a big improvement in the potentially linear time of the recursion schema and justifies another special implementation.

For any categorical data type, the implementation of maps can always be optimised. Maps always return a structure identical to their argument, with the only change being the application of the lifted function at each singleton node in the structure. So it never makes sense to disassemble and reassemble the structure. Reductions can usually be optimised, but since they require communication, it depends on the data type's constructors.

There are two other special forms of homomorphism for which special forms are known: they are called *upwards* and *downwards accumulations*.

How does the equational transformation system come into it?

Calculational software development takes the following view: there's usually an obvious solution to a given specification, especially one that's expressed as a comprehension. One typical solution is: generate all the things that might be solutions and check if one of them is. This kind of solution is obvious, but impractically expensive.

The existence of a transformation system means that this initial solution can be transformed, step by step, into a more efficient one. Along the way, the obviousness is usually lost. If all of the transformations preserve correctness, then the final solution is sure to be correct, so it doesn't matter if it's now hard to understand.

There are two other important benefits of the calculational methodology. First, software builders are forced to face up to choices; they must, at certain stages of the derivation, decide to make *this* transformation or *that* one. This is in contrast to traditional software building where the first algorithm that occurs to the programmer is often the one that gets used. Second, the record of the derivation is a useful kind of documentation for the program — it explains how the program was built and, if the choice points are documented, it also

explains *why* it was built that way. When a program has to be altered, the derivation can quickly suggest the point at which a different decision needs to be made; and the new derivation from that point introduces a consistent set of changes to the program.

Where do the equations come from?

They arise in two ways. The first kind of equation comes from considering the recursion schema. Recall that

$$h = \text{eval_hom}(p_1, p_2, _)$$

Applying both sides to a single node, and to a larger tree, we get:

$$\begin{aligned} h \cdot \text{Leaf} &= p_1 \\ h \cdot \text{Join} &= p_2 \cdot h \times id \times h \end{aligned}$$

The second kind of equation arises from the uniqueness of the homomorphism from a data type and its algebra to any other related algebra. Suppose that we have two algebras: one based on set P with operations

$$\begin{aligned} p_1 &: A \rightarrow P \\ p_2 &: P \times A \times P \rightarrow P \end{aligned}$$

and other based on set Q with

$$\begin{aligned} q_1 &: A \rightarrow Q \\ q_2 &: Q \times A \times Q \rightarrow Q \end{aligned}$$

Then f is an algebra homomorphism from p to Q if it respects the operation structure, that is

$$\begin{aligned} f \cdot p_1 &= q_1 \\ f \cdot p_2 &= q_2 \cdot f \times id \times f \end{aligned}$$

Now we know that there is a unique homomorphism from $A\mathfrak{J}$ to the algebra based on P (call it h) and another unique homomorphism from $A\mathfrak{J}$ to the algebra based on Q (call it h'). The “promotion theorem” says that then

$$f \cdot h = h'$$

The expansion of this result in terms of the p_i and q_i gives many useful equations.

You've shown how software development works, and how the model keeps programmers from knowing about the messy details of parallelism. What happens on the implementation side?

The implementer and compiler writer are responsible for providing an implementation that computes homomorphisms. They could just decide to implement the recursion schema directly, either sequentially or in the obvious parallel way, using a fork for the concurrent recursive calls. For modestly parallel machines this is not a bad way to implement homomorphisms.

However, usually there will be performance benefits to building more clever implementations for the special cases, whenever they occur. For trees, if we assume that one processor is allocated to each node of the tree, maps take constant time. Reductions, upwards and downwards accumulations can all make use of tree contraction or extensions of it, and so can be computed in $O(\log n)$ time on shared-memory MIMD machines [1], and distributed-memory MIMD machines with hypercube or hypercube-like interconnect [14, 18]. Reduction can even be computed in the same time bound using only $n/\log n$ processors.

What other categorical data types have been built?

The first type to be investigated was the type of join or concatenation lists, and more is known about them than any other type [2, 5, 6]. Such lists have three constructors:

$$\begin{aligned} [] & : \mathbf{1} \rightarrow A * \\ [.] & : A \rightarrow A * \\ \# & : A * \times A * \rightarrow A * \end{aligned}$$

The first constructor makes an empty list, the second a singleton list, and the third concatenates two lists to make a longer one. Concatenation is associative and the empty list is its identity.

