Algebra of Programming
Lecture notes

Prof. Dr. Stefan Milius

Leon Vatthauer

March 22, 2024

Contents

1 Introduction

This is a summary of the course "Algebra des Programmierens" taught by Prof. Dr. Stefan Milius in the winter term 2023/2024 at the FAU [1]. The course is based on [2] with [1] as a reference for category theory.

Goal of the course is to develop a mathematical theory for semantics of data types and their accompanying proof principles. The chosen environment is the field of category theory.

1.1 Functions

A function $f : X \to Y$ is a mapping from the set X (the domain of f) to the set Y (the codomain of f). More concretely f is a relation $f \subseteq X \times Y$ which is

- *left-total*, i.e. for all $x \in X$ exists some $y \in Y$ such that $(x, y) \in f$;
- *right-unique*, i.e. any $(x, y), (x, y') \in f$ imply $y = y'$.

Often, one is also interested in the symmetrical properties, a function is called

- *injective* or *left-unique* if for every $x, x' \in X$ the implication $f(x) = f(x') \to x = x'$ holds;
- *surjective* or *right-total* if for every $y \in Y$ there exists an $x \in X$ such that $f(x) = y$;
- *bijective* if it is injective and surjective.

Example 1.1.
1. The identity function $id_A : A \to A$, $id_A(x) = x$
2. The constant function $b! : A \to B$ for $b \in B$ defined by $b!(x) = b$
3. The inclusion function $i_A : A \to B$ for $A \subseteq B$ defined by $i_A(x) = x$
4. Constants $b : 1 \to B$, where $1 := *$. The function b is in bijection with the set B.
5. Composition of function $f : A \to B, g : B \to C$ called $g \circ f : A \to C$ defined by $(g \circ f)(x) = g(f(x))$.
6. The empty function ¡ $: \emptyset \to B$
7. The singleton function $! : A \to 1$

1.2 Data Types

Programs work with data that should ideally be organized in a useful manner. A useful representation for data in functional programming is by means of *algebraic data types*. Some basic data types (written in Haskell notation) are

```
data Bool = True | False
data Nat  = Zero | Succ Nat
```

These data types are declared by means of constructors, yielding concrete descriptions how inhabitants of these types are created. *Parametric data types* are additionally parametrized by another data type, e.g.

```
data Maybe a    = Nothing | Just a
data Either a b = Left a  | Right b
data List a     = Nil     | Cons a (List a)
```

[1] Friedrich-Alexander-Universität Erlangen-Nürnberg

Such data types (parametric or non-parametric) usually come with a principle for defining functions called recursion and in richer type systems (e.g. in a dependently typed setting) with a principle for proving facts about recursive functions called induction. Equivalently, every function defined by recursion can be defined via a *fold*-function which satisfies an identity and fusion law, which replace the induction principle. Let us now consider two examples of data types and illustrate this.

1.2.1 Natural Numbers

The type of natural numbers comes with a fold function $foldn : C \to (Nat \to C) \to Nat \to C$ for every C, defined by

$$\begin{aligned} foldn\ c\ h\ zero &= c \\ foldn\ c\ h\ (suc\ n) &= h\ (foldn\ c\ h\ n) \end{aligned}$$

Example 1.2. Let us now consider some functions defined in terms of $foldn$.

- $iszero : Nat \to Bool$ is defined by

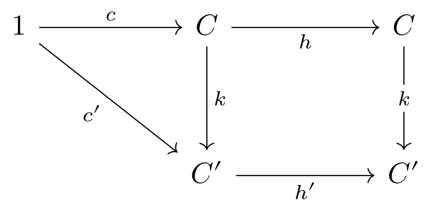
$$iszero = foldn\ true\ false!$$

- $plus : Nat \to Nat \to Nat$ is defined by

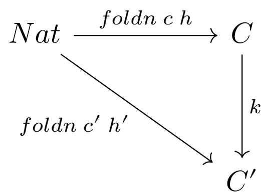
$$plus = foldn\ id(\lambda f\ n.succ(f\ n))$$

Proposition 1.3. *$foldn$ satisfies the following two rules*

1. ***Identity**: $foldn\ zero\ succ = id_{Nat}$*
2. ***Fusion**: for all $c : C$, $h, h' : Nat \to C$ and $k : C \to C'$ with $kc = c'$ and $kh = h'k$ follows $k \circ foldn\ c\ h = foldn\ c'\ h'$, or diagrammatically:*



implies



Proof. Both follow by induction over an argument $n : Nat$:

1. **Identity**:

   Case 1. $n = zero$

$$foldn\ zero\ succ\ zero = zero = id\ zero$$

   Case 2. $n = succ\ m$

$$\begin{aligned} foldn\ zero\ succ\ (succ\ m) &= succ(foldn\ zero\ succ\ m) \\ &= succ\ m && \text{(IH)} \\ &= id(succ\ m) \end{aligned}$$

2. **Fusion**:

   Case 1. $n = zero$

$$k(foldn\ c\ h\ zero) = k\ c = c' = foldn\ c'\ h'\ zero$$

   Case 2. $n = succ\ m$

$$\begin{aligned} k(foldn\ c\ h\ (succ\ m)) &= k(h(foldn\ c\ h\ m)) \\ &= h'(k(foldn\ c\ h\ m)) \\ &= h'(foldn\ c'\ h'\ m) && \text{(IH)} \\ &= foldn\ c'\ h'\ (succ\ m) \end{aligned}$$

□

Example 1.4. The identity and fusion laws can in turn be used to prove the following induction principle:

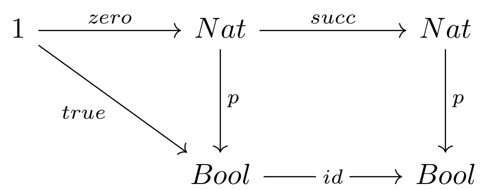
For any predicate $p : Nat \to Bool$,

1. $p\ zero = true$ and
2. $p \circ succ = p$

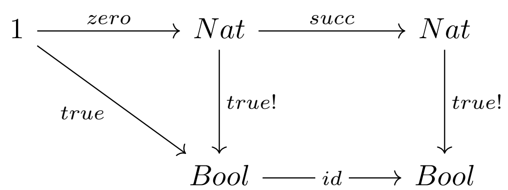
implies $p = true!$. This follows by

$$\begin{aligned} & p \\ =\ & p \circ (foldn\ zero\ succ) && (\textbf{Identity}) \\ =\ & foldn\ true\ id && (\textbf{Fusion}) \\ =\ & true! \circ (foldn\ zero\ succ) && (\textbf{Fusion}) \\ =\ & true!. && (\textbf{Identity}) \end{aligned}$$

Where the first application of **Fusion** is justified, since the diagram



commutes by the requisite properties of p, and the second application of **Fusion** is justified, since the diagram



trivially commutes.

1.2.2 Lists

2 Category Theory

2.1 Special Objects

2.2 Duality

2.3 Functors

2.4 Natural Transformations

2.5 Functor Algebras

2.6 Functor Coalgebras

2.7 (co)Limits

3 Constructions

3.1 CPO

3.2 Initial Algebra Construction

3.3 Terminal Coalgebra Construction

References

[1] J. Adámek, H. Herrlich, and G. Strecker, *Abstract and concrete categories.* Wiley-Interscience, 1990.

[2] E. Poll and S. Thompson, 'Algebra of programming by richard bird and oege de moor, prentice hall, 1996 (dated 1997).,' *Journal of Functional Programming*, vol. 9, no. 3, pp. 347–354, 1999.