Refining algebraic data types

Andrei Lapets [*]

December 29, 2006

Abstract

Our purpose is to formalize two potential refinements of single-sorted algebraic data types – subalgebras and algebras which satisfy equivalence relations – by considering their categorical interpretation. We review the usual categorical formalization of single- and multi-sorted algebraic data types as initial algebras, and the correspondence between algebraic data types and endofunctors. We introduce a relation on endofunctors which corresponds to a subtyping relation on algebraic data types, and partially extend this relation to multi-sorted algebras. Finally, we explore how this relation can help us understand single-sorted algebraic data types which satisfy equivalence relations.

1 Algebraic Data Types as Initial Algebras

In programming languages with type systems, a type usually represents a family of values, such as the space of 32-bit integers: $\texttt{Int32} \cong \mathbb{Z}/2^{32}\mathbb{Z}$. Some programming languages (popular examples include Haskell [Je99] and ML [MTHM97]) have type systems which allow a programmer to define her own families of values by means of a recursion equation. For example, the data type representing inductively constructed binary trees with integer values at the leaves can be defined by the equation

$$\texttt{IntTree} = \texttt{Int32} \uplus (\texttt{IntTree} \times \texttt{IntTree}).$$

Typically, names must be assigned to the functions which allow the programmer to construct values of a data type. The diagram below illustrates this particular example:

$$\begin{tikzcd}
\texttt{Int32} \arrow[dr, "\texttt{Leaf}"'] & & \texttt{IntTree} \times \texttt{IntTree} \arrow[dl, "\texttt{Node}"] \\
& \texttt{IntTree} &
\end{tikzcd}$$

The two functions $\texttt{Leaf} : \texttt{Int32} \to \texttt{IntTree}$ and $\texttt{Node} : \texttt{IntTree} \times \texttt{IntTree} \to \texttt{IntTree}$ can effectively be viewed as operators in an algebra which has no axioms. Together, they induce an isomorphism

---

[*] al@eecs.harvard.edu

$$[\texttt{Leaf}, \texttt{Node}] : \texttt{Int32} \uplus (\texttt{IntTree} \times \texttt{IntTree}) \to \texttt{IntTree}.$$

In general, if an algebraic data type has k such constructor functions, we will denote them $\omega_1, \ldots, \omega_k$. We will call the collection $\Omega = \{\omega_1, \ldots, \omega_k\}$ the signature of an algebra. Now, if we assume that the collection of types in a hypothetical programming language is a category $\mathcal{K}$, any recursion equation can be converted to an open form to obtain an endofunctor $F : \mathcal{K} \to \mathcal{K}$. In our example, F is then defined to be

$$\begin{aligned} F(X) &= \texttt{Int32} \uplus (X \times X) \\ F(f) &= [1_{\texttt{Int32}}, f \times f], \end{aligned}$$

where $f \in \mathcal{K}(A, B)$ for any $A, B \in \mathrm{Obj}(\mathcal{K})$. Note that if there is a corresponding signature Ω, we can decompose the functor into a disjoint sum of the domains of each of the operators in Ω:

$$F(X) = \biguplus_{\omega_i \in \Omega} F_{\omega_i}(X),$$

where $\forall i$, $\mathrm{dom}(\omega_i) = F_{\omega_i}(X)$. In this case, the operators in Ω induce an arrow $[\omega_1, \ldots, \omega_n] : F(X) \to X$. This suggests a definition for an algebra.

Definition 1.1 Given an endofunctor $F : \mathcal{K} \to \mathcal{K}$, an object $A \in \mathcal{K}$ is an F-algebra if there exists an arrow $a : F(A) \to A$ in $\mathcal{K}(F(A), A)$ [Pie91].

We can now see that the object $\texttt{IntTree} \in \mathrm{Obj}(\mathcal{K})$ is indeed an F-algebra, since there exists an arrow $[\texttt{Leaf}, \texttt{Node}] : F(\texttt{IntTree}) \to \texttt{IntTree}$. The collection of F-algebras forms a category, with homomorphisms between these algebras acting as arrows.

