To the Graduate Council:

I am submitting herewith a dissertation written by Hong Gee Chen entitled "Implementing Categorical Programming Languages." I have examined the final copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

J. Robin B. Cockett, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council:

Vice Provost
and Dean of The Graduate School

IMPLEMENTING CATEGORICAL PROGRAMMING LANGUAGES

A Dissertation

Presented for the

Doctor of Philosophy Degree

The University of Tennessee, Knoxville

Hong Gee Chen

December 1990

Copyright © Hong Gee Chen, 1990
All rights reserved

To the late Earl Chen Chow [周娥] and

the late Chuan Shui Chen [陳泉水].

ACKNOWLEDGEMENTS

I would like to thank my advisor, J. Robin B. Cockett, and the other members of my committee, J. D. Birdwell, B. J. MacLennan, J. H. Poore, and M. G. Thomason, for their guidance, criticisms, and encouragement.

The years I have spent in the UTK Computer Science Department have been pleasant and stimulating. I wish to express my gratitude to all the caring friends for making my experience here as rewarding personally as it has been intellectually. Finally, I want to thank my relatives and my wife Chun Huey [純惠] for their support and encouragement.

ABSTRACT

While the use of category theory in the specification and construction of programming languages is increasing, there is still a considerable gap between categorical definition and what is needed to specify and implement a programming language. A categorical framework is needed to organize different computational settings uniformly. This framework should facilitate the description and verification of their operational semantics and programming environment so that the correctness of their implementations can be verified.

This dissertation is designed to help fill this gap. It presents categorical combinator theories which provide a uniform and polymorphic method for specifying varieties of categories such as cartesian closed, distributive, and list arithmetic categories. Using these theories, we show how the combinator expressions can be type checked and reduced to a canonical form. Furthermore, simple abstract machines have been developed to facilitate the efficient implementation of these languages.

TABLE OF CONTENTS

CHAPTER PAGE

1. Introduction and Overview 1

1.1 Categorical Combinators 2

1.2 Categorical Programming 4

1.3 The Organization of this Dissertation 7

2. Background 11

2.1 Basic Categorical Concepts 11

2.2 Semantics 16

2.2.1 Free Term Algebras (over Sets) 17

2.2.2 The Clone of a Theory **T** 21

2.2.3 List Arithmetic Equational Theories 22

2.2.4 Translation to Combinators 26

3. Categorical Combinator Theories 28

3.1 Type Expressions and Classes 29

3.2 Combinator Expressions 31

3.3 Type Checking 35

3.4 Inference rules 37

4. Categorical Combinator Reduction System 42

4.1 A Description of the System 42

4.2 Classes of Expressions 45
4.3 The Reduction Rules 48
4.4 Properties of the Reduction System 54

5. Abstract Machines for Evaluation 57
5.1 Chain Rewriting 57
5.2 The Greedy Abstract Machine 61
5.3 A Partially Lazy Abstract Machine 65
5.4 Graph Reduction 71

6. Conclusion 74
6.1 Contributions 74
6.2 Directions for Future Research 75

BIBLIOGRAPHY 77

APPENDIX 84

VITA 106

LIST OF FIGURES

FIGURE **PAGE**

1.1 Categorical Programming Environment 5

CHAPTER 1

Introduction and Overview

Combinators provide an alternative representation of the λ-calculus [14, 23]; they have computational significance as their rewritings are generally simpler than those of the λ-calculus itself. This has led researchers [52, 80] to investigate programming languages whose underlying computational machinery is motivated by combinator rewritings, as it was perceived that in this approach lay a computational advantage. In practice, this computational promise was hard to exploit. In particular, the simplistic use of combinators resulted in unwieldy expressions simply to move data to where functions could be applied. These problems led to the development of alternative combinator sets and the introduction of supercombinators in order to obtain more efficient implementations.

The introduction of new combinators raised an underlying issue of how to arrive at the right sets of combinators for an application. This issue is similar to the more traditional problem of finding the right data structures and primitive operations for an application.

1.1 Categorical Combinators

With the connections between cartesian closed categories and the λ-calculus becoming firmly established, see Lambek [55] and Lambek and Scott [56], there became available another combinator style language into which the equational style (typed) λ-calculus expressions could be translated: namely, categorical notation. Unlike classical combinators, little was known about the rewriting systems available for category combinators. Indeed, it only gradually became clear that the availability of this representation could be exploited in this direction.

In categories, composition rather that λ-abstraction is the fundamental construct. This means that categorical combinators can cover a wide range of settings including those in which exponentiation is lacking. Hagino in his thesis [42] made the important connection between categorical data structures and the setting. He observed that data structures not only determine the theoretical setting but they also determine reduction rules. This means that a programming language can be totally specified by its declared data structures.

The ADJ group [34, 35] provided a very clean categorical view of data structures as initial algebras. Hagino realized that programming languages need final algebra constructions as well. The most obvious of these is the product. Once one allows that final constructs are also data structures, it is possible to introduce exponentiation into a setting by declaration. Conversely, of course, it is also possible to exclude it by not making the declaration.

This philosophy of language specification is very attractive for it makes a wide

range of data structures readily available and programmers can choose the complete setting to suit the application. However, little is gained by this flexibility if the data structures are inefficiently implemented. The reductions associated with a declared data structure must, therefore, be deeply embedded in the computational mechanism and this implies a departure from the philosophy which translates all computation into a fixed set of combinators.

There are other important effects of following this categorical route to programming languages:

1. The underlying machinery is polymorphic (parametric) so that a high degree of data abstraction can be obtained with benefits in software development and reusability.

2. Every category has an associated equational style logic. There is, therefore, a more friendly syntax available which is equipped with a proof system which can be exploited in supporting program development.

3. The most interesting categorical systems are strongly normalizing (it is not possible to create non-terminating programs) and this can provide significant advantages in validation and verification.

4. Initial algebra constructs not only do not destroy termination properties when introduced, but also behave in the correct formal manner.[1] This

[1] In settings with fixed points it is a well-known fact of mathematical folklore that this is not the case.

greatly facilitates verification and code transformation for efficiency which utilize the properties of these data structures.

When the components of categorical programming are allowed to accumulate what emerges is a dramatically different philosophy towards software development. The traditional fixed point semantics with open ended recursion, for example, has little or no role: most categorical settings of interest already possess the strong normalization property. This fact affects the whole style of programming. It changes the emphasis from ease of programming to that of securing formally correct programs. It is this change in emphasis which we find the most exciting, for it neatly by-passes many of the fundamental pitfalls inherent in the traditional approaches. Furthermore, it brings programming into line with mathematical intuition and practice.

1.2 Categorical Programming

This dissertation is primarily concerned with developing the theory of categorical combinators and connecting this to the design and implementation of an efficient evaluation machinery. This is, of course, only a small part of the full categorical programming language story. The dotted boxes and arrows in figure 1.1 indicates what remain to be done in this broader effort; however, it is perhaps worth discussing some of these other components in order to put the current effort in perspective.

It is not proposed that a programmer should program directly in combinators.

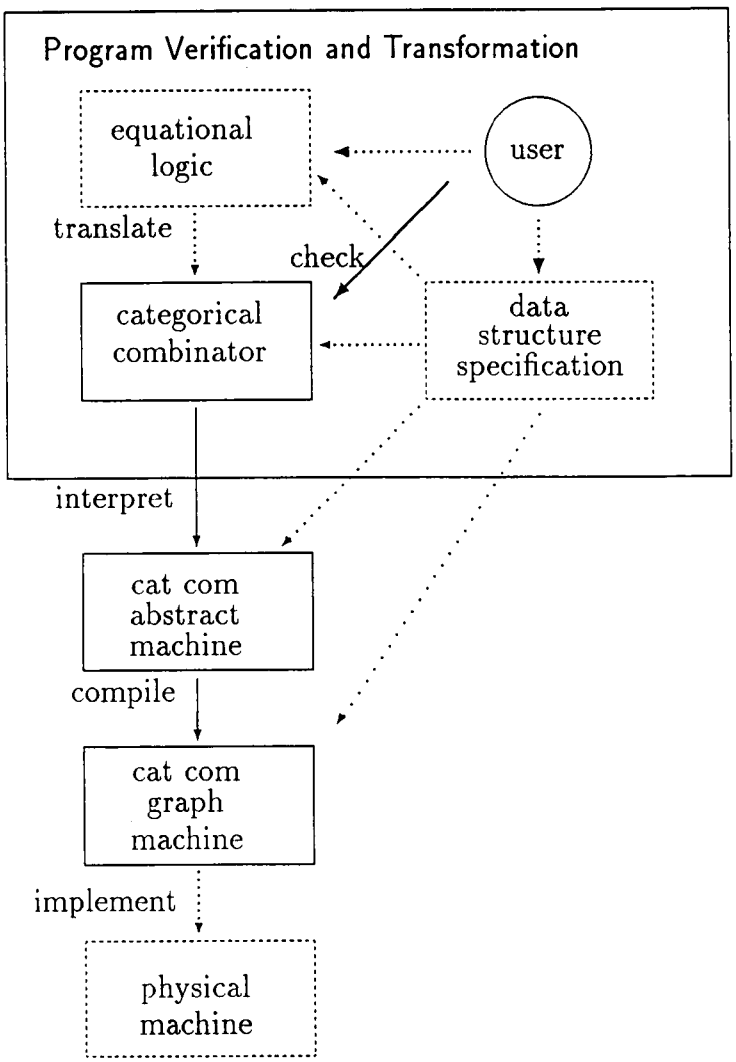


Figure 1.1: Categorical Programming Environment

In fact, combinators provide, on the whole, a rather unfriendly language. Rather, it is proposed that he should use a mixture of equational logic and combinators for programming. However, it is proposed that the mechanical evaluation should be performed at the combinator level.

Thus, it is necessary to be able to translate from equational notation to combinators. Furthermore, it is desirable that this translation be efficient. This is no small problem area! Not only must there be automatic means of associating equational logics with categorical combinator theories (something which is still a somewhat ad hoc business), but there must also be generic and efficient translation mechanisms.

In addition, a programmer should be allowed to declare data structures. These declarations must then be incorporated into the syntax, support environment, and, to ensure efficiency, deep in the evaluation mechanisms. Hagino has already proposed [42] a discipline for these declarations which ensures consistency and the strong normalization property: unfortunately, this discipline does not cover some interesting examples (e.g. distributive categories and the list arithmetic setting of a locos). However, it does indicate that a suitable discipline for data structure declarations can be obtained.

An increasingly important component of any programming language is its support environment. This encompasses not simply debugging aids but also the formal tools for verification and manipulation. Perhaps the most significant advantages of the categorical approach to programming are in this area as its formal underpinnings mirror mathematical intuition closely.

Finally, it is necessary to consider how such a language might be implemented. Wraith [82] has shown how the weak version of Hagino's system can be implemented in the polymorphic λ-calculus but suggested that it might be valuable to look for more direct implementation techniques. In this dissertation we develop a more direct approach beginning with a simplified abstract machine for evaluating categorical combinators which is similar to the categorical abstract machine, **CAM**, of Curien [22] in that the simplified machine also performs an eager evaluation. While the CAM extends its machine to allow the ML style lazy evaluation, our simplified machine is modified to do chain rewritings assisted by an accounting system which then incorporates a scheme to share subexpressions during the reduction. This modified machine is a graph reducer and follows the reduction style proposed by Hagino, but is only partially lazy. Thus, a fully lazy graph reduction machine is a desirable addition.

1.3 The Organization of this Dissertation

This introductory chapter gives a preview of the categorical programming environment and a summary of this dissertation.

Chapter 2 describes the background. It first introduces the basic categorical concepts used in this dissertation. These concepts are then applied to organize a rather familiar algebraic system so that it is easy to connect to the categorical combinator theories discussed in the next chapter and also to reveal the organizing power of category theory.

Chapter 3 presents a simplified version of categorical combinator theories whose generalized version is in [21]. A categorical combinator theory has four components: a set of classes, a set of type constructors, a combinator base, and a set of axioms. The classes are primitive data types and the type constructors are used to construct recursive data types. The combinator base consists of combinators, which are polymorphic maps. That is maps which start and end at polymorphic data types and which can be associatively composed. Finally, these combinators satisfy the axioms of the combinator theory.

Combinator theories differ significantly from the categorical specification language introduced by Hagino as we do not start from functors and adjoint structures. Rather we start from type constructors and maps between these (the combinator base). We feel that this is a slightly cleaner starting point as it avoids the difficulties introduced by variance. In fact it allows us to dispense entirely with the functorial calculus which was used by Hagino to calculate varity.

In order to describe a programming semantics, it is necessary to augment the theoretical setting with a reduction system. Crucial to defining a sensible programming semantics is that the reduction system is compatible, in some defined sense, with the theoretical system. In standard rewriting systems it is usual to simply invert the reduction rules to obtain the theory. It is an important realization that in combinator reduction systems it is quite possible, and even desirable, that the theory is stronger than that implied by the reductions. This divergence led to the consideration of weak versions of data structures, particularly of the natural number object in Lambek and Scott [56]. It also underpins the reason the

data structures in the polymorphic λ-calculus are weak: it is a rewriting setting.

Chapter 4 presents a combinator reduction system where the combinators are divided into three classes: the *active* combinators, the *distributive* combinators, and the *constructive* combinators. Once this division is made we can define what a *closed* and *canonical* combinator expression look like. The purpose of a reduction system is to reduce any closed combinator expression to a canonical one. This is done by removing, using the reduction rules, any occurrence of an active which has a *context.* In order for this to work the reduction rules must satisfy certain conditions which are similar to the requirements in Huet [45].

