

# Limitations of SML in Computational Category Theory

by

Tod H. Matola

B.S., Youngstown State University (1992)

Submitted to the Department of Mathematics  
in partial fulfillment of the requirements for the degree of

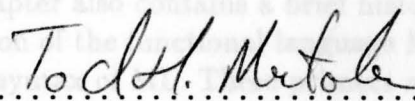
Master's of Science in Mathematics

at the

YOUNGSTOWN STATE UNIVERSITY

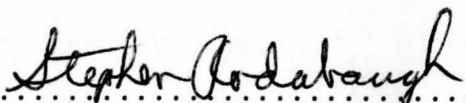
June 1994

Signature of Author .....



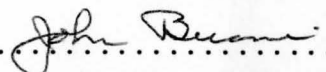
Department of Mathematics  
June 5, 1994

Certified by .....



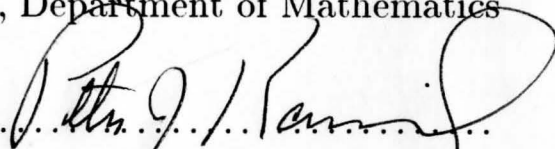
Stephen Rodabaugh Ph.D.  
Professor of Mathematics  
Thesis Supervisor

Certified by .....



John Buoni Ph.D.  
Chairman, Department of Mathematics

Accepted by .....



Peter Kasvinsky Ph.D.  
Dean, School of Graduate Studies

# Limitations of SML in Computational Category Theory

by

Tod H. Matola

## Contents

Submitted to the Department of Mathematics  
on June 5, 1994, in partial fulfillment of the  
requirements for the degree of  
Master's of Science in Mathematics

## Abstract

The main theme of this paper is to provide a basic introduction to the notion of category theory in computer science. It will include primer sections or chapters on category theory and functional programming. The category theory tutorial will discuss the basic concepts of the definition a category, and categorical limits, including the special limits (products and equalizers). Next, we have the functional programming (FP) tutorial: it contains a description of functional languages. This chapter also contains a brief history of FP. The tutorial is then further refined into description of the functional language ML and providing a basic reference of useful functions and syntax of ML. These primers and tutorials gives the necessary insight for the final illustration of a categorical proof encoded in the programming language of SML/NJ (standard meta-language of New Jersey). The illustration makes use of the constructions and definitions from the tutorial chapters, and provides a sense of cohesion to the material.

Thesis Supervisor: Stephen Rodabaugh Ph.D.

Title: Professor of Mathematics

3.2	First Approach	33
3.2.1	Universal Categorical Constructions	35
3.2.2	Limits	39
3.2.3	Limitations of the First Approach	42
3.3	Second Approach	44
3.3.1	Categorical Datatype	44
3.3.2	The SML Abstract Datatype: $\lambda$ Set	45
3.3.3	Universal Constructions	45

# Contents

3.3.4	Limits	51
3.3.5	Limitations of the Second Approach	55
A	ML Source Code	57
A.1	Auxiliary List Functions	57
A.2.1	The Complete Definition of Abstract Datatype 'a Set	59
A.2.2	FINSET Complete Definition	63
<b>1</b>	<b>Category Theory</b>	<b>1</b>
1.1	Introduction	1
1.2	Categories: the Basic Definition	2
<b>2</b>	<b>Functional Programming</b>	<b>18</b>
2.1	Introduction	18
2.2	History	20
2.3	ML Tutorial	21
2.3.1	Basic Types and Values	23
2.4	Identifiers and Bindings	27
<b>3</b>	<b>An illustration of a computational category theory in SML</b>	<b>31</b>
3.1	Introduction	31
3.2	First Approach	33
3.2.1	Universal Categorical Constructions	35
3.2.2	Limits	39
3.2.3	Limitations of the First Approach	42
3.3	Second Approach	44
3.3.1	Categorical Datatype	44
3.3.2	The SML Abstract Datatype: 'a Set	45
3.3.3	Universal Constructions	48

3.3.4	Limits . . . . .	51
3.3.5	Limitations of the Second Approach . . . . .	55
<b>A</b>	<b>ML Source Code</b>	<b>57</b>
A.1	Auxiliary List Functions . . . . .	57
A.2	Types and Functions . . . . .	59
A.2.1	The Complete Definition of Abstract Datatype 'a Set . . . . .	59
A.2.2	<b>FINSET</b> Complete Definition . . . . .	63
A.3	The Limit Code . . . . .	63
A.3.1	First Approach . . . . .	63
A.3.2	Second Approach . . . . .	68
	1-4 An example of a cone of the diagram $D$ . . . . .	12
	1-5 The limit cone of $D$ . . . . .	13
	1-6 Diagram of subcase 2.2 . . . . .	15
	1-7 Rewrite of the diagram of subcase 2.2 . . . . .	16
	1-8 $E$ the equaliser of $D$ . . . . .	17
	<b>Bibliography</b>	<b>74</b>

# List of Figures

1-1	Illustration of categorical diagrams . . . . .	8
1-2	Illustration of finite category with inclusion as non-identity arrows . . . . .	8
1-3	Illustration of equalizer candidate. . . . .	11
1-4	An example of a cone of the diagram $\mathbf{D}$ . . . . .	12
1-5	The limit cone of $\mathbf{D}$ . . . . .	13
1-6	Diagram of subcase 2.2 . . . . .	15
1-7	Rewrite of the diagram of subcase 2.2 . . . . .	16
1-8	$E$ the equalizer of $\mathbf{D}$ . . . . .	17

# Chapter 1

## Category Theory

### 1.1 Introduction

Category theory is the study of objects and morphisms, includes the algebra of functions and the objects they operate on. For example, an object of a category could be a set, a topological space, a group, or a domain or codomain of a function. A morphism of a category is a model of a changing quantity, a continuous mapping, a homomorphism, or an action translating an input to an output. So why is this study important? Category theory gives a formal mechanism of organizing the abstraction of the discipline in question (mathematics or computer science) to comprehend the discipline's content in a new light. So, we can use what we have learned in one area and apply it to the less well-known area (the one we are working with). According to [15], category theory was founded by Eilenberg and Mac Lane in 1945 as they developed algebraic topology. It has since rapidly spread throughout many areas of mathematics and is now being applied to computer science.

A category, as mentioned earlier, is a collection of objects with arrows or morphisms that act on the objects: for example, sets and functions between sets, or topological spaces and the continuous mappings between the spaces. If category theory is in essence the algebra of functions, then the primary mode of operation on the functions will have

to be composition. In other words, to be able to express a “complicated” function, with its domain and codomain in a category it should be possible to express it using only composition and simpler functions.

If we perceive a computer program as a function and view the function as being built from many smaller functions, we quickly realize that a formal mechanism for collecting, classifying, and abstracting these functions could be category-theoretic.

There are several other justifications for the use of category theory in computer science. The first is that machines are dynamic systems of finite sets of states and we have functions that take us from state to state, so what better way to organize but with an algebra. Next, we have the issue of data types and operations on and between the types, which could be viewed as a large portion of computer programming; in programming, we have programs, known as compilers, that guarantee that our programs are well-defined and will stay within the boundaries of type restrictions, as long as they follow the prescribed paths of how to get from one type to another. This type checking prevents us from making mistakes that do not make sense in the real world, so that our programs behave the way we intended. Programming is also the ability to take a real world event and translate it into an internalized version that illustrates the real event. Again, we have a potential application for category theory and its organization, and how it might benefit us in these translation problems. So we want a formal collection and comprehension of the syntax, logic and semantics of the computer filtered by category theory.

*Example 1.2.1.* The category  $\text{SET}$  is comprised of the following data:

## 1.2 Categories: the Basic Definition

1. The objects of  $\text{SET}$ , collectively denoted  $|\text{SET}|$ , forms the class of all sets. If  $X$

Let us formally present the definition of a category  $C$ :

**Definition 1.2.1.** A category  $C$  comprises the following data.

1. A collection of **objects**;  $\text{SET}(X, Y)$ , or, if  $X$  and  $Y$  are understood,  $f \in \text{SET}$ .
2. A collection of **arrows** (or **morphisms**), mappings on or actions between the objects of  $C$ ; having  $\text{codom}(f) = \text{dom}(g)$ , the morphism  $g \circ f \in \text{SET}$ . This

3. The **domain** of an arrow  $f$  is an object assigned to  $f$  that describes the source or inputs of  $f$  (often denoted  $\text{dom}(f)$  or  $f : A \rightarrow B$ ). The **codomain** of an arrow  $f$  is an object assigned to  $f$  that describes the target or outputs of  $f$  (often denoted  $\text{codom}(f)$  or  $f : A \rightarrow B$ );

4. The **composition operator** assigns two arrows  $f$  and  $g$ , with  $\text{codom}(f) = \text{dom}(g)$ , a composition arrow  $g \circ f : \text{dom}(f) \rightarrow \text{codom}(g)$ , satisfying the associative law:

$$\forall f : A \rightarrow B, g : B \rightarrow C, \text{ and } h : C \rightarrow D \text{ (with } A, B, C \text{ and } D \text{ all objects of } \mathbf{C} \text{ and not necessarily distinct), } h \circ (g \circ f) = (h \circ g) \circ f;$$

5. There is a **identity arrow** for each object  $A$ , denoted  $\text{id}_A : A \rightarrow A$ , that satisfy the identity law:

$$\forall f : A \rightarrow B, \text{id}_B \circ f = f \text{ and } f \circ \text{id}_A = f;$$

□

Now, that we have a formal definition of a category, we will cite several examples to illustrate the definition. These are not the only categories in existence just some basic comprehensible examples that are easily visualized.

**Example 1.2.1.** The category **SET** is comprised of the following data:

1. The **objects** of **SET**, collectively denoted  $|\mathbf{SET}|$ , forms the class of all sets. If  $X$  is a set, we write  $X \in |\mathbf{SET}|$ .
2. The **morphisms** of **SET** are the functions between sets. If  $f : X \rightarrow Y$  is a function, we write  $f \in \mathbf{SET}(X, Y)$ , or, if  $X$  and  $Y$  are understood,  $f \in \mathbf{SET}$ .
3. There is a binary operation, **composition**, denoted  $\circ$ , which generates from  $f, g \in \mathbf{SET}$ , having  $\text{codom}(f) = \text{dom}(g)$ , the morphism  $g \circ f \in \mathbf{SET}$ . This



morphism has domain =  $\text{dom}(f)$  and codomain =  $\text{codom}(g)$ , and is defined by:

$$(g \circ f)(x) = g(f(x)).$$

4. The objects, morphisms, and composition of **SET** satisfy the following axioms:

- (a) **Associativity:**  $\forall f, g, h \in \mathbf{SET}$  with  $\text{codom}(f) = \text{dom}(g)$  and  $\text{codom}(g) = \text{dom}(h)$ ,  $h \circ (g \circ f) = (h \circ g) \circ f$ .
- (b) **Identity morphism:**  $\forall X \in |\mathbf{SET}|$ ,  $\exists id_X \in \mathbf{SET}(X, X)$ ,  $\forall f \in \mathbf{SET}$ ,  $\text{dom}(f) = X$ ,  $\forall g \in \mathbf{SET}$ ,  $\text{codom}(g) = X$ ,  $f \circ id_X = f$  and  $id_X \circ g = g$ .

□

**Example 1.2.2.** The category **FINSET** is similar to **SET** with one major difference: **SET** is complete, **FINSET** is only finitely complete. The objects of **FINSET** are finite sets, where all other stipulations are the same as for **SET**.

□

Next, we will introduce the **partial ordered set** or **poset**, a fundamental mathematical concept that yields a category similar to the previous examples. It is a common notion throughout mathematics and logic, and thus is our next example of a category.

**Definition 1.2.2.** Let  $R$  be a binary relation on a set  $X$ . Then  $R$  is a **partial order** iff the following hold:

1.  $R$  is reflexive (i.e.,  $\forall x \in X, xRx$ );
2.  $R$  is antisymmetric (i.e.,  $\forall x, y \in X, xRy$  and  $yRx \Rightarrow x = y$ );
3.  $R$  is transitive (i.e.,  $\forall x, y, z \in X, xRy$  and  $yRz \Rightarrow xRz$ ).

Let  $R$  be a partial order, denoted  $\leq$ . Let  $X$  be a set then  $(X, R)$  or  $(X, \leq)$ , is called a partially ordered set.

□

Just as for **SET** we can define a category based on the notion of the poset, which is normally denoted **POSET** and is defined as follows.

**Example 1.2.3.** The category **POSET** is comprised of the following data:

1. The **objects** of **POSET**, collectively denoted  $|\mathbf{POSET}|$ , forms the class of all posets, and we write  $(X, \leq) \in |\mathbf{POSET}|$ .
2. The **morphisms** of **POSET** are the order preserving functions between posets. If  $f : (X, \leq) \rightarrow (Y, \leq)$  is a morphism, we write  $f \in \mathbf{POSET}((X, \leq), (Y, \leq))$ , or if  $(X, \leq)$  and  $(Y, \leq)$  are understood,  $f \in \mathbf{POSET}$ .
3. There is a **composition operation**, denoted  $\circ$ , which is inherited from **SET** (i.e., it can be shown that the composition of monotone maps is again monotone). The associativity of the composition operator is also inherited from **SET**.
4. **Identity morphism** is defined as follows:

Let  $(X, \leq)$  be a poset. Then we clearly see that  $id : X \rightarrow X$  is order preserving.

□

Now that we have the basic definition of a category, and some classic examples of categories, we will use a discrete example to further re-enforce this definition. It will actually be an object of an earlier category **FINSET**. Which will lead to the next topic-categorical diagrams and how they enrich category theory, but first the definition of the discrete category  $C$ .