Definition 1.2 Given an endofunctor $F : \mathcal{K} \to \mathcal{K}$ and F-algebras A and B, an arrow $h \in \mathcal{K}(A, B)$ is an F-homomorphism if the following diagram commutes [Pie91]:

$$\begin{tikzcd} F(A) \arrow[r, "F(h)"] \arrow[d, "a"'] & F(B) \arrow[d, "b"] \\ A \arrow[r, "h"'] & B \end{tikzcd}$$

Proposition 1.3 Given an endofunctor $F : \mathcal{K} \to \mathcal{K}$, let A be the initial object in the category of F-algebras with a corresponding arrow $a : F(A) \to A$. It is the case that a is an isomorphism, which means that

$$F(A) \cong A.$$

Proof. Because F is a functor, there exists an arrow $F(a) : F(F(A)) \to F(A)$, which means $F(A)$ is also an F-algebra, and there exists an F-homomorphism a^{-1} such that the following diagram commutes:

$$\begin{tikzcd}
F(A) \arrow[r, "F(a^{-1})"] \arrow[d, "a"'] & F(F(A)) \arrow[d, "F(a)"] \\
A \arrow[r, "a^{-1}"'] & F(A)
\end{tikzcd}$$

Thus, a is an isomorphism. □

The object A in the above proposition is the fixed point of F, up to isomorphism, and we denote this object $\mathrm{Fix}(F)$. In our example, `IntTree` is, by definition, the fixed point of the functor F, and in practice, $[\texttt{Leaf}, \texttt{Node}]$ is automatically defined to be an isomorphism.

We henceforward assume that there exists a closed cartesian category $\mathcal{K}$ that represents the collection of types in a hypothetical programming language, and that every algebraic data type must correspond uniquely, up to isomorphism, to the fixed point of a functor. Thus, in defining an algebraic data type[1], the programmer is allowed to specify any endofunctor [2] over $\mathcal{K}$ that has a fixed point.

2 A Relation on Algebras

While the ability to define initial algebras allows programmers to define a large variety of algebraic data types, it does not allow them to capture a natural relationship between an algebra and its subalgebras. We can formalize this relationship by providing a relation on functors which correspond to single-sorted algebraic data types.

Definition 2.1 Given two endofunctors $F : \mathcal{K} \to \mathcal{K}$ and $G : \mathcal{K} \to \mathcal{K}$, we say that $G \preceq F$ if for any $A \in \mathrm{Obj}(\mathcal{K})$ such that there exists an arrow $a : F(A) \to A$, there exists an arrow $i : G(A) \to F(A)$:

$$\begin{tikzcd}
F(A) \arrow[d, "a"] & G(A) \arrow[l, "i"] \arrow[dl, dotted, "a \circ i"] \\
A &
\end{tikzcd}$$

---

[1] For simplicity, we restrict ourselves to monomorphic, single-sorted algebraic data types.

[2] In practice, only endofunctors defined using finite expressions consisting of sums, products, and exponents of elements in $\mathcal{K}$ will be encountered, but our analysis applies to all endofunctors.

Proposition 2.2 The relation $\preceq$ is a partial order.

Proof. It is obviously reflexive. If $H \preceq G$ and $G \preceq F$, we then know that the following diagram commutes for any $A \in \mathrm{Obj}(\mathcal{K})$,

$$\begin{tikzcd}
F(A) \arrow[d] & G(A) \arrow[l] \arrow[dl] & H(A) \arrow[l] \arrow[dll, bend left] \\
A & &
\end{tikzcd}$$

so $H \preceq F$, and it is indeed transitive. To see that it is antisymmetric, notice that $G \preceq F$ and $F \preceq G$ if and only if for every object A, $F(A) \cong G(A)$. $\square$

The relation $\preceq$ on functors induces a natural subtyping relation on initial algebras (and thus, algebraic data types).

Proposition 2.3 If $G \preceq F$, then there exists a G-homomorphism $i : \mathrm{Fix}(G) \to \mathrm{Fix}(F)$.

Proof. Note that $G \preceq F$ implies that every F-algebra is also a G-algebra, and $\mathrm{Fix}(F)$ is an F-algebra, so $\mathrm{Fix}(F)$ is also a G-algebra. Because $\mathrm{Fix}(G)$ is initial in the category of G-algebras, there must exist a G-homomorphism $i : \mathrm{Fix}(G) \to \mathrm{Fix}(F)$. $\square$

Consequently, given any catamorphism f with domain $\mathrm{Fix}(F)$, we can construct a catamorphism $f \circ i$ with domain $\mathrm{Fix}(G)$. In a type system, the relation $\preceq$ can be used in a formal typing rule corresponding to subsumption.

