Semantics of Categories in `Aldor`

A D Kennedy
Department of Physics & Astronomy
The University of Edinburgh

`Time-stamp: <16-JUL-2001 12:34:36.00 adk@MAXWELL>`

Abstract

We consider some questions about the semantics of `Aldor` regarding the way that types can be considered on an equal footing with any other objects in the language. After a digression into the relationship between `Aldor` categories and mathematical categories, we shall discuss the more practical issue of the limitations of the `define` keyword and what the compiler should do about them.

Contents

1 Introduction

Four of the main principles of `Aldor` are:

1. All objects are *first class*.
2. Names are bound to constant values as in λ calculus.
3. Types, both domains and categories, are treated in the same way as any other objects.
4. All type inferences can be determined at compile time.

We shall consider the implications of these principles.

2 First Class Objects

An object is *first class* if its value can be expressed as an anonymous constant expression.

Examples:

```
1@SingleInteger -- is an anonymous integer constant,

(x: SingleInteger): SingleInteger +-> x * x
  -- is an anonymous function constant
  -- (a.k.a. a lambda expression)

with {+: (%,%) -> %; inv: % -> %}
  -- is an anonymous category constant

add {
  Rep == SingleInteger;
  import from Rep;
  (x: %) + (y: %): % == per((rep x) + (rep y));
  inv(x: %): % == per(1 / rep x)
  }  -- is an anonymous domain constant
```

2.1 Categorical Equality

Although there is no explicit domain which implements categories in `Aldor`, it is interesting to consider the categorical properties that such a domain would have. In other words, what operations are performed

on categories by the compiler, albeit implicitly. We may denote some of these as

```
member? : (D: Domain, C: Category) -> Boolean;
= : (CA: Category, CB: Category) -> Boolean
```

for example. Membership is implictly used in verifying type satisfaction, or explicitly by the `has` operation. Equality of two categories A and B is implicitly defined by considering two domains DA and DB which explicitly assert that they belong to A and B respectively. If DA satisfies B and DB satisfies A then the compiler considers A and B to be equal; if not then it considers them unequal.

Since the mathematical meanings of these operations are not computable in general, `Aldor` implements them in a restrictive way. Category equality is defined to be an equivalence relation $\equiv$. If $=$ represents categorical equality in the mathematical sense, then $CA = CB \Rightarrow CA \equiv CB$, but the converse is not necessarily true.

A domain is a `member` of a category only if it explicitly declares itself to be

```
D : Join(..., C, ...)
```

and not in the mathematical sense of just happening to define all the necessary methods and to satisfy the (implicit) axioms.

The question which concerns us here is the precise definition of categorical equality in `Aldor`. When should two categories be considered equal to each other? This should be different from the question as to when the compiler consider them to be equal in practice, as otherwise we would have abolished the possibility of compiler bugs by definition.

Obviously, two equal categories must contain the same set of signatures. The order of the signatures is unimportant, but the names associated with each signature are required to be the same. This means that the multiplication operation in the category `Group` must be called $*$, which is why there is a separate category `AbelianGroup` which has a multiplication operation called $+$, even though logically an `AbelianGroup` should satisfy `Group`. This means that a `Ring` is a `Join(AbelianGroup, Group)` rather than trying to inherit `Group` in two different ways.[1]

[1]Quite apart from the fact that a `Ring` is not strictly a `Group` under multiplication because of the zero element.

Even if two categories containe the same set of signatures they are not mathematically equivalent unless they satisfy equivalent axioms. We alluded to the fact that the equivalence of axiom schema is undecidable before,[2] but `Aldor` cannot even apply the simpler equivalence relation of the categories having syntactically equal axioms because there is no way of stating the axioms in `Aldor`. The usual argument for not having a way of expressing the axioms is that there is not very much one can usefully do with them if one had them, so they might as well be relegated to documentation anyhow.[3]

If we were to agree with these two observations then we might conclude that a good definition for categorical equality in `Aldor` would be that two categories are equal iff they have exactly the same set of signatures with the same names. It is not obvious whether this is the intended definition, or whether a yet more restrictive definition is the goal. For instance, we could define two categories as equal iff they have identical source code, or perhaps identical source code in the same file.