**Example 1.2.4.** The category  $C$  is comprised of the following data:

1. The **objects** of  $C$ :

(a)  $X = \{ 1, 2, 3 \}$ ;

(b)  $Y = \{ 4, 5, 6 \}$ ;

(c)  $Z = \{ 7, 8, 9 \}$ ;

2. The **morphisms** of  $C$  are  $id_X$ ,  $id_Y$ , and  $id_Z$ , together with the following functions:

$f : X \rightarrow Y$  by

$$f(x) = \begin{cases} 4 & \text{if } x = 3 \\ 5 & \text{if } x = 1 \\ 6 & \text{if } x = 2 \end{cases}$$

$g : Y \rightarrow Z$  by

$$g(x) = \begin{cases} 7 & \text{if } x = 5 \\ 8 & \text{if } x = 4 \\ 9 & \text{if } x = 6 \end{cases}$$

$h : X \rightarrow Z$  by

$$h(x) = \begin{cases} 7 & \text{if } x = 1 \\ 8 & \text{if } x = 3 \\ 9 & \text{if } x = 2 \end{cases}$$

3. Note: Composition can be defined in only one way (*i.e.*, inherited from **SET**) and that  $f \circ g = h$ . Thus,  $C$  satisfies the five parts of our definition of a category.

□

WILLIAM F. WEAVER LIBRARY  
YOUNGSTOWN STATE UNIVERSITY

**Definition 1.2.3.** Let  $C$  be a category. **Diagram** is any collection of vertices and directed edges, consistently labeled with objects and morphisms of  $C$ .

□

**Definition 1.2.4.** Let  $C$  be a category. Diagram is said to **Commute** if for any two pairs of vertices  $X$  and  $Y$ , all paths in the diagram from  $X$  to  $Y$  are equal, in the sense of that each path in the the diagram determines an arrow and these arrows are equal in  $C$ .

□

Figure 1-1: Illustration of categorical diagrams

Now we turn our attention to the issue of categorical diagrams. These are representations of a category in the form of a directed graph. It is a collection of vertices, which represent the objects of the category and directed edges, which are the arrows of the category. It is important that this graph accurately represent the domains and codomains of the particular arrow, so if there is a function from  $A$  to  $B$  the graph would reflect this by representing the arrow as an edge of the graph from the node representing  $A$  to the node representing  $B$ . Figure 1-1 is an example of the diagram associated with category  $C$  defined in example 1.2.4.



Figure 1-2: Illustration of finite category with inclusion as non-identity arrows

WILLIAM F. MAAG LIBRARY  
YOUNGSTOWN STATE UNIVERSITY

### 1.3 Some Categorical Constructions

In category theory there are the categories which we have described and then there are further abstractions, objects which are made from the categories themselves. These are known as categorical constructions which are common to groups of categories or operations that take place in the whole category, and not on any particular object or arrow. These constructions are common to all categories of category theory, because they guarantee certain characteristics about the the particular category. So we might think of a categorical construction as being a function or black box.

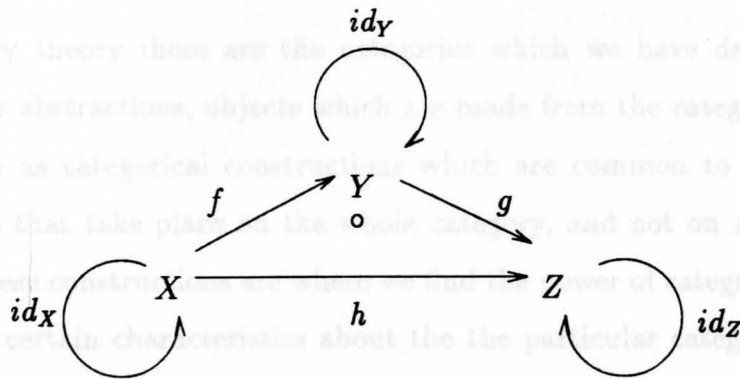
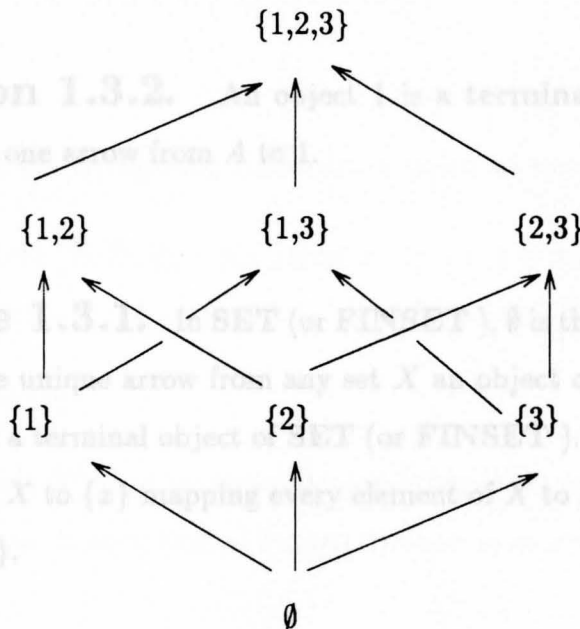


Figure 1-1: Illustration of categorical diagrams

**Definition 1.3.1.** An object  $0$  is an initial object if, for every object  $A$  there exists exactly one arrow from  $0$  to  $A$ .

**Definition 1.3.2.** An object  $1$  is a terminal object if, for every  $A$ , there exists exactly one arrow from  $A$  to  $1$ .

**Example 1.3.1.** In  $\text{SET}$  (or  $\text{FINSET}$ ),  $\emptyset$  is the only initial object. The empty function is the unique arrow from any set  $X$  to  $\emptyset$ . Each singleton set  $\{x\} \in \text{SET}$  is a terminal object of  $\text{SET}$  (or  $\text{FINSET}$ ). And, for every set  $X$ , there is a function from  $X$  to  $\{x\}$  mapping every element of  $X$  to  $x$ , and is the only total function from  $X$  to  $\{x\}$ .



Now, the next type of construction is that of a product. The notion of a product is characterized in the category of sets  $\text{SET}$  by the Cartesian product, denoted  $A \times B$ , where  $A$  and  $B$  are sets. An object of  $A \times B$  is the ordered pair  $(x, y)$ , where  $x \in A$  and  $y \in B$ .

### 1.3 Some Categorical Constructions

In category theory there are the categories which we have described and then there are further abstractions, objects which are made from the categories themselves. These are known as categorical constructions which are common to groups of categories or operations that take place on the whole category, and not on any particular object or arrow. These constructions are where we find the power of category theory, because they guarantee certain characteristics about the the particular category. So we might think of a categorical construction as being a function or black box.

There are several categorical constructions that are common to all categories, so we label them **universal constructions**. We will start with the simplest of these.

**Definition 1.3.1.** An object  $0$  is an **initial object** if, for every object  $A$  there exists exactly one arrow from  $0$  to  $A$ .

□

**Definition 1.3.2.** An object  $1$  is a **terminal object** if, for every  $A$ , there exists exactly one arrow from  $A$  to  $1$ .

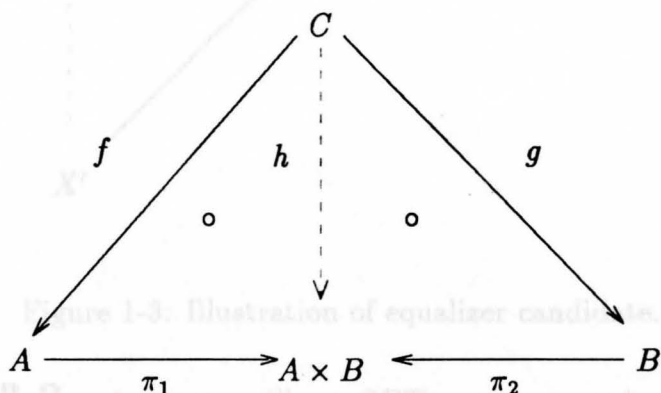
□

**Example 1.3.1.** In **SET** (or **FINSET**),  $\emptyset$  is the only initial object. The empty function is the unique arrow from any set  $X$  an object of **SET** to  $\emptyset$ . Each singleton set  $\{x\} \in \mathbf{SET}$  is a terminal object of **SET** (or **FINSET** ). And, for every set  $X$ , there is a function from  $X$  to  $\{x\}$  mapping every element of  $X$  to  $x$ , and is the only total function from  $X$  to  $\{x\}$ .

□

Now, the next type of construction is that of a **product**. The notion of a **product** is characterized in the category of **SET** by the *Cartesian product*, denoted  $A \times B$ , where  $A$  and  $B$  are sets. An object of  $A \times B$  is the ordered pair  $(x, y)$ , where  $x \in A$  and  $y \in B$ .

**Definition 1.3.3.** A product of two objects  $A$  and  $B$  is  $A \times B$  together with the two arrows  $\pi_1 : A \times B \rightarrow A$  and  $\pi_2 : A \times B \rightarrow B$  known as *projection mapping or arrows*. Such that, for any object  $C$  and pair of arrows  $f : C \rightarrow A$  and  $g : C \rightarrow B$  must be factored uniquely through  $h : C \rightarrow A \times B$  where the following diagram



**Example 1.3.2.** Again we will use SET as our example category to illustrate the categorical construction equalizer. Assume, we have two arrows  $f, g \in \text{SET}$  defined by  $f(x) = 2x$  and  $g(x) = x + 1$ . The equalizer  $e$  is defined by  $e(x) = x$  if  $f(x) = g(x)$  and  $e(x) = \perp$  otherwise. The equalizer  $e$  commutes (i.e.,  $\pi_1 \circ h = f$  or  $\pi_2 \circ h = g$ ).

$$X = \{x \in A \text{ and } f(x) = g(x)\}.$$

□

The next basic categorical construct is called an **equalizer** and is defined as follows.

**Definition 1.3.4.** An arrow  $e : X \rightarrow A$  is an **equalizer** (or **pre-equalizer**) of a pair of arrows  $f : A \rightarrow B$  and  $g : A \rightarrow B$ , where  $X \subset A$  if:

1.  $f \circ e = g \circ e$ ;
2. for any  $e' : X' \rightarrow A$  an equalizer candidate (i.e., satisfying  $f \circ e' = g \circ e'$ ),  $\exists! h : X' \rightarrow X$  such that  $e' \text{ factors through } e$ , or  $e \circ h = e'$ ;

□

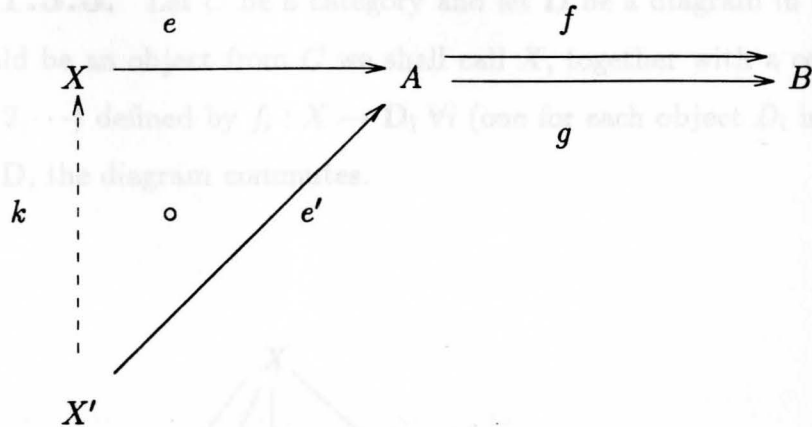


Figure 1-3: Illustration of equalizer candidate.

**Example 1.3.2.** Again we will use **SET** as our example category to illustrate the categorical construction equalizer. Assume, we have two arrows  $f, g \in \mathbf{SET}$  defined by  $f, g : A \rightarrow B$  and let  $X \subset A$  be defined by:

$$X = \{x \in A \text{ and } f(x) = g(x)\}.$$

Then the inclusion function  $e : X \rightarrow A$ , where  $x \in X$  to the same  $x$  considered as an element of  $A$ , is an equalizer of  $f$  and  $g$ .

(i.e.,  $f \circ e = g \circ e$ ) and the diagram commutes. □

Terminal objects, products and equalizers are categorical constructions. These are a special case of what is known as a categorical limit, which is the next area of discussion. We will formally describe this topic as well as give a few examples to help clarify the notion of a limit. Before, we can describe limits we need a definition of a common component of categorical diagrams, called the limiting cone.



**Definition 1.3.5.** Let  $C$  be a category and let  $D$  be a diagram in  $C$ . Then a **cone** over  $D$ , would be an object from  $C$  we shall call  $X$ , together with a collection of arrows  $\{f_i : i = 1, 2, \dots\}$  defined by  $f_i : X \rightarrow D_i \forall i$  (one for each object  $D_i$  in  $D$ ), so that for each  $g_i \in D$ , the diagram commutes.

□

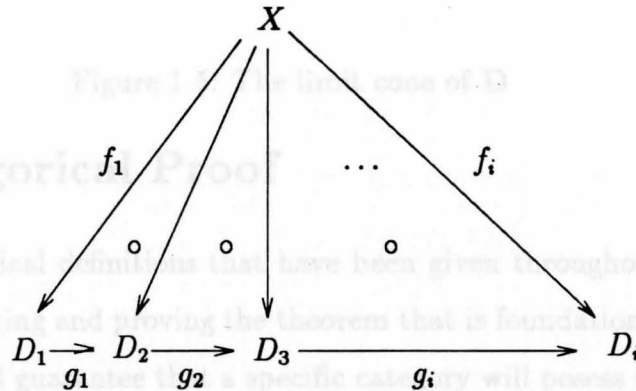


Figure 1-4: An example of a cone of the diagram  $D$

**Definition 1.3.6.** A **limit** of  $D$  is a cone  $\{f_i : X \rightarrow D_i, \forall i\}$  such that for any other cone  $\{f'_i : X' \rightarrow D_i, \forall i\}$  that  $\exists ! h : X' \rightarrow X$  so that it factors uniquely through  $h$  (i.e.,  $f'_i \circ h = f_i$ ) and the diagram commutes.