There are a variety of ways to employ such a relation in formalizing useful constructs. For example, given two subalgebras, we can formalize the notion of a maximum subalgebra which is contained in both. Let $\mathcal{E}$ be the category of endofunctors of $\mathcal{K}$; for any two endofunctors $F, G \in \mathrm{Obj}(\mathcal{E})$, there exists an arrow $\eta : G \to F$ if and only if $G \preceq F$. Since $\preceq$ is a partial order, this is indeed a category.

Definition 2.4 Let G_1, G_2 and F be functors such that $G_1 \preceq F$ and $G_2 \preceq F$. If there exists in $\mathcal{E}$ a pullback G' (whose existence ensures that the diagram

$$\begin{tikzcd}
G' \arrow[r] \arrow[d] \arrow[dr, dotted] & G_2 \arrow[d] \\
G_1 \arrow[r] & F
\end{tikzcd}$$

commutes), we call G' the intersection of G_1 and G_2, which we can denote $G_1 \cap G_2$.

If such a functor $G_1 \cap G_2$ indeed exists, it is clear by Proposition 2.3 that there are functions $i_1 : \mathrm{Fix}(G_1 \cap G_2) \to \mathrm{Fix}(G_1)$ and $i_2 : \mathrm{Fix}(G_1 \cap G_2) \to \mathrm{Fix}(G_2)$. Furthermore, because $G_1 \cap G_2$ is a pullback, for any other G'' for which there exist such functions i_1' and i_2', there must exist a function $i' : \mathrm{Fix}(G'') \to \mathrm{Fix}(G_1 \cap G_2)$.

3 Multi-sorted Algebras

So far, we have only provided a correspondence between functors and single-sorted algebraic data types. However, in many type systems, it is possible to define a collection of mutually recursive algebraic data types. A simple generalization of the definition of F-algebras is sufficient to formalize the structure of such data types.

Definition 3.5 Let $\overline{F} = \{F_1, \ldots, F_n\}$ be a collection of functors where $\forall i$, $F_i : \mathcal{K}^n \to \mathcal{K}$. We call a collection of objects $\overline{A} = \{A_1, \ldots A_n\} \subset \mathrm{Obj}(\mathcal{K})$ an $\overline{F}$-algebra if for all i, there exist arrows a_i such that the diagram

$$\begin{tikzcd}
F_i(A_1, \ldots, A_n) \arrow[d, "a_i"] \\
A_i
\end{tikzcd}$$

commutes.

An $\overline{F}$-homomorphism can be defined as a collection of homomorphisms $\{h_1, \ldots, h_n\}$ by extending the definition of an F-homomorphism in an analogous manner to n separate cases. Furthermore, we can extend Proposition 1.3 and thus the notion of a fixed point to $\overline{F}$ – we reuse our notation, and denote this collection of objects $\mathrm{Fix}(\overline{F})$.

Proposition 3.6 Assume we have a collection of endofunctors $\overline{F} = \{F_1, \ldots, F_n\}$ where $\forall i$, $F_i : \mathcal{K}^n \to \mathcal{K}$. Let $\overline{A} = \{A_1, \ldots A_n\}$ be a collection of objects which forms an $\overline{F}$-algebra with corresponding arrows $a_1, \ldots, a_n$ such that for any other $\overline{F}$-algebra $\overline{B} = \{B_1, \ldots B_n\}$, there exists an $\overline{F}$-homomorphism $\{h_1, \ldots, h_n\}$ with an arrow $h_i : A_i \to B_i$ for all i. It is then the case that all of the a_i are isomorphisms, which means that for all i,

$$F_i(A_1, \ldots, A_n) \cong A_i.$$

Proof. There exists for each F_i an arrow $F_i(a_1, \ldots, a_n) : F_i(F_1(A_1, \ldots A_n), \ldots, F_n(A_1, \ldots A_n)) \to F_i(A_1, \ldots A_n)$ because F_i is a functor, which means the set of objects $\{F_1(A_1, \ldots A_n), \ldots, F_n(A_1, \ldots A_n)\}$ is also an $\overline{F}$-algebra, and there exists an $\overline{F}$-homomorphism with an arrow a_i^{-1} for all i such that the following diagram