To see why these considerations are not entirely trivial let us ask whether even the last of these putative definition is sufficient. Consider a functor which produces categories, such as

```
F(n: SingleInteger): Category
  == with {dim: SingleInteger == n}
```

Do different evaluations of F with equal arguments[4] produce the same category? In other words, is $F(2) = F(1+1)$? A special case of this arises for nullary functors,

```
F(): Category == with {};

random: () -> Boolean; -- randomly true
```

[2]This is also why the Knuth–Bendix algorithm is not an algorithm.

[3]Axioms can be specified as functions with signatures `name: Tuple % -> Boolean` which are required to be true for all values of their arguments. Explicitly specifying these functions is one way of (i) including the axioms in the formal specification of a category, and (ii) automatically getting the axioms documented by tools such as `Aldordoc`. Adding a new keyword `axioms` with syntactic similarity to `default` within category definitions might permit the "required to be true for all values of their arguments" quantifier to be made explicit, remove the need for the axiom per se to be specified in a `default` clause (with the meaningless possibility of overriding it), allow it to be formatted specially in documentation, and save time and space by not requiring any code to be generated for the functions.

[4]Bear in mind that $1 + 1$ is not equal to 2 in all categories, and that `SingleInteger` might not mean what we think it means.

```
R(): Category ==
  if random() then
    with {a: SingleInteger}
  else
    with {b: Float}
```

Is $F() = F()$? If not, is

```
local C: Category == F();
C = C
```

true? Under what conditions should $R() = R()$?

While we could define categorical equality to always be false this would not be a very useful thing to do, so we need to find a satisfactory non-trivial definition which works for all these cases.

3 λ Binding

Names may be associated with values (objects) at compile time by declaration. We would like to view a declaration of the form

```
local x: SingleInteger == <value>;
<free expression>
```

as being equivalent to the λ binding

```
((x: SingleInteger): SingleInteger
  +-> <free expression>)(<value>)
```

where the `<free expression>` is built out of the symbol x and the exports of the domain `SingleInteger`, such as `+,*,1,0`. Its meaning is identical to textually substituting `<value>` for the symbol x throughout `<free expression>`. Strictly speaking, we mean that the definition

```
local x: SingleInteger == <value>
```

could be replaced by the macro

```
local macro x == (<value>) @ SingleInteger
```

ignoring subtleties such as that

```
local x: SingleInteger == 23;
local x: String == "twenty three"
```

cannot be implemented by typeless macro substitution.

3.1 Free Names

This association may be "factored", meaning that the λ expression

```
(x: SingleInteger): () +-> <free expression>
```

may be defined and compiled separately from its application to `<value>`. If a name is declared to be a dummy argument belonging to some domain in this way then we shall say that the name *belongs to* that domain. In this case it may be used in any expression built from methods of that domain where an object of type `%` is required. It is assumed to be a transcendental element of type `%` in the free construction over the domain).

At first sight this seems to be just a trivial restatement of the previous paragraph where the meaning of `<free expression>` was defined as a macro, but it is fundamentally different because of the assertion that x is *transcendental* over its domain. Consider the expression `1/(x - 23)` to clarify the difference: the λ expression with this body, `(x: SingleInteger): SingleInteger +-> 1/(x - 23)` is perfectly well defined since $x \neq 23$ by assumption. Nevertheless, a division by zero exception occurs when it is applied to the value 23. in categorical language evaluating the λ expression with the dummy argument x bound to the value 23 is constructing the image of the expression $1/(x-23)$ in the quotient domain of free constructions over x modulo the equivalence relation $x = 23$. A more subtle example is that

$$\frac{x-23}{x^2-529} = \frac{1}{x+23}$$

in the free construction $\mathbb{Z}(x)$, and thus the quotient construction could evaluate

```
((x: singleinteger): fraction singleinteger +->
  (x - 23)/(x^2 - 529))(23)
```

to 1/46). Using these semantics would tell us what algebraic simplifications an optimiser should be allowed to do. Of course, in practice the compiler will generate code for the λ expression which will generate a divide by zero exception, so we need to be careful in how we define function evaluation.

