2.2.2 The category corresponding to a functional programming language A functional programming language has:

FPL–1 Primitive data types, given in the language.

FPL–2 Constants of each type.

FPL–3 Operations, which are functions between the types.

FPL–4 Constructors, which can be applied to data types and operations to produce derived data types and operations of the language.

The language consists of the set of all operations and types derivable from the primitive data types and primitive operations. The word 'primitive' means given in the definition of the language rather than constructed by a constructor. Some authors use the word 'constructor' for the primitive operations.

2.2.3 If we make two assumptions about a functional programming language and one innocuous change, we can see directly that a functional programming language L corresponds in a canonical way to a category $C(L)$.

A–1 We must assume that there is a do-nothing operation id_A for each type A (primitive and constructed). When applied, it does nothing to the data.

A–2 We add to the language an additional type called 1, which has the property that from every type A there is a unique operation to 1. We interpret each constant c of type A as an arrow $c : 1 \longrightarrow A$. This incorporates the constants into the set of operations; they no longer appear as separate data.

A–3 We assume the language has a composition constructor: take an operation f that takes something of type A as input and produces something of type B, and another operation g that has input of type B and output of type C; then doing one after the other is a derived operation (or program) typically denoted $f; g$, which has input of type A and output of type C.

Functional programming languages generally have do-nothing operations and composition constructors, so A–1 and A–3 fit the concept as it appears in the literature. The language resulting from the change in A–2 is operationally equivalent to the original language.

Composition must be associative in the sense that, if either of $(f; g); h$ or $f; (g; h)$ is defined, then so is the other and they are the same operation. We

must also require, for $f : A \longrightarrow B$, that $f; \mathrm{id}_B$ and $\mathrm{id}_A; f$ are defined and are the same operation as f. That is, we impose the **equations** $f; \mathrm{id}_B = f$ and $\mathrm{id}_A; f = f$ on the language. Both these requirements are reasonable in that in any implementation, the two operations required to be the same would surely do the same thing.

2.2.4 Under those conditions, a functional programming language L has a category structure $C(L)$ for which:

FPC–1 The types of L are the objects of $C(L)$.

FPC–2 The operations (primitive and derived) of L are the arrows of $C(L)$.

FPC–3 The source and target of an arrow are the input and output types of the corresponding operation.

FPC–4 Composition is given by the composition constructor, written in the reverse order.

FPC–5 The identity arrows are the do-nothing operations.

The reader may wish to compare the discussion in [Pitt, 1986].

Observe that $C(L)$ is a *model* of the language, not the language itself. For example, in the category $f; \mathrm{id}_B = f$, but in the language f and $f; \mathrm{id}_B$ are different source programs. This is in contrast to the treatment of languages using context free grammars: a context free grammar generates the actual language.

2.2.5 Example As a concrete example, we will suppose we have a simple such language with three data types, `NAT` (natural numbers), `BOOLEAN` (true or false) and `CHAR` (characters). We give a description of its operations in categorical style.

(i) `NAT` should have a constant $0 : 1 \longrightarrow$ `NAT` and an operation `succ` : `NAT` $\longrightarrow$ `NAT`.

(ii) There should be two constants `true`, `false` : $1 \longrightarrow$ `BOOLEAN` and an operation $\neg$ subject to the equations $\neg \circ$ `true` = `false` and $\neg \circ$ `false` = `true`.

(iii) `CHAR` should have one constant $c : 1 \longrightarrow$ `CHAR` for each desired character c.

(iv) There should be two type conversion operations `ord` : `CHAR` $\longrightarrow$ `NAT` and `chr` : `NAT` $\longrightarrow$ `CHAR`. These are subject to the equation `chr` $\circ$ `ord` = $\mathrm{id}_{\texttt{CHAR}}$. (You can think of `chr` as operating modulo the number of characters, so that it is defined on all natural numbers.)

An example program is the arrow 'next' defined to be the composite `chr` $\circ$ `succ` $\circ$ `ord` : `CHAR` $\longrightarrow$ `CHAR`. It calculates the next character in order.

This arrow 'next' is an arrow in the category representing the language, and so is any other composite of a sequence of operations.

2.2.6 The objects of the category $C(L)$ of this language are the types `NAT`, `BOOLEAN`, `CHAR` and 1. Observe that typing is a natural part of the syntax in this approach.

The arrows of $C(L)$ consist of all programs, with two programs being identified if they must be the same because of the equations. For example, the arrow

$$\texttt{chr} \circ \texttt{succ} \circ \texttt{ord} : \texttt{CHAR} \longrightarrow \texttt{CHAR}$$

just mentioned and the arrow

$$\texttt{chr} \circ \texttt{succ} \circ \texttt{ord} \circ \texttt{chr} \circ \texttt{ord} : \texttt{CHAR} \longrightarrow \texttt{CHAR}$$

must be the same because of the equation in (iv).