$$\begin{tikzcd}
F_i(A_1, \ldots A_n) \arrow[r, "F_i(a_1^{-1}, \ldots, a_n^{-1})"] \arrow[d, "a_i"'] & F_i(F_1(A_1, \ldots A_n), \ldots, F_n(A_1, \ldots A_n)) \arrow[d, "F_i(a_1, \ldots, a_n)"] \\
A_i \arrow[r, "a_i^{-1}"'] & F_i(A_1, \ldots A_n)
\end{tikzcd}$$

commutes for every i. Thus, for all i, a_i is an isomorphism. □

Intuitively, a single-sorted algebra was modelled as a solution of a recursion equation in a single variable, and $\mathrm{Fix}(\overline{F})$ is modelled as a set of solutions of n equations, each in up to n variables.

While it is possible to extend our relation $\preceq$ to all $\overline{F}$-algebras, a more simple extension will suffice for the purposes of our discussion. This is because we are interested only in those $\overline{F}$-algebras which are subalgebras of single-sorted algebras.

Definition 3.7 Given a collection of n functors $\overline{G}$, we say that $\overline{G} \preceq F$ if for any $A \in \mathrm{Obj}(A)$ and corresponding arrow $a : F(A) \to A$, there exists for each i an arrow γ_i such that the following diagram commutes:

$$\begin{tikzcd}
F(A) \arrow[d, "a"] & G_i(\overbrace{A, \ldots, A}^{n}) \arrow[l, "\gamma_i"] \arrow[dl, dotted, "a \circ \gamma_i"] \\
A &
\end{tikzcd}$$

Note that we can extend Proposition 2.3 to this case as well: $\overline{G} \preceq F$ implies the existence of a collection of functions $\{\gamma_1, \ldots, \gamma_n\}$ where for all $A_i \in \mathrm{Fix}(\overline{G})$, there is some $\gamma_i : A_i \to \mathrm{Fix}(F)$.

4 Algebras with Equivalence Relations

The usual notion of a coequalizer can be utilized to specify a single-sorted algebra which satisfies some equivalence relation (typically, a set of axioms).

Definition 4.1 Given an endofunctor $F : \mathcal{K} \to \mathcal{K}$ which has a fixed point $\mathrm{Fix}(F)$ and a binary equivalence relation $R \subset \mathrm{Fix}(F) \times \mathrm{Fix}(F)$, an object $A \in \mathrm{Obj}(F)$ is an (F, R)-algebra if there exists an arrow $a : F(A) \to A$ such that given the two projection functions π_1 and π_2, $a \circ \pi_1 = a \circ \pi_2$, so that the following diagram commutes:

$$\begin{tikzcd}
R \arrow[r, bend left, "\pi_2"] \arrow[r, bend right, "\pi_1"'] & F(A) \arrow[r, "a"] & A
\end{tikzcd}$$

The category of (F, R)-algebras is a subcategory of the category of F-algebras. The initial object in the category of (F, R)-algebras, which we denote F/R if it exists, would then be exactly the algebra we want. Because any (F, R)-algebra A is also an F-algebra, we know there exists an F-homomorphism $\phi : \mathrm{Fix}(F) \to A$. In the case where $A = F/R$, the function ϕ takes an element in $\mathrm{Fix}(F)$ and chooses an explicit representation of its equivalence class in F/R. It can be called an interpretation of $\mathrm{Fix}(F)$. However, this alone is not so useful in practice, because there still needs to be a concrete representation of every element in the algebra.

To what sort of concrete representations are we limited? We can observe that for every equivalence class y in an (F, R)-algebra, there exists at least one $x \in \phi^{-1}(y) \subset \mathrm{Fix}(F)$ which can act as a representative of that class. Thus, one natural way to obtain a restriction is to state that we are limited to representations which are themselves fixed points of collections of functors, as this would require no extensions to our hypothetical programming language.