Despite these irritating details, the concept of having *variables* or *free names* in this form is very attractive. Indeed, it would not be acceptable if we were required to know the value of every variable at compile time. A reasonable desire for semantic uniformity then leads us to want the meaning of an expression to depend only upon the type a variable has been declared to have and not on its value, which we might not yet know.

Let us now see what pitfalls all these reasonable requirements lead us into.

4 Why Define?

The basic problem is that if we write

```
Foo: Category == with {baz: SingleInteger}
```

then all we should be allowed to know about `Foo` is that it is a category, and therefore only methods which explicitly act on categories can use it (e.g., the `Aldor` built-in `has`). If we want to write the natural statement

```
Bar: Foo == add {baz: SingleInteger == 23}
```

then because `Foo: Category` the forgetful functor is applied to the domain produced by the `add` clause (which wants to export the signature `baz: SingleInteger`, i.e., it has the fully-qualified value of `add baz: SingleInteger == 23 @ with baz: SingleInteger`) to produce a domain with no visible exports. This is a problem, as we presumably hoped to do something with `Foo`, having gone to the trouble of defining it.

`Aldor` tries to solve this problem with the `define` keyword, which declares that the *value* of the assignee is made visible. In the case of the preceding example this means that if we write

```
define Foo: Category == with {baz: SingleInteger}
```

then when `Foo` is mentioned on the right hand side of the `:` (element of) operator its value is visible. This means that

```
Bar: Foo == add {baz: SingleInteger == 23}
```

will now use the value of `Foo` — namely, the constant category `with Baz: SingleInteger` — rather than its type (`Category`), and hence the signature `baz: SingleInteger` will be visible in the domain `Bar`.

Unfortunately this just hides the problem under the rug. We wanted the references to the variable `Foo` to depend only on its type and not its value because its value might not be known if it occurs as a "factored λ binding". The problem crawls out from the other end of the rug when we consider dependent types.

Consider the anonymous function

```
(D: Group, x: D): D +-> x * x;
```

At compile time we do not know the value of the actual arguments D and x, but we do know that the type of D is the category `Group` (strictly speaking, the category constant assigned to the name `Group`), and that of x is D. We cannot require that the value of the right hand side of the `:` operator be used because it is not known at compile time (or possibly ever) in this case.

If we were to try an extra level of dependency, as in

```
(C: Category, D: C, x: D): D +-> ???
```

then we cannot contruct any interesting body (the body x itself is not very interesting): the dependency of types is a kind of homological functor.

The `define` keyword has no useful definition in the case of dependent types because there is no value to make visible, so we can recast our original problem in such a way that there is still a problem:

```
((Foo: Category, Bar: Foo): SingleInteger +-> baz)
  (with {baz: SingleInteger},
   add {baz: SingleInteger == 23})
```

We would like this to be semantically equivalent to binding the names `Foo` and `Bar` locally,

```
(): SingleInteger +-> {
  Foo: Category == with {baz: SingleInteger};
  Bar: Foo == add {baz: SingleInteger == 23};
  baz }
```

but what must we write to obtain the equivalent of

```
(): SingleInteger +-> {
  define Foo: Category == with {baz: SingleInteger};
  Bar: Foo == add {baz: SingleInteger == 23};
  baz }
```

bearing in mind that the definition

```
fizz ==
  (Foo: Category, Bar: Foo): SingleInteger +-> baz
```

and its application

```
fizz(with {baz: SingleInteger},
     add {baz: SingleInteger == 23})
```

may appear in separately compiled modules?

We could give up the elegant identification of the binding of local variables with that of dummy arguments, i.e., that

```
a: SingleInteger == 43;
print("sin(~a) = ~a, whereas exp(~2) = ~a~n")
  (<< sin a, << a, << exp a)
```

means the same as

```
((a: SingleInteger): () ==
  print("sin(~a) = ~a, whereas exp(~2) = ~a~n")
    (<< sin a, << a, << exp a))(43)
```

but not only is this inelegant but it also does not rid of the underlying problem, which is that the value associated with a name might not be known at compile time.