Observe that `NAT` has constants $\texttt{succ} \circ \texttt{succ} \circ \ldots \circ \texttt{succ} \circ 0$ where `succ` occurs zero or more times. In the exercises, n is the constant defined by induction by $1 = \texttt{succ} \circ 0$ and $n + 1 = \texttt{succ} \circ n$.

Composition in the category is composition of programs. Note that for composition to be well defined, if two composites of primitive operations are equal, then their composites with any other program must be equal. For example, we must have

$$\texttt{ord} \circ (\texttt{chr} \circ \texttt{succ} \circ \texttt{ord}) = \texttt{ord} \circ (\texttt{chr} \circ \texttt{succ} \circ \texttt{ord} \circ \texttt{chr} \circ \texttt{ord})$$

as arrows from `CHAR` to `NAT`. This is handled systematically in 3.5.8 using the quotient construction.

This discussion is incomplete, since at this point we have no way to introduce n-ary operations for $n > 1$, nor do we have a way of specifying the flow of control. The first will be remedied in Section 5.3.14. Approaches to the second question are given in Section 5.7.6 and Section 14.2. See also [Wagner, 1986a]. Other aspects of functional programming languages are considered in 5.3.14 and 5.4.8.

2.2.7 Exercise

1. Describe how to add a predicate '`nonzero`' to the language of this section. When applied to a constant of `NAT` it should give `true` if and only if the constant is not zero.

3.5.8 The category of a programming language We described in 2.2.6 the category $C(L)$ corresponding to a simple functional programming language L defined there. We can now say precisely what $C(L)$ is.

The definition of L in 2.2.5 gives the primitive types and operations of the language. The types are the nodes and the operations are the arrows of a graph. This graph generates a free category $F(L)$, and the equations imposed in 2.2.5(ii) and (iv) (each of which says that two arrows of $C(L)$ must be equal) generate a congruence relation as just described. The resulting quotient category is precisely $C(L)$.

When one adds constructors such as record types to the language, the quotient construction is no longer enough. Then it must be done using sketches. The construction just given is in fact a special case of a model of a sketch (see Section 4.6).

3.5.9 Functorial semantics Functors provide a way to give a meaning to the constructs of the language L just mentioned. This is done by giving a functor from $C(L)$ to some category suitable for programming language semantics, such as those discussed in 2.4.3.

We illustrate this idea here using a functor to **Set** as the semantics for the language described in 2.2.5. **Set** is for many reasons unsuitable for programming language semantics, but it is the natural category for expressing our intuitive understanding of what programming language constructs mean.

Following the discussion in 2.2.5, we define a semantics functor $\Sigma : C(L) \longrightarrow \mathbf{Set}$. To do this, first we define a function F on the primitive types and operations of the language.

(i) $F(\texttt{NAT})$ is the set of natural numbers. The constant 0 is the number 0 and $F(\texttt{succ})$ is the function which adds 1.

(ii) $F(\texttt{BOOLEAN})$ is the set $\{\texttt{true}, \texttt{false}\}$. The constants `true` and `false` are the elements of the same name, and $F(\neg)$ is the function which switches true and false.

(iii) $F(\texttt{CHAR})$ is the set of 128 ASCII symbols, and each symbol is a constant.

(iv) $F(\texttt{ord})$ takes a character to its ASCII value, and $F(\texttt{chr})$ takes a number n to the character with ASCII code n modulo 128.

Let $F(L)$ be the free category generated by the graph of types and operations, as in 2.6.16. By Proposition 3.1.15, there is a functor $\widehat{F} : F(L) \longrightarrow \mathbf{Set}$ which has the effect of F on the primitive types and operations.

This functor $\widehat{F}$ has the property required by Proposition 3.5.4 that if $\sim$ is the congruence relation on $F(L)$ generated by the equations of 2.2.5(ii) and (iv), then $f \sim g$ implies that $\widehat{F}(f) = \widehat{F}(g)$ (Exercise 5). This means that there is a functor $\Sigma : C(L) \longrightarrow \mathbf{Set}$ (called F_0 in Proposition 3.5.4) with the property that if x is any primitive type or operation, then $\Sigma(x) = F(x)$.

The fact that Σ is a functor means that it preserves the meaning of programs; for example the program (path of arrows) $\texttt{chr} \circ \texttt{succ} \circ \texttt{ord}$ ought to produce the next character in order, and in fact

$$\Sigma(\texttt{chr} \circ \texttt{succ} \circ \texttt{ord})$$

does just that, as you can check. Thus it is reasonable to refer to Σ as a possible semantics of the language L.

We will return to this example in Section 4.3.12. The construction of $C(L)$ and Σ are instances of the construction of the theory of a sketch in Section 7.5.