Definition 4.2 Given an endofunctor $F : \mathcal{K} \to \mathcal{K}$ and a collection of endofunctors $\overline{G}$ such that $\overline{G} \preceq F$, we say that a collection of possibly partial functions $\{\phi_1, \ldots, \phi_k\}$ is an interpretation of $\mathrm{Fix}(F)$ if we have $\forall i$, $\phi_i : \mathrm{Fix}(F) \to A_j$ for some $A_j \in \mathrm{Fix}(\overline{G})$, and if $[\gamma_{j_1} \circ \phi_1, \ldots, \gamma_{j_k} \circ \phi_k] : \mathrm{Fix}(F) \to \mathrm{Fix}(F)$ is a total function on $\mathrm{Fix}(F)$.

We can now express a restriction on the set of equivalence relations a programmer should be allowed to specify if such a feature is to be included in a programming language. Given a functor F, it must be the case that for any relation $R \subset \mathrm{Fix}(F) \times \mathrm{Fix}(F)$ which can be specified, there must exist: (i) some collection $\overline{G} \preceq F$ of functors such that there exists some fixed point $\mathrm{Fix}(\overline{G})$ with the usual inclusion functions $\gamma_i : A_i \to \mathrm{Fix}(F)$ for $A_i \in \mathrm{Fix}(\overline{G})$, (ii) some collection $\overline{\phi} = \{\phi_1, \ldots, \phi_k\}$ of arrows which constitute an interpretation of $\mathrm{Fix}(F)$, and (iii) an initial (F, R)-algebra A with F-homomorphism h from $\mathrm{Fix}(F)$ to A such that the following diagram commutes:

$$\begin{tikzcd}
 & R \arrow[d, shift right, "\pi_1"'] \arrow[d, shift left, "\pi_2"] \\
F(\mathrm{Fix}(F)) \arrow[r, "F(h)"] \arrow[d, "1_{\mathrm{Fix}(F)}"] & F(A) \arrow[d, "a"] \\
\mathrm{Fix}(F) \arrow[r, bend left, "h"] \arrow[r, bend right, "h\circ[\gamma_{j_1}\circ\phi_1,\ldots,\gamma_{j_k}\circ\phi_k]"'] & A
\end{tikzcd}$$

We call an equivalence relation which satisfies these conditions a valid equivalence relation. Note that the programmer need not be able to specify every valid relation R; a language designer would only need to ensure that every relation which can be specified is indeed valid.

5 Overview and Future Work

We have demonstrated how simple concepts from category theory can help define a subtyping relation on single-sorted algebraic data types, and by extending it partially have also hinted at how it can be extended to multi-sorted algebras in general. One natural next step is to actually define the full extension, and formalize unions and intersection of multi-sorted algebras using pullbacks and pushouts. A formalization of algebras defined in terms of their own subalgebras may also inform categorical models of generalized algebraic data types.

We have also shown how this relation can be used to better understand how algebraic data types which satisfy an equivalence relation might look in a programming language. It is possible that by extending the relation on functors to collections of functors and treating fixed points of collections of functors as objects in some category, we can obtain a much cleaner condition on equivalence relations which does not require the notion of an interpretation. Also, given a valid relation R and a functor F, it would ideally be useful to be able to find a functor or collection of functors whose fixed point would be *minimal* with respect to the set of distinct elements in F/R. Likewise, given an algebra and all of its valid subalgebras, it would be immensely useful in the design of a language to be able to specify inductively a set of valid equivalence relations which can be defined on that algebra.

References

[Je99] Simon Peyton Jones and John Hughes (editors). Haskell 98: A non-strict, purely functional language. Technical report, February 1999.

[Mis] Michael Mislove. An introduction to domain theory – notes for a short course. available at: `http://www.dimi.uniud.it/lenisa/notes.pdf`, 2003.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997. MIL r 97:1 1.Ex.

[Pie91] Benjamin C. Pierce. *Basic category theory for computer scientists*. MIT Press, Cambridge, MA, USA, 1991.

[ST97] Donald Sannella and Andrzej Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing*, 9(3):229–269, 1997.

[ST99] Donald Sannella and Andrzei Tarlecki. Algebraic methods for specification and formal development of programs. *ACM Comput. Surv.*, 31(3es):10, 1999.