A NOTE ON CATEGORICAL DATATYPES

G.C.Wraith
Department of Mathematics, University of Sussex

Abstract

It is shown how Hagino's categorical datatypes can be expressed in the polymorphic typed λ-calculus. This gives a way of passing from a description of a datatype in terms of its universal properties, to a representation in terms of λ-expressions.

A Summary of Hagino's categorical datatypes

Hagino's ideas for a more categorical programming language involve two sorts of type declarations:

product X(t) =	sum X(t) =
$d_1 : X \to e_1$,	$c_1 : e_1 \to X$,
.....	.....
$d_n : X \to e_n$	$c_n : e_n \to X$
end	end

The notation is mine, not Hagino's. The symbol X stands for a type constructor identifier, and t for a possibly empty list of covariant type parameters. The body of the declaration is a possibly empty list of destructor/constructor declarations. These play a special role in the definition of X, and correspond to the canonical projections/injections of a product/coproduct. The e_k are type expressions in which the definiend X and the members of t may occur covariantly. The type declaration would have the following effects:

1) X is added to the type constructor environment.
2) The destructors/constructors are added to the value environment.
3) A value, also denoted by X of type

$$(t \to t') \to X(t) \to X(t')$$

expressing the functoriality (or strength) of X is added to the value environment.

If the expressions e_k do not contain X, we simply have products (i.e. records) and sums (i.e. unions). We must interpret recursive products and sums using the notions of terminal coalgebra and initial algebra of an endofunctor A. In the product case A(X) is given by the product of the e_k's and in the sum case by the sum of the e_k's.

Recall that an A-coalgebra is a pair (Y,η) where

$$\eta : Y \to A(Y)$$

and that a map $f: Y \to Y'$ defines a map of A-coalgebras from (Y,η) to (Y',η') if $f;\eta' = \eta;A(f)$. Here f;g denotes the map f followed by the map g. Dually an A-algebra is a pair (X,ε) where

$$\varepsilon : A(X) \to X$$

and a map $f : X \to X'$ defines a map of A-algebras from (X,ε) to (X',ε') if $\varepsilon;f = A(f);\varepsilon'$.

We will denote a terminal A-coalgebra by

$$(\nu X.A(X), d)$$

and an initial A-algebra by

$$(\mu X.A(X), c)$$

when they exist (in which case they are unique up to isomorphism). The maps d and c must be isomorphisms. To see why, consider the A-algebra

$$(A(\mu X.A(X)), A(c))$$

Let $u : \mu X.A(X) \to A(\mu X.A(X))$ be the unique A-algebra map, so that

$$c;u = A(u);A(c).$$

Since we have an A-algebra map from $(A(\mu X.A(X)), A(c))$ to $(\mu X.A(X), c)$ by c, it follows from the uniqueness that

$$u;c = Id(\mu X.A(X))$$

whence

$$c;u = A(u);A(c) = A(Id(\mu X.A(X))) = Id(A(\mu X.A(X)))$$

so that c is an isomorphism with inverse u. An entirely dual argument shows that d is an isomorphism.

It may make these notions more familiar to think of μ and ν as least and greatest fixedpoint operators. However this would hide the fact that c and d are isomorphisms not equalities. It is unfortunate that in the past types have been treated as elements of an algebra (leading to the idea o recursive equations for types) rather than as objects of a category, for which the notion of equality is meaningless. We can only compare types vi values (i.e. maps) which mediate between them. It is the fact that μX.A(X is the underlying object of an initial A-algebra which is the important aspect. That it should be a fixedpoint (up to isomorphism) is in some sense an accident. Besides, thinking in terms of equations obscures the categorical duality between μ and ν. We can cope with mutually recursive product/sum type declarations by using parameters in the usual way.

To summarize, Hagino's categorical programming language is based on categories with:

1) finite products - terminal object 1
   binary products ×
2) finite coproducts - initial object ∅
   binary sums +
3) exponentials →
4) terminal coalgebras ν
5) initial algebras μ

So for example 1 is given by

```
product unit = end
```

and list(α) = μX.(1+α×X) , the type of finite lists, by

```
sum list(α) =
    nil : unit → list,
    cons : α×list → list
end
```

and stream(α) = νX.α×X , the type of infinite lists, by

```
product stream(α) =
    head : stream → α,
    tail : stream → stream
end
```

Similarly we have game = μX.list(X) given by

```
sum game = moves : list(game) → game end
```

and cascade(α) = νX.α×list(X), the type of finitely branching, possibly infinite α-labelled trees, given by

```
product cascade(α) =
   label : cascade → α,
   shower : cascade → list(cascade)
end
```

We could describe this system as a polymorphically typed combinatory algebra, but we forebear to go into the details, which can be had from [Hagino 1]. In this reference, Hagino shows that his calculus, which I will refer to as Cλ, is strongly normalizing.

Note that we have made a sharp distinction between lists (they must be finite) and streams (they must be infinite). They are quite different sorts of type constructor. Because we have strong normalization there is no possibility of fudging the distinction with normal order evaluation. Indeed we have decoupled types from evaluation order. Of course, the print

function cannot be represented in Cλ. Nevertheless, we still have a wealth of functions for manipulating infinite data structures.

We shall show that Cλ can be interpreted as a fragment of 2Tλ, the polymorphic typed λ-calculus. This gives another way of proving strong normalization, and explains why the various "strange" constructions of standard types in 2Tλ, which have been discovered over the years by various authors, really work. It also explains why recursive type definitions and fixedpoint operators for values are redundant for practical purposes in 2Tλ. This in turn answers a question posed in [Reynolds 2, page 113] about principal type schemes and type inference for "infinite recursively defined" types. Essentially, the 2Tλ expressions which encode functions for handling infinite data structures in Cλ use a continuation style of representation; the continuation types arise from the use of quantified type variables in 2Tλ.

2Tλ The polymorphic typed λ-calculus

We will use capital letters for types, lowercase for values. We suppose that we start with a given set of type variables, and a given set of value variables.

Types are formed as follows:

1) A type variable **X** is a type.
2) If **S** and **T** are types so is **S→T**.
3) If **X** is a type variable and **T** a type, **ΠX.T** is a type.

We treat α-convertible expressions, that is those that differ by a consistent change of bound variables, as denoting the same thing. When a type expression **A** is thought of as parametrized by a type variable **X**, we shall write **A(X)** to emphasize its dependence on **X**, and we shall write **A(T)** for the expression obtained by substituting the type expression **T** for **X** in **A**, with appropriate change of bound variables to avoid capture. A type with no free variables is called a closed type.

Values are defined as follows:

1) A value variable is a value.
2) If **f** and **g** are values, the 'application' **f g** is a value.
3) If **f** is a value and **T** is a type then the 'instance' **f T** is a value.
4) If **f** is a value and **x** is a value variable and **T** is a type then the 'value abstraction' **λx∈T.f** is a value.
5) If **f** is a value and **X** is a type variable, not occurring free in the type of a free variable of **f**, then the 'type abstraction' **ΛX.f** is a value.

A typing is a relation **f:T** between values **f** and types **T** satisfying the rules:

1) If $f:S\to T$ and $g:S$ then $(f\ g):T$.
2) If $f:\Pi X.S(X)$ then $(f\ T):S(T)$.
3) If $f:S$ then $\lambda x{\in}T.f:T\to S$.
4) If $f:T$ then $\Lambda X.f:\Pi X.T$.

We may define inductively the relations

$$co(A,X) \qquad contra(A,X)$$

between types A and type variables X, expressing the idea that A(X) depends covariantly or contravariantly on X.

1) $co(X,X)$.
2) If Y is a variable distinct from X then $co(Y,X)$ and $contra(Y,X)$.
3) If $co(S,X)$ and $contra(T,X)$ then $co(T\to S,X)$ and $contra(S\to T,X)$.
4) If $co(S,X)$ then $co(\Pi Y.S,X)$.
5) If $contra(S,X)$ then $contra(\Pi Y.S,X)$.

If $co(A(X),X)$ then we can define the strength of A as a value $\langle A\rangle$ with the closed type

$$\langle A\rangle : \Pi X.\Pi Y.(X\to Y)\to A(X)\to A(Y)$$

by induction. We omit the details. We abbreviate $\langle A\rangle X\,Y\,f$ to $A(f)$ to fit in with categorical practice, with apologies.

Similarly, if $contra(A(X),X)$ we have

$$\langle A\rangle : \Pi X.\Pi Y.(X\to Y)\to A(Y)\to A(X)$$

giving the action of A as a contravariant functor.

Now suppose that A(X) is covariant in X. We make the definition:

$$\mu X.A(X) = \Pi X.(A(X)\to X)\to X$$

and we define:

$$\varphi = \Lambda X.\lambda f{\in}A(X)\to X.\lambda w{\in}(\mu Y.A(Y)).w\ X\ f$$
$$: \Pi X.(A(X)\to X)\to(\mu Y.A(Y))\to X$$

and

$$s = \lambda h{\in}A(\mu Y.A(Y)).\Lambda X.\lambda f{\in}(A(X)\to X).f\ (A(\varphi\ X\ f)\ h)$$
$$: A(\mu X.A(X))\to(\mu X.A(X))$$

The following proposition has been known for some time:

Proposition

$(\mu X.A(X),s)$ is a weakly initial A-algebra. That is, for any A-algebra (S,c) we have a value $f:\mu X.A(X)\to S$ such that $s;f$ β-reduces to $A(f);c$.