On the one hand consider

```
define Foo: (): Category ==
  if godel?()
    then with {baz: SingleInteger}
    else with {};
Bar: Foo() == add {baz: SingleInteger == 23};
```

where `godel?` is a function whose insides we do not want to mess with. Does the use of `define` require that all functions appearing on the right hand side are not only computable but are also computed? Is the following legal?

```
import from OperatingSystemInterface;
define Foo: (): Category ==
  if getenv("BAZZIT") = "YES"
  then with {baz: SingleInteger}
  else with {};
Bar: Foo() == add {baz: SingleInteger == 23}
```

4.1 Why Category-Valued Functions?

On the other hand, there are examples where we want to have category-producing functions. The following is an example where we want to extend any category C to a new category which has an additional export:

```
Complete(C: Category): Category ==
  C with {floop: Integer -> %}
```

The problem is that when we try to use this

```
bar: Complete Ring ==
  Integer add {floop(i: Integer): % == sample + i}
```

or even the more explicitly `defined`

```
define Foo: Category == Complete Ring;
bar: Foo ==
  Integer add {floop(i: Integer): % == sample + i}
```

the compiler has to invoke the function `Complete` at compile time so that it can determine the value of the exported category in order to tell whether `floop` is exported or not.

What does the `Complete` functor defined above buy us over using the anonymous Category constant

```
C with {floop: Integer -> %}
```

directly? What is wrong with writing

```
bar: Ring with {floop: Integer -> %} ==
  Integer add {floop(i: Integer): % == sample + i}
```

and avoiding all of these problems?

The reason becomes clearer if we consider what the `Complete` functor was really meant to do originally: instead of taking a category and joining the new export `floop: Integer -> %` to it it was meant to take a category-valued functor and add the new export to its value.[5] The original definition was something like

```
Complete: (Tuple Type -> Category)
  -> (Tuple Type -> Category)
```

where the idea as to start with a functor like

```
LinearSpace: Field -> Category
```

and produce

```
Complete(LinearSpace):
  (Field, OrderedField) -> Category ==
    (F: Field, R: Ring): Category +->
      LinearSpace(F) with {floop: R -> %}
```

The correct syntax and semantics for doing this in `Aldor` is unclear. In principle there ought to be a way of getting hold of the domain and codomain of a mapping (i.e., given $f : A \to B$ there should be some way of obtaining $\mathrm{dom}\, f$ and $\mathrm{cod}\, f$), and there should be some operations for assembling bigger `Tuple`s from smaller ones.

At present we partially circumvent these problems by defining `Complete` as a macro, but this is unsatisfactory because

1. It is not strongly typed, so we could not overload it if for some bizarre reason we wanted to.

[5] The exports to be added were a little more interesting too.

2. The semantics of exporting macros into libraries is obscure (can they be `import`ed or do they have to be `#include`d?)
3. We still cannot get `Complete` to be a map from functors to functors.
4. Semantic errors in a macro definition produce error messages when the macro is expanded rather than when it is defined, which can be confusing.

In any case, surely the purpose of `Aldor` is to get the syntax and semantics of second-order types *correct*; after all, everything can always be expressed using macros and assembler!

4.2 What is the Answer?

The only solution to this dilemma seems to be to extend the meaning of `define` to insist that all function occuring within the scope of a `define` construct must be evaluatable (and presumably will be evaluated) at compile time, but the concept of "evaluatable" would need to be defined carefully (are recursive functions allowed, for instance?).

The main objection to this is that it becomes rather ad hoc, and falls short of the simple and elegant mathematical model which `Aldor` purports to have, but there seems to be no other reasonable way of simultaneously satisfying the four principles listed in the introduction.

A What does the Compiler Actually Do?