An abstract machine for a combinator reduction system can be divided into two parts: the traversal method and the reduction of each combinator. The traversal method not only affects the components needed by the machine but also the way each combinator is reduced. In chapter 5, a simple by-value machine is introduced first for clarity. The principle of this evaluation is that the subexpression that has been traversed must be in canonical form. The machine does not require closures or any of the complication required to achieve lazy rewriting; moreover, the standard reduction rules can be directly applied to each redex. Thus, it provides a solid beginning to the design of abstract machines for these languages.

A partially lazy evaluation machine used by Hagino[42] is discussed next. This machine avoids reducing any subexpression of a distributive combinator unless it is deemed necessary. Furthermore, we introduce an accounting system which helps avoid unnecessary traversal. It also manages subexpression sharings, so

that when a subexpression is evaluated, the evaluation will benefit all who share this structure.

The fact that the machine is performing a partially lazy evaluation complicates the structure of the machine. In particular, it is necessary to form closures (suspended evaluations) and to do all the storage management required for graph reduction. We see that a partially lazy evaluation involves a lot of work. The investigate of a fully lazy machine is therefore necessary.

Finally, conclusions and directions for future work are given in chapter 6.

CHAPTER 2

Background

Category theory is a calculus of programs: it studies objects (types), and morphisms (programs) between them and focuses only on the way the morphisms can be combined. This external approach to computation, i.e. hiding the underlying detail representation and implementation from the computation reasoning, is particularly appropriate to computer programming[26]. In this chapter we introduce some categorical concepts that are used in this dissertation. The reader is referred to category theory textbooks like [3, 38] for further details. We then show how algebraic systems can be viewed as a categorical system leading to an example which is complemented as a programming language later in the dissertation.

2.1 Basic Categorical Concepts

Definition 2.1 *A* **category X** *is a sextuple* $\mathbf{X} = (\mathbf{X}_0, \mathbf{X}_1, \delta_0, \delta_1, i, \cdot)$ *where* $\mathbf{X}_0$ *is a class of objects,* $\mathbf{X}_1$ *is a class of maps,* δ_0 *is the domain map,* δ_1 *is the codomain map,* i *is the identity map,* $\cdot$ *is binary composition, such that*

(i) The structure $(\mathbf{X}_0, \mathbf{X}_1, \delta_0, \delta_1)$ *is a directed graph. That is,* $\delta_0, \delta_1 : \mathbf{X}_1 \rightarrow \mathbf{X}_0$ *and a map* $f \in \mathbf{X}_1$ *is written* $f : \delta_0(f) \rightarrow \delta_1(f)$ *to indicate its domain* $\delta_0(f)$ *and codomain* $\delta_1(f)$.

(ii) Composition: $f \cdot g \in \mathbf{X}_1$ *is defined iff* $\delta_1(f) = \delta_0(g)$. *Furthermore,* $\delta_0(f \cdot g) = \delta_0(f)$, $\delta_1(f \cdot g) = \delta_1(g)$, *and whenever the composition is defined, it is associative, that is,* $(f \cdot g) \cdot h = f \cdot (g \cdot h)$.

(iii) Identity: there is a map $i : \mathbf{X}_0 \to \mathbf{X}_1 : X \to i_X$ *which associates to each object the identity map on that object. That is, a map* $i_X : X \to X$ *such that whenever the compositions are defined,* $i_X \cdot f = f$ $g \cdot i_X = g$ *for all* $f, g \in \mathbf{X}_1$

Examples of categories are given by

(i) **Set**, the category of sets: with objects the class of all sets and maps, functions between sets.

(ii) $\mathbf{Mat}_K$, the category of matrices over a field K: with objects the natural numbers 0,1,2,3,$\cdots$ and maps $f : n \to m$ given by $n \times m$ matrices with the usual composition. Note that there is only one matrix $0 \to n$ (and $n \to 0$) given by the zero map.

(iii) The algebras of any algebraic theory, $\mathbf{T}$, form a category, **Set**, whose objects are algebras and whose maps are algebra homomorphisms.

Definition 2.2 *An object is a* **final object** *if there is a unique map from every object to it. The designated final object of a category shall be denoted by 1.*

Dually an object is an **initial object** if there is a unique map from it to every object. In **Set** the empty set $\{\}$ is an initial object (in fact unique in this regard)

and any singleton set, $\{\cdot\}$, is a final object, since $\{\}$ has a unique map to every object and every object has a unique map to a singleton set in **Set.**

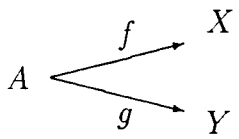
Definition 2.3 *An* **element** *is a map from the final object* 1.

Let **X** be a category and X and Y be objects of **X.**

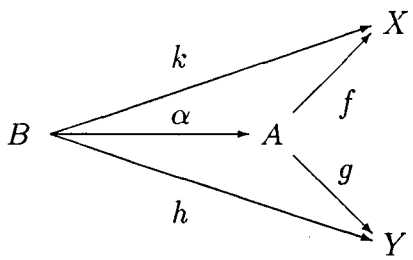
Definition 2.4

(i) A **product** *of X and Y is a triple (A, f, g) such that*

(1) it is a candidate product of X and Y, i.e., $\delta_0(f) = \delta_0(g) = A$, $\delta_1(f) = X, \delta_1(g) = Y$

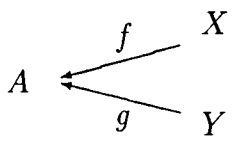


(2) It is the couniversal candidate product of X and Y. If (B, k, h) is any candidate product of X and Y then there is a unique map $\alpha : B \to A$ with $\alpha \cdot f = k$ and $\alpha \cdot g = h$

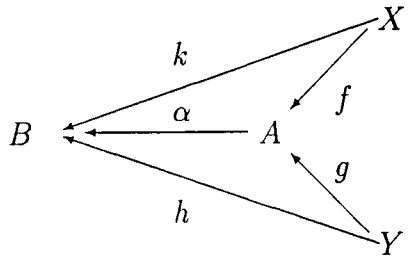


(ii) Dually (A, f, g) is a **coproduct** *of X and Y if*

(1) (A, f, g) is a candidate coproduct, i.e.,



(2) (A, f, g) *is the universal candidate coproduct of* X *and* Y*. Taht is, if* (B, k, h) *is any candidate product of* X *and* Y *then there is a unique map* $\alpha : A \to B$ *with* $f \cdot \alpha = k$ *and* $g \cdot \alpha = h$



Definition 2.5 *A* **functor** $F : \mathbf{X} \to \mathbf{Y}$ *consists of two maps* $F = (F_0, F_1)$ *where* $F_0 : \mathbf{X}_0 \to \mathbf{Y}_0$, $F_1 : \mathbf{X}_1 \to \mathbf{Y}_1$ *such that*

(i) F *is a map of directed graphs, i.e.,* $\delta_0(F_1(f)) = F_0(\delta_0(f)), \delta_1(F_1(f)) = F_0(\delta_1(f))$.

(ii) F *preserves composition:* $F_1(a \cdot b) = F_1(a) \cdot F_1(b)$ *whenever the composition is defined.*

(iii) F *preserves identity maps:* $F_1(i_X) = i_{F_0(X)}$

Let $\mathbf{C}$ be a directed graph and $\mathbf{X}$ a category.

Definition 2.6

(i) A $\mathbf{C}$*-diagram in* $\mathbf{X}$ *is a directed graph morphism* $F : \mathbf{C} \to \mathbf{X}$.

(ii) Let $F, G : \mathbf{C} \to \mathbf{X}$ *be* $\mathbf{C}$*-diagrams then* $\alpha : F \Rightarrow G$ *is a* **natural transformation** *if* $\alpha : \mathbf{C}_0 \to \mathbf{X}_1; C \mapsto \alpha_C$ *such that for any* $f : A \to B$ *of* $\mathbf{C}$ *the following diagram commutes:*

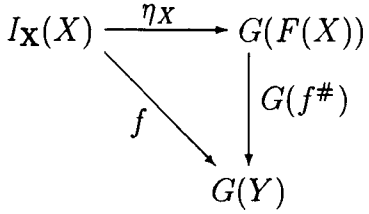
$$\begin{array}{ccc} F(A) & \xrightarrow{F(f)} & F(B) \\ {\scriptstyle \alpha(A)}\downarrow & & \downarrow{\scriptstyle \alpha(B)} \\ G(A) & \xrightarrow[G(f)]{} & G(B) \end{array}$$

Note that functors are maps from one category to another. As natural transformations are maps from one "functor" to another they provide a primitive polymorphism as they are maps defined on families of objects. An example is the transformation $reverse(S) : list(S) \to list(S)$ which satisfied the following if $f : S \to S'$ then $reverse(list\{f\}(x)) = list\{f\}(reverse(x))$[26]

$$\begin{array}{ccc} S & \xrightarrow{f} & S' \\ & & \\ \mathrm{list}(S) & \xrightarrow{\mathrm{list}(f)} & \mathrm{list}(S') \\ {\scriptstyle \mathrm{reverse}_S}\downarrow & & \downarrow{\scriptstyle \mathrm{reverse}_{S'}} \\ \mathrm{list}(S) & \xrightarrow[\mathrm{list}(f)]{} & \mathrm{list}(S') \end{array}$$

Let $\mathbf{X}$ and $\mathbf{Y}$ be a pair of categories.

Definition 2.7 *A pair of functors* $F : \mathbf{X} \to \mathbf{Y}$ *and* $G : \mathbf{Y} \to \mathbf{X}$ *with a natural transformation* $\eta : I_{\mathbf{X}} \to G \cdot F$ *is an* **adjunction** *if for each object* $Y \in \mathbf{Y}$ *and* $\mathbf{X}$*-morphism* $f : X \to G(Y)$ *there is a unique* $\mathbf{Y}$*-morphism* $f^{\#} : F(X) \to Y$ *such that the following triangle commutes.*



The cartesian product of objects X and Y in **Set** defined by $X \times Y = \{(x,y)|x \in X, y \in Y\}$ forms a bifunctor $_ \times _ : \mathbf{Set} \times \mathbf{Set} \rightarrow \mathbf{Set}$. This bifunctor is a right adjoint to the diagonal functor $\triangle : \mathbf{Set} \rightarrow \mathbf{Set} \times \mathbf{Set}$.

Another way of characterizing an adjunction $F \dashv G : \mathbf{X} \rightarrow \mathbf{Y}$ uses the induced natural isomorphism

$$\mathbf{Y}(F(X), Y) \cong \mathbf{X}(X, G(Y)).$$

Thus the product adjunction described above can be expressed as

$$\mathbf{Set} \times \mathbf{Set}(\triangle\ (\mathrm{C}), (\mathrm{A}, \mathrm{B})) \cong \mathbf{Set}(C, A \times B).$$

2.2 Semantics

This section introduces some elementary term categories which are a preliminary to the combinator theories introduced later. First the free term algebra is described and then the categorical form of the same structure is elucidated. Finally, list arithmetic is integrated into this structure.

2.2.1 Free Term Algebras (over Sets)

The concept of free term algebras introduced briefly here is basic to computation in all algebraic languages. The author refers to [17] for more detailed discussion on these concepts.

Definition 2.8 *A* **carrier** Ω *is a triple* (S, F, σ) *where* S *is a set of sorts,* F *is a set of functions, and* σ *is a* **signature** *that is a map*

$$\sigma : F \rightarrow list(S) \times S; f \mapsto ([s_1, \cdots, s_n], s_0)$$

Given a carrier Ω we may define a set $V(\Omega)$ of *types* and a set $\tau(\Omega)$ of *terms* associated with the carrier. The *types* $V(\Omega)$ are recursively defined by

(i) 1 is a type,

(ii) $s \in S$ is a type,

(iii) If v_1, v_2 are types then $v_1 \times v_2$ is a type.

Given a set of typed variables, where each variable x is assigned a sort $\tau(x)$ in S

$$\tau : X \rightarrow S; x \mapsto \tau(x)$$

the terms are defined recursively by

(i) x is a term of type $\tau(x)$ if $x \in X$,

(ii) If $\sigma(f) = ([s_1, \cdots, s_n], s_0)$ and $t_1, \cdots, t_n$ are terms of, respectively, type $s_1, \cdots, s_n$, then $f(t_1, \cdots, t_n)$ is a term of type s_0,

(iii) () is a term of type 1,

(iv) If t_1 has type v_1, t_2 has type v_2, then (t_1, t_2) has type $v_1 \times v_2$,

(iv) If t is a term of $v_0 \times v_1$ then $p_0(t)$ is term of type v_0 and $p_1(t)$ is term of type v_1.

Thus in the algebra Ω =(S,F,σ) each term has a type

$$\tau^* : T(\Omega) \rightarrow V(\Omega).$$

In order to complete the definiton of an algebraic theory we need to define the notion of a variable base. As we are dealing with many sorted theories it is important to keep careful account of the variable used in an equation or a proof (as empty sorts can produce invalid proofs).

A variable base is defined as follows:

(i) () is a variable base for 1,

(ii) If x is a variable of type s then x is a variable for s,

(iii) If $\vec{x_1}, \vec{x_2}$ are variable bases for v_1 and v_2, and $\vec{x_1}$ and $\vec{x_2}$ have no variables in common then $(\vec{x_1}, \vec{x_2})$ is a variable base for $v_1 \times v_2$.

Definition 2.9 *An equational theory is a pair* (Ω, E) *where* Ω *is the carrier of the algebra,* E *is a set of equations, that is, triples*

$$t_1 =_{\vec{x}} t_2$$

with t_1, t_2 *are terms and* $\vec{x}$ *a variable base such that* t_1, t_2 *are of the same type and* $\vec{x}$ *contains all variables of* t_1 *and* t_2.