Proof: Simply take f to be $\varphi\ S\ c:\mu X.A(X)\to S$.

What I believe is new, is that we have a dual result. We make the definitions:

$$\nu X.A(X) = \Pi Y.(\Pi X.(X\to A(X))\to X\to Y)\to Y$$

where Y is not a free variable of A(X).

$$\phi = \Lambda X.\lambda f{\in}X\to A(X).\lambda x{\in}X.\Lambda Y.\lambda h{\in}(\Pi Z.(Z\to A(Z))\to Z\to Y).h\ X\ f\ x$$
$$: \Pi X.(X\to A(X))\to X\to(\nu Y.A(Y))$$

$$\eta = \lambda w{\in}(\nu X.A(X)).w\ A(\nu X.A(X))\ \Lambda X.\lambda f{\in}X\to A(X).\lambda x{\in}X.A(\phi\ X\ f)\ (f\ x)$$
$$: (\nu X.A(X))\to A(\nu X.A(X))$$

Proposition

$(\nu X.A(X),\eta)$ is a weakly terminal A-coalgebra. That is, for any A-coalgebra (S,d) there is a value $f:S\to\nu X.A(X)$ such that $f;\eta$ β-reduces to $d;A(f)$.

Proof: Just take $f=\phi\ S\ d:S\to\nu X.A(X)$.

Neglecting for the moment the question of whether we have actual rather than weak initiality and terminality, we can argue for the "poly" versions of the standard types as follows:

If $\mathbb{0}$ and 1 denote empty and unit types respectively

$$\mathbb{0} = \mu X.\mathbb{0} = \Pi X.(\mathbb{0}\to X)\to X = \Pi X.1\to X = \Pi X.X$$

$$1 = \mu X.1 = \Pi X.(1\to X)\to X = \Pi X.X\to X$$

If $\times$ and $+$ denote product and sum, then using exponential adjointness

$$X\to Y = \mu Z.X\to Y = \Pi Z.(X\to Y\to Z)\to Z = \Pi Z.(X\to Y\to Z)\to Z$$

$$X+Y = \mu Z.(X+Y) = \Pi Z.((X+Y)\to Z)\to Z$$
$$= \Pi Z.((X\to Z)\to(Y\to Z))\to Z$$
$$= \Pi Z.(X\to Z)\to(Y\to Z)\to Z$$

Note that if we make the definition

$$\Sigma X.A(X) = \Pi Y.(\Pi X.A(X) \to Y) \to Y$$

where Y is not a free type variable of A(X), then we could have

$$\nu X.A(X) = \Sigma X.X \times (X \to A(X))$$

which perhaps brings out the duality better. $(X \times -, X \to -)$ is an adjoint pair dual to the adjoint pair $(- \to X, - \to X)$.

Various "magic formulae" have long been known for representing certain datatypes and functions for manipulating them in terms of the λ-calculus. For example:

$$pair = \lambda x.\lambda y.\lambda f.f\ x\ y$$

is a classic. What we have now is a piece of machinery for producing such representations from categorical specifications of the datatype in terms of its universal properties. First describe the datatype in Hagino's categorical language using **sum** and **product** declarations. Then translate into Cλ, then into 2Tλ, and finally throw away the type information. Of course, this gives in principle a way of compiling Hagino's language, once one has chosen a particular mechanism for evaluating λ-terms. Whether there exist more direct methods that bypass the translation into λ-terms would seem to be an interesting question (especially in the light of the Categorical Abstract Machine), suggesting that we should search for simple operational semantics of Cλ.

By way of example, here is the result of applying this translation scheme to get a representation of **hd** and **tl** for streams:

Define:

$$fst = \lambda p.p\ (\lambda x.\lambda y.x) \qquad snd = \lambda p.p\ (\lambda x.\lambda y.y)$$
$$hd = \lambda s.s\ (\lambda f.\lambda a.fst\ (f\ a))$$
$$tl = \lambda s.s\ (\lambda f.\lambda a.\lambda g.g\ f\ (snd\ (f\ a)))$$

These definitions are got by stripping types from:

$$fst = \Lambda X.\Lambda Y.\lambda p{:}X \times Y.p\ X\ (\lambda x{:}X.\lambda y{:}Y.x)$$
$$snd = \Lambda X.\Lambda Y.\lambda p{:}X \times Y.p\ Y\ (\lambda x{:}X.\lambda y{:}Y.y)$$
$$hd = \Lambda X.\lambda s{:}stream(X).s\ X\ (\Lambda Y.\lambda f{:}Y \to X \times Y.\lambda a{:}Y.fst\ X\ Y\ (f\ a))$$
$$tl = \Lambda X.\lambda s{:}stream(X).s\ stream(X)\ (\Lambda Y.\lambda f{:}Y \to X \times Y.\lambda a{:}Y.\Lambda Z.\lambda g{:}(\Pi W.(W \to X \times Y) \to W \to Z).\ g\ Y\ f\ (snd\ X\ Y\ (f\ a)))$$

which in turn are got from decoding **hd** and **tl** from the η expression for the case $A(Y) = X \times Y$ in $stream(X) = \nu Y.X \times Y$. I hope this gives the general

idea of the process.

It is the thesis of this note that by a categorical model of 2Tλ we should mean something in which the type constructors we have described

$$\pi\ 1 + \times \to \mu\ \nu$$

are interpreted as genuine limits and colimits, that is as initial object, terminal object, sum, product, exponential, initial algebra, terminal coalgebra, respectively. This means that as well as identifying values with their normal forms for β-reduction, we must also impose η-rules in the model. The η-rules are

$$\lambda x{:}T.(f\ x) = f \qquad x\ \text{not free in}\ f$$

$$\varphi\ \mu X.A(X)\ \varepsilon = \lambda x{:}\mu X.A(X).x$$

$$\phi\ \nu X.A(X)\ \eta = \lambda x{:}\nu X.A(X).x$$

using the notation above. For example, the η-rule for π (take $A(X) = X$) asserts that the values

$$\lambda x{:}(\Pi X.X).x \qquad \lambda x{:}(\Pi X.X).x\ (\Pi X.X)$$

of type $(\Pi X.X) \to (\Pi X.X)$ are to be identified. It is not clear to me whether Coquand and Breazu-Tannen's minimal extensional model or Hyland's PER model satisfy these conditions. Extensionality (1 is a generator) would seem an unnecessarily strong condition for a categorical model in general. P. Freyd, in e-mail notes on "Structors" has pointed out that we have more than simply weakly initial/terminal objects. We have objects that come equipped with canonical maps which are preserved by composition with other maps. This is a notion stronger than what has hitherto been called weak initiality/terminality.

We see that Cλ corresponds to a fragment of 2Tλ in which Π appears only either at the outermost level or concealed within ×, +, π, 1, μ or ν. This fragment already contains the functorial strengths ⟨A⟩ without explicit recursion; instances of this phenomenon (e.g. **listrec**) have been noted before [Reynolds 2]. It is a consequence of the fact that universal constructions are automatically functorial. The implicit recursion in the functorial types gives rise to implicitly recursive values.

We have type inference for this fragment, because if we simply omit the Π quantifiers at the outermost level we are left with a free algebraic theory of types, generated by the binary operation → and the user named type constructors. The unification theorem asserts that in a free algebraic theory (in the sense of Lawvere, i.e. a category with finite products in which all the objects are finite powers of a basic object) any finite diagram having a cone has a limiting cone. A value expression in the

fragment we are considering gives rise to such a finite diagram, with vertices representing type constraints between subexpressions, and arrows representing substitutions of expressions for variables. A typing corresponds to a cone on the diagram, and a principle type scheme to a limiting cone. This is really a consequence of "taming" quantification by adjoining constant type constructors to replace certain patterns of occurrence of Π anywhere but at the outermost level.

I would like to thank Andy Pitts for much useful advice at Sussex, and Anders Kock for the same at Aarhus. Martin Hyland convinced me at Trondheim of the usefulness of PER models.

References

I am grateful to the referee for pointing out references to the work of Mendler and Constable.

K.B.Bruce, A.R.Meyer (1984)
The semantics of second order polymorphic lambda-calculus.
SLNCS 173 pp 131-144

J.Bell (1988)
Toposes and Local set Theories.
Oxford University Press

T. Coquand, V.Breazu-Tannen (1988)
Extensional Models of Polymorphism.
T.C.S. 59 pp 85-114

Jon Fairbairn (1985)
Design and Implementation of a simple typed language based on the lambda-calculus.
University of Cambridge
Technical Report No. 75

T. Hagino (1987)
A Typed Lambda Calculus with Categorical Type Constructors.
Preprint.

T. Hagino (1987)
A Categorical Programming Language.
Thesis. Edinburgh University.

D.B.MacQueen, R.Sethi, G.Plotkin (1984)
An Ideal Model for Recursive Polymorphic Types.
11-th Annual ACM Symposium on the Principles of Programming Languages.

P. Mendler (1987)
Inductive Definitions in type Theory.
Thesis - Cornell University.

Mendler, Constable (1985)
Recursive Definitions in Type Theory.
LNCS 193 pp 61-78

J.C.Reynolds (1984)
Polymorphism is not Set-Theoretic.
SLNCS 173 pp 145-156

J.C.Reynolds (1985)
Three approaches to type structure.
TAPSOFT. SLNCS 185 pp 97-138

A.Pitts (1987)
Polymorphism is Set-Theoretic Constructively.
Preprint. University of Sussex.