This appendix is Tom Ashby's notes on what the `Aldor` compiler currently does regarding category-valued functions and `define` keywords.

```
#include "axllib"

-- Example 1

define A : Category == IntegerNumberSystem with;
a : A == Integer add;

i : a := 14;
```

This first example shows the standard way of constructing a new category. Note that the value of the constant A in the domain `Category` (i.e., a `with` expression defining some exports) is used to constrain the type of the domain a.

```
#include "axllib"

-- Example 4

define Apple: Category ==
  IntegerNumberSystem with {Apple?: % -> Boolean};
apple: Apple ==
  Integer add {Apple?(a: %): Boolean == false};

z: apple := 20;
```

The fourth example demonstrates something that is conceptually identical, but involves the addition of an export to the original category.

```
#include "axllib"

-- Example 3

define BananaFunctor(C: Category): Category ==
  C with {Banana?: % -> Boolean}

define BananaINS ==
  BananaFunctor(IntegerNumberSystem);

BananaInteger: BananaINS ==
  Integer add {Banana?(i: %): Boolean == false;}

-- The compiler yacks at this:
-- banana : BananaInteger := 12;
```

The third example presents an alternative way of achieving what the fourth example does, but using a function to construct the category with the extra export. In terms of the function it makes sense to handle the category purely in terms of its type (i.e., a member of `Category`) using the standard abstraction mechanism — that is, the type supports `Join` and `with` operations and the value is irrelevant. However, the value returned from the function (and thus the type

of `BananaInteger`) is not available at compile time without in some sense calling the function then. Thus we have only the return type of the function available, so the type of `BananaInteger` is obscured.

The root of the issue here seems to be a tension between the first class nature of categories and the type system. Does it make any sense to have first class categories if you cannot do anything with them? Also, as the language now stands, there appears to be an inconsistency between the declaration of categories (when constrained by `:Category`) as locals, and the view of locals being simple function applications — i.e., it should be possible to replace local definitions with an anonymous function that achieves the same result. This is merely an extrapolation of our original issue — consider the function version of example 1:

```
  ( (A:Category, a:A):() +-> {i : a := 14; ()} )
(IntegerNumberSystem, Integer)
```

which currently causes the compiler to complain. The implicit `import from` in example 1 is causing compile time evaluation in its commonest form and providing a meaning for the literal 14. As it now stands, the compiler ignores the `define` keyword. I am assuming that it is intended to indicate that the value associated with the identifier can be used (in this case to provide the type of another value) rather than just treating it as a black box with a type — that is the following two lines are intuitively the same:

```
  define A:Category == with {...}
  A == with {...}
```

In a similar vein, it may be an idea to make `with` expressions generative. This would then sidestep the question of equality of categories, in that automatically any instance of a `with` expression, whether it had the same exports as another or not, would not be equal to it. This ties in with the fact that domains are not members of named categories unless they specifically join, as it were.

```
#include "axllib"

-- Example 2

Functor(C:Category):Category == C with;
define B : Category == Functor(IntegerNumberSystem);
```

```
b : B == Integer add;

j : b := 16;
```

As a point of interest, the compiler does not complain at example 2. My first guess is that the unnecessary function call is being removed by some optimisation step. However, this produces inconsistent semantics in the program.

My initial thoughts are that it is somewhat pointless to have "first class" categories. I can only think of two uses for functions that return categories as such. The first is a form of short hand to add an export to an already existing category without having to write the whole category out. This was where the original problem stems from (the `Complete` function in the Paraldor QCD code). This calls for the compile time evaluation of the function to produce the required category against which an already written domain will be matched. Unfortunately, in the awkward case the function could be non-terminating, and obviously this is impossible to decide in general. The second use of a category returning function would be to take run-time input to decide on the typing of the rest of the program (although the distinction between run-time and compile-time becomes a little blurred here). How one would match domains to the resulting category is unclear however. Perhaps there is some mind-bending way that domains could be constructed as a result of user input, but I cannot think of one off the top of my head. Also, I suspect that this would require the grouping together of the category and the domain to be churned out of the function — there would be little point in having the category on its own. In an even more vague way I think that this package could be the basis of dynamic typing (i.e. run-time query of whether a certain function exists for a type) in the language if someone looked into it, but on the other hand Aldor was only meant to be statically typed as far as I know.

Acknowledgements

This document is based on many discussions with Tom Ashby, Martin Dunstan, Balínt Joó, and Stephen Watt.