List homomorphisms are functions from list algebras to algebras of the form

$$\begin{aligned} p_1 & : \mathbf{1} \rightarrow P \\ p_2 & : A \rightarrow P \\ p_3 & : P \times P \rightarrow P \end{aligned}$$

for some set P . The function p_3 inherits the associativity property and has identity p_1 .

The recursion schema for lists is

```
eval_hom (p1, p2, p3, l)
case l of
  [] :: p1 (1)
```

```

[a] :: p2 (a)
l1 ++ l2 :: p3 ( eval_hom (p1, p2, p3, l1,
                        eval_hom (p1, p2, p3, l2))

```

List *maps* lift functions such as $f : A \rightarrow B$ to corresponding functions on lists and are homomorphisms of the form

$$\begin{aligned}
[] &: \mathbf{1} \rightarrow B^* \\
f &: A \rightarrow B \\
++ &: B^* \times B^* \rightarrow B^*
\end{aligned}$$

while list *reductions* are homomorphisms of the form

$$\begin{aligned}
e &: \mathbf{1} \rightarrow A \\
id &: A \rightarrow A \\
\otimes &: A \times A \rightarrow A
\end{aligned}$$

where \otimes is the operation of a monoid with identity $e\mathbf{1}$.

Unlike trees, lists are a *separable* type. This means that *any* homomorphism on lists can be expressed as the composition of a map and a reduction. Thus instead of implementing the general recursion schema, it suffices to implement the two special schemas for maps and reductions.

Many derivations involving lists have been done. Some small-scale implementations have been built [8], and quite a lot is known about code generation for list programs [22, 23].

Other types that have been built include bags or multisets, graphs [21], and arrays [4]. Thus categorical data types generalises languages such as Gamma [3], Parallel SETL [16], and NESL [7], as well as more general object-oriented approaches such as Mentat [15] and C** [17].

When I'm deriving programs, how can I decide whether to use one algorithm or another? Don't I need to know the complexity of each one? But I might not even know what kind of machine the program will run on.

A useful model should have one further property: it should be permeable to cost information. The balance here is a little tricky. On the one hand, cost information cannot depend on the target architecture because it isn't usually known when the program is being developed. Even if it were, the amount of detail required to determine execution time costs is usually prohibitive. On the other hand, some cost information must be provided to make these decisions.

It's fundamentally impossible to provide precisely the information we would like here.

However, it is possible to provide a reasonable approximation that should work in practice. The whole subject is complex. You can find a detailed discussion in [23].

So can you sum up the properties of categorical data types?

Categorical data types are a parallel computation model that is suitable for long-term development of parallel software.

- It is architecture-independent, because programs are just compositions of operations on objects of data types. The way in which these operations are translated to patterns of computation and communication on target machines is invisible at the programming level.
- It is abstract because it hides all of the details of the decomposition of operations into threads and their communication. Most of the complexity involved in using massive parallelism is hidden.
- There is a methodology for developing correct software based on correctness-preserving equational transformations. These transformations arise directly from each data type's construction.
- The operations on each data type are all instances of the computation of homomorphisms, and there is a general recursion schema that computes them all. Thus programs are efficiently implementable on any architecture into which the recursion schema can be embedded in a way that allows all communication to be local. Further optimisations can be achieved by implementing certain homomorphisms, maps for example, in even more direct ways.
- Because implementations are well-structured, it is often possible to make cost information available to programmers in an architecture independent way.
- The recursion schema makes it easy to adjust the parallelism of an implementation to match the size of the problem instance to the number of processors available to execute it. Thus generating different code for different size machines is easy. Machines that use parallelism in different ways at different levels (superscalar architectures, for example) can also be exploited.

The basic theory of categorical data types and their application to parallel computation has been worked out. It is now ready to be tried as a methodology on real problems. Many aspects of the approach can be tried at low cost – for example, the evaluation of homomorphisms can be implemented by an object with a single method, *evaluate_homomorphism*. The parameters to this object are two other objects: data type algebras, and target algebras. Data type algebras have two sets of methods, one set corresponding to constructors and the second to destructors (which appear, at least implicitly in the case statements of

the recursion schemas). Target algebra objects have methods corresponding to their component functions, p_i . Implementing the approach in a functional language is even easier, as pattern matching is usually built-in. Both of these approaches could be used to provide categorical data type libraries in any suitable language.

Another opportunity is the use of the methodology for developing interesting parallel algorithms. Derivational software development is a bit short of examples of new and practical algorithms found using it. We have recently developed a family of fast parallel algorithms for search problems in structured text that we do not believe could have been easily found from first principles.

Further information on work being done at Queen's University can be retrieved by ftp from `ftp.qucis.queensu.ca` in directory `pub/skill`. This directory also includes an extensive bibliography of work on architecture-independent parallelism.

References

- [1] K. Abrahamson, N. Dadoun, D.G. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. In *Proceedings of the Twenty-Fifth Allerton Conference on Communication, Control and Computing*, pages 624–633, September 1987.
- [2] R. Backhouse. An exploration of the Bird-Meertens Formalism. In *Proceedings of the International Summer School on Constructive Algorithmics*, Hollum, Ameland, The Netherlands, September 1989.
- [3] J.P. Banâtre and D. Le Métayer. Introduction to Gamma. In J.P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, pages 197–202. Springer Lecture Notes in Computer Science 574, June 1991.
- [4] C.R. Banger. *Construction of Multidimensional Arrays as Categorical Data Types*. PhD thesis, Queen's University, Kingston, Canada, 1994.
- [5] 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.
- [6] R.S. Bird. Algebraic identities for program calculation. *The Computer Journal*, 32(2):122–126, February 1989.
- [7] G.E. Blelloch. NESL: a nested data parallel language. Technical Report CMU-CS-92-103, School of Computer Science, Carnegie-Mellon University, January 1992.
- [8] W. Cai and D.B. Skillicorn. Evaluation of a set of message-passing routines in transputer networks. In A.R. Allen, editor, *Proceedings of the WoTUG 92 World Transputer Users Group, "Transputer Systems – Ongoing Research"*, pages 24–36. IOS Press, April 1992.
- [9] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

- [10] M. Cole. Parallel programming, list homomorphisms and the maximum segment sum problem. In D. Trystram, editor, *Proceedings of Parco 93*. Elsevier Series in Advances in Parallel Computing, 1993.
- [11] Jack J. Dongarra, Rolf Hempel, Anthony J. G. Hey, and David W. Walker. A proposal for a user-level message-passing interface in a distributed memory environment. Technical Report TM-12231, Oak Ridge National Laboratory, October 1992.
- [12] Alan Gibbons and Wojciech Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, 1988.
- [13] J. Gibbons. *Algebras for Tree Algorithms*. D.Phil. thesis, Programming Research Group, University of Oxford, 1991.
- [14] J. Gibbons, W. Cai, and D.B. Skillicorn. Efficient parallel algorithms for tree accumulations. *Science of Computer Programming*, to appear.
- [15] A. Grimshaw. Easy-to-use object-oriented parallel processing with Mentat. *IEEE Computer*, 26, No.5:39–51, May 1993.
- [16] S. Flynn Hummel and R. Kelly. A rationale for parallel programming with sets. *Journal of Programming Languages*, 1:187–207, 1993.
- [17] J.R. Larus, B. Richards, and G. Viswanathan. C***: A large-grain, object-oriented, data-parallel programming language. Technical Report TR1126, University of Wisconsin-Madison, November 1992.
- [18] E.W. Mayr and R. Werchner. Optimal routing of parentheses on the hypercube. In *Proceedings of the Symposium on Parallel Architectures and Algorithms*, June 1993.
- [19] 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.
- [20] S.L. Peyton-Jones and David Lester. *Implementing Functional Programming Languages*. Prentice-Hall International Series in Computer Science, 1992.
- [21] P. Singh. Graphs as a categorical data type. Master’s thesis, Computing and Information Science, Queen’s University, Kingston, Canada, 1993.
- [22] D.B. Skillicorn and W. Cai. Equational code generation: Implementing categorical data types for data parallelism. In *Proceedings of TENCON '94*, Singapore, 1994.
- [23] D.B. Skillicorn and W. Cai. A cost calculus for parallel functional programming. submitted. Also appears as Department of Computer Science Technical Report 92-329.