The following are the inference rules for equational theories.

Reflexive:

$$\frac{[]}{t =_{\vec{x}} t}$$

where $\vec{x}$ is any variable base for t.

Symmmetric:

$$\frac{[t_1 =_{\vec{x}} t_2]}{t_2 =_{\vec{x}} t_1}$$

Transitive:

$$\frac{[t_1 =_{\vec{x}} t_2, t_2 =_{\vec{x}} t_3]}{t_1 =_{\vec{x}} t_3}$$

Pairing:

$$\frac{[t_1 =_{\vec{x}} t_2, s_1 =_{\vec{x}} s_2]}{(t_1, s_1) =_{\vec{x}} (t_2, s_2)}$$

Unpairing:

$$\frac{[(t_1, s_1) =_{\vec{x}} (t_2, s_2)]}{t_1 =_{\vec{x}} t_2}$$

$$\frac{[(t_1, s_1) =_{\vec{x}} (t_2, s_2)]}{s_1 =_{\vec{x}} s_2}$$

Projection:

$$\frac{[]}{p_0(t_1, t_2) =_{\vec{x}} t_1}$$

$$\frac{[]}{p_1(t_1, t_2) =_{\vec{x}} t_2}$$

Substitution:

$$\frac{[t_1 =_{\vec{x}} t_2, s_1 =_{\vec{y}} s_2]}{\sigma_{[\vec{x}:=s_1]}(t_1) =_{\vec{y}} \sigma_{[\vec{x}:=s_2]}(t_2)}$$

where $\tau^*(\vec{x}) = \tau^*(s_1) = \tau^*(s_2)$

An *algebra* A of $\mathbf{T} = (\Omega, E)$ is a set which is indexed over the sorts

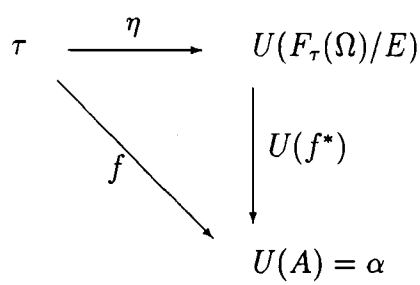
$$A_0 \xrightarrow{\alpha} S$$

together with maps for each $f \in F$

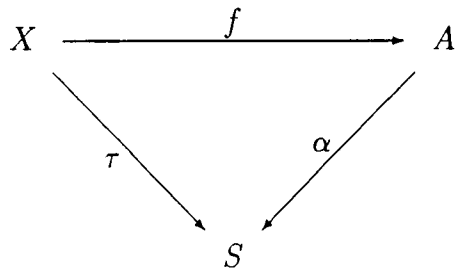
$$f : \alpha^{-1}(s_1) \times \cdots \times \alpha^{-1}(s_n) \longrightarrow \alpha^{-1}(s_0)$$

where $\sigma(f) = ([s_1, \cdots, s_n], s_0)$ which satisfies the equations E in the obvious manner.

The equalities in the algebra identify terms. This allows us to build free algebras $F_\tau(\Omega)/E$ on any indexed set of variables, $\tau : X \to S$, by using the least equivalence class which is forced by the axioms and equations. This gives the following universal diagram:



where



commutes, implying that $F(\tau) = F_\tau(\Omega)/E$ is left adjoint to U.

2.2.2 The Clone of a Theory T

A category is defined by the specification of its objects, maps, composition, and identities. Given $\mathbf{T} = (\Omega, E)$ then the following forms a category $\mathcal{C}(\mathbf{T})$, called the "clone of **T**".

Objects: Objects are types.

Maps: They are the formal maps $\{\vec{x} \mapsto t\}$ such that

$$\{\vec{x} \mapsto t\} : \tau(x) \rightarrow \tau(t)$$

where $\vec{x}$ is a variable base for the term t (i.e., $\vec{x}$ contains all free variables of t).

The names of variables do not matter: thus, the following two formal maps are equal:

$$\{\vec{x} \mapsto t\} = \{\vec{y} \mapsto t_{[\vec{x}:=\vec{y}]}\}$$

whenever $\tau(\vec{x}) = \tau(\vec{y})$.

Composition

$$\{\vec{x} \mapsto t_1\}.\{\vec{y} \mapsto t_2\}$$

is defined whenever $\tau(t_1) = \tau(\vec{y})$ and is

$$\{\vec{x} \mapsto t_{2[\vec{y}:=t_1]}\}$$

So composition is given by substitution.

Identities

$$\{\vec{x} \mapsto \vec{x}\}$$

It is easy but tedious to check that.

Proposition 2.1 $\mathcal{C}(\mathbf{T})$ *is a category with finite products.*

We note $(v_1 \times v_2, \{\vec{x} \mapsto p_0(\vec{x})\}, \{\vec{x} \mapsto p_1(\vec{x})\})$ is the product of v_1 and v_2 and $() : v_1 \to 1$ defines the final map.

2.2.3 List Arithmetic Equational Theories

In the previous section we saw how a category can be constructed from a many-sorted equational theory. In this section we wish to show how list-arithmetic can be added to the setting.

This involves three steps. First, we must add new types corresponding to list construction. Second, we must add the term constructions which are associated with list construction. To do this it is actually necessary to use abstracted maps in the terms. This is a form of quantification. Third, we must add the equations peculiar to list construction.

Let $\mathbf{T} = (\Omega, E)$ be an equational theory then we shall form a category $\mathcal{L}(\mathbf{T})$ by defining the types, terms, and equations in a similar manner to the above construction.

Objects/Types

(i) 1 is a type,

(ii) $s \in S$ is a type,

(iii) If v_1, v_2 are types, then $v_1 \times v_2$ is a type,

(iv) If v is a type, then $list(v)$ is a type.

Terms

(i) () is a term of type 1,

(ii) nil is a term of type $list(v)$ for any type v,

(iii) If t is a term of type v and t' is a term of type $list(v)$ then $cons(t, t')$ is a term of type $list(v)$,

(iv) If $f \in F$ has signature $([s_1, \cdots, s_n], s_0)$ and t_i is a term of type s_i then $f(t_1, \cdots, t_n)$ is a term of type s_0,

(v) If $\{\vec{x} \mapsto t_0\} : v_1 \to v_2$ and $\{(\vec{y}, \vec{z}) \mapsto t_1\} : v_0 \times v_2 \to v_2$, t_2 is a term of type $list(v_2)$, and t_3 is a term of type v_1 then

$$act\{\vec{x} \mapsto t_0, (\vec{y}, \vec{z}) \mapsto t_1\}(t_2, t_3)$$

is a term of type v_2.

The variables of an abstract map in a term are regarded as being bound. This means that one can change their names without changing the meaning of the term.

The category $\mathcal{L}(\mathbf{T})$ is then formed by taking the maps to be abstracted maps of this term algebra with composition substitution as usual.

It remains only to specify the special axioms which the list-arithmetic must satisfy. These are

[L.1] $act\{\vec{x} \mapsto t_0, (\vec{y}, \vec{z}) \mapsto t_1\}(nil, t_3) = t_{0[\vec{x}:=t_3]}$

[L.2] $act\{\vec{x} \mapsto t_0, (\vec{y}, \vec{z}) \mapsto t_1\}(cons(t_2, t_2'), t_3) = t_{1[\vec{y}:=t_2, \vec{z}:=act\{\vec{x} \mapsto t_0, (\vec{y}, \vec{z}) \mapsto t_1\}(t_2', t_3)]}$

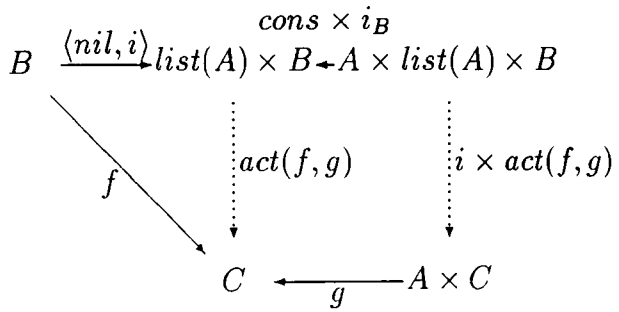
[L.3]

$$\frac{\begin{bmatrix} t_{[x':=nil]} =_{\vec{x}} t_0, \\ t_{[x':=cons(\vec{y},w)]} =_{((\vec{y},w),\vec{x})} t_{1[\vec{z}:=t_{[x':=w]}]} \end{bmatrix}}{act\{\vec{x} \mapsto t_0, (\vec{y}, \vec{z}) \mapsto t_1\}(x', \vec{x}) =_{(x',\vec{x})} t(x', \vec{x})}$$

For example, we may use the act construction to perform an "append" as follows:

$$\begin{aligned} & act\{x \mapsto x, (y, z) \mapsto cons(y, z)\}(cons(a_1, nil)cons(a_2, nil)) \\ =_{[L.2]} \; & cons(a_1, act\{x \mapsto x\}, (y, z) \mapsto cons(y, z)\}(nil, cons(a_2, nil))) \\ =_{[L.1]} \; & cons(a_1, cons(a_2, nil)) \end{aligned}$$

The axiom [$L.3$] expresses the uniqueness of the arrow $act(f,g)$ in the diagram below:



The list-arithmetic introduced here is parameterized. That is the comparison map $act(f,g)$ has domain the product of a list with another object which is the parameter. This is descibed further in [17].

It is also possible in a similar manner to add other constructs to the setting, for example, exponentiation. The result gives a notation which is somewhat more clumsy than standard λ-calculus notation although expressively equivalent. The relevant axioms are

[E.1] $eval(curry\{(\vec{x},\vec{y}) \mapsto t\}(t_1,t_2) = t_{[\vec{x}:=t_1,\vec{y}:=t_2]}$

[E.2]

$$\frac{eval(t_1,\vec{x}) = h}{t_1 =_{\vec{y}} curry\{(\vec{y},\vec{x}) \mapsto h\}(\vec{y})}$$

where $curry\{(\vec{x},\vec{y}) \mapsto t_0\}$ and $eval$ have to be introduced as terms with the natural typing.

2.2.4 Translation to Combinators

The use of variables in this notation, while being clear in the simplest situation becomes increasingly cumbersome as more complex constructions are added. In particular, it should be noted that the substitutions required for list-arithmetic are rather unwieldly.

Categorical notation, however, does not use variables. The elimination of variables, which is an important aspect of combinators, can actually increase the expressive power. In the next chapter we shall formally introduce a polymorphic categorical notation which we call categorical combinators. To finish this section we remind the reader that it is relatively simple to translate from the equational notation introduced into categorical notation. We shall illustrate this by sketching how the translation for $\mathcal{C}(\mathbf{T})$ works.

The translation starts with an abstract map

$$\{\vec{x} \mapsto t\}$$

and proceeds as follows.

(i) If t is a variable x and $\vec{x} = (\vec{x_0}, \vec{x_1})$ and $x \in \vec{x_i}$ then

$$\mathcal{T}(\{\vec{x_0}, \vec{x_1}) \mapsto x\}) = p_i.\mathcal{T}(\{\vec{x_i} \mapsto x\})$$

(ii) $\mathcal{T}(\{x \mapsto x\}) = i$,

(iii) $\mathcal{T}(\{\vec{x} \mapsto f(t_1, \cdots, t_n)\}) = \langle \mathcal{T}(\vec{x} \mapsto t_1), \cdots, \mathcal{T}(\vec{x} \mapsto t_n) \rangle.f$

So for example,

$$\mathcal{T}(\{(x,y) \mapsto f(x,g(y,x))\}) = \langle p_0, i, \langle p_1.i, p_0.i\rangle.g\rangle.f$$

That this translation process is actually an equivalence of categories is another story. For further details see [19].

CHAPTER 3

Categorical Combinator Theories

Categorical combinator theories can model many computational settings. The theories provide an organizing framework in which many programming language features can be uniformly discussed. This chapter presents both the concepts of categorical combinator theories and the use of these concepts in organizing different categories.

A categorical combinator theory C consists of the following data:

- a set $\mathcal{K}$ of *classes*
- a set $\mathcal{F}$ of *type constructors* with a signature $\Sigma_{\mathcal{F}} : \mathcal{F} \rightarrow \mathit{list}(\mathcal{K}) \times \mathcal{K}$
- a set $\mathcal{C}$ of (categorical) *combinators* with combinator signatures $\Sigma_{\mathcal{C}}$.
- a set $\mathcal{E}$ of inference rules.

These data: classes, type constructors, combinators, and inference rules correspond roughly to the following concepts: categories, functors, natural transformations, and categorical axioms respectively. In fact a model for a categorical combinator theory associates a category with each class in $\mathcal{K}$, an object operation (not necessarily a functor) for each symbol in $\mathcal{F}$, a parameterized class of maps for each combinator in $\mathcal{C}$, and an exiom to each inference rule of $\mathcal{E}$.

3.1 Type Expressions and Classes

From the classes $\mathcal{K}$ and the type constructors $\mathcal{F}$, we can construct *type expressions* as defined below.

Definition 3.1 *A* **type expression**, T, *of the triple* $\langle \mathcal{K}, \mathcal{F}, \Sigma_{\mathcal{F}} \rangle$ *and its* **class**, $\tau_{\mathcal{K}}(T)$ *is given recursively by*

(i) each class $\mathbf{X} \in \mathcal{K}$ *has a stock of* type variables $X_1, X_2, \cdots$, *each of which has class* $\mathbf{X}$, *that is,* $\tau_{\mathcal{K}}(X_i) = \mathbf{X}$.

(ii) If $F \in \mathcal{F}$ *with* $\Sigma_{\mathcal{F}}(F) = ([\mathbf{X}_1, \cdots, \mathbf{X}_n], \mathbf{X}_0)$ *and* $T_1, \cdots, T_n$ *are type expressions with* $\tau_{\mathcal{K}}(T_i) = \mathbf{X}_i$ *then* $F(T_1, \cdots, T_n)$ *is a type expression with* $\tau_{\mathcal{K}}(F(T_1, \cdots, T_n)) = \mathbf{X}_0$

Thus, the type expressions are simple terms of a many sorted algebra where the sorts are referred to as classes and the operations as type constructors. A type variable can be instantiated to a type expression. We shall denote the type expressions of $\mathbf{C}$ by $\mathcal{T}\mathbf{C}$. When context $\mathbf{C}$ is clear we shall omit it, referring to $\mathcal{T}\mathbf{C}$ as $\mathcal{T}$.

Example 3.1 (type constructors for PLOC and NCC)

(i) The type constructors for NCC, a cartesian categories with a parameterized natural number object, are given by:

$$\mathcal{K} = \{\mathbf{X}\}$$

$$\mathcal{F} = \{1, \times, \Rightarrow, N\}$$

$$\Sigma_{\mathcal{F}} : \mathcal{F} \to list(\mathcal{K}) \times \mathcal{K}; \quad \begin{array}{l} 1 \mapsto ([], \mathbf{X}) \\ \times \mapsto ([\mathbf{X}, \mathbf{X}], \mathbf{X}) \\ \Rightarrow \mapsto ([\mathbf{X}, \mathbf{X}], \mathbf{X}) \\ N \mapsto ([], \mathbf{X}) \end{array}$$

(ii) The type constructors for a prelocos, PLOC, are given by:

$$\mathcal{K} = \{\mathbf{X}\}$$

$$\mathcal{F} = \{1, \times, +, list\}$$

$$\Sigma_{\mathcal{F}} : \mathcal{F} \to list(\mathcal{K}) \times \mathcal{K}; \quad \begin{array}{l} 1 \mapsto ([], \mathbf{X}) \\ \times \mapsto ([\mathbf{X}, \mathbf{X}], \mathbf{X}) \\ + \mapsto ([\mathbf{X}, \mathbf{X}], \mathbf{X}) \\ list \mapsto ([\mathbf{X}], \mathbf{X}) \end{array}$$

□

If X, Y, and Z are type variables of class $\mathbf{X}$ then the following

$$list(X) \quad X \times Y \quad X + Y \quad X \Rightarrow Y \quad list(X) + Y \quad Y \Rightarrow (list(X) \times Z)$$

are type expressions (where the obvious infix notation has been used.)

3.2 Combinator Expressions

The signature, $\Sigma_{\mathcal{C}}$, of the combinators $\mathcal{C}$ is given by a map

$$\Sigma_{\mathcal{C}} : \mathcal{C} \to list(\mathcal{T}^2) \times \mathcal{T}^2 : \alpha \mapsto ([(D_1, C_1), \cdots, (D_n, C_n)], (D_0, C_0))$$

where $\mathcal{T}$ is the type expressions and $\tau_K(D_i) = \tau_K(C_i)$. The type expression pairs (D_i, C_i) are called map types and shall usually be written $D_i \to C_i$, where D_i is the domain of the map type and C_i the codomain.

In the following definition, the map type of a combinator $\tau_{\mathcal{C}}(\alpha) = T_1 \to T_2$ is expressed more conventionally by the notation $\alpha : T_1 \to T_2$.

Definition 3.2 *A categorical* **combinator expression** *and its* **map signature** *is defined recursively by*

(i) For each type variable X of class $\mathbf{X} \in \mathcal{K}$ there is an identity combinator i_X with type signature $i_X : X \to X$

(ii) For each pair map type $T_1 \to T_2$ there is a stock of combinator variables

$$x_1, \cdots, x_n, \cdots$$

each of which has type signature $x_i : T_1 \to T_2$

(iii) If c is a combinator expression and σ is a substitution of the type variables of c, then c_σ is a combinator expression with $\tau_{\mathcal{C}}(c_\sigma) = \tau_{\mathcal{C}}(c)_\sigma$

(iv) If $\alpha \in \mathcal{C}$ and $\Sigma_{\mathcal{C}}(\alpha) = ([T_1 \to T_1', \cdots, T_n \to T_n'], T_0 \to T_0')$ and $\beta_1, \cdots, \beta_n$ are combinator expressions with type signature $\tau_{\mathcal{C}}(\beta_i) = T_i \to T_i'$,

then $\alpha(\beta_1, \cdots, \beta_n)$ *is a combinator expression with map type* $\alpha(\beta_1, \cdots, \beta_n) : T_0 \to T_0'$

(v) If $c_1 : T_1 \to T_2$ *and* $c_2 : T_2 \to T_3$ *are combinators, then* $c_1 \cdot c_2$ *is a combinator expression and* $c_1.c_2 : T_1 \to T_3$.

Example 3.2 (Signatures for NCC)

For a cartesian categories with a parameterized natural number object, the set of basic combinators is

$$\mathcal{C} = \{i, !, p_0, p_1, \langle _ , _ \rangle\}, eval, curry, 0, s, n$$

with signatures as follows:

identity map	$\Sigma_{\mathcal{C}}(i) = ([], X \to X)$
terminal map	$\Sigma_{\mathcal{C}}(!) = ([], X \to 1)$
first product projection	$\Sigma_{\mathcal{C}}(p_0) = ([], X \times Y \to X)$
second product projection	$\Sigma_{\mathcal{C}}(p_1) = ([], X \times Y \to Y)$
product pairing	$\Sigma_{\mathcal{C}}(\langle _ , _ \rangle) = ([Z \to X, Z \to Y], Z \to X \times Y)$
evaluation	$\Sigma_{\mathcal{C}}(eval) = ([], X \times (X \Rightarrow Y) \to Y)$
currying	$\Sigma_{\mathcal{C}}(curry(_)) = ([X \times Y \to Z], Y \to X \Rightarrow Z)$
zero	$\Sigma_{\mathcal{C}}(0) = ([], 1 \longrightarrow N)$
successor	$\Sigma_{\mathcal{C}}(s) = ([], N \longrightarrow N)$
number action	$\Sigma_{\mathcal{C}}(n) = ([X \longrightarrow Y, Y \longrightarrow Y], N_\sigma \times X_\sigma \longrightarrow Y_\sigma)$

□

Example 3.3 (Signatures for PLOC)

For a prelocos[17] the set of basic combinators is

$$\mathcal{C} = \{i, !, p_0, p_1, b_0, b_1, d_0, d_1, cons, split, act, \langle _\ ,\ _ \rangle, \langle _\ ;\ _ \rangle\}$$

with signatures as follows:

$$\begin{aligned}
\text{identity map} \quad & \Sigma_{\mathcal{C}}(i) = ([], X \to X) \\
\text{terminal map} \quad & \Sigma_{\mathcal{C}}(!) = ([], X \to 1) \\
\text{first product projection} \quad & \Sigma_{\mathcal{C}}(p_0) = ([], X \times Y \to X) \\
\text{second product projection} \quad & \Sigma_{\mathcal{C}}(p_1) = ([], X \times Y \to Y) \\
\text{first coproduct embedding} \quad & \Sigma_{\mathcal{C}}(b_0) = ([], X \to X + Y) \\
\text{second coproduct embedding} \quad & \Sigma_{\mathcal{C}}(b_1) = ([], Y \to X + Y) \\
\text{first distributor} \quad & \Sigma_{\mathcal{C}}(d_0) = ([], (X + Y) \times Z \to (X \times Z) + (Y \times Z)) \\
\text{second distributor} \quad & \Sigma_{\mathcal{C}}(d_1) = ([], Z \times (X + Y) \to (Z \times X) + (Z \times Y)) \\
\text{list constructor} \quad & \Sigma_{\mathcal{C}}(cons) = ([], X \times list(X) \to list(X)) \\
\text{list destructor} \quad & \Sigma_{\mathcal{C}}(split) = ([], list(X) \to 1 + X \times list(X)) \\
\text{primitive recursion} \quad & \Sigma_{\mathcal{C}}(act) = ([Y \to Z, X \times Z \to Z], list(X) \times Y \to Z) \\
\text{product pairing} \quad & \Sigma_{\mathcal{C}}(\langle _\ ,\ _ \rangle) = ([Z \to X, Z \to Y], Z \to X \times Y) \\
\text{coproduct cases} \quad & \Sigma_{\mathcal{C}}(\langle _\ ;\ _ \rangle) = ([X \to Z, Y \to Z], X + Y \to Z)
\end{aligned}$$

□

Note that it is often more convenient and expressive to write:

$$\langle Z \to X, Z \to Y \rangle : Z \to X \times Y$$

instead of

$$\Sigma_c(\langle -, - \rangle) = ([Z \to X, Z \to Y], Z \to X \times Y).$$

The type signature of the identity combinator $i_X : X \to X$ is intended to be indifferent to the object type within its class **X**. This polymorphic feature enables the identity combinator to be used in any context. That is, the type variable, X, can be instantiated to a closed type expression or indeed to any type variable, thus

$$\sigma_{[X:=Y]}$$

$$(i_X)_{[X:=Y]} = i_Y$$

and if Y is an instance — e.g., the terminal object, 1, then

$$\sigma_{[X:=1]}(i_X) = i_1.$$

As type substitution is often required, we shall annotate the combinators by type substitutions.

Example 3.4 (annotated combinator expressions)

(i) Appending two lists (or adding numbers represented by $list(1)$) is defined in prelocos as the unannotated combinator expression

$$act(i, cons)$$

Using the signatures defined above, the annotations required are

$$act_{[Y:=list(X),Z:=list(X)]}((i_X)_{[X:=list(X)]}, cons) : list(X) \times list(X) \to list(X)$$

(ii) Appending two one-element lists may be written as the unannotated combinator expression

$$\langle !, ! \cdot nil \rangle \cdot cons \cdot \langle i, i \rangle \cdot act(i, cons).$$

The annotated expression for appending two one-element lists is

$$\langle !_{[X:=Z]}, !_{[X:=Z]} \cdot nil_{[X:=1]} \rangle \cdot cons_{[X:=1]} \cdot \langle (i_X)_{[X:=list(1)]}, (i_X)_{[X:=list(1)]} \rangle_{[Z,X,Y:=list(1)]} \cdot$$

$$act_{[X:=1,Y:=list(1),Z:=list(1)]}((i_X)_{[X:=list(1)]}, cons_{[X:=1]})$$

□

It is clear that the annotation considerably complicates the combinator expressions and it would be nice if we could get rid of it completely. Fortunately, this is possible. We may simply follow the flavor of Milner's scheme [66] for automatically calculating the most general type substitutions required to ensure that the map, as written, can be consistently typed. This means that the expressions shown above can be written and soundly interpreted in an unannotated form

$$act(i, cons)$$

and

$$\langle !, !.\ nil \rangle .\ cons\ .\langle i, i \rangle .\ act(i, cons)\ .$$

The type matches required are obtained through unification, and the expression only makes sense if all the unifications, which are required, work. The unification process automatically provides a most general annotation called the principle typing and its implementation is discussed in the next section.

3.3 Type Checking

The first problem in implementing any programming language is simply to recognize a syntactically correct expression. In polymorphic functional languages

the actual syntax is relatively simple, the problem is to ensure that the expression can be typed. This is done following Milner [66] by caculating the most general typing which is possible for the expression (called the principle typing).

Below we outline a simple system for calculating the principle type of a categorical combinator expression. The system uses unification. As we are using variables which have assigned classes, it must be remembered that two variables belonging to different classes cannot be unified. Otherwise the unification is standard.

The principle map type is calculated as follows:

(i) The principle map type of the identity map of a class is

$$i : X \longrightarrow X$$

where X is a variable of that class.

(ii) If c_1 and c_2 are combinator expressions with principle types

$$c_1 : T_1 \longrightarrow T_1'$$

$$c_2 : T_2 \longrightarrow T_2'$$

where the two map types have no variables in common, then $c_1 \cdot c_2$ has principle map type

$$c_1 \cdot c_2 : (T_1 \longrightarrow T_2')_{unifier(T_1', T_2)}$$

(iii) If γ is a combinator with signature

$$\gamma(T_1 \rightarrow T_1', \cdots, T_n \rightarrow T_n') : T_0 \rightarrow T_0'$$

and

$$c_1, \cdots, c_n$$

are combinators with principle types

$$c_1 : S_i \longrightarrow S_i'$$

where variables of the map type of any two distinct combinators are disjoint, then $\gamma(c_1, \cdots, c_n)$ has principle type

$$\gamma(c_1, \cdots, c_n) : (T_0 \longrightarrow T_0')_{unifier([T_1 \to T_1', \cdots, T_n \to T_n'], [S_1 \to S_1', \cdots, S_n \to S_n'])}$$

Given any valid annotation of a combinator expression, it is clear that it must be a substitution of this principle typing. This can be shown by an easy structural induction on the components.

3.4 Inference rules

The final part of a combinator theory C, are the axioms. These we style as an equational theory with the usual proof rules, except for one small modification: namely, the variables (and their typing) used in a proof must be carried along. Given that this can be done automatically, it greatly promotes the possibility of automating proofs in specific systems.

Definition 3.3

(i) An **equation**, $c_1 =_X c_2$, *is a pair of combinator expressions,* c_1 *and* c_2, *of the same map type together with a set of combinator variables,* X, *which*

includes those used in the expressions.[1]

(ii) An equational **inference rule** *is a pair*

$$\frac{c_1 =_{X_1} c'_1, \ldots, c_n =_{X_n} c'_n}{c_0 =_{X_0} c'_0}$$

in which the premise is a set of equations and the consequent is a single equation. When the premise is empty, the inference rule is an **axiom** *and it will be written as a single equation.*

(iii) The **inference rules** $\mathcal{A}$ *of a combinator theory,* C, *consist of a set of inference rules.*

(iv) The valid inferences of a combinator theory are those whose consequent can be proved from the premise equations using the inference rules of, A, *and the axioms and inference rules listed below:*

[HE.1] Reflexive:

$$c = c$$

[HE.2] Symmetric:

$$\frac{c_1 =_X c_2}{c_2 =_X c_1}$$

[HE.3] Transitive:

$$\frac{c_1 =_X c_2, c_2 =_X c_3}{c_1 =_X c_3}$$

[1]We shall use the convention of omitting the variable base when it contains exactly the variables in the expressions.

[HE.4] Associative composition:

$$c_1.(c_2.c_3) = (c_1.c_2).c_3$$

[HE.5] Composition identity:

$$c.i = c$$

$$i.c = c$$

[HE.6] Substitution:

$$\frac{c_1 =_X c'_1, .., c_n =_X c'_n, w =_{X'} w'}{w_{[x_1:=c_1,...,x_n:=c_n]} =_X w'_{[x_1:=c'_1,...,x_n:=c'_n]}}$$

where $X' = \{x_1, ..., x_n\}$.

We can now complete the specification of our examples by adding the axioms.

Example 3.5 (Axiom systems)

(i) The axioms of **NCC**, the combinator theory of a cartesian closed category, are:

- Terminal:

$$f.! =!$$

- Products:

$$\langle f, g\rangle.p_0 = f, \langle f, g\rangle.p_1 = g, \langle x.p_0, x.p_1\rangle = x$$

- Exponentiation:

$$\langle x, y.\, curry(z)\rangle.\, eval = \langle x, y\rangle.z$$

$$\frac{\langle p_0, p_1.g\rangle.\, eval = z}{curry(z) = g}.$$

- Natural number:

$$\langle x,y\rangle.p_0 = x,\ \langle x,y\rangle.p_1 = y,\ \langle z.p_0, z.p_1\rangle = z$$

$$z.! =!,\ \langle x.0,y\rangle.n(f,g) = y.f,\ \langle x.s,y\rangle.n(f,g) = \langle x,y\rangle.n(f,g).g$$

$$\frac{\langle !.0,i\rangle.z = f,\ \langle p_0.s,p_1\rangle.z = z.g}{z = n(f,g)}.$$

(ii) The axioms of **PLOC**, the combinator theory of a prelocos, are as follows:

- Final object:

$$f.! =!$$

- Products:

$$\langle f,g\rangle.p_0 = f, \langle f,g\rangle.p_1 = g, \langle x.p_0, x.p_1\rangle = x$$

- Coproducts:

$$b_0.\langle h;k\rangle = h, b_1.\langle h;k\rangle = k, \langle b_0.h; b_1.h\rangle = h$$

- Distributors:

$$\langle x.b_0, y\rangle.d_0 = \langle x,y\rangle.b_0,\quad \langle x.b_1,y\rangle.d_0 = \langle x,y\rangle.b_1$$

$$\langle x, y.b_0\rangle.d_1 = \langle x,y\rangle.b_0,\quad \langle x,y.b_1\rangle.d_1 = \langle x,y\rangle.b_1$$

$$\langle\langle p_0.b_0, p_1\rangle; \langle p_0.b_1, p_1\rangle\rangle.d_0 = i,\quad d_0.\langle\langle p_0.b_0, p_1\rangle; \langle p_0.b_1, p_1\rangle\rangle = i$$

$$\langle\langle p_0, p_1.b_0\rangle; \langle p_0, p_1.b_1\rangle\rangle.d_1 = i,\quad d_1.\langle\langle p_0, p_1.b_0\rangle; \langle p_0, p_1.b_1\rangle\rangle = i$$

- List actions:

$$\langle\langle u, w\rangle.\, cons, x\rangle.\, act(f,g) = \langle u, \langle w, x\rangle.\, act(f,g)\rangle.g$$

$$\langle w.\, nil, x\rangle.\, act(f,g) = x.f$$

$$\frac{\langle p_0.\, cons, p_1\rangle.z = \langle p_0.p_0, \langle p_0.p_1, p_1\rangle.z\rangle.g, \langle !.\, nil, i\rangle.z = f}{z = act(f,g)}.$$

□

CHAPTER 4

Categorical Combinator Reduction System

This chapter discusses how an operational semantics can be associated with a categorical combinator theory. The operational semantics is a reduction system which is obtained by orienting some of the valid identities of the theory. In addition the reduction system must satisfy certain properties which are described below which make it suitable for computation.

4.1 A Description of the System

In a categorical combinator reduction system the primitive combinators of the underlying categorical combinator theory are classified into three distinct groups:

Active: The *active* combinators are those which initiate a rewriting. When a reduction is completed all such combinators (which have a context) will have been removed.

Distributive: The *distributive* combinators are those which satisfy the (left) distributive law:

$$z.\delta(x_1, ..., x_n) = \delta(z.x_1, ..., z.x_n).$$

Constructives: These are the remainder of the combinators.

Both cartesian closed categories and the list arithmetic setting of a prelocos can be turned into combinator reduction systems by classifying the combinators and adding reduction rules.

Example 4.1 (Examples of combinator reduction systems)

(i) The active combinators of **NCC** are:

$$n, \quad p_0, \quad p_1, \quad eval\,.$$

The distributive combinators are:

$$\langle\, -, - \,\rangle, \quad !\,.$$

The constructive combinators are:

$$0, \quad s, \quad curry\,.$$

The reduction rules for an **NCC** operational semantics are:

$$\begin{aligned}
\langle x, y\rangle.p_0 &\Longrightarrow x \\
\langle x, y\rangle.p_1 &\Longrightarrow y \\
\langle x, y.\, curry(z)\rangle.\, eval &\Longrightarrow \langle x, y\rangle.z \\
\langle x.0, y\rangle.n(f,g) &\Longrightarrow y.f \\
\langle x.s, y\rangle.n(f,g) &\Longrightarrow \langle x, y\rangle.n(f,g).g\,.
\end{aligned}$$

(ii) The active combinators of **PLOC** are:

$$act,\ \ d_0,\ \ d_1,\ \ p_0,\ \ p_1,\ \ split,\ \ \langle\ -;-\ \rangle\ .$$

The distributive combinators are:

$$\langle\ -,-\ \rangle,\ \ !\ .$$

The constructive combinators are:

$$nil,\ \ cons,\ \ b_0,\ \ b_1.$$

The reduction rules are:

$$\begin{aligned}
\langle x,y\rangle.p_0 &\Longrightarrow x\\
\langle x,y\rangle.p_1 &\Longrightarrow y\\
z.b_0.\langle x;y\rangle &\Longrightarrow z.x\\
z.b_1.\langle x;y\rangle &\Longrightarrow z.y\\
\langle x.b_0,y\rangle.d_0 &\Longrightarrow \langle x,y\rangle.b_0\\
\langle x.b_1,y\rangle.d_0 &\Longrightarrow \langle x,y\rangle.b_1\\
\langle x,y.b_0\rangle.d_1 &\Longrightarrow \langle x,y\rangle.b_0\\
\langle x,y.b_1\rangle.d_1 &\Longrightarrow \langle x,y\rangle.b_1\\
\langle\langle x,y\rangle.\,cons,z\rangle.\,act(f,g) &\Longrightarrow \langle x,\langle y,z\rangle.\,act(f,g)\rangle.f\\
\langle !.\,nil,z\rangle.\,act(f,g) &\Longrightarrow z.g\\
!.\,nil\,.\,split &\Longrightarrow !.b_0\\
x.\,cons\,.\,split &\Longrightarrow x.b_1\ .
\end{aligned}$$

□

4.2 Classes of Expressions

This section defines classes of combinators which are used to analyze the properties of the reduction system. Some classes are only of special interest when a particular method of evaluation is considered. For example a weak-canonical is central to the evaluation style introduced by Hagino[42] but is not used in the by-value machine.

Definition 4.1

(i) A combinator expression is a **ground** *expression if it contains no (combinator) variables.*

(ii) A combinator is respectively, **canonical**, **weakly canonical**, *or* **closed**, *if it is recursively defined by:*

[C.1] If δ is distributive and $c_1, ..., c_n$ are any one of the above then $\delta(c_1, \cdots, c_n)$ also is. Terminals, in particular, always satisfy this condition.

[C.2] If c is respectively canonical, weakly canonical, or closed, $f_1, \cdots, f_n$ are ground, and β is respectively constructive, not active, or any combinator, then $c.\beta(f_1, \cdots, f_n)$ is respectively canonical, weakly canonical, or closed.

(iii) A combinator is **fully canonical** *if it is canonical and only uses primitive constructives which have no arguments.*

It is clear that a fully canonical combinator is canonical, a canonical is weakly canonical, and a weakly canonical is a closed combinator expression. Below are some canonical expressions for the example systems above.

Example 4.2 (Canonical expressions)

(i) The following are canonical combinators in **NCC**:

$$\langle !, ! \rangle, \quad !.0.s.s.s, \quad !.\, curry(p_0.s)$$

All except the last is fully canonical.

(ii) The following are canonical combinators in **PLOC**:

$$!, \quad !.b_1.b_0, \quad \langle !, !.b_0 \rangle.b_1, \quad \langle !.b_0, !.\, nil \rangle.\, cons, \quad \langle !, \langle !, !.\, nil \rangle.\, cons \rangle.\, cons$$

All canonicals in **PLOC** are fully canonical.

□

Notice that the last expression of 4.2 (i) is not fully canonical as *curry* has the argument, $p_0.s$, which is unevaluated in the sense that it contains an active. We call $p_0.s$ a *contextless* subexpression. In general, a subexpression is contextless when its occurrence is in an argument of a non-distributive combinator. Fully canonical expressions are distinguished by not having any contextless subexpressions.

Lemma 4.1 *If x is any combinator and c is a closed combinator, then $x.c = c$.*

Proof. Considering **[C.1]**: terminal combinators are left absorbing, and if $c_1, .., c_n$ are left absorbing and δ is distributive then

$$x.\delta(c_1, ..., c_n) = \delta(x.c_1, ..., x.c_n) = \delta(c_1, ..., c_n)$$

so that the constructed expression is left absorbing. Considering **[C.2]**: if c is left absorbing then certainly the composition $c.g$ is left absorbing.

□

Thus, closed combinators are left absorbing. This knowledge can be rather useful in an implementation, as there is no point in forming a closure whose code is closed. For example, when processing a closure whose code is a distributive combinator expression, the effect is to form the closure of the context with each argument of the distributive. However, this is unnecessary for any argument which is already closed.

Example 4.3 (Closed expressions)

(i) The following are closed combinator expressions in **NCC**:

$$\langle !.0.s.s, !.0.s\rangle.n(i,s), \quad \langle !.0.s.s.s, !\rangle.\langle p_0, p_1.\, curry(p_0.s)\rangle.\langle eval, p_1\rangle.$$

The last is also a weak canonical.

(ii) The following are closed combinator expressions in the **PLOC** semantics:

$$!, \quad !.b_1.\langle b_0; b_1\rangle, \quad \langle !, !.b_0\rangle.p_1,$$

$$\langle\langle !, \langle !, !.\, nil\rangle.\, cons\rangle.\, cons, \langle !, !.\, nil\rangle.\, cons\rangle.\, act(i, cons).$$

□

When a combinator is evaluated lazily (i.e. from the right end) it is important to know when the right segment cannot be evaluated any further. To capture this notion we have the following definition.

Definition 4.2 *A combinator is* **right canonical** *if it is recursively defined by:*

[D.1] if $g_1, \cdots, g_n$ are ground, and β is constructive then $\beta(g_1, \cdots, g_n)$ is right canonical.

[D.2] if $d_1, \cdots, d_n$ are right canonical and δ is distributive then $\delta(d_1, \cdots, d_n)$ is right canonical.

[D.3] If d_1 and d_2 are right canonical then $d_1.d_2$ is right canonical.

The idea of right-canonicals is that when they are composed with canonicals then the result can be turned into a canonical by using distributions only. As we shall shortly discover this means that the composition of a canonical with a right canonical is a precanonical.

4.3 The Reduction Rules

The idea of a reduction is that it should reduce a closed combinator to a canonical. Therefore, it is desirable for structurally distinct canonicals to be distinct in the underlying combinator theory. Unfortunately, this is too much to

demand, as canonicals can contain arbitrary contextless subexpressions. Instead, we will demand that fully canonical expressions are *separated* by the combinator theory, that is two such expressions cannot be proven equal.

It is clear that, as every active in a canonical must be contextless, the reduction in some sense reduces to the problem of eliminating those actives with context from the expression. The *context* of an occurrence is formalized as follows:

Definition 4.3

(i) The **context** *of an occurrence of an active primitive combinator subexpression, $\alpha(c_1, ..., c_n)$, in a closed term, c is defined as follows:*

[Cx.1] If c associatively matches $c'.\alpha(c_1, ..., c_n).c''$ the occurrence has context c'.

[Cx.2] If c associatively matches $c'.\delta(c_1, ..., c_n).c''$, in which δ is distributive, and the occurrence is in c_i which is closed, then the occurrence has the same context as in c_i.

[Cx.3] If c associatively matches $c'.\delta(c_1, ..., c_n).c''$, in which δ is distributive, and the occurrence is in c_i, then the occurrence has the same context as the occurrence in $c'.c_i$.

A context is **direct** *if only the first two rules of the definition are used, it is* **indirect** *if the last rule is used.*

(ii) An occurrence is said to be **contextless** *if it has no context.*

Notice how the fact that the composition can be associated in various different ways is overcome by using associative matching. A closed expression in which every active occurrence is contextless is said to be *precanonical.*

Lemma 4.2

(i) A combinator expression c is precanonical if and only if performing all distributive rewritings on c gives a canonical.

(ii) If an occurrence has an indirect precanonical context, then there are distributions which can turn the context into a direct canonical context.

(iii) Any closed expression is either precanonical or contains an occurrence of an active whose context is precanonical.

Proof.

(i) As a canonical contains no actives with context, reversing distributions cannot introduce actives with context. Thus, any combinator which can be reduced to a canonical using just distributions must be.

Only a finite number of applications of distributions are possible on any expression, and any order of application will eventually give the same result. However, a precanonical in which no distributions can be performed is clearly a canonical.

(ii) The context can be turned into a direct precanonical context by performing all the distributions encountered in the derivation of the fact that it is a context.

(iii) If the expression is not precanonical, consider the leftmost active with a context. Its context cannot have active occurrences with context as they would appear further to the left. Thus, it must be precanonical.

□

To approach the restrictions which define a legal reduction rule, we need to define classes of expressions which can be completed to closed and canonical expressions by substitution. These are the terms that will be matched and used to create reductions in the process of rewriting.

Definition 4.4

(i) A combinator expression, t, is respectively **open** *or* **open canonical** *with* **context variables** $cx(t)$ *if it is recursively defined by:*

[Op.1] A variable, x, is both open and open canonical and has context $cx(x) = \{x\}$.

[Op.2] If δ is a distributive combinator and $c_1, ..., c_n$ are a set of, respectively, open or open canonical expressions, in which no variable occurs as both a context variable and a non-context variable, then $\delta(c_1, ..., c_n)$ is, respectively, open or open canonical with context variables $\bigcup cx(c_i)$.

[Op.3] If β is, respectively, any combinator or a constructive combinator, $y_1, \cdots, y_n$ are, respectively, arbitrary combinator expressions or variables, and t is, respectively, an open or an open canonical, with no

context variables in common with $y_1, \cdots, y_n$, *then* $t.\beta(y_1, \cdots, y_n)$ *is, respectively, open or an open canonical with context variables* $cx(t)$.

(ii) A **linear** *combinator expression is one in which no variable has two distinct occurrences.*

Open canonicals are open. The variables which are not context variables in an open expression are called *contextless* variables as they are contextless subexpressions. An important property of an open (resp. open canonical) expressions is that they can be turned into a closed (resp. canonical) expression by substitution.

Lemma 4.3

(i) If the context variables of an open expression are substituted by closed expressions and the contextless variables by ground expressions, then the resulting expression is closed.

(ii) If the context variables of an open canonical expression are substituted by canonicals and the contextless variables by ground expressions, then the resulting expression is canonical.

Therefore, the reason for distinguishing the context variables so carefully is that they must generally be substituted by special expressions in order to obtain special forms. They also play the same role as the variables in traditional rewriting systems.

Definition 4.5

(i) A **reduction rule** *is of the form*

$$t.\alpha(x_1, \cdots, x_m) \Rightarrow r$$

where the left-hand side, called the **redex**, *must have the same domain and codomain as the right-hand side, and*

- $\alpha(x_1, \cdots, x_m)$ *is an active combinator, called the* **actor** *of the rule, and* $x_1, ..., x_m$ *are distinct combinator variables which do not occur elsewhere on the left.*
- t *is called the* **template** *of the rule and must be a linear open canonical.*
- r *is a* **contractum** *and must be an open expression such that both the context variables are contained in the redex context variables and the non-context variables are contained in the redex non-context variables.*

(ii) $t.\alpha(y_1, ..., y_n) \Rightarrow r$ **matches** $c.\alpha(c_1, ..., c_n).c'$, *if there is a substitution* σ *such that* $c = t_\sigma$ *and* $(y_i)_\sigma = c_i$ *for* $i = 1, ..., n$, *and every context variable is substituted by a closed expression. The* **reduct**, *which results from this application, is then* $r_\sigma.c'$.

The reason the template is required to be linear is that it is computationally advantageous if there is no need to unify context and contextless expressions. The results above ensure that a reduct is necessarily closed. One should verify that the rewriting rules given in the examples above do satisfy this definition.

If we were to do a by-value evaluation of expressions then this would involve finding occurrences of actives with precanonical contexts and evaluating the active on the distributed version of that context. With this strategy in mind lemma 4.2 shows that we may always reduce a closed expression while it is not precanonical, *provided* that each legal closure formed from a canonical and an active has an applicable reduction rule.

A *closure* is a pair of combinator expressions

$$c_1 :: c_2$$

such that $c_1.c_2$ is a valid expression. For example the redex of any reduction rule may be viewed as a closure of an open canonical with an active combinator

$$t :: \alpha(x_1, \cdots, x_n).$$

When a lazy evaluation is performed and it is necessary to suspend a computation, the suspended computation is held as a closure.

4.4 Properties of the Reduction System

In this section the conditions under which a combinator theory together with reduction rules form a categorical combinator reduction system are stated.

Definition 4.6

(i) A set of reduction rules is said to be **complete** *if every legal closure of a canonical with an active has an applicable reduction rule.*

(ii) If every such closure has precisely one applicable reduction rule the set of reduction rules is said to be **deterministic**.

(iii) An **irreducible** *combinator is a combinator which cannot be further reduced by either the application of a reduction rule or a distribution.*

(iv) A **categorical combinator reduction system** *is a combinator theory together with*

- *a classification of the primitive combinators into actives, distributives, and constructives,*
- *a set of reduction rules,*

such that this data is consistent with the theory, the theory separates fully canonical expressions, and the reduction rules are complete and deterministic.

The point of these conditions can now be made clear:

Theorem 4.4 *If* **R** *is a categorical combinator reduction system, then reduction is confluent on closed expressions and an expression is irreducible if and only if it is canonical.*

The above remarks show that, when the reduction rules are complete, an expression is irreducible if and only if it is canonical. It remains only to establish the confluence of the rewriting system. This is a relatively simple consequence of the fact that the reduction system is deterministic and linear [45]. The determin-

ism guarantees that there are no non-trivial critical pairs and any linear system with only trivial critical pairs is confluent whether terminating or not.

CHAPTER 5

Abstract Machines for Evaluation

If combinator reduction systems are to be used as programming languages, a key concern would be whether they can be efficiently implemented. In this chapter it is shown how one can implement these reduction systems using abstract machines.

5.1 Chain Rewriting

The traditional notion of a redex is, in these systems, broken into the pair $t.a$ where t is a template and a is an active. The fact that we distinguish the active combinators allows the rewriting to be implemented as a "chain rewriting," which is described below.

A closed combinator expression can always be associatively matched to a chain of primitive combinators expressions, whose arguments will contain ground expressions,

$$a_1.a_2.a_3.\ldots.a_{n-2}.a_{n-1}.a_n \ .$$

This is called the outermost composition chain. Identity maps can always be eliminated from this chain as, being a closed expression, it must be a non-trivial composition.

As composition is associative, the arrangement of the brackets does not matter. It is normal, however, to assume that the composition is in a right or left bracketed form, which allows us to regard any chain as either a backward or a forward list in which the empty list is the identity. In chain rewriting we need both forms: the composition is split into a backward list (left associated), followed by a forward list (right associated). The place where these lists meet is indicated by a $\star$ and is the position of the *rewriting head*

$$(..(((i.a_1).a_2).a_3).....a_i) \star (a_{i+1}.(....(a_{n-2}.(a_{n-1}.(a_n.i)))..)\ .$$

The basic idea of a chain rewriting is that the rewriting head starts at one end of the chain and traverses the chain once in the course of a reduction. If the head starts at the left end, the rewriting is greedy or eager, while if it starts at the right end, it is lazy. One can also introduce variants which are partially lazy, for example, by greedily evaluating to weak canonical form and lazily evaluating to canonical form. This method of evaluation was used by Hagino[42] and was also used in a pilot implementation for this thesis (see below).

Whenever there is an active immediately to the right of the head, the chain rewriting system tries to apply the reduction rules associated with that active. To achieve this, however, it may require that certain arguments of distributives in the left chain are evaluated. To do this, the chain to the left of that distributive combinator is distributed inside to give closures in each argument, and the required argument is evaluated. An argument is demanded whenever it must match a non-variable expression in the template.

Example 5.1 (A simple chain rewriting system)

Consider the following rules:

$$a_1 \star f \Rightarrow a_2\star \qquad a_1 \star g \Rightarrow a_3\star$$

$$a_2 \star f \Rightarrow a_3\star \qquad a_2 \star g \Rightarrow a_1\star$$

$$a_3 \star f \Rightarrow a_1\star \qquad a_3 \star g \Rightarrow a_1 \star .$$

We may perform a rewriting on the term

$$\star a_1.f.g.g$$

by moving the head from left to right as follows:

$$\begin{aligned} \star a_1.f.g.g &\Rightarrow a_1 \star f.g.g \\ &\Rightarrow a_2 \star g.g \\ &\Rightarrow a_1 \star g \\ &\Rightarrow a_3 \star . \end{aligned}$$

□

In the partially lazy evaluation, the rewriting has not finished when the head reaches the right end as there may be distributives with unevaluated arguments. The final sweep is a movement of the head back to the left. Every time the head encounters a distributive, it does a complete evaluation of its arguments by distributing the chain to the left inside and working on the closure in each argument.

For the **PLOC** system this gives the following chain rewriting; this is a trace from the **charity** system (see appendix).

```
Charity Interpreter

version 0.4


char> set:{trace}?

trace is on

char> <[!,!],[!,!]>.act(i,cons)?

<[!,!],[!,!]>_act(i,_cons)

(1) act: <[!,!],[!,!]>*act(i,_cons)

   (2) done: [!,!]*

(1) act(cons): <[!,!],[!,!]>_act(i,_cons)

(3) move: <!,<[!],[!,!]>_act(i,_cons)>*cons

(4) pr(p0): [!|<[!],[!,!]>_act(i,_cons)]*

   (5) canonical: !*

(4) pr(p1): [!|<[!],[!,!]>_act(i,_cons)]

   (6) act: <[!],[!,!]>*act(i,_cons)

      (7) done: [!]*

   (6) act(cons): <[!],[!,!]>_act(i,_cons)

   (8) move: <!,<[],[!,!]>_act(i,_cons)>*cons

   (9) pr(p0): [!|<[],[!,!]>_act(i,_cons)]*

      (10) canonical: !*

   (9) pr(p1): [!|<[],[!,!]>_act(i,_cons)]

      (11) act: <[],[!,!]>*act(i,_cons)
```

```
        (12) done: []*
      (11) act(nil): <[],[!,!]>_act(i,_cons)
        (13) done: []*
      (11) act(nil): <[],[!,!]>_act(i,_cons)
      (14) canonical: [!,!]*
   (9) pr: [!|[!,!]*]
   (15) canonical: [!,!,!]*
(4) pr: [!|[!,!,!]*]

[!,!,!,!]
```

The prolog list notation has been used above for convenience:

$$\langle x, y \rangle . \, cons \equiv [x|y], \quad !. \, nil \equiv [],$$

$$[x, y, ..] \equiv \langle x, \langle y, ... \rangle . \, cons \rangle . \, cons$$

The * denotes the active head and the underscores denote the closures or inactive heads.

5.2 The Greedy Abstract Machine

In a chain rewriting the left and right chains can be regarded as stacks. In an eager evaluation the left stack will always hold a canonical or a "value" while the right stack will hold the active combinators or the "code". Except for the

Table 5.1: A greedy abstract machine ($\beta(\vec{c})$ is constructive)

input			*output*		
Value	**Code**	**Dump**	**Value**	**Code**	**Dump**
V	$!.C$	D	$!$	C	D
V	$\beta(\vec{c}).C$	D	$V.\beta(\vec{c})$	C	D
$V.b_i$	$\langle c_0; c_1\rangle.C$	D	V	c_i	$c(C).D$
V	$\langle c_0, c_1\rangle.C$	D	V	c_0	$d(\langle -, c_1\rangle, V).c(C).D$
$\langle V_0, V_1\rangle$	$p_i.C$	D	V_i	C	D
$V.nil$	$split.C$	D	$V.b_0$	C	D
$V.cons$	$split.C$	D	$V.b_1$	C	D
$\langle V_0.b_i, V_1\rangle$	$d_0.C$	D	$\langle V_0, V_1\rangle.b_i$	C	D
$\langle V_0, V_1.b_i\rangle$	$d_1.C$	D	$\langle V_0, V_1\rangle.b_i$	C	D
$\langle V.nil, V_1\rangle$	$act(f,g).C$	D	V_1	f	$c(C).D$
$\langle\langle V_0, V\rangle.cons, V_1\rangle$	$act(f,g).C$	D	$\langle V, V_1\rangle$	$act(f,g)$	$l(V_0,g).C(C).D$
V	[]	$c(C).D$	V	C	D
V_0	[]	$d(\langle -, c_1\rangle, V').D$	V'	c_1	$e(\langle V_0, -\rangle).D$
V_1	[]	$e(\langle V_0, -\rangle).D$	$\langle V_0, V_1\rangle$	[]	D
V_1	[]	$l(V_0,g).D$	$\langle V_0, V_1\rangle$	g	D
V	[]	[]	**stop**		

complication which arises when an active causes an evalution of the arguments of a primitive combinator on the "value" stack, we have the design of a machine.

In order to handle the above complication it is necessary to add a third stack on which we may "dump" partially completed computations which have initiated some other computation. This gives the following form to the abstract machine (see table 5.1). Notice that this is very similar in form to the CAM machine and indeed to Landin's SECD machine.

Example 5.2 (Appending two lists)

Suppose one wishes to append two lists:

$$\langle [!], [!] \rangle . act(i, cons)$$

then the starting state of the greedy abstract machine is

$$([], \langle (!.[], !.nil.[]) . cons.[], \langle !.[], !.nil.[] \rangle . cons.[] \rangle . act(i.[], cons.[]).[], [])$$

and the transitions are:

$\rightarrow$ $([],\langle !.[],!.nil.[]\rangle.cons.[],d(\langle -,\langle !.[],!.nil.[]\rangle.cons.[]\rangle,[]).c(act(i.[],cons.[]).[]).[])$

$\rightarrow$ $([],!.[],d(\langle -,!.nil.[]\rangle,[]).c(cons.[]).d(\langle -,\langle !.[],!.nil.[]\rangle.cons.[]\rangle,[]).c(act(i.[],cons.[]).[]).[])$

$\rightarrow$ $(!.[],[],d(\langle -,!.nil.[]\rangle,[]).c(cons.[]).d(\langle -,\langle !.[],!.nil.[]\rangle.cons.[]\rangle,[]).c(act(i.[],cons.[]).[]).[])$

$\rightarrow$ $([],!.nil.[],d(\langle !.[],-\rangle,[]).c(cons.[]).d(\langle -,\langle !.[],!.nil.[]\rangle.cons.[]\rangle,[]).c(act(i.[],cons.[]).[]).[])$

$\rightarrow$ $(!.[],nil.[],d(\langle !.[],-\rangle,[]).c(cons.[]).d(\langle -,\langle !.[],!.nil.[]\rangle.cons.[]\rangle,[]).c(act(i.[],cons.[]).[]).[])$

$\rightarrow$ $(nil.!.[],[],d(\langle !.[],-\rangle,[]).c(cons.[]).d(\langle -,\langle !.[],!.nil.[]\rangle.cons.[]\rangle,[]).c(act(i.[],cons.[]).[]).[])$

$\rightarrow$ $(\langle !.[],nil.!.[]\rangle.[],[],c(cons.[]).d(\langle -,\langle !.[],!.nil.[]\rangle.cons.[]\rangle,[]).c(act(i.[],cons.[]).[]).[])$

$\rightarrow$ $(\langle !.[],nil.!.[]\rangle.[],cons.[],d(\langle -,\langle !.[],!.nil.[]\rangle.cons.[]\rangle,[]).c(act(i.[],cons.[]).[]).[])$

$\rightarrow$ $(cons.\langle !.[],nil.!.[]\rangle.[],cons.[],d(\langle -,\langle !.[],!.nil.[]\rangle.cons.[]\rangle,[]).c(act(i.[],cons.[]).[]).[])$

$\rightarrow$ $([],\langle !.[],!.nil.[]\rangle.cons.[],d(\langle cons.\langle !.[],nil.!.[]\rangle.[],-\rangle,[]).c(act(i.[],cons.[]).[]).[])$

$\rightarrow$ $([],!.[],d(\langle -,!.nil.[]\rangle,[]).c(cons.[]).d(\langle cons.\langle !.[],nil.!.[]\rangle.[],-\rangle,[]).c(act(i.[],cons.[]).[]).[])$

$\rightarrow$ $(!.[],[],d(\langle -,!.nil.[]\rangle,[]).c(cons.[]).d(\langle cons.\langle !.[],nil.!.[]\rangle.[],-\rangle,[]).c(act(i.[],cons.[]).[]).[])$

$\rightarrow$ $([],!.nil.[],d(\langle !.[],-\rangle,[]).c(cons.[]).d(\langle cons.\langle !.[],nil.!.[]\rangle.[],-\rangle,[]).c(act(i.[],cons.[]).[]).[])$

$\rightarrow$ $(!.[],nil.[],d(\langle !.[],-\rangle,[]).c(cons.[]).d(\langle cons.\langle !.[],nil.!.[]\rangle.[],-\rangle,[]).c(act(i.[],cons.[]).[]).[])$

$\rightarrow$ $(nil.!.[],[],d(\langle !.[],-\rangle,[]).c(cons.[]).d(\langle cons.\langle !.[],nil.!.[]\rangle.[],-\rangle,[]).c(act(i.[],cons.[]).[]).[])$

$\rightarrow$ $(\langle !.[],nil.!.[]\rangle,[],c(cons.[]).d(\langle cons.\langle !.[],nil.!.[]\rangle.[],-\rangle,[]).c(act(i.[],cons.[]).[]).[])$

$\rightarrow$ $(\langle !.[],nil.!.[]\rangle,cons.[],d(\langle cons.\langle !.[],nil.!.[]\rangle.[],-\rangle,[]).c(act(i.[],cons.[]).[]).[])$

$\rightarrow$ $(cons.\langle !.[],nil.!.[]\rangle,[],d(\langle cons.\langle !.[],nil.!.[]\rangle.[],-\rangle,[]).c(act(i.[],cons.[]).[]).[])$

$\rightarrow$ $(\langle nil.!.[],cons.\langle !.[],nil.!.[]\rangle.[]\rangle,act(i.[],cons.[]).[],l(!.[],cons.[]).[])$

$\rightarrow$ $(cons.\langle !.[],nil.!.[]\rangle.[],i.[],c([]).l(!.[],cons.[]).[])$

$\rightarrow$ $(cons.\langle !.[],nil.!.[]\rangle.[],[],c([]).l(!.[],cons.[]).[])$

$\rightarrow$ $(cons.\langle !.[],nil.!.[]\rangle.[],[],l(!.[],cons.[]).[])$

$\rightarrow$ $(\langle !.[],cons.\langle !.[],nil.!.[]\rangle.[]\rangle.[],cons.[],[])$

$\rightarrow$ $(cons.\langle !.[],cons.\langle !.[],nil.!.[]\rangle.[]\rangle.[],[],[])$

□

5.3 A Partially Lazy Abstract Machine

This section presents a partially lazy abstraction machine M_w for a list arithmetic cartesian closed category. In common with lazy rewriting schemes, it is necessary for such a machine to have some mechanism for forcing the rewriting to completion. The machine accomplishes this by having different modes: a forward mode which evaluates to a weak canonical, a backward mode which completes the evaluation to a canonical. This means the M_w consists of four components: a *mode*, a *left* stack, a *right* stack, and a *dump* stack. The left and right stacks hold, respectively, the left and right chains of the chain rewriting system, while the dump holds suspended state and processing information.

The machine actually has three modes: the forward, $\uparrow$, the backward, $\downarrow$, and the stopped, $\bullet$, mode. The machine starts in the forward mode with the expression to be processed in the right stack and all the other stacks empty. In the forward mode the right stack acts like "code" in that its contents cause actions to be performed on the left stack. The aim of the forward mode is to produce a weak canonical by completely emptying the right stack. This is done by passing constructive and distributive combinators across to the left stack and carrying out the actions associated with the active combinators.

When the right stack is emptied in the forward mode the machine switches to the backward mode. The purpose of this mode is to generate a canonical in the right stack. This is achieved by passing constructives across to the right stack and processing the arguments of any distributive combinators which are

encountered. Generally, to process an argument of a distributive, the machine will be thrown back into the forward mode again. When the left stack is empty in the backwards mode and there are no dumped states, the machine stops.

It is not very clear from the above description what the role of the dump might be. The dump stack holds two different sorts of objects: *continuations* and *demands*. A continuation, $c(C)$, is used in the forward mode to store a continuation of the code or right stack. A demand,

$$\begin{pmatrix} I \\ E \\ A \\ C \end{pmatrix},$$

has four components: an indicator, I, which together with the result of the current evaluation determines the "pop action" (which restores the original environment in some predefined way), an environment, E, an argument array, A, and a code continuation, C.

Whenever an active combinator is encountered in the forward mode it will cause a continuation or demand to be dumped. For example an active may cause the current evaluation to be suspended in favor of the evaluation of an argument of the left stack, in which case it is necessary to dump the current state of the machine until the evaluation of this child calculation has been completed. Encoded into the information which is dumped must of course be the information required to reinitiate the calculation when the child calculation is completed.

It is sometimes necessary for the machine to store computations which are

partially complete (as they have been initiated as a side effect of another calculation). In these situations it is necessary to hold a closure, $E :: C$, where E is an environment (left stack) and C is the code (right stack). It is also sometimes necessary to hold nested closures, as in $(E :: C_0) :: C_1$.

Example 5.3 (Evaluation of Simple code)

(i) Suppose one wishes to evaluate:

$$\langle !.b_0, !.b_1\rangle.\langle p_1, p_0\rangle.p_1$$

then the starting state of the M_w is

$$(\uparrow, [], \langle !.b_1.[], !.b_0.[]\rangle.\langle p_1.[], p_0.[]\rangle.p_1.[], [])$$

and the transitions are:

$$\begin{aligned}
&\rightarrow (\uparrow, \langle !.b_1.[], !.b_0.[]\rangle.[], \langle p_1.[], p_0.[]\rangle.p_1.[], [])\\
&\rightarrow (\uparrow, \langle p_1.[], p_0.[]\rangle.\langle !.b_1.[], !.b_0.[]\rangle.[], p_1.[], [])\\
&\rightarrow (\uparrow, \langle !.b_1.[], !.b_0.[]\rangle.[], p_0.[], c([]).[])\\
&\rightarrow (\uparrow, [], !.b_1.[], c([]).c([]).[])\\
&\rightarrow (\uparrow, !.[], b_1.[], c([]).c([]).[])\\
&\rightarrow (\uparrow, b_1.!.[], [], c([]).c([]).[])\\
&\rightarrow (\uparrow, b_1.!.[], [], c([]).[])\\
&\rightarrow (\uparrow, b_1.!.[], [], [])\\
&\rightarrow (\downarrow, b_1.!.[], [], [])
\end{aligned}$$

Table 5.2: M_w Weak Canonical State Transitions

input				*output*			
mode	**left**	**right**	**dump**	**mode**	**left**	**right**	**dump**
↑	S	$[]$	$[]$	↓	S	$[]$	$[]$
↑	S_0	$S_1 :: C$	D	↑	S_1	C	D
↑	$S_0 :: C_0$	C_1	D	↑	S_0	C_0	$c(C_1).D$
↑	S	$[]$	$c(C).D$	↑	S	C	D
↑	S	$!.C$	D	↑	$!.[]$	C	D
↑	S	$\langle C_1, C_2\rangle.C$	D	↑	$\langle C_1, C_2\rangle.S$	C	D
↑	S	$curry(C_1).C$	D	↑	$curry(C_1).S$	C	D
↑	S	$b_0.C$	D	↑	$b_0.S$	C	D
↑	S	$b_1.C$	D	↑	$b_1.S$	C	D
↑	S	$nil.C$	D	↑	$nil.S$	C	D
↑	S	$cons.C$	D	↑	$cons.S$	C	D
↑	$curry(C).S$	$[]$	$\begin{pmatrix} \uparrow p_1 \\ S_1 \\ \langle C_1, _\rangle \\ C_2 \end{pmatrix}.D$	↑	$\langle S_1 :: C_1, S\rangle.[]$	C	$c(C_2).D$
↑	$\langle C_1, C_2\rangle.S$	$eval.C$	D	↑	S	C_2	$\begin{pmatrix} \uparrow p_1 \\ S \\ \langle C_1, _\rangle \\ C \end{pmatrix}.D$
↑	$\langle C_1, C_2\rangle.S$	$p_0.C$	D	↑	S	C_1	$c(C).D$
↑	$\langle C_1, C_2\rangle.S$	$p_1.C$	D	↑	S	C_2	$c(C).D$
↑	$b_0.S$	$\langle C_1; C_2\rangle.C$	D	↑	S	C_1	$c(C).D$
↑	$b_1.S$	$\langle C_1; C_2\rangle.C$	D	↑	S	C_2	$c(C).D$
↑	$nil.S$	$list(C_1, C_2).C$	D	↑	$[]$	C_1	$c(C)$
↑	$cons.$ $\langle A_1, A_2\rangle.S$	$list(C_1, C_2).C$	D	↑	$\langle S :: A_1,$ $(S :: A_2) ::$ $list(C_1, C_2)\rangle.[]$	C_2	$c(C).D$

Table 5.3: M_w Canonical State Transitions

input				*output*			
mode	**left**	**right**	**dump**	**mode**	**left**	**right**	**dump**
$\downarrow$	$[]$	C	$[]$	$\bullet$	**STOP**		
$\downarrow$	$!.S$	C	D	$\downarrow$	$[]$	$!.C$	D
$\downarrow$	$\langle C_1, C_2\rangle.S$	C	D	$\uparrow$	S	C_1	$\begin{pmatrix}\downarrow p_0\\ S\\ \langle -, C_2\rangle\\ C\end{pmatrix}.D$
$\uparrow$	S_1	$[]$	$\begin{pmatrix}\downarrow p_0\\ S\\ \langle -, C_2\rangle\\ C\end{pmatrix}.D$	$\downarrow$	S_1	$[]$	$\begin{pmatrix}\downarrow p_0\\ S\\ \langle -, C_2\rangle\\ C\end{pmatrix}.D$
$\downarrow$	$[]$	C_1	$\begin{pmatrix}\downarrow p_0\\ S\\ \langle -, C_2\rangle\\ C\end{pmatrix}.D$	$\uparrow$	S	C_2	$\begin{pmatrix}\downarrow p_1\\ S\\ \langle C_1, -\rangle\\ C\end{pmatrix}.D$
$\uparrow$	S_2	$[]$	$\begin{pmatrix}\downarrow p_1\\ S\\ \langle C_1, -\rangle\\ C\end{pmatrix}.D$	$\downarrow$	S_2	$[]$	$\begin{pmatrix}\downarrow p_1\\ S\\ \langle C_1, -\rangle\\ C\end{pmatrix}.D$
$\downarrow$	$[]$	C_2	$\begin{pmatrix}\downarrow p_1\\ S\\ \langle C_1, -\rangle\\ C\end{pmatrix}.D$	$\downarrow$	$[]$	$\langle C_1, C_2\rangle.C$	D
$\downarrow$	$curry(C_0).S$	C	D	$\downarrow$	S	$curry(C_0).C$	D
$\downarrow$	$nil.S$	C	D	$\downarrow$	S	$nil.C$	D
$\downarrow$	$cons.S$	C	D	$\downarrow$	S	$cons.C$	D
$\downarrow$	$b_0.S$	C	D	$\downarrow$	S	$b_0.C$	D
$\downarrow$	$b_1.S$	C	D	$\downarrow$	S	$b_1.C$	D

$$\rightarrow \quad (\downarrow, !.[], b_1.[], [])$$

$$\rightarrow \quad (\downarrow, [], !.b_1.[], [])$$

$$\rightarrow \quad (\bullet, [], !.b_1.[], [])$$

(ii) Suppose one wishes to evaluate:

$$\langle\langle !.b_0, !.b_1\rangle, curry(p_0.p_1)\rangle.\, eval$$

then the starting state of the M_w is

$$(\uparrow, [], \langle\langle !.b_0.[], !.b_1.[]\rangle.[], curry(p_0.p_1.[]).[]\rangle.\, eval.[], [])$$

and the transitions are:

$$\rightarrow \quad (\uparrow, \langle\langle !.b_0.[], !.b_1.[]\rangle.[], curry(p_0.p_1.[]).[]\rangle.[], eval.[], [])$$

$$\rightarrow \quad (\uparrow, [], curry(p_0.p_1.[]).[], \begin{pmatrix} \uparrow p_1 \\ [] \\ \langle\langle !.b_0.[], !.b_1.[]\rangle.[], _\rangle \\ [] \end{pmatrix}.[])$$

$$\rightarrow \quad (\uparrow, curry(p_0.p_1.[]).[], [], \begin{pmatrix} \uparrow p_1 \\ [] \\ \langle\langle !.b_0.[], !.b_1.[]\rangle, _\rangle \\ [] \end{pmatrix}.[])$$

$$\rightarrow \quad (\uparrow, \langle\langle !.b_0.[], !.b_1.[]\rangle.[], []\rangle.[], p_0.p_1.[], c([]).[])$$

$$\rightarrow \quad (\uparrow, [], \langle !.b_0.[], !.b_1.[]\rangle.[], c(p_1.[]).c([]).[])$$

$$\rightarrow \quad (\uparrow, \langle !.b_0.[], !.b_1.[]\rangle.[], [], c(p_1.[]).c([]).[])$$

$$\rightarrow (\uparrow, \langle !.b_0.[], !.b_1.[]\rangle.[], p_1.[], c([]).[])$$

$$\rightarrow (\uparrow, [], !.b_1.[], c([]).c([]).[])$$

$$\rightarrow (\uparrow, !.[], b_1.[], c([]).c([]).[])$$

$$\rightarrow (\uparrow, b_1.!.[], [], c([]).c([]).[])$$

$$\rightarrow (\uparrow, b_1.!.[], [], c([]).[])$$

$$\rightarrow (\uparrow, b_1.!.[], [], [])$$

$$\rightarrow (\downarrow, b_1.!.[], [], [])$$

$$\rightarrow (\downarrow, !.[], b_1.[], [])$$

$$\rightarrow (\downarrow, [], !.b_1.[], [])$$

$$\rightarrow (\bullet, [], !.b_1.[], [])$$

□

5.4 Graph Reduction

The description of the above abstract machine still does not give a very efficient implementation. For example, many steps are taken doing empty continuations: clearly these may be optimized out of the machine. Also, consider the computation of

$$\langle !.b_0, !.b_1\rangle.\langle p_1, p_1\rangle$$

it is clear that one repeats the calculation of the first argument twice. This is why graph reduction techniques, see for example [52, 30], are really required for

this method of evaluation. This was done on an ad hoc basis for the **Charity** system already.

Subexpression sharing The purpose of sharing in the reduction is to cut down the amount of copying. In lazy systems it also allows computations to be shared. The behavior of sharing is related explicitly to the form of the reduction rule.

In a reduction rule, if the subexpression of a redex is used more than once in the contractum to construct the reduct, then the subexpression should be shared, even though it is used in two different positions of the reduct. Two clear examples are the distributive reduction:

$$\mathrm{W}.\delta(\vec{c}) \Longrightarrow \delta(\mathrm{W}.c_0,\cdots,\mathrm{W}.c_n)$$

and the action of a list

$$\langle\langle X_1, X_2\rangle.cons, Y\rangle * act(f,g) \Longrightarrow \langle X_1, \langle X_2, Y\rangle.act(f,g)\rangle.g$$

A simple reference counting strategy can be used to keep track of the sharing. Since subexpressions are discarded by decrementing the reference count during the reduction, it is clear that when the reference count of a subexpression becomes 0, then that storage space can be reclaimed.

Accounting An accounting system is used to keep track of the state of evaluation of each subexpression, that is whether it is canonical, weak canonical, or closed. This information can be calculated when a combinator expression is

parsed, and it can be updated whenever a reduction is applied to the combinator expression. This information means that there is never an attempt to evaluate a canonical.

CHAPTER 6

Conclusion

6.1 Contributions

The main contribution of this research is to have closed considerably the gap between the definition of category and what is needed to have an efficient implementation of such a language. The contributions of this work are indicated by the solid boxes and arrows of figure 1.1.

The categorical combinator theories provide a powerful framework for organizing programming language features uniformly. Some examples have been shown here and [21] has further descriptions. When compared with the CAM[22], which is designed specifically for cartesian closed categories, the categorical combinator theories are more extensive. When compared with the CPL[42], the categorical combinators are more flexible because they are not restricted by the functors and adjoints.

The categorical combinator reduction system gives us confidence that the computational setting is basically sound. This provides a solid foundation for the design of the abstract machines and a good basis for proving that the implementations are correct. The analysis of the system is done by connecting it to the linear rewriting system of Huet[45]. This simplifies the understanding of the

verification.

The design of the abstract machines in chapter 5 brings out the requirements of having such an efficient language. Although the techniques used in the design are not new, the effort to integrate them into this setting is novel. The accounting system which helps avoid useless traversal is a direct implementation of the semantic calculation which correctly reflects the status of a subexpression. It is integrated into the language with very little cost. The sharing system, which turns a tree representation into a graphical representation, is directly related to the handling of distributive combinators, and is taken care of by the storage management during the reduction.

This work has brought us closer the goal of having a categorical programming language. However, there is much to be done before such a system becomes widely acceptable.

6.2 Directions for Future Research

Since my work, which focuses on the theory of organizing the language features and the design of their abstract machines for an efficient implementation, is just a small portion of the categorical programming story, it is reasonable to expect that much work remains to be done. In the following, some future work that is perceived to be essential is discussed.

The data structure specification, which gives the capability of having abstract data types, has been studied. Implementations which involve the management of

the representation of data structures have been experimented with grouping the data type, its operation, and type signatures in an object-oriented fashion.

Distributive logic, which is an equational style of programming, will enhance the detection of common subexpressions and avoid the burden of dealing with common subexpressions at the level of combinators. The theory of distributive logic has been studied [19] and its translation into the combinator logic has been designed and tested independently. However, the tool still has to be integrated into these settings.

When the above facilities are provided, examples of this style of programming will be studied. Even though it is perceived that many computational problems will benefit from by this language, it is not certain what resources are needed to fully exploit the benefits of this language.

Regarding the design of an efficient machine, the abstract machines should be further improved to become a machine level instruction that can be directly executed by a host machine. This may open a new way of optimizing the current implementation. This problem has been studied in other programming languages, such as the G-machine [50] or Tim[30]; hopefully some of these techniques can be incorporated into this situation.

Other work that needs to be done includes type checking for equational logic expressions, the search for a good representation of the primitive combinators, and the completion of a by-need abstract machine.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] A.V. Aho and J.D. Ullman, *Principles of Compiler Design*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1979.

[2] M.A. Arbib and E.G. Manes, "Machines in a Category: An Expository Introduction", *SIAM Review* , Vol 16, No. 2, April 1974.

[3] M.A. Arbib and E.G. Manes, *Arrows, Structures, and Functors: The Categorical Imperative.* Academic Press, New York, 1975.

[4] M.p. Atkinson, P. Buneman, and R. Morrison(Eds), *Data Types and Persistence*, Spriner-Verlag, Berlin, 1988.

[5] S. Baden, *Berkeley FP User's Manual*, Rev. 4.1, (July 1983).

[6] J. Backus, "Can Programming be Liberated from von Neuman Style? A Functional Style and Its Algebra or Programs." *Communications of ACM*, Vol 21, 8 (1978).

[7] M. Barr and C. Wells, *Toposes, Triples and Theories.*, A Series of Comprehensive Studies in Mathematic Vol. 278, Springer-Verlag, (1985).

[8] G. Berry, Stable models of typed lambda calculi, *Automata, languages, and programming*, proceeding of the 5th colloquium (LNCS 62) (Undine, Italy, July 17-21, 1978), C. Bohm (Ed.), Springer-Verlag, Berlin, pp. 72-89, 1978.

[9] M. Boscarol, L. Carlucci, and G. Levi (Eds.), *Foundations of Logic and Functional Programming* Lecture Notes in Computer Science, 306, Springer-Verlag, 1986.

[10] R.M. Burstall and J.A. Goguen, "Putting Theories Together to Make Specification." *Proceedings of 5th International Joint Conference on Artificial Intelligence*, pp. 1045-1058, (1977).

[11] R.M. Burstall, D.B. MacQueen, and D.T. Sanella, "HOPE: an experimental applicative language." *Proceedings of the ACM Lisp Conference*, (1980).

[12] R.M. Burstall and J.A. Goguen, "An Informal Introduction to Specifications using Clear." *The Correctness Problem in Computer Sciences*, Academic Press, pp. 185-213, 1981.

[13] R.M. Burstall and B. Lampson, "A Kernel Language for Abstract Data Types and Modules." *Lecture Notes in Computer Science*, Vol. 173, pp. 1-50 (1984).

[14] A. Church, *The Calculi of Lambda Conversion*, Ann. Math Stud., No. 6, Princeton University Press, 1951.

[15] W. Clocksin and C. Mellish, *Programming in Prolog*, Springer-Verlag, Berlin, (1981).

[16] J.R.B. Cockett, *Locally Recursive Categories*, preprint: Univ. of Tennessee at Knoxville, Dept. of Comp. Sc., (1986)

[17] J.R.B. Cockett, "List-Arithmetic Distributive Categories: Locoses," University of Tennessee at Knoxville, Dept. of Computer Science, Technical Report CS-88-76, 1988 (to be appeared in JPAA).

[18] J.R.B Cockett, "On the Decidability of Objects in a Locos," *Contemporary Mathematics*, Vol. 92, Amer. Math. Soc., pp. 23–46, 1989.

[19] J. R. B. Cockett, *Distributive Logic*, Univ. of Tennessee at Knoxville, Dept. of Comp. Sc., Technical Reprot, CS-89-01, (1989)

[20] J.R.B. Cockett, H.G. Chen, L.R. Smith, *Preliminary users manual for* **charity**, University of Tennessee at Knoxville, Dept. of Computer Science,

[21] J.R.B. Cockett, H.G. Chen, *Categorical Combinators* University of Tennessee at Knoxville, Dept. of Computer Science, presented at Banff, Cananda, Preprint, 1989.

[22] G. Cousineau, P-L Curien, M. Mauny, *The categorical abstract machine*, Lecture Notes in Computer Science 201, Springer-Verlag, Berlin (1985).

[23] H.B. Curry, R. Feys, and W. Craig, *Combinatory Logic*, Vol. 1, Studies in Logic and the Found. of Math., North-Holland Publishing Company, Amsterdam, (1958).

[24] J. Darlington, P. Henderson, and D.A. Turner, *Functional Programming and its Applications: An Advanced Course*, Cambridge University Press, Cambridge, 1982.

[25] J. Darlington, *Functional Programming* , Duce (ed), Academic Press, 1984.

[26] P. David, S. Abramsky, A. Poigne, and David Rydeheard, *Category Theory and Computer Programming.* Lecture Notes in Computer Science, 240, Springer-Verlag, 1986.

[27] N. Dershowitz, *Termination of Rewriting*, Technical Report, No. UIUCDCS-R-85-1220, Department of Computer Science, Univ. of Illinois at Urbana-Champaign, Urbana, Illinois, 1985.

[28] E.J.W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, (1976).

[29] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification 1*, EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1985

[30] J.H. Fasel and R.M. Keller (eds.), *Graph Reduction*, Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1987.

[31] R.W. Floyd, "Assigning meanings to programs", *Proc. Amer. Math. Soc. Symposia in applied mathematics*, Vol. 19, pