Prospects for Category Theory in Aldor

Saul Youssef
Center for Computational Science
Boston University
youssef@bu.edu

December 22, 2008

Abstract

Ways of encorporating category theory constructions and results into the Aldor language are discussed. The main features of Aldor which make this possible are identified, examples of categorical construtions are provided and a suggestion is made for a foundation for rigorous results.

1 Introduction

Category theory[1, 2, 3] is one of the most important unifying ideas in mathematics. It is both a candidate for the foundations and provides the large scale structure relating separate areas of mathematics. Naturally, one would expect that the elements of category theory, including objects, morphisms, categories, functors, natural transformation and adjoints, would be basic indispensible elements of modern mathematical software. The high level organization of mathematical software libraries would naturally be categorical and would allow category theory to relate separate areas and allow results from category theory to be encorporated across their entire domain of applicability. It is very striking to the author that this picture has not emerged at all, in spite of the fact that the mathematical importance of category theory has been understood since the 1940s and even though category theory plays a significant role in computer science[4, 5, 6, 7]. Before discussing the

prospects for following this program using the new language Aldor[8], we attempt some explanation as to why this has not happened already.

The most obvious computational difficulty with category theory is the variety of categories of interest. The following categories, for example

objects	morphisms
sets	functions
groups	group homomorphisms
elements of a preorder	$\leq$
integers n, m	$n \times m$ matrices over a ring
points x, y in a topological space	paths from x to y modulo homotopy
categories	functors
functors F, G from category C to C'	natural transformations from F to G

are all of interest. It is hard, however, to find a language construction for which these are all special cases. For instance, in a typical object oriented language, one would certainly want to be able to treat the objects satisfying some base class as objects in a category. Morphisms, however, are then not member functions and already don't naturally fit into the object oriented framework or even have the same type. Even in the cases when these difficulties have been overcome in C++[9] and the functional language ML[10], success in implementing category theory has not lead to attractive general purpose mathematical software. We can suppose that even if category theory is implemented correctly, a particular system may be too awkward or constraining to keep up with the free flow of mathematical ideas in a more general setting. It is important, then, to understand "awkwardness." However, as with poorly understood diseases, we have no cause or mechanism, we only have symptoms. The symptoms are simple constructive mathematical facts which can only be expressed in the system with difficulty. To keep this problem in mind it helps to remember some selection of representative categorical and non-categorical facts such as

1. An associative algebra can be considered to be a Lie algebra where $[x, y] = xy - yx$.

2. Every vector space has a dual. A finite dimensional vector spaces is isomorphic to its dual.

3. Fix an object X in category C. Let objects be morphisms with codomain X and let morphisms from $A \xrightarrow{\alpha} X$ to $B \xrightarrow{\beta} X$ be morphisms $A \xrightarrow{\phi} B$ such that $\beta \circ \phi = \alpha$. This is again a category.

4. The composition of two functors is again a functor.

5. Right adjoints preserve limits, left adjoints preserve colimits.

One of the main reasons that Aldor is particularly interesting in this context relates to "awkwardness" as much as to category theory. We have noticed how often the combination of parametric polymorphism, curried functions, dependent types and extends turns an awkward fact into an easily expressible fact. For example, if A is an associative algebra in Aldor, 1) above is completely expressed in Aldor by

```
extend A: LieAlgebra == A add { bracket(x:%,y:%):% == x*y - y*x }
```

Aldor offers the hope that its new features may contain key missing elements needed for general purpose software that includes category theory.

2 Foundations

There are several potential ways to implement category theory in Aldor. It is helpful to first focus on a restricted solution where morphisms are required to be Aldor functions. More general approaches along the same lines are discussed in Section 9.

Consider an Aldor category with no signatures

```
Domain:Category == with
```

Let objects in the mathematical sense be Aldor domains satisfying Domain. If A and B are such domains, let Hom(A,B) be the collection of Aldor functions with signature $A \to B$. Let the composition of $A \xrightarrow{f} B$ and $B \xrightarrow{g} C$ be the Aldor function

```
(a:A):C +-> g f a
```

and let the identity morphism of an object A be the Aldor function

```
(a:A):A +-> a
```

we claim that this is, in fact, a category. Although we have no proof, we are confident of this claim because of expected properties of Aldor functions and Domains. We expect, for instance, that composition of Aldor functions is associative. This suggests that

- **PreAxiom**: Domain is a category.

be taken as part of our "model of computation." In more detail, we expect

- **Axiom**: Domain is a category with finite products, coproducts, initial, final and exponential objects.

based on expected properties of the Aldor functions, "Record" and "Union" types. If **Axiom** is sufficient for rigorous arguments, it would be especially satisfying since the definition of a category could serve as the foundation for "regular" mathematics[1] and the existence of a particular such category could serve as the foundation for computation.

Given **Axiom** we consider a "Set" to be any domain satisfying

```
Set:Category == Domain with
    =: (%,%) -> Boolean
```

where "=" is an equivalence relation. Morphisms in this category are Aldor functions `f:A->B` where `x=y` implies `f x = f y`. Since each Set is a Domain and each Set morphism is an Aldor function, we only have to prove that composition of Set morphisms is a Set morphism and that identities are Set morphisms to prove that Set is a category. Note that in introducing a category like Set, one is implicitly establishing the convention that the authors of domains satisfying Set must provide a "=" function which is, in fact, an equivalence relation. This convention is required because the Aldor compiler itself has no way of checking whether "=" has the required property. Similarly, we are forced to introduce the convention that the author of any Aldor function `f` from Set A to Set B must insure that it is, in fact, a Set morphism. The same story repeats for more complicated categories. The fact that domains G,H satisfying

```
Group:Category == Set with
    *: (%,%) -> %
    inv:  %  -> %
    1:       %
```

are really groups and that any `f:G->H` is really a group homomorphism is, by necessity, a convention[11]. These conventions imply that there is exactly one mathematical category for each Aldor Category. For example, if one wants to consider, for some reason, the category with groups as objects and Aldor functions as morphisms, one should define a new

```
Group2:Category == Group with
```

to avoid confusing Group morphisms (group homomorphisms) with Group2 morphisms (Aldor functions). Fortunately, this kind of distinction seems to be rarely needed. The reliance on these conventions is less unsatisfactory than it might seem because both objects like G and H and morphisms like f will normally be produced by free construction or other functors where one does have a proof that one is producing groups and group homomorphisms.

Composition of morphisms can be done with native Aldor function composition, but it is sometimes convenient to also have composition and identities via functions as in

```
Id(Obj:Category):Category == with
    id: (A:Obj) -> (A->A)
    default
        id(a:A):(A->A) == (a:A):A +-> a

Compose(Obj:Category):Category == with
    compose: (A:Obj,B:Obj,C:Obj) -> (A->B,B->C) -> (A->C)
    default
        compose(A:Obj,B:Obj,C:Obj)(f:A->B,g:B->C):(A->C) ==
              (a:A):C +-> g f a

MathCategory(Obj:Category):Category == Id Obj with Compose Obj
```

For each category like Group, we will have an Aldor category Group and and an Aldor "package" GroupCategory which inherits at least MathCategory(Group) and has extra exports such as products and coproducts (section 4) according to the extra categorical properties of the particular category of groups.

3 Functors and Adjoints

Given objects as Aldor domains and morphisms as particular types of Aldor functions, a functor from category ObjA to category ObjB can be most directly represented as a domain constructor with signature `Arrow ObjA -> Arrow ObjB` where

```
Arrow(Obj:Category):Category == with
    Domain:   Obj
    CoDomain: Obj
    morphism: Domain -> CoDomain
```

represents a single morphism together with its domain and codomain. Not any such function will do, of course and, as usual, the author of a functor is responsible for insuring that any such domain constructor preserves diagram shapes and commuting diagrams.

Although the above solution works in Aldor, it is preferable to have a less direct treatment of functors and to informally refer to any Aldor domain constructor $L : \text{ObjA} \to \text{ObjB}$ as a "functor" from ObjA to ObjB with the understanding that L has an adjoint $R : \text{ObjB} \to \text{ObjA}$ with natural isomorphism $\text{Hom}_{\text{ObjB}}(L\,A, B) \simeq \text{Hom}_{\text{ObjA}}(A, R\,B)$. The point is that functors usually come in adjoint pairs and if the adjoint isomorphism is known, then the action of L and R on morphisms is provided by the category defaults of

```
Adjoint(ObjA:Category,ObjB:Category, L:ObjA->ObjB, R:ObjB->ObjA):
Category == with
    >>: (A:ObjA,B:ObjB, L A ->   B ) ->  (    A -> R B  )
    <<: (A:ObjA,B:ObjB,   A -> R B ) ->  (  L A ->   B  )

    unit:   (A:ObjA) -> (     A -> R L A )
    counit: (B:ObjB) -> ( L R B ->     B )

    left: (X:ObjA,Y:ObjA,X->Y) -> ( L X -> L Y )
    right:(X:ObjB,Y:ObjB,X->Y) -> ( R X -> R Y )
    default
        unit  (A:ObjA):(    A -> R L A) == >>(  A, L A, id L A)
        counit(B:ObjB):(L R B ->     B) == <<(R B,   B, id R B)
        left (X:ObjA,Y:ObjA,f:X->Y):(L X->L Y) ==
             <<(  X,L Y,compose(X,R L Y,R L X)(f,unit(Y)) )
```

```
    right(X:ObjB,Y:ObjB,f:X->Y):(R X->R Y) ==
        >>(R X,  Y,compose(L R X,X,Y)(counit(X),f))
```

Note that only the two halves of the adjoint isomorphism ">>" and "<<" have to be provided. The unit and counit natural transformations and the action of L and R on morphisms ("left" and "right") are provided by default. This highlights the importance of Aldor category defaults since this seems to be the place where facts from category theory can be used most effectively.

4 Limits and Colimits

Given these variants of Adjoint

```
RightAdjoint(ObjA:Category,ObjB:Category,L:ObjA->ObjB):Category == with
    Right: ObjB -> ObjA
    Adjoint(ObjA,ObjB,L,Right)

LeftAdjoint(ObjA:Category,ObjB:Category,R:ObjB->ObjA):Category == with
    Left: ObjA -> ObjB
    Adjoint(ObjA,ObjB,Left,R)
```

we can define category limits and colimits as right and left adjoints of the diagonal functor. For example, consider products and coproducts of exactly two objects. If Obj is some category, let

```
Two(Obj:Category):Category == with
    one: Obj
    two: Obj
```

be the category of pairs of Obj objects satisfied by the domain

```
Pair(Obj:Category,A:Obj,B:Obj):Two Obj == add
    one: Obj == A
    two: Obj == B
```

Letting

```
Diagonal(Obj:Category)(X:Obj):Two Obj == Pair(Obj,X,X) add
```

the "diagonal functor" for the category Obj is Diagonal(Obj) : Obj $\to$ Two Obj. We can then define products and coproducts of two objects

```
Product  (Obj:Category):Category ==
     RightAdjoint (    Obj,Two Obj,Diagonal Obj) with
CoProduct(Obj:Category):Category ==
     LeftAdjoint  (Two Obj,    Obj,Diagonal Obj) with
```

just as they are defined in category theory. A domain satisfying Product(Set), for example, must supply Right : Two Set → Set which can be implemented with the Aldor Record. A domain satisfying CoProduct(Set) must provide Left : Two Set → Set which can be done with the Aldor Union. Arbitrary products, coproducts and limits and colimits in general follow in a similar fashion.

5 Adding New Categories

To give a feeling for how one introduces a new category, let's pretend that we are inventing groups by adding structure to monoids. First, we define a category

```
Group:Category == Monoid with
    inv: % -> %
```

and decide what conventional properties inv and group morphisms must have. We notice that there is a forgetful functor from Group to Set and a corresponding free construction

```
Forget(G:Group):Set == G add
FreeGroup:LeftAdjoint(Set,Group,Forget) == add
```

and that the category of groups has, say, finite products and coproducts

```
GroupCategory: MathCategory Group with
                     Product Group with
                   CoProduct Group with
== add
```

Given this code, the compiler conveniently informs you that Forget is fine as it is, FreeGroup is missing the free construction Left : Set → Group and its adjoint isomorphism, GroupCategory is missing the product constructor Right : TwoGroup → Group, the CoProduct construction Left : TwoGroup → Group and their associated adjoint isomorphisms. It is easy

to then implement the free group constructor with lists of elements from the set, as usual, and it is easy to implement the product with a Record and the coproduct with a Union. Even better than this, one can implement the product and coproduct adding an "inv" signature to the product and coproduct from the category of Monoids. Best of all, one can use the adjoint functor theorem to notice that left adjoints preserve colimits so one can use the coproduct from Set and apply Left to get the coproduct in Group.

6 Slice Categories

Slice categories are one of the "awkwardness" tests from section 1. Given a category Obj, we would like to fix a particular object X and consider a new category as described in Section 1. Let

```
Slice(Obj:Category,X:Obj):Category == with { slice: % -> X }
```

be the new category in question. We know that slice categories always have a final object where

```
Final(Obj:Category):Category == with
    1: Obj
    1: (A:Obj) -> (A->1)
```

is the Final object category. This fact can be encorporated in the package part of the slice category:

```
SliceCategory(Obj:Category,X:Obj):MathCategory Slice(Obj,X) with
                                          Final Slice(Obj,X) == add
    1:Slice(Obj,X) == add
        Rep == X; import from Rep
        slice:% -> X == (x:X):X +-> rep x
    1(A:Slice(Obj,X)):(A->1) == (a:A):1 +-> slice a pretend 1
```

Slice categories again show the importance of parametric polymorphism.

7 Skeletal Categories

Any preorder P can be considered to be a "skeletal" category where there is a single morphism from $p \in P$ to $q \in P$ if and only if $p \leq q$. Although

morphisms are not functions in this case, this can be done in the framework we have discussed so far by making use of the seemingly pointless

```
Categorify(T:Type):Category == with
    value: T
```

which "turns any type into a category." Given a Preorder P, the skeletal category is Categorify P and the associated package

```
Skeletal(P:Preorder): MathCategory Categorify P with
    homList: (A:Categorify P,B:Categorify P) -> List A->B
== add
    homList(A:Categorify P,B:Categorify P):SingleInteger == add
        import from P
        if (value$A) <= (value$B) then { [(a:A):B +-> never] }
        else { [] }
```

For example, the integers with their normal preorder considered as a skeletal category is obtained by

```
IntegerSkeleton: MathCategory Categorify Integer == Skeletal Integer
```

Morphisms in skeletal categories can't be evaluated as functions, but they can nevertheless be composed, which is all that is required categorically.

8 n–Morphisms

Given a category Obj, define a new category by letting the new objects be morphisms with domain and codomain such as $A \xrightarrow{f} B$ and let a new morphism from $A \xrightarrow{f} B$ to $A' \xrightarrow{g} B'$ be a pair of old morphisms $(A \xrightarrow{\alpha} A', B \xrightarrow{\beta} B')$ such that the square commutes. Composition of such "2–morphisms" is defined in the obvious way. This procedure can clearly be iterated. In order to capture 2–Morphisms, 3–Morphisms etc., one can use recursive Aldor Categories. One should actually obtain all the n–morphisms together with the basic identities and composition by redefining

```
MathCategory(Obj:Category):Category == Id Obj with Compose Obj
         with MathCategory Arrow Obj
```

supplying identities and composition for 1–morphisms, 2–morphisms, ... It is not too suprising that the Aldor compiler objects to this and refuses to have a domain with an infinite number of signatures, even if they are all provided by default. However, it is easy to truncates the recursion above, and this works. Recursive categories are possible in Aldor and this is mostly unexplored territory as far as I know.

9 General Categories

There are interesting categories where morphisms are not functions and so one ultimately wants to move beyond this assumption. This can be done in Aldor by providing a domain constructor $\mathrm{Hom}(A, B)$ for each category which produces the domain of morphisms from A to B. One would then modify MathCategory like so:

```
MathCategory(Obj:Category,Morphisms:Category,Hom:(Obj,Obj)->Morphisms):
Category == with
    id: (A:Obj) -> Hom(A,A)
    compose: (A:Obj,B:Obj,C:Obj) -> (Hom(A,B),Hom(B,C)) -> Hom(A,C)
```

where id and compose no longer have defaults and there can be more than one category for each Aldor category. This solution also allows us to distinguish monomorphisms, epimorphisms and isomorphisms from morphisms in general and to encode simple facts such as the composition of two monomorphisms is again a monomorphism. Development of this approach follows the Aldor function solution closely. However, because the current pre-release version of Aldor (1.1.12p6) does not handle dependent type "Objects" such as

```
(Obj:Category,Morphisms:Category,Hom:(Obj,Obj)->Morphisms,
         Cat:MathCategory(Obj,Morphisms,Hom))
```

this approach is clumbersome at the moment. This approach would also allow the category of categories where morphisms are functors and functor categories where morphisms are natural transformations to be treated as categories in the same sense as Group and Set above. Limiting ourselves to Aldor functions means that these categories must be represented informally.

10 Summary

Aldor does seem to have a sufficient collection of ideas to express category theory effectively. The most critical ideas seem to be Aldor "Categories," parametric polymorphism, dependent types and category defaults. Accepting the limitation that morphisms must be Aldor functions, one obtains concise representations of categories, morphisms, functors, adjoints and natural transformations. Within this framework, simple facts from category theory can be encorporated. We suggest that an axiom that Domain is a category with product, coproduct, initial, final and exponential object captures what one needs to assume to provide a convenient basis for rigorous arguments about the correctness of Aldor applications.

References

[1] Saunders Mac Lane, *Categories for the Working Mathematician*, Springer (1997).

[2] Michael Barr and Charles Wells, *Toposes, Triples and Theories*, `http://www.cwru.edu/artsci/math/wells/pub/ttt.html` (2001).

[3] Chris Hillman, *A Categorical Primer*, `http//www.math.washington.edu/~hillman/papers.html` (2001).

[4] Tatsuya Hagino, *A Typed Lambda Calculus with Categorical Type Construction*, in Category Theory and Computer Science, ed: D.H.Pitt and D.E.Ryderheard, Springer (1987).

[5] R.F.C. Walters, *Categories and Computer Science*, Cambridge (1991).

[6] Richard Bird, Oege DeMoor, *The Algebra of Programming*, Prentice-Hall, (1996).

[7] Robin Cockett, *Strong Categorical Datatypes I*, in the International Meeting on Category Theory, ed: R.A.G. Seely, AMS (1991).

[8] S.Watt, P.A. Broadbery, S.S. Dooley, P. Iglio, J.M. Steinbach and R.S.Sutor, in proceedings of the International Symposium on Symbolic and Algebraic Computation, ISSAC 94, ACM Press, (1994). For recent

Aldor activities, see `http://www.aldor.org`. Note that "Category" is a keyword in Aldor and is not a category in the mathematical sense.

[9] J.J.Barton and L.R.Nackman, *Scientific and Engineering Programming in C++*, Addison-Wesley Longman (1994).

[10] D.E.Rydeheard and R.M Burstall, *Computational Category Theory*, Prentice-Hall (1988).

[11] E.Poll and S.Thompson, *Integrating Computer Algebra Reasoning through the Type System of Aldor* in Frontiers of Combining Systems: Frocos2000, ed: H. Kirchner and C.Ringeissen, Springer (2000).