□

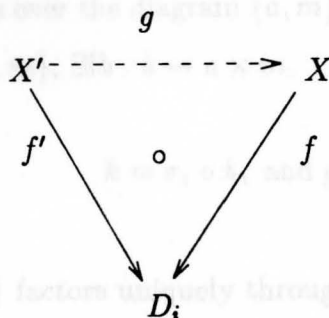


Figure 1-5: The limit cone of  $\mathbf{D}$

## 1.4 A Categorical Proof

In view of the categorical definitions that have been given throughout this chapter, we are now capable of stating and proving the theorem that is foundation of the illustration. It is a criteria that will guarantee that a specific category will possess all of its limits or it is complete. The theorem takes the form of an inductive proof about complete categories.

**Theorem 1.4.1.** If  $C$  has terminal objects, (finite) products, and equalizers, then  $C$  is (finitely) complete.

*Proof.* By induction on the number of nodes and edges.

Case 1. No nodes, and no edges:

Terminal object is the limit over this diagram.

Case 2. Let  $\mathbf{D}$  be a nonempty diagram with limit  $(a, \{f_n : n \text{ is a node in } \mathbf{D}\})$ .

Subcase 2.1. Add a node  $m$  to  $\mathbf{D}$  to get  $\mathbf{D}'$ .

Claim. The limit over  $\mathbf{D}'$  is  $(a \times n, \{f_n \circ \pi_1 : n \text{ in } \mathbf{D}\} \cup \{\pi_2\})$ .

*Proof of Claim.* No new triangles have been set up, so commutativity in  $\mathbf{D}'$  plus its limit is automatic.

Consider another cone over  $\mathbf{D}' : (b, \{g_n : n \in D\} \cup \{g_m\})$ .

1.  $(b, \{g_n : n \in D\} \cup \{g_m\})$  is a cone over  $\mathbf{D}$ . So,  $\exists! k : b \rightarrow a, \forall n \in \mathbf{D}, g_n = f_n \circ k$ .

2.  $(b, \{k, g_m\})$  is a cone over the diagram  $\{a, m\}$  (with no edges). Since  $(a \times m, \{\pi_1, \pi_2\})$  is the limit over  $\{a, m\}$ ,  $\exists! h : b \rightarrow a \times m$ ,

$$k = \pi_1 \circ h, \text{ and } g_m = \pi_2 \circ h.$$

3. Show.  $\{g'_n s\} \cup \{g_m\}$  factors uniquely through  $\{f_n \circ \pi'_1 s\} \cup \{\pi_2\}$  via  $h$ .

$\{g'_n s\}$

$$\begin{aligned} g_n &= f_n \circ k \\ &= f_n \circ (\pi_1 \circ h) \\ &= (f_n \circ \pi_1) \circ h \end{aligned}$$

$\{g_m\}$   $g_m = \pi_2 \circ h$  (from above).

Uniqueness Suppose  $\exists \bar{h} : b \rightarrow a \times m$ , so that both the following hold:

(a)  $\forall n, g_n = (f_n \circ \pi_1) \circ \bar{h}$ .

(b)  $g_m = \pi_2 \circ \bar{h}$ .

Using (a):  $\forall n, g_n = f_n \circ (\pi_1 \circ \bar{h})$  by the uniqueness of  $k$ , with respect to these factorizations,

$$k = \pi_1 \circ \bar{h}$$

Using (b), we have:  $k = \pi_1 \circ \bar{h}$  and  $g_m = \pi_2 \circ \bar{h}$ . By uniqueness of  $h$  with respect to these factorizations  $\bar{h} = h$ .

Therefore the limit over  $D'$  is verified for subcase 2.1.

Subcase 2.2. Add an edge to  $D$  to get new diagram  $D'$ .

Have  $(E, e)$  equalizer of  $g \circ f_{n_1}$  and  $f_{n_2}$ :

$$g \circ f_{n_1} \circ e = f_{n_2} \circ e \text{ and the limit.}$$

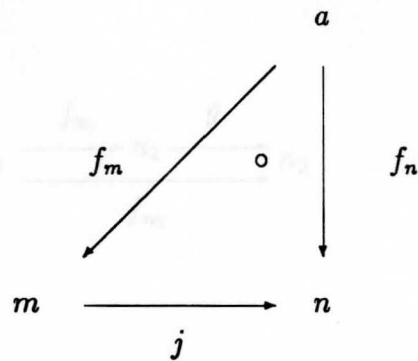


Figure 1-6: Diagram of subcase 2.2

Claim.  $(E, \{f_n \circ e : n \text{ node in } \mathbf{D}\})$  is the limit of  $D'$ .

Cone. We note there are two cases:  $\mathbf{D}$  is a cone over  $D'$ .

(i)  $j$  is already an edge.

*Proof of (i).* Let  $m$  and  $n$  be nodes with some edge  $m \rightarrow_j n$  from  $\mathbf{D}$ . Then  $f_m, f_n$  make a commutative triangle. Then

Now since  $(E, e)$  is the equalizer,  $\exists! k : E' \rightarrow E$ ,

$$\begin{aligned} k \circ e \circ h &= f_n \circ e = (j \circ f_m) \circ e \\ &= j \circ (f_m \circ e) \end{aligned}$$

Claim. Each  $g_n = (f_n \circ e) \circ h$ .

*Subproof.* Previously have

(ii)  $j = g$ .

*Proof of (ii).* Now, if  $m = n_1, n = n_2$  and  $j = g$ , then  $f_{n_2} \circ e = g \circ (f_{n_1} \circ e)$  from equalization.

Therefore  $(E, \{f_n \circ e : n \text{ node in } \mathbf{D}\})$  is a cone over  $\mathbf{D}$ .

Claim.  $(E, e)$  is a unique equalizer with respect to previous claim.

Limiting Cone. Let  $(E', \{g_n : n \text{ node in } \mathbf{D}\})$ .

Note:  $(E', \{g_n : n \text{ node in } \mathbf{D}\})$  is a cone over  $\mathbf{D}$ .

$$\exists! k : E' \rightarrow a, \forall n \in \mathbf{D}, g_n = f_n \circ k.$$

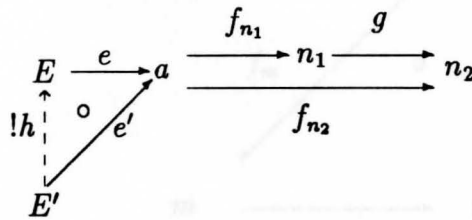


Figure 1-7: Rewrite of the diagram of subcase 2.2

Claim.  $(E', k)$  “equalizes”  $g \circ f_{n_1}$  and  $f_{n_2}$ .

*Subproof.* Since  $(E', \{g_n : n \text{ node in } \mathbf{D}\})$  is a cone over  $\mathbf{D}'$ ,

$$g_{n_2} = g \circ g_{n_1}.$$

So:

$$f_{n_2} \circ k = g \circ f_{n_1} \circ k.$$

Now since  $(E, e)$  is the equalizer,  $\exists! h : E' \rightarrow E$ ,

$$k = e \circ h.$$

Claim. Each  $g_n = (f_n \circ e) \circ h$ .

*Subproof.* Previously have:

$$g_n = f_n \circ k$$

$$f_n \circ (e \circ h)$$

$$(f_n \circ e) \circ h$$

Claim.  $h$  is unique with respect to previous claim.

*Subproof.* Let  $\hat{h} : E' \rightarrow E, \forall g_n$ ,

$$g_n = (f_n \circ e) \circ \hat{h}.$$

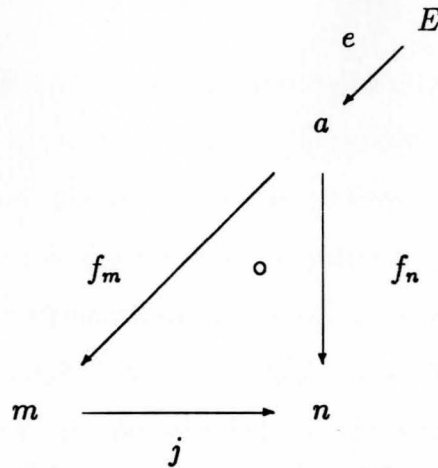


Figure 1-8:  $E$  the equalizer of  $D$ .

But, we have  $k$  from the equalization, that is unique with respect to:

$$g_n = f_n \circ k.$$

Since  $g_n = (f_n \circ e) \circ \hat{h} = f_n \circ (e \circ \hat{h})$  then

$$e \circ \hat{h} = k.$$

But  $h$  is unique with respect to factoring  $k$  through  $e$ , so

$$\hat{h} = h.$$

Therefore,  $(E, \{f_n \circ e : n \text{ node in } D\})$  is the limit of  $D'$ .

Therefore, by subcases 1 and 2, we see that if  $C$  has terminal objects, (finite) products, and equalizers, then  $C$  is (finitely) complete.

□

In conclusion, we have seen the definition of a category, several examples of categories (**FINSET** in particular), universal constructions, and a fairly technical categorical proof. Which is all of the category theory that is required for the SML illustration of categorical limits as functional programs. Now we will move to the background material of functional programming, which will lead us to the discussion of SML, the language we use in the illustration.

## Chapter 2

# Functional Programming

### 2.1 Introduction

The problem facing most modern programming languages is they are constructed with the organization of the hardware and software as the primary set of restrictions rather than the problem of representing mathematics as the major datatype. By getting away from the machine-related problems, our specification or programming language could simplify the process by which we convert a real world situation into a working piece of code. Some types of computer languages are doing just that, by attempting to make the programs take on a human-friendly form rather than a machine-friendly form. This way we could let the machine perform memory management or selection processes rather than having the programmer include all of the low level specification that many languages require. These specifications would still be required but computing techniques are becoming so advanced that the difficulties of these specifications could be mapped out during the language design phase. The language should be simple to read, so that the process of translating the real world situation into the language would be less taxing. Therefore, we want a small set of language primitives that we can build from and expand into a complex, situation-dependent set of programs. We also need a way to insure that the programs are as error free as possible, providing the user with feedback and security.

The language would also require strong development of mathematical concepts, since they are the foundations of many situations. Thus the language should build a bridge that spans the gaps by redefining how we solve problems with the computer. One attempt at such a language is the area of functional programming.

What is a functional programming language? It is an attempt computers, via the programming language, resemble human thought, with respect to the notion of functions. By the notion of functions, we are referring to the common mathematical definition of a function. So we need to allow functions to be classified as first class datatypes, allowing them to become an argument to other functions as well as accepting arguments themselves. For example, a function can be composed with another function, creating a new function, or we use auxiliary functions as simplifications of other functions. This conjecture could give rise to a computer-related definition of a function. First, a function has a domain, the set of inputs, and a range, the set of outputs. The domain and range could be viewed as the type information about the function or what type of data that is "input" to the function along with what is the type of the outputs. We would then classify the function by the type of the range or outputs.

**Example 2.1.1.** If we had the function "add" (commonly denoted by '+'), which accepts two arguments, we would then check if the arguments were of the same type or from the function's domain. Then the function would be described by a rule that would provide a mechanism for translating the arguments. From this rule, if it was defined properly (*i.e.*, closed under the operation), we could determine or infer the type of the result, or what the range of the function would be. So in our example, if we had '2 + 2' we knew that addition of two integers is guaranteed to give us an integer, we see that our '+' function would be of type "integer".

We can see several flaws in the example, which I will point out and clarify when we have better terminology. □



## 2.2 History

- 1930: Alonzo Church [3] pioneered the concept of  $\lambda$ -calculus in hopes of formalizing computers. It was an attempt at the theory of functionality and was thought to be the first functional language. It proved to be logically inconsistent and required a type mechanism to overcome these flaws. Church did however point out three important facts that are still relevant to functionality today:
  - Variable names - defined by the common mathematical terms *e.g.*,  $x$ ,  $y$
  - Terms or expressions defined by groupings or orderings of functions, *e.g.*,  $x^2$  or  $y = x + 1$
  - Functional Abstraction - defined by the application of the term or expression, *e.g.*,  $f(x) = y$
- 1958: MacCarthy [7] developed LISP Processing language (LISP) which has similarities to the  $\lambda$ -calculus described above. It also lacked a type mechanism, which brought about the dialect of SCHEME. This was an attempt to force LISP to behave as modern functional languages behave.
- 1965: P. Landin [5] developed ISWIM (If you See What I Mean), which is viewed as the origin of the ML style languages.
- 1978: J. Backus [1] defined FP: a language of combinators. It was the first time that it was possible to reason about programs or develop a consistent validation of a program with respect to logic. One major fault was the lack of variable names, which can over-complicate matters.
- 1978: R. Milner [4] introduced ML (Meta-Language) which was designed to aid in the study of programming of mathematical proofs. Its key feature was its original type system combined with abilities of ISWIM.

- 1985: D. Turner [14] proposed the Miranda programming language which uses the ML type system but terms are evaluated with *lazy* rules. This evaluation technique uses the shortest or *laziest* path for expressions.

The concept of datatype was designed to aid in the correct use of a variable; the type mechanism or system checks how a datatype is against how it is supposed to be used. This helps the person confronted with the task of repairing all the program's errors or bugs. The compiler will report any misuse-usage to the programmer by means of error messages; then it is up to the programmer to track down the bug and attempt a repair.

## 2.3 ML Tutorial

An introductory tutorial of the *functional* programming language Standard ML is the next topic of discussion, but first we give a brief overview of the outstanding features of Standard ML:

- ML is a *functional* programming language, *i.e.*, functions are first-class data objects. The principal control mechanism in ML is recursive function application.
- ML is an *interpreter based* language. Every phrase read is analyzed, compiled, and executed, and the value of the phrase is reported, together with its type, interactively.
- ML is *strongly typed*. Every legal expression has a type which is determined automatically by the compiler. Strong typing guarantees that no program can incur a type error at run time, a common source of bugs.
- ML has a *polymorphic* type system. Each legal phrase has a uniquely-determined most general typing that determines the set of contexts in which that phrase may be legally used.

- ML supports *abstract datatypes*. Abstract types are a useful mechanism for program modularization. New types are defined together with a set of functions on objects of that type. The details of the implementation are hidden from the user of the type, achieving a degree of isolation that is crucial to program maintenance.
- ML is *statically scoped*. ML resolves identifier references at compile time, leading to more modular and more efficient programs.
- ML has a type-safe *exception* mechanism. Exceptions are a useful means of handling unusual or deviant conditions arising at run-time.

Since standard ML is an interpreter based language the compiler takes the program source code and translates it interactively into machine instructions. So, that when the ML compiler is invoked it returns by saying:

```
%) sml
```

```
Standard ML of New Jersey, Version 0.93, February 15, 1993
```

```
val it = () : unit
```

```
-
```

where the “val” ML keyword says that variable ‘‘it’’ is of unit type, and ‘‘it’’ is the variable name given to the top level of the interpreter. The “-” is the ML compiler prompt waiting for input. The form of the interpreter is similar to that of a LISP interpreter using the “read-eval-print” dialogue, where we enter a expression, it is read, evaluated, and the result is printed back to the terminal, assuming all goes well.

**Example 2.3.1.** Here is a sample session with the compiler:

```
- 10 + 10;  
val it = 20 : int
```

In this example the user entered “10 + 10” and ML responded with “val it = 20 : int”, saying that we are at the outermost level of evaluation as well as evaluating and typing the expression to “20” and “: int” respectively.

□

If our expression was incorrect there are three basic types of problems that the compiler can report. They are syntax errors, type errors and run-time faults. The intention of this chapter is to introduce the reader to SML/NJ, so a full language description is not included. This means that most of the important details will be introduced and detailed explanations can be obtained from the reference materials included.

Syntax errors are when you incorrectly use a part of the SML grammar, such as misspelling a key word or leaving out punctuation. Type errors occur when the type checking system can not resolve the type a function or variable, due to incomplete input or errors in the definition. We will have more to say about types and type checking in a later section of this chapter. Finally, we have run-time faults or results that were caused by ill-defined actions, such as division by zero.

### 2.3.1 Basic Types and Values

We begin our definition of ML by analyzing the use of the term type. In ML, a type is a collection of values, where a value is an object that is manipulated by a computer or computer program. For example, we have a set of integers, a set of character strings, or boolean values. This is not an exhaustive list of basic types in ML, so we need to develop a clear description of type so that we can expand our list to include the many other types necessary for SML.

**Definition 2.3.1.** A type is a classification of objects or operations on objects, such as integers or floating point addition. They identify the properties or requirements of the objects or operators. The type is also the domain and range values of function or group of functions that map a set of one class to another class. We use the type of an expression to abstractly describe the rule that will guarantee that if an expression has a value that the type of the value will be that of the expression, or that expression are well formed. In mathematics we refer to this as the closure property.

□

As we can see this definition can not be clear cut as in the traditional sense of types in programming. ML allows for polymorphic types or “variable typed” type. By doing this we allow a greater sense of abstraction to our data, by not forcing it to be a predefined static type, but allowing a dynamic use of the type. ML also allows the user to define what they call an abstract type or a previously nonexistent type (by nonexistent type I mean it is not included in the core language definition and is supplied by the user). ML also treats functions as first class citizens, by allowing them to be treated as if they were data, assigning them types. So we see that a function can return another function as a result of some processing or modification. This is the same view mathematics has of functions. So again we see composition enter into our description of the computers language. These topics will further be described after a preliminary amount of material is covered.

We want to introduce other formally defined types of the core ML language, we will start by showing a simple example of a boolean expression:

### Example 2.3.2.

```
- not (true andalso ( false orelse true));  
val it = false : bool
```

Here we are introduced to the boolean values *true* and *false* along with their operators *andalso* and *orelse*.

This gives a simple working example of how the ML compiler handles an expression of primitive core types and operators on those types.

1. **UNIT**: this consists of a single value, denoted (). It is used whenever an expression has no interesting value, or a function is to have no argument.
2. **BOOLEAN**: ML supports a boolean type, which describes "TRUE" and "FALSE."
3. **INT**: the set of positive and negative integers as normally viewed. There are several basic operations defined  $+$ ,  $-$ ,  $*$ ,  $\text{div}$ , and  $\text{mod}$  which are the normal binary arithmetic operations which map two integers to an integer. There is also a set of relational operators  $<$ ,  $\leq$ ,  $>$ ,  $\geq$ ,  $=$ , and  $<>$  which compare two integers and return a boolean value describing the comparison.
4. **REAL**: the set of floating point numbers or numbers in a decimal notation. Each is represented by an integer followed by a decimal point with one or more following digits, or E is to denote the exponential notation. The decimal point or E can be dropped as long as one or the other is present to allow ML to distinguish between integers and reals. The conversion from one type to another must be explicit, due to the required hardware difference during processing. There are several basic operations denoted  $+$ ,  $-$ ,  $*$ ,  $/$ , and  $\text{mod}$ , which are the normal arithmetic operations which map two reals to a real. Neither  $\text{mod}$  nor  $\text{div}$  (for reals) is defined but there is normal real valued division, denoted  $/$ . There are also other functions, including  $\text{sin}$ ,  $\text{sqrt}$ , and  $\text{exp}$ .
5. **STRING**: the set of finite sequences of ascii characters. Examples include, "123", "abc" or "Thesis". Strings are written in the conventional form between double quotes. There only specific meaning to ML is to denote the user's intention. There is a built-in size function, as well as append operator, denoted  $\text{"~"}$ .

6. **LIST:** these are homogeneous or “single type” collections of any of the allowed types. This is the first occurrence of a polymorphic or “variable typed” type. This means that we can have a list of strings or a list of integers, but not a mixed list of strings and integers. There are two basic notations for lists: the conventional way and the abbreviated form. The conventional way, finding its origins in LISP, is an empty list or an expression followed by another list. We write the empty list as `nil`, and a non-empty list as `h::t`, where `h` is an expression describing the type of the list, and `t` is another list of the same type as `h`. We refer to `h` as the head of the list, `t` as the tail of the list and `::` operator as “cons” from the LISP operator. For example, `1,2,3` would be `1::2::3::nil`, or more generally, `e1 :: e2 :: ... :: nil`. This allows us to describe infinite collections without actually using the explicit description of each element. The abbreviated form uses recursion, where you define one or more base cases and one or more recursive cases. The base case should be `nil` since it functions as a delimiter for lists. Then the recursive cases constructed using the “cons” operator. We note that ML uses a compressed format for lists where `[]` represents `nil` and a list is placed in the brackets, separated by commas, such as `[1,2,3]`. It is important to note that `nil` and `::` are still the primary form.
7. **TUPLES:** this is a non-homogeneous collection of values. It is also of polymorphic type. The type of the tuple is represented as the cartesian product of the types contained in the tuple. Thus a tuple of the form `(3, “abc”)` would have a type of `int * string`. Thus we have an ordered pair of values and corresponding cross product of types. These tuples are not restricted to only two components but rather can be thought of as  $n$ -tuples where  $n \geq 2$ . The components are listed separated by commas and equality is determined component by component.

These are already defined in the compiler and are the primitive set of types that all other types are built from.

## 2.4 Identifiers and Bindings

A common programming practice is to include variable names with expressions and values so that they can be referenced without having to explicitly refer to them on every occurrence. ML provides a system for handling this matter. It is known as value binding, which is dependent on textual context of its definition, and not on some notion of past value assignment. We refer to these occurrences as **value identifiers** or **variables**. All identifiers must be declared before they are used.

### Example 2.4.1.

```
- val x = 15 + 15;
val x = 30 : int

- val y = 10;
val y = 10 : int
- val foo = [ x, y ];
val foo = [ 30, 10 ] : int list

- val x = 3.5;
val x = 3.5 : real
- val stuff = " a b c";
val stuff = " a b c" : string
```

We see that keyword “val” is used to inform the compiler that what is to follow is a binding. Thus  $x$  is bound to the value “30” and is of type integer,  $y$  is bound to the value “10” and is of type integer,  $foo$  is bound to an integer list consisting of the values of  $x$  and  $y$ . Thus the phrase `val x = 15 + 15;` is a value binding. To determine a binding, ML evaluates the right-hand side of an equation and then sets the left-hand



side to this value. Once an identifier is set, it can never be changed. This is known as referential transparency, where the binding is permanent and fixed as opposed to an assignment statement in another language such as Pascal, where the value is maintained and updated. Hence bindings are more closely related to `const` declarations rather than `var` declarations. We illustrate multiple bindings of a variable with the following example.

### Example 2.4.2.

```
- val x = 15 + 15;
val x = 30 : int
- val x = 3.5;
val x = 3.5 : real
- val x = " a b c";
val x = " a b c" : string
- val stuff = x;
val stuff = " a b c" : string
```

We see that that `x` is hidden by rebinding it "3.5" and "a b c". We see that if we have multiple binds of the same identifier, there must be a rule for determining which binding is to be used. This is the one that occurs textually closest to the use of the identifier, which is similar to conventional programming languages. Thus when we bind `stuff` to the value of `x` we see the closest occurrence is the string " a b c".

These value bindings or environments (a collection of bindings) can be used to construct larger, more complex environments, so we are not limited in how many levels of bindings we use to construct a value. To this point, we have been using value bindings and predefined functions, but we now move toward a more complex topic: **function bindings**. This will allow the user a convenient means of describing complex functional descriptions and guarantee they are of the specified type.

We will begin by making some basic characteristics of a functions in ML. Most functions are evaluated by a *function identifier* followed by the arguments that it acts upon, for example  $f(e_1, e_2, \dots)$ , where  $f$  is the function identifier and  $e_1, e_2, \dots$  are the arguments to  $f$ . This syntax is known as prefix notation, since the function precedes the list of arguments. ML uses another syntax for a core set of predefined functions, it is infix notation since the function is embedded in the list of arguments. An illustration of this would be “1 + 2” where the function identifier is “+”, or addition of integers.

The general form of a function is  $ee'$  where  $e$  is the expression that describes the function. In the simplest case  $e$  is just a single functional identifier, but we could have a complex list of identifiers describing the function identifier. The way the ML compiler evaluates the expression  $ee'$  is first by evaluating  $e$  obtaining a function identifier and then by evaluating  $e'$  obtaining a value. We then attempt to resolve the function based on the type of the function. A function in ML is a value and all values have a type associated with them. A function has a compound type that has functions as members. We see the form of a function type to be  $\alpha \rightarrow \beta$ , where  $\alpha$  is known as the domain type and  $\beta$  is the range type. So then a function can be evaluated if its arguments are of type  $\alpha$  and return values of type “ $\beta$ ”.

### Example 2.4.3.

```
- size;
val it = fn : string -> int
- val how_big = size;
val how_big = fn : string -> int
- size "abcdefg";
val it = 7 : int
- how_big("abcde");
val it = 5 : int
std_in:6.1-6.10 Error: operator and operand don't agree (tycon mismatch)
operator domain: string
```

```
operand:          int
in expression:
  size (+ : overloaded (10,8))
- size "10+8";
val it = 4 : int
```

□

Up to this point, we have seen description of functional programming, the history of functional programming, and a working model (SML) of a functional language. This has also been an introduction or primer for some of the necessary SML syntax. We now have most of the components needed to begin our development of categorical limits as a functional programs.

### 3.1 Introduction

In researching computational category theory, two approaches were encountered ([2] and [12]). Both of these approaches will be the focus of the next chapter, with the work form the conference proceedings [2] being the starting point, since it seems to be simpler approach. This will be followed by the later work [12]. Each illustration will be followed by the limitation of each attempt.

We now look at what is needed to develop categories as a program. A categories is a class of objects and a class of arrows together with the structure that organizes it. In programming we have a class of types with the structure that organizes or manipulates these types. Thus we try to see a logical connection from a category to a computer program. Now we develop a schema or mapping for converting these structure in categories into structure in programs.

In categories we see that the structure is related to functions that pick out common information. They are the target and source functions which return an object for each arrow that occurs in the category, the identity function, returning an arrow for each

## Chapter 3

# An illustration of a computational category theory in SML

### 3.1 Introduction

In researching computational category theory, two approaches were encountered ([2] and [12]). Both of these approaches will be the focus of the next chapter, with the work from the conference proceedings [2] being the starting point, since it seems to be simpler approach. This will be followed by the later work [12]. Each illustration will be followed by the limitation of each attempt.

We now look at what is needed to develop categories as a program. A categories is a class of objects and a class of arrows together with the structure that organizes it. In programming we have a class of types with the structure that organizes or manipulates these types. Thus we try to see a logical connection from a category to a computer program. Now we develop a scheme or mapping for converting these structure in categories into structure in programs.

In categories we see that the structure is related to functions that pick out common information. They are the target and source functions which return an object for each arrow that occurs in the category, the identity function, returning an arrow for each

object and composition which for two composable arrows returns their composed arrow. Rarely does one work with a category and no other structure about that category, so with these four functions, we still need some more information. The additional information is the type of the objects and the definition of the arrows, along with some common categorical constructions.

To make computational category theory usable product, the programs have to be constructed in such a general way that they will remain reusable, just like the mathematical counterparts. The common constructions (*i.e.*, products, equalizers) must be designed so that they can be recycled or applied to range of different categories. This is accomplished in ML by the use of binding, we can rebind the functions to the specifics of the various categories. Then these functions can be passed as data to other functions, so we encounter the problem of high-order programming. High-order programming is precisely the ability of being able to use function as first class data, for example, being allowed to have a function that returns another function as its result.

There is one final major requirement that is necessary from the prospective language. This is polymorphism, or the ability of a function or datatype to be applicable to a range of values, for example, the datatype "list" is a good candidate. It can be defined to be of type "real", "strings" or "tuples" of which is again polymorphic, so we can have "many types" lists. There exist functions which are polymorphic by nature such as `isnumber` which would have a type of `'a -> bool`. This means that any argument given to our function `isnumber` whether it is a letter, string, or some other type, will be evaluated to see if it is a number, returning "true" if it is a number and "false" otherwise. So we see when any type is passed to the function it returns a boolean value, thus showing that `isnumber` ranges over the domain of all possible datatypes and maps them to a boolean value. Thus we see that our choice for ML is not without reason.

```
ident: set_object -> set_arrow;
```

```
comp: set_arrow * set_arrow -> set_arrow;
```

Would be the type information that we are looking for and the formal ML definition would be:

## 3.2 First Approach

We must start by finding a way to represent our categories in ML, we might be to accomplish this using a list of objects and a list morphisms, together with a composition arrow. But we would quickly see that would be insufficient in that we would be attempting to build infinite lists to describe our categories. In programming languages and math we know that if we can express a pattern that will exhausted all of the possibilities then we have a much better solution that trying to list all possible elements and this is the precise thing that we make use of.

So we will define a type (*i.e.*, programming language datatype) for our “object” and also for our “arrow”. Along with this we need four functions to help us capture the structure necessary for the proper definition of a category. They would be:

```
source: arrow -> object
target: arrow -> object
identity: object -> arrow
composition: arrow * arrow -> arrow
```

We note that in the above description the terms `arrow` and `object` represent the respective datatypes.

**Example 3.2.1.** Suppose we had the category `FINSET` as our goal (*i.e.*, we want to implement it as a ML program). We could see that our above description would require:

```
type set_object, set_arrow;
source: set_arrow -> set_object;
target: set_arrow -> set_object;
ident: set_object -> set_arrow;
comp: set_arrow * set_arrow -> set_arrow;
```

Would be the type information that we are looking for and the formal ML definition would be:

```

type set_object = int list;
type set_arrow = set_object * (int->int) * set_object;

fun set_eq([],_:set_object,[]:set_object) = true
  | set_eq(hd1::tl1:set_object , hd2::tl2:set_object) = if hd1 = hd2 then
                    set_eq(tl1:set_object , tl2:set_object)
                    else false
  | set_eq( hd::tl:set_object , []:set_object ) = false
  | set_eq( []:set_object , hd::tl:set_object ) = false;

exception no_composition;

fun set_source((a,f,b):set_arrow) = a;
fun set_target((a,f,b):set_arrow) = b;
fun set_ident(a:set_object) = (a, fn x => x, a):set_arrow;
fun set_comp((a,f,b):set_arrow,(c,g,d):set_arrow) =
  if set_eq(b,c) then (a, fn x => g(f(x)), d):set_arrow

```

We see that a few more issue arise when we try to implement the category of **FINSET** . The first is the fact that there is no primitive type “set”, so we use integer lists instead. The next problem comes from composition function need to be able to determine the element-wise equality of our domains to see if the arrows are composable, and ML does not provide such a function so it is up to the user. Finally we see exception in use, so that if `set_comp` determines that two arrows do not have the same domain, then we raise an exception warning the user that composition is impossible.

So the product function would look something like this:

□

```

fun set_product(a, b) =

```

Other categories can be constructed in similar manner, but the main focus of this paper is to evaluate the ML functions on the category **FINSET** . So we will now begin

to look at how other categorical structure is obtained through computational category theory.

### 3.2.1 Universal Categorical Constructions

Rarely are categories worked with only the properties of how they are defined, as seen in the category chapter, we have what is known as universal constructions. They provided added structure that is applied across various classes of categories, in particular terminal objects, products, and equalizers. Just as we have seen that it is possible to describe categories in ML, there is a need to describe the universal constructions in ML. This is where the difficulties began to develop.

Following the scheme of deciding the type and data in question (product) and then developing the program. So we see that a product is an  $object \times object \rightarrow object$  and the projection functions. So a final type for product would be  $object \times object \rightarrow object \times arrow \times arrow$ .

We need two auxiliary functions to implement the product function in ML. The first is `pre` which operates on list elements and by prepending an element to a list. The second will take two lists and return a single list that represents the cross product of the lists. The following are the auxiliary functions:

```
fun pre(x, []) = []  
  | pre(x, y::l) = (x,y)::pre(x,l);
```

```
fun pairs([], l) = []  
  | pairs(x::m, l) = pre(x,l) @ pairs(m, l);
```

So the product function would look something like this:

```
fun set_product(a, b) =  
  let val p = pairs(a,b);  
      val i = (p, fn(x,y)=>x, a)
```



```

    and j = (p, fn(x,y)=>y, b);
in (p, i, j) end;

```

However the product  $(p,i,j)$  (i.e.,  $(A \times B, \pi_1, \pi_2)$ ) produced by this function has a universal property that for any  $C, f : C \rightarrow A$ , and  $g : C \rightarrow B$  there is a unique  $h : C \rightarrow A \times B$  such that  $f = h \circ i$  (i.e.,  $f = h \circ \pi_1$ ) and  $g = h \circ j$  (i.e.,  $g = h \circ \pi_2$ ). So the function product will be modified to yield a forth result the unique function  $h$  that

from objects (C) and pairs of arrows (f,g). So the type of product is now:

```

product: object*object -> object*object*arrow*

```

```

(object*arrow*arrow -> arrow)

```

Note that the \* represents cross product of the type. This same expression could be rewritten using the short notation introduced above, and would look like the following:

```

product: 'o * 'o -> 'o * 'o * 'o * ('o * 'a * 'a -> 'a)

```

To make this addition to the function product we need another auxiliary function `ap` which is used to apply a set arrow to a set object. It would be defined by:

```

fun ap((a,f,b), x) = f x;

```

So the new product would be the same until the addition of the unique function  $h$  as seen below:

```

val set_terminal = let val t = [0];
exception no_product; fun m(a) = (a, fn x => 0, t)
in (t, a) end;

```

```

fun set_product(A, B) =
  let val P = pairs(A,B);
      val pi_1 = (P, fn(x,y)=>x, A)
          and pi_2 = (P, fn(x,y)=>y, B);
  in fun h(Q,f,g) =

```

```

    if (set_eq(set_source(f),Q) andalso set_eq(set_source(g), Q)
        andalso set_eq(set_target(f), A) andalso set_eq(set_target(g),B))
    then (Q, fn x => (ap(f,x), ap(g,x)), P)
    else raise no_product
in (P, pi_1, pi_2, h) end;

```

We see in the above example the use of exception which is ML mechanism for handle runtime error or exceptions, so that rather causing the function to stop execution of the interpreter it executes an exception which if the necessary handling routines were in place we would be able to recover. This illustrations has no need for such recovery other than warning the user that the input provided does not product a product, by printing the message "no\_product".

The next universal construction that is implemented is the terminal object. So the type of a terminal object (*i.e.*, an unique arrow from any object  $A$  to the terminal object "1") would be:

```
terminal: object * ( object -> arrow)
```

So then the terminal = (t,m) where t is the terminal object ("1") and for any object a we see  $m(a) : \rightarrow t$  is the unique arrow to t. We see the ML definition would look like the following:

```

val set_terminal = let val t = [0];
                    fun m(a) = (a, fn x => 0, t)
                    in (t, m) end;

```

The final universal construction that is necessary for our development of categorical limits is that of an equalizer. Recall that an equalizer of  $f : A \rightarrow B$  and  $g : A \rightarrow B$  is an object  $E$  and an arrow  $e : E \rightarrow A$  which satisfies the following conditions:

1.  $f \circ e = g \circ e$ ;

2. whenever  $E'$  and  $e' : E' \rightarrow A$  such that  $f \circ e' = g \circ e'$ , there is a unique  $h : E' \rightarrow E$  such that  $e' = h \circ e$ .

Now as we have done for the other constructions, a type must be defined then we can precede to the development of the function. The type would be defined as follows:

```
equalizer: 'a * 'a -> 'o * 'a * ('o * 'a -> 'a)
```

In **SET** (or **FINSET**) we take our set  $E = \{x \mid x \in A \text{ and } f(x) = g(x)\}$ . Also we note that our unique arrow  $h$  is really just  $e'$  restricted to  $E$ , so we could define an auxiliary function `eql` to help us with `set_equalizer`. The type of `eql` would be:

```
eql: 'int list * (int -> 'a) * (int -> 'a) -> (int list)
```

and the function would be:

```
fun eql([], f, g) = []
  | eql(x::l, f, g) = if f x = g x then x::eql(l,f,g)
                      else eql(l,f,g);
```

In view of this auxiliary function we can now construct the equalizer that we need. It would be defined as follows:

```
exception mediate_equalizer;
exception no_equalizer;
```

```
fun set_equalizer((A,f,B),(A',g,B')) =
  if (set_eq(A,A') andalso set_eq(B,B')) then
    let val E = eql(A,f,g);
        fun h(E',(E'',e',A')) =
          if (set_eq(E'',E') andalso set_eq(A',A)) then (E',e',E)
          else raise mediate_equalizer;
    in (E, (E, fn x => x, a), h) end
  else raise no_equalizer;
```

### 3.2.2 Limits

We now want to start to put together some these universal constructions to aid us in the computation of finding the limit of a finite diagram in **FINSET**. As we saw in the category chapter a categorical proof usually involves diagram chasing where more information about the category can be obtained from the diagram. So we are going to see that this is how to approach the computation. Which means we need to know how to represent a diagram and other related categorical constructions, so to start a type definition is necessary.

Since `int` is the type of our objects, our graph will contain integer sets (*i.e.*, `set_objects`), and integer functions (*i.e.*, `set_arrows`). This will be the starting point.

**Example 3.2.2.** The illustration of a diagram as represented in ML. We will start with the types definitions of the necessary components of a categorical diagram:

```
type node = int;
type edge = int;
type graph_object = node list * edge list * (edge -> node) * (edge -> node);
type ('o, 'a)diagram = graph_object * (node -> 'o) * (edge -> 'a);
type ('o, 'a)cone = ('o, 'a)diagram * 'o * (node -> 'a);
type ('o, 'a)limit_cone = ('o, 'a)cone * (('o, 'a)cone -> 'a);
```

□

Based on these types a type for the categorical limit could be constructed, it could be represented:

```
Limit: ('o, 'a)diagram -> ('o, 'a)limit_cone
```

But some additional information is required, the source, target, identity, and composition functions for the particular category would be necessary. They could be added to our definition, but it would become rather length, so a definition or value binding will be used to express which category we are working with. So:

```
type ('o, 'a)category = ('a->'o)*('a->'o)*('o->'a)*('a*'a->'o)
```

With this new datatype a “category”, a slightly more accurate definition of what a limit is, can be given:

```
Limit: ('o, 'a)category * ('o, 'a)diagram -> ('o, 'a)limit_cone
```

Recall from the first chapter the final section a categorical proof, it illustrates what is necessary to guarantee that a limit exist. If a category has a terminal object, binary products and equalizers then it has all finite limits. So again, we must add more restraints to our definition of what a limit is:

```
type ('o, 'a)Terminal = 'o * ('o->'a);
type ('o, 'a)Product = ('o*'a*'a) * (('o*'a*'a)->'a);
type ('o, 'a)Equalizer = 'a * ('a->'a);
```

```
type ('o, 'a)category_tpe = ('o, 'a)category *
                             ('o, 'a)Terminal *
                             ('o, 'a)Product *
                             ('o, 'a)Equalizer;
```

Which will yield the correct datatype for categorical limits.

```
Limit: ('o, 'a)category_tpe * ('o, 'a)diagram -> ('o, 'a)limit_cone
```

This means that limit of a category can iteratively be constructed by splitting the computation into pieces. Finding the products over all the nodes, and then taking into account all the edges. This construction built form a diagram that has no nodes (or edges), since the limit of this is terminal object of the category. We then start looking at what happens as we add nodes to the base case, until we have added all the node that are required. Once this is complete we move to the next stage of the construction, a diagram that has all of the nodes and no edges. We then add the edges to our diagram

one at a time examining at each step how the new diagram (one with one more edge) affects the the old diagram. This is continued until all the edges are added. The final two steps are to take the product over all the nodes, and add in the effect of the edges thus resulting in the limit of the diagram.

Now that the technique is mapped out for how we utilize the mathematical proof to in our construction, we will start to define the rest of the necessary ML functions. We will start with the function `limitplusnode`.

```
limitplusnode: ('o, 'a)limit_cone * node * 'o -> ('o, 'a)limit_cone
```

```
fun limit_plus_node(((a,c),h_limit), m, new) =
  let val ((p,i,j), h_prod) = product(a, new);
      val a' = p;
      fun c'(n) = if n=m then j else comp(i,c(n));
      val h'_limit(b',d') = let val h1 = h_limit(b',d')
                            in h_prod(b',h1,d'(m)) end
  in ((a',c'), h'_limit) end;
```

We see now that our next function would be similar to the above function where we add a node, only we will add an edge. This means that it will be a similar construction only the use of the equalizer instead of the product is required.

So then the development of the functions that will take the product over the all the nodes and the function that take the limit of the diagram including the edge is all that remain to get a function that will compute limits of finite diagrams. So we see that the product of all the could be obtain by the following:

```
fun limit_of_nodes([], Dn) = terminal, fn n => raise no nodes)
  | limit_of_nodes(n::N, Dn) = limit_plus_node(
                                limit_of_nodes(N, Dn),n, Dn);
```

And the function to obtain the limit over the diagram with the edges would be:

```

fun limit_of_edges((n, [], d0, d1), Dn, De) = limit_of_nodes(N, Dn)
  | limit_of_edges((N, e::E, d0, d1), Dn, De) =
    limit_plus_edge(limit_of_edges((N, E, d0, d1), Dn, De),
                    d0 e, d1 e, e, De e)

```

We see that all of these functions are dependent on the composition, terminal object, product, and equalizer of the particular category. So we must include them as arguments to our function limit which would look like the following:

```

fun limit((source, target, ident, comp), terminal, product, equalizer), D) =
  let fun limit_plus_node(...) = ...;
      fun limit_plus_edge(...) = ...;
      fun limit_of_nodes(...) = ...;
      fun limit_of_edges(...) = ...;
  in limit_of_diagram D end;

```

### 3.2.3 Limitations of the First Approach

The largest limitation that I faced in extracting the functions from the descriptions was the problem of inconsistent notation. This seems to be a small matter, but one strong points of category theory is to provide a common naming convention across categories. When this name convention is disregarded, the task of trying to redesign the ML functions became extremely laborious and difficult. So I have tried to provide a “cleaner” more categorical approach to computational category theory.

The next major rough spot was that lack of complete source or documentation. In describing, the less trivial constructions (*i.e.*, limits) the authors choose to leave out important details, as well as most auxiliary functions and provided no motivations for why incomplete descriptions were provided. This is probably the largest reason for approaching the subject from a limitation point of view. If the source material was carefully organized and thoroughly described maybe the reader would find reason for following the

material. So this is the beginning of why computational category theory is limited in ML.

Now for explicit examples of where problems occurred in the development of functions that were supposed to be complete. As stated both approaches uses the normal, basic category **FINSET**, and still the material was insufficient to adequately describe common starting point for logical development of a formal system. The first approach begins to falter in it's definition of the "set\_objects", they defined them to:

```
type set_object = int list;
```

**Example 3.2.3.** As we can see this is not a polymorphic type as they suggest as requirement for the chosen language. This causes considerable confusion when one attempts to develop a function for products. As we by the type information that the SML interpreter gives:

```
val set_product = fn
  : int list * int list
  -> (int * int) list * ((int * int) list * ('a * 'b -> 'a) * int list)
    * ((int * int) list * ('c * 'd -> 'd) * int list)
    * (set_object * set_arrow * set_arrow
      -> set_object * (int -> int * int) * (int * int) list)
```

Since we would have a hard time trying to force the compiler into realizing that a (int \* int) list is an int list. This is relatively small problem, we could attempt a patch by using 'a list rather than limiting ourself to only integers.

The next major problem was the amount of detail that the easier material was presented with, followed by a less trivial calculation of a finite limit, that was treated trivially. The authors thoroughly developed the universal construction with sufficient detail, then announce that the next construction is less trivial and proceed to start leaving out detail.



They claim that there is a close connection to the proof of the theorem, then do not even state the theorem formally. They informally describe how the proof could be constructed, which is based on the categorical diagram. The diagram and several of the cases that are necessary are not addressed. This lack of detail as the complexity of the functions increase, leaves a strong feeling of distrust and confusion.

### 3.3 Second Approach

Now we turn to the second approach to computational category that we see in [12]. Here are using a more sophisticated approach to what is involved in defining a category, but the authors choose a somewhat unorthodox introduction to categories. They approach datatypes with greater thought, but when they begin to implement the universal categorical constructions they develop the less intuitive route of dual constructions (*i.e.*, coproducts, coequalizers and colimits) and develop the more natural route (*i.e.*, products, equalizers and limits) as dual functions. So although, they improve their approach the overcomplicate matters and give no formal justifications as to why.

#### 3.3.1 Categorical Datatype

If a category can be described by the datatypes of the objects and arrows, along with the four function discussed earlier, a datatype for a category would be the next step. Since we require these types to be polymorphic or ranging over a variety of types. Then we could represent the datatype objects as 'o and the type of arrows as 'a. So then the source and target functions would have type 'a -> 'o, the identity function would have type 'o -> 'a and the type of composition would be 'a \* 'a -> 'a (note: \* represents the cross product of types). The type of a category uses four types and is constructed from a 4-tuple as seen below:

#### Example 3.3.1.

datatype ('o, 'a)Cat =

`cat of ('a->'o)*('a->'o)*('o->'a)*('a*'a->'a)`

This example introduces the “datatype” definitions in ML, there is a key word `datatype`, which signals a type definition. Then the type identifier `('o, 'a)Cat` which is parameterize type displaying the type of the categories' objects and arrows, followed by the data constructor `cat`, the ML keyword `of` and type information.

□

### 3.3.2 The SML Abstract Datatype: 'a Set

In this section we begin developing a SML abstract datatype “a Set” that will be the type we use for our category `FINSET`. One main step in developing a program is the choice of the type of the data that is to be manipulated by the program. We should see that this is not a unique solution for this problem, since programs and the problems they are written for are extremely varied. Rarely if ever is there a direct mapping of the datatype to the real world data that the program expresses. One often represent data in various forms that are different from how it exist in the real world or how humans perceive it in the context of the problem. Often it is necessary to provide more structure with the chosen datatype than we need to express the real world data.

What we do need to guarantee is that certain operations are defined and have a given behavior. It is often important to separate the representation from the use of the type. There is a ‘interface’ between the representation and it’s use, so that the representation can be changed without requiring the rewriting of the programs that make use of the type. This is known as **data abstraction** and is a common concept in computer programming. We also note that for this concept to be effective we need to limit what is available to the user, they should only have access to the interface or functions that act on the newly constructed type.

Now we start by explaining the various components of the of the abstract type “a Set”. This we will see is the foundation of what we construct our category `FINSET`

from. The first obstacle that we encounter is what primitive type is our “a Set” going to be constructed from. The SML “list” of the elements of the set is going to be the choice.

**Example 3.3.2.** The following is an example of what an abstract datatype for set looks like in ML. We see a key word `abstype` signals the start of the definition, followed by the type label and how the type is constructed or what is known as the “interface to the type”. We see “a Set” is polymorphic and are based on the list. What follows is the interface or the set functions that allow us to manipulate the type. With this approach, where we have interface to the type, we can build into the interface specific type components, such as equality or membership.

```

abstype 'a Set = set of 'a list
with
  val emptyset = set([]);
  fun mkset [] = emptyset
    | mkset (hd::tl) = set(hd::tl)
  fun member(x,set(l)) = list_member(x,l)
  fun seteq(set([],),set([],)) = true
    | seteq(set(hd1::tl1),set(hd2::tl2)) = if hd1 = hd2 then
                                             seteq(set(tl1),set(tl2))
                                             else false
    | seteq(set(hd::tl),set([],)) = false
    | seteq(set([],),set(hd::tl)) = false
  : The rest of the interface
end

```

Note: this is not a complete list of the interface for the set used in later constructions. The primitive type “list” is building block of Set, so we need to bootstrap a more complex of 'a list environment. This has been incorporated into the definition of Set and a complete collection of the necessary list functions are in the appendix.

So in **FINSET** objects are of type `'a Set` where `'a` is the polymorphic type variable that describes type of the elements of that set.

**Example 3.3.3.** So using this user defined type `'a Set` we could ML interpretation of **FINSET** as follows:

```

fun set_s(set_arrow(a,_,_)) = a;
fun set_t(set_arrow(_,_,b)) = b;
fun set_ident(a) = set_arrow(a,fn x => x,a);
fun set_comp(set_arrow(c,g,d),set_arrow(a,f,b)) =
  val if seteq(b,c) = true then set_arrow(a,fn x => g(f(x)),d)
      else raise non_composable_pair;

val FinSet = cat(set_s,set_t,set_ident,set_comp);

```

Which yields that following type information:

```

val set_s = fn : 'a Set_Arrow -> 'a Set
val set_t = fn : 'a Set_Arrow -> 'a Set
val set_ident = fn : 'a Set -> 'a Set_Arrow
val set_comp = fn : ''a Set_Arrow * ''a Set_Arrow -> ''a Set_Arrow
val FinSet = cat (fn,fn,fn,fn) : (''a Set,''a Set_Arrow) Cat

```

So it is clear that we can build a category as a datatype, then abstractly describe the content of the prospective category (**FINSET**), from the four functions discussed earlier.

### 3.3.3 Universal Constructions

As in the earlier section we examined what is needed to implement terminal object, products, and equalizers in the category **FINSET**. The terminal object of is almost identical to the terminal object this approach, which is just a value binding representing the relationship singleton sets along with constant function.

**Example 3.3.4.** This has three parts: the definition of the SML datatype, the actual construction of the function, and the use of the type constructor to force it use the defined datatype.

```
datatype ('o, 'a)TerminalObj = terminal of 'o * ('o -> 'a);
```

```
val st_terminal =  
  let val t = mkset([ttrue])  
  in (t, fn a => set_arrow(a,fn x => ttrue,t)) end;
```

```
val set_terminal = terminal(st_terminal);
```

The interpreter would then respond with the following:

```
datatype ('a, 'b) TerminalObj
```

```
  con terminal : 'a * ('a -> 'b) -> ('a, 'b) TerminalObj
```

```
val st_terminal = (-,fn) : 'a Tag Set * ('b Tag Set -> 'b Tag Set_Arrow)
```

```
val set_terminal = terminal (-,fn) : ('a Tag Set, 'a Tag Set_Arrow) TerminalObj
```

□

To ensure that elements of our categories are disjoint, we develop a type definition that will allow us clearly see the elements are distinct. This process could be thought

of as tagging the elements or labeling them with markers so we can be sure as to where they came from. But we must be sure that the type of elements is closed under the requisite operations: pairing for products and a constant element (which we call `ttrue`) for terminal objects. So the following definition will accomplish this:

```
datatype 'a Tag =
  just of 'a
  | pink of ('a Tag)
  | blue of ('a Tag)
  | pair of ('a Tag)*('a Tag)
  | ttrue;
```

So we are ready to develop the universal construction product. Using the SML datatype “a Set” solves the earlier problems that products faced, that of a non-polymorphic type for set objects. We also modify the function to clean up the organization and naming conventions. So as above in terminal object, we have a three part definition of categorical construction product in **FINSET**.

**Example 3.3.5.** This is the implementation of the categorical products: we have the SML datatype for products, the definition of a function for **FINSET**, and the application of the type constructor.

```
datatype ('o,'a)Product =
  product of 'o * 'o -> ('o*'a*'a) * ('o*'a*'a->'a);
```

```
fun st_prdt(a,b) =
  let val a_cross_b = mapset(pair,X(a,b))
      val proj_a =
        set_arrow(a_cross_b,fn pair(y,z) => y, a)
      val proj_b =
        set_arrow(a_cross_b,fn pair(y,z) => z, b)
```

```
val univ =
```

```
  fn (p,f1,f2) =>
```

```
    set_arrow(p,
```

```
      fn y => pair(OF(f1,y) ,OF(f2,y)),
```

```
      a_cross_b )
```

```
  in ((a_cross_b,proj_a,proj_b),univ) end;
```

```
val set_product = product(st_prdt);
```

The interpreter would then respond with the following:

```
datatype ('a,'b) Product
```

```
con product
```

```
: ('a * 'a -> ('a * 'b * 'b) * ('a * 'b * 'b -> 'b)) -> ('a,'b) Product
```

```
val st_prdt = fn
```

```
  : ''a Tag Set * ''a Tag Set
```

```
  -> (''a Tag Set * ''a Tag Set_Arrow * ''a Tag Set_Arrow)
```

```
  * (''a Tag Set * ''a Tag Set_Arrow * ''a Tag Set_Arrow
```

```
  : ''a Tag Set -> ''a Tag Set_Arrow)
```

```
val set_product = product fn : (''a Tag Set, ''a Tag Set_Arrow) Product
```

□

The final construction needed is the equalizer, it is similar to the earlier construction, except we incorporate the auxiliary functions into the main function. We again see a better use of categorical names and standards. Equalizers will also be defined in the three part definition:

**Example 3.3.6.** This is the implementation of the categorical equalizers: we have the SML datatype for equalizers, the definition of a function for **FINSET**, and the

application of the type constructor.

```
datatype ('o,'a)Equalizer = equalizer of 'a * 'a ->
```

```
Example 3.3.7. We need to define ('o * 'a) * ('o * 'a -> 'a);  
and finally limits. The follow would be the SML implementations of such categorical  
fun st_equalizer(f,g) =  
  let val a = source(FinSet)(f)  
      val e = filter_by((fn y => (OF(f,y))=(OF(g,y))),a)  
  in ((e,set_arrow(e, fn x => x,a)),  
      (fn (e1,h1) => set_arrow(e1,fn y => OF(h1,y),e))) end;  
val set_equalizer = equalizer(st_equalizer);
```

The interpreter would then respond with the following:

```
datatype ('a,'b) Equalizer  
  con equalizer : ('b * 'b -> ('a * 'b) * ('a * 'b -> 'b)) -> ('a,'b) Equalizer  
datatype ('o,'a)Complete_Cat =  
val st_equalizer = fn  
  : ''a Set_Arrow * ''a Set_Arrow  
  -> (''a Set * ''a Set_Arrow) * (''a Set * ''a Set_Arrow -> ''a Set_Arrow)  
val set_equalizer = equalizer fn : (''a Set,'a Set_Arrow) Equalizer
```

### 3.3.4 Limits

We now turn our attention to the development of finite limits in **FINSET**. We again must remember the proof that guarantees us that if we have terminal objects, products and equalizer then categories is finitely complete. Again the way we prove this is by induction on the nodes and edges of a diagram. So we need to develop all of the



datatypes or definitions for the various diagram related components of a limit. In the following example 3.3.7 we collect all the necessary pieces.

**Example 3.3.7.** We need to define a node, an edge, a graph, a diagram, a cone and finally limiting cone. The follow would the SML implementations of such categorical objects.

```
datatype Node = word of string;
datatype Edge = word' of string;
datatype Graph = graph of (Node Set)*(Edge Set)*(Edge->Node)*(Edge->Node);
datatype ('o,'a)Diagram = diagram of Graph*(Node ->'o)*(Edge->'a);
datatype ('o,'a)Cone = cone of 'o * ('o,'a)Diagram * (Node ->'a);
datatype ('o,'a)Cone_Arrow = cone_arrow of ('o,'a)Cone * 'a * ('o,'a)Cone;
type ('o,'a)Limiting_Cone =
  ('o,'a)Cone * (('o,'a)Cone -> ('o,'a)Cone_Arrow)
type ('o,'a)Limit =
  ('o,'a)Diagram -> ('o,'a)Limiting_Cone;
datatype ('o,'a)Complete_Cat =
  complete_cat of ('o,'a)Cat * ('o,'a)Limit;
```

□

**Example 3.3.8.** This is a sample of SML code that describes a categorical diagram. It is an example of the definitions of nodes, graphs, diagrams and edges. This diagram could then be used as an argument to the limit function, which requires a diagram to iterate on. This example is included to illustrate the how an abstract description can provide all the information needed to prove the existence of a limit with in a category.

```
val G = let val N = mkset([word "a", word "b",word "c"])
        val E = mkset([word' "f" , word' "g"])
```

```

    fun src (word' "f") = word "a"
      | src (word' "g") = word "a"
    fun tgt (word' "f") = word "b"
      | tgt (word' "g") = word "c"
  in graph( N, E, src, tgt) end;

```

But authors choose to use the dual universal constructions which were on the focus of this thesis, but enough information is available to help the reader understand the dual concepts of dual diagrams and colimits. A dual

```

val a_set = mkset[just "A", just "B",just "C",just "D"];
val b_set = mkset[just "C", just "D", just "E", just "F", just "G"];

```

**Example 3.3.9.** We can now cite the SML code necessary for constructing a categorical limit using the second approach.

```

fun f_fn (just "A") = just "C"
  | f_fn (just "B") = just "C"
  | f_fn (just "C") = just "D"
  | f_fn (just "D") = just "E";
fun dual_graph(graph(N,E,s,t)) = graph(N,E,t,s);
fun g_fn (just "A") = just "B"
  | g_fn (just "B") = just "E"
  | g_fn (just "C") = just "B"
  | g_fn (just "D") = just "F";
fun nodes_to_sets (word "a") = a_set
  | nodes_to_sets (word "b") = b_set
  | nodes_to_sets (word "c") = c_set;
fun edges_to_arrows (word' "f") = set_arrow(a_set,f_fn,b_set)
  | edges_to_arrows (word' "g") = set_arrow(a_set,g_fn,c_set);

```

```
val po_diagram = diagram(G, nodes_to_sets, edges_to_arrows);
```

□

But in the second implementation [12] the authors choose to use the dual universal constructions which were on the focus of this thesis, but enough information is available to help the reader understand the dual concepts of dual diagrams and colimits. A dual diagram is a similar diagram, only we reverse the arrows. Where a colimit is related to least upper bound, as opposed to the greatest lower bound.

**Example 3.3.9.** We can now cite the SML code necessary for constructing a categorical limit using the second approach.

```
val to_p_e_FinSet =
  to_p_e_cat( FinSet, set_terminal,
             set_product, set_equalizer);

fun dual_graph(graph(N,E,s,t)) = graph(N,E,t,s);

fun dual_diagram(diagram(g,nm,em)) = diagram(dual_graph(g),nm,em);

fun co_apex_arrow(cocone_arrow(cone1,arrow,cone2)) = arrow;

fun dual_cone(cocone(a,D,f),univ) =
  let val result_cone = cone(a, dual_diagram(D), f)
  in val universal =
      fn (c1 as cone(a1,D1,f1)) =>
        let val c2 = cocone(a1, dual_diagram(D1), f1)
        in cone_arrow( c1,
                      co_apex_arrow(univ c2),
```

```

    result_cone) end
  in (result_cone,universal) end;
fun dual_colimit(F) =
  fn D => dual_cone(F(dual_diagram D));
val complete_FinSet =
  complete_cat( FinSet,
    limit(finite_limit(
      to_p_e_cat( FinSet,
        set_terminal,
        set_product,
        set_equalizer)))));

```

□

### 3.3.5 Limitations of the Second Approach

The largest problem that kept reappearing was the fact that not all of the code that was supposed to be given was given. This means that to extract a working function from the text was difficult because not all of the specifications were available or vague at best. Therefore, to develop these functions ourselves we had to base our work on the given material and create the functions that would fill in the holes, which at times was extremely difficult since we were unsure of their actual motivations. This is the reason behind not accepting this as working functions and describing the limitations of each approach.

Next, was the problem of why the authors choose the coconstructions over the normal functions. It was clear that the coconstruction were less intuitive and more complex to encode. The authors might have realized something profound about these coconstruc-

tions, but they kept it a secret and failed to present any motivation as to why they choose a much more difficult path and then failed to complete that. We had some conjecture as to why the colimits were attempted, and it was that maybe they saw the colimit as outputs and the colimit as the best output for the job, but this is only speculation and not thoroughly developed. This is why throughout the paper, we adopted the more common approach of limits verses colimits.

Therefore in conclusion, we have seen a brief picture of category theory, followed by a function programming. These two ideas we combined to form an illustration of how one could convert mathematics into functional programs. This all was developed in hopes of some day meshing the complex workings of mathematics and computer programming languages to simplify the process of software development.

## A.1 Auxiliary List Functions

In section 2, a collection of functions that were used to complement the type interface for Set. Since "Set" is constructed out of the primitive type list, we need to be able to manipulate lists. These functions then are inherited by the new type and by this method we can bootstrap our interface into behaving properly.

Basically the functions should be straight forward implementations of Lisp like list functions. The majority of the functions were compiled from the reference material on ML, so that any further questions could be directed to the ML texts from which the functions were obtained.

```
fun cons a x = a::x;
```

```
fun consrev x a = a::x;
```

```
fun append x y = x@y;
```

```
fun list_member(e, []) = false
```

```
  | list_member(e,(a::x)) = if e = a then true
```

```
                           else list_member(e,x);
```

```

fun list_remove(a, []) = []
  | list_remove(a, (a::x)) = if a = a then list_remove(a, x)
                             else a::list_remove(a, x);

```

## Appendix A

```

  | list_map(f, a::x) = f(a)::list_map(f, x);

```

## ML Source Code

```

  | filter p [] = [];

```

### A.1 Auxiliary List Functions

In section a collection of functions that were used to complement the type interface for Set. Since “Set” is constructed out of the primitive type list, we need to be able to manipulate lists. These functions then are inherited by the new type and by this method we can bootstrap our interface into behaving properly.

Basically the functions should be straight forward implementations of lisp like list functions. The majority of the functions were compiled from the reference material on ML, so that any further questions could be directed to the ML texts from which the functions were obtained.

```

fun cons a x = a::x;

```

```

fun consonto x a = a::x;

```

```

fun append x y = x@y;

```

```

fun list_member(e, []) = false

```

```

  | list_member(e, (a::x)) = if e = a then true

```

```

                             else list_member(e, x);

```

```

fun list_remove(e,[]) = []
  | list_remove(e,(a::x)) = if e=a then list_remove(e,x)
                           else a::list_remove(e,x);

fun list_map(f,[]) = []
  | list_map(f,a::x) = f(a)::list_map(f,x);

fun filter p (a::x) = if p a then a::filter p x else filter p x
  | filter p [] = [];

fun exists p (a::x) = if p a then true else exists p x
  | exists p [] = false;

fun all p (a::x) = if p a then all p x else false
  | all p [] = true;

fun accumulate f a (b::x) = accumulate f (f a b) x
  | accumulate f a [] = a;

fun reduce f a (al::rest) = f al (reduce f a rest)
  | reduce f a [] = a;

fun cart(x,y) = reduce( fn a => fn Z =>
                      reduce( fn b => fn Z => (a,b)::Z) Z y) [] x;

val link = accumulate append [];
val revonto = accumulate consonto;
val rev = revonto [];

```

## A.2 Types and Functions

Here is a collection of the functions and types used to construct the category **FINSET**. This is the abstract type definition for Set, along with some necessary function from the interface. This is followed by the type information for defining a category, and the definition of a **FINSET**.

### A.2.1 The Complete Definition of Abstract Datatype 'a Set

This is a complete collection for the SML abstract datatype Set. It includes all of the other types and auxiliary function needed in the second approach to constructing categorical limits. There several function given here that were useful in developing the function that are not referred to anywhere in the paper, such as functions for printing various tagged sets, or even printing a set itself with the bracket notation.

```
datatype 'a Tag =
  just of 'a
| pink of ('a Tag)
| blue of ('a Tag)
| pair of ('a Tag)*('a Tag)
| ttrue;

fun stringwith (front, sep, back) list
  = let fun sepback [] = back
        | sepback [a] = a^back
        | sepback (a::x) = a^sep^sepback x
      in front^sepback list end;

datatype Node = word of string;
```



```
datatype Edge = word' of string;
```

```
fun strofNode (word x) = x;
```

```
fun stroftagNode (just(x)) = "just"^strofNode(x)  
  | stroftagNode (blue(x)) = "blue"^stroftagNode(x)  
  | stroftagNode (pink(x)) = "pink"^stroftagNode(x)  
  | stroftagNode (pair(x,y)) = "pair"^stroftagNode(x)^stroftagNode(y)  
  | stroftagNode (ttrue) = "ttrue";
```

```
(* fun strofEdge *)
```

```
val codeof0 = ord "0";
```

```
fun charofdigit n = chr(n + codeof0);
```

```
fun stringofnat n = if n < 10 then charofdigit n  
  else stringofnat(n div 10)^charofdigit(n mod 10);
```

```
fun stringofint n = if n < 0 then "~"^stringofnat(~n)  
  else stringofnat n;
```

```
fun strofitag (just(x)) = "just"^stringofint(x)  
  | strofitag (blue(x)) = "blue"^strofitag(x)  
  | strofitag (pink(x)) = "pink"^strofitag(x)  
  | strofitag (pair(x,y)) = "pair"^strofitag(x)^strofitag(y)  
  | strofitag (ttrue) = "ttrue";
```

```

exception empty_set;
exception the_nil_fn;

abstype 'a Set = set of 'a list
with
  val emptyset = set([]);
  fun X(set([],set([])) = set(cart([],[]))
    | X(set(a1::x1),set(a2::x2)) = set(cart(a1::x1, a2::x2))
    | X(set([],set(a::x)) = set(cart([], a::x))
    | X(set(a::x),set([])) = set(cart(a::x, []));
  fun mkset[] = emptyset
    | mkset (hd::tl) = set(hd::tl)
  fun filter_by( f, set([])) = set(filter f [])
    | filter_by( f, set(a::x)) = set(filter f (a::x) );
  fun is_empty( set(s)) = length(s) = 0
  fun nil_fn (set(x)) = raise the_nil_fn
  fun singleton(x) = set([x])
  fun remove(x,set(l)) = set(list_remove(x,l))
  fun union(set(s), set(t)) = set(s@t)
  fun member(x,set(l)) = list_member(x,l)
  fun intersection( set([], s2) = set([])
    | intersection(set(h::t), s2) =
      let val tset as (set(tl)) = intersection(set(t),s2)
      in if member(h,s2) then set(h::tl) else tset end
  fun seteq(set([],set([])) = true
    | seteq(set(hd1::tl1),set(hd2::tl2)) = if hd1 = hd2 then
      seteq(set(tl1),set(tl2))
    else false
end;

```

```

A.2.2 | seteq(set(hd::tl),set([])) = false
      | seteq(set([]),set(hd::tl)) = false
      fun singleton_split(set(nil)) = raise empty_set
      | singleton_split(set(x::s)) = (x,remove(x,set(s)))
datatype fun split(s) =
          let val (x,s') = singleton_split(s) in (singleton(x),s')
          end;
datatype fun diff(set(x),a) = if member(a,set(x)) then remove(a,set(x))
                               else set(x);
datatype ('a,'b)Set =
  fun mapset(f,set(l)) = set(list_map(f,l));
  fun card(set([])) = 0
  fun card(set(h::t)) = if seteq(singleton(h),set([])) then 0
                         else 1+card(set(t));
  fun prtintset(set([])) =
    stringwith("{",",",","}" (list_map(stringofint, []))
  | prtintset(set(a::x)) =
    stringwith("{",",",","}" (list_map(stringofint, a::x))
  fun prttagset(set([])) =
    stringwith("{",",",","}" (list_map(strofitag, []))
  | prttagset(set(a::x)) =
    stringwith("{",",",","}" (list_map(strofitag, a::x))
A.3 | fun prtNodetagset(set([])) =
      stringwith("{",",",","}" (list_map(stroftagNode, []))
      | prtNodetagset(set(a::x)) =
        stringwith("{",",",","}" (list_map(stroftagNode, a::x))
      end;
type set_object = int list;

```

## A.2.2 FINSET Complete Definition

Here is the complete definition of FINSET , along with various other necessary type information.

```
datatype 'a Set_Object = set_object of 'a Set;

datatype 'a Set_Arrow = set_arrow of ('a Set)*('a->'a)*('a Set);

exception non_composable_pair;

datatype ('o,'a)Cat =
  cat of ('a->'o)*('a->'o)*('o->'a)*('a*'a->'a);

fun set_s(set_arrow(a,_,_)) = a;
fun set_t(set_arrow(_,_,b)) = b;
fun set_ident(a) = set_arrow(a,fn x => x,a);
fun set_comp(set_arrow(c,g,d),set_arrow(a,f,b)) =
  if seteq(b,c) = true then set_arrow(a,fn x => g(f(x)),d)
  else raise non_composable_pair;

val FinSet = cat(set_s,set_t,set_ident,set_comp);
```

## A.3 The Limit Code

### A.3.1 First Approach

This is a complete collection of the code used in the first approach to the finite limits attempt.

```
type set_object = int list;
```

```

type set_arrow = set_object * (int->int) * set_object;

fun set_eq([], []:set_object) = true
(* | set_eq(hd1::tl1:set_object , hd2::tl2:set_object) = if hd1 = hd2 then
exception no_composition;
    set_eq(tl1:set_object , tl2:set_object)
else false
| set_eq( hd::tl:set_object , []:set_object ) = false
| set_eq( []:set_object , hd::tl:set_object ) = false;
fun target(set_arrow(a,f,b)) = b; *)

exception no_composition;

type ('a,'b)Terminal = 'a * ('b->'a);
type ('a,'b)Product = ('a*'b->'a) * (('a*'b->'a)->'a);

fun set_source((a,f,b):set_arrow) = a;

fun set_target((a,f,b):set_arrow) = b;

fun set_ident(a:set_object) = (a, fn x => x, a):set_arrow;

fun set_comp((a,f,b):set_arrow,(c,g,d):set_arrow) =
  if set_eq(b,c) then (a, fn x => g(f(x)), d):set_arrow
  else raise no_composition;

fun pre(x, []) = []
  | pre(x, y::l) = (x,y)::pre(x,l);

```

```

fun pairs([], 1) = []
  | pairs(x::m, 1) = pre(x,1) @ pairs(m, 1);

(* fun source(set_arrow(a,f,b)) = a;
exception tough_one;
fun OF(set_arrow(a,f,b),y) = if member(y,a) then f y
                             else raise tough_one;

fun target(set_arrow(a,f,b)) = b; *)

type ('o,'a)Terminal = 'o * ('o->'a);
type ('o,'a)Product = ('o*'a*'a) * (('o*'a*'a)->'a);
type ('o,'a)Equalizer = 'a * ('a->'a);

fun ap((a,f,b), x) = f x;

exception no_product;

fun set_product(a, b) =
  let val p = pairs(a,b);
      val i = (p, fn(x,y)=>x, a)
          and j = (p, fn(x,y)=>y, b)
      fun h(q,f,g) =
          if (set_eq(set_source(f),q) andalso set_eq(set_source(g), q)

```

```

andalso set_eq(set_target(f), a) andalso set_eq(set_target(g),b))
type ('o,'a)category =
  then (q, fn x => (ap(f,x), ap(g,x)), p)
  else raise no_product
in (p, i, j, h) end;

```

```

val set_terminal = let val t = [0];
  fun m(a) = (a, fn x => 0, t)
  in (t, m) end;

```

```

fun eql([], f, g) = []
  | eql(x::l:int list, f, g) = if f(x) = g(x) then x::eql(l,f,g)
  else eql(l,f,g);

```

```

exception mediate_equalizer;

```

```

exception no_equalizer;

```

```

fun set_equalizer((a,f,b):set_arrow,(a',g,b'):set_arrow) =
  if (set_eq(a,a') andalso set_eq(b,b')) then
    let val e = eql(a,f,g);
      fun h(e',(e'',i',a')) =
        if (set_eq(e'',e') andalso set_eq(a',a)) then (e',i',e)
        else raise mediate_equalizer;
    in (e, (e, fn x => x, a), h) end
  else raise no_equalizer;

```

```

type ('o,'a)category = (set_arrow->set_object)*
                                (set_arrow->set_object)*
                                (set_object->set_arrow)*
                                (set_arrow*set_arrow->set_object);

type ('o,'a)category_tpe = ('o,'a)category *
                                ('o,'a)Terminal *
                                ('o,'a)Product *
                                ('o,'a)Equalizer;

exception no_nodes;

fun limit(((source,target,ident,comp),terminal,product,equalizer), D)
= let fun limit_plus_node(((a,c),h_limit), m, new) =
      let val ((p,i,j), h_prod) = product(a, new);
          val a' = p;
          fun c'(n) = if set_eq(n,m) then j else comp(i,c(n));
          val h'_limit(b',d') = let val h1 = h_limit(b',d')
                                in h_prod(b',h1,d'(m)) end;
      in ((a',c'), h'_limit) end;

      fun limit_plus_edge(((a,c),h_limit),l,m,e,f) =
          let val (r, q, h_eq) = equalizer((a,comp(f,c(l)),m)(a,c(m),m));
              val a' = r;
              fun c'(n) = comp(c(n),q);
              val h'_limit(E',k) = let val d(n) = comp(c(n),k);
                                      val h1 = h_limit(E',k);
                                      in h_eq(E',(E',h1,a')) end;
          end;
end;

```



```

datatype ('o,'a) in ((a',c'), h'_limit) end;

datatype ('o,'a) fun limit_of_nodes([], Dn) = (terminal, fn n => raise no_nodes)
  | limit_of_nodes(n::N, Dn) =
  | limit_of_nodes(N, Dn), n, Dn);

datatype ('o,'a) fun limit_of_diagram((n, [], d0, d1), Dn, De) = limit_of_nodes(N, Dn)
  | limit_of_diagram((N, e::E, d0, d1), Dn, De) =
  | limit_of_diagram((N, E, d0, d1), Dn, De), d0 e, d1 e, e, De e);

datatype ('o,'a) in D;
  complete_cat of ('o,'a) Cat = ('o,'a) Limit
  in limit_of_diagram(D);

```

### A.3.2 Second Approach

This is a complete collection of the code used in the second approach to the finite limits attempt.

```

datatype Graph = graph of (Node Set)*(Edge Set)*(Edge->Node)*(Edge->Node);

datatype ('o,'a) Diagram = diagram of Graph*(Node ->'o)*(Edge->'a);

datatype ('o,'a) Product = product of 'o * 'o -> ('o*'a*'a) * ('o*'a*'a->'a);

datatype ('o,'a) Equalizer = equalizer of 'a * 'a ->
  ('o * 'a) * ('o * 'a -> 'a);

datatype ('o, 'a) TerminalObj = terminal of 'o * ('o -> 'a);

```

```

datatype ('o,'a)Cone = cone of 'o * ('o,'a)Diagram * (Node ->'a);

datatype ('o,'a)Cone_Arrow = cone_arrow of ('o,'a)Cone * 'a * ('o,'a)Cone;

type ('o,'a)Limiting_Cone =
  ('o,'a)Cone * (('o,'a)Cone -> ('o,'a)Cone_Arrow)

type ('o,'a)Limit =
  ('o,'a)Diagram -> ('o,'a)Limiting_Cone;

datatype ('o,'a)Complete_Cat =
  complete_cat of ('o,'a)Cat * ('o,'a)Limit;

fun dual_graph(graph(N,E,s,t)) = graph(N,E,t,s);

fun dual_diagram(diagram(g,nm,em)) = diagram(dual_graph(g),nm,em);

fun co_apex_arrow(cocone_arrow(cone1,arrow,cone2)) = arrow;

fun dual_cone(cocone(a,D,f),univ) =
  let val result_cone = cone(a, dual_diagram(D), f)
      val universal =
        fn (c1 as cone(a1,D1,f1)) =>
          let val c2 = cocone(a1, dual_diagram(D1), f1)
              in cone_arrow( c1,
                            co_apex_arrow(univ c2),
                            result_cone) end
      in result_cone
  end

```

```

in (result_cone,universal) end;

      fn y => pair(OF(X1,y) ,OF(X2,y)),

fun dual_colimit(F) = a_cross_b
  fn D => dual_cone(F(dual_diagram D));

fun source(cat(src,trg,id,comp)) = src;

exception tough_one;

fun st_equalizer(f,g) =
  fun OF(set_arrow(a,f,b),y) = if member(y,a) then f y
    else raise tough_one;
  val e = filter_by((fn x => x.a),
    in ((e,set_arrow(e,fn x => x.a)),
      (fn (a1,a2) => set_arrow(a1,fn y => OF(a1,y),e))) end;

val st_terminal =
  let val t = mkset([ttrue])
  in (t, fn a => set_arrow(a,fn x => ttrue,t)) end;
datatype ('a, 'b) TO_P_E_Cat =
val set_terminal = terminal(st_terminal);
datatype ('a, 'b) TerminalObj =
datatype ('a, 'b) Product =
datatype ('a, 'b) Equalizer;

fun st_prdt(a,b) =
  let val a_cross_b = mapset(pair,X(a,b))
  val to_val proj_a =
    to_p_e set_arrow(a_cross_b,fn pair(y,z) => y, a)
  val proj_b = reduct, set_equalizer);
    set_arrow(a_cross_b,fn pair(y,z) => z, b)
  val univ =
  fun limit(F fn (p,f1,f2) => F(diagram D));

```

```

        set_arrow(p,
                fn y => pair(OF(f1,y) ,OF(f2,y)),
                a_cross_b )
    in ((a_cross_b,proj_a,proj_b),univ) end;

val set_product = product(st_prdt);

fun st_equalizer(f,g) =
    let val a = source(FinSet)(f)
        val e = filter_by((fn y => (OF(f,y))=(OF(g,y))),a)
    in ((e,set_arrow(e, fn x => x,a)),
        (fn (e1,h1) => set_arrow(e1,fn y => OF(h1,y),e))) end;

val set_equalizer = equalizer(st_equalizer);

datatype ('o, 'a) TO_P_E_Cat =
    to_p_e_cat of ('o, 'a)Cat *
                ('o, 'a)TerminalObj *
                ('o, 'a)Product *
                ('o, 'a)Equalizer;

val to_p_e_FinSet =
    to_p_e_cat( FinSet, set_terminal,
                set_product, set_equalizer);

fun limit(F) = fn D => cone(F(diagram D));

```

```

val complete_FinSet =
  complete_cat( FinSet,
    limit(finite_limit(
      to_p_e_cat( FinSet,
        set_terminal,
        set_product,
        set_equalizer)))));

```

- [1] J. Backus. Can programming be liberalized? Yes, Neumann style! A functional style and its algebra of programs. *The ACM*, 21:123-140, 1978.
- [2] R. Burstall and D. Rydeheard. Computing with categories. In David Pitt, editor, *Category theory and computer programming: tutorial and workshop, number 249 in Lecture Notes in Computer Science*, pages 506-519, Guildford, U.K., September 1985. Springer-Verlag.
- [3] A. Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.
- [4] M. Gordon, R. Müller, L. Morris, N. Newey, and C. Wadsworth. A metalanguage for interactive proofs in lcf. *Symposium on Principles of Programming Languages*, pages 119-130. ACM, 1978.
- [5] P. Landin. The next 700 programming languages. *Communications of ACM*, 9:157-164, 1956.
- [6] Xavier Leroy and Michel Mauny. *The Caml Light system, release 9.5 documentation and user's manual*, September 1992.
- [7] J. MacCarthy. *Lisp 1.5 Programmer's Manual*. MIT Press, Cambridge Mass., 1962.
- [8] Laurence G. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

# Bibliography

- [1] J. Backus. Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21:133–140, 1978.
- [2] R. Burstall and D. Rydeheard. Computing with categories. In David Pitt, editor, *Category theory and computer programming : tutorial and workshop*, number 240 in Lecture Notes in Computer Science, pages 506–519, Guildford, U.K., September 1985. Springer-Verlag.
- [3] A. Church. *The Calculi of Lambda Conversions*. Princeton University Press, 1941.
- [4] M. Gordon, R. Milner, L. Morris, N. Newwey, and C. Wadsworth. A metalanguage for interactive proofs in lcf. Symposium on Principles of Programming Languages, pages 119–130. ACM, 1978.
- [5] P. Landin. The next 700 programming languages. *Communications of ACM*, 9:157–164, 1966.
- [6] Xavier Leroy and Michel Mauny. *The Caml Light system, release 0.5 documentation and user's manual*, September 1992.
- [7] J. MacCarthy. *Lisp 1.5 Programmer's Manual*. MIT Press, Cambridge Mass., 1962.
- [8] Laurence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

- [9] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. Foundations of Computing Series. The MIT Press, Cambridge Mass, 1991.
- [10] Chris Reade. *Elements of Functional Programming*. Addison-Wesley Publishing Company, 1989.
- [11] S. E. Rodabaugh. Fuzzy sets with applications: An introductory outline. Introduction to Category Theory, 1992 September.
- [12] David E. Rydeheard and Rod M. Burstall. *Computational Category Theory*. Prentice Hall International, 1988.
- [13] S. Sokolowski. *Applicative high order programming: Standard ML in practice*. Chapman and Hall computing series. Chapman and Hall, London; New York, 1st ed. edition, 1991.
- [14] D. Turner. Miranda: a non-strict functional language with polymorphic types. In *Functional Programming languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 1–16. Springer Verlag, 1985.
- [15] R. F. C. Walters. *Categories and Computer Science*. Cambridge University Press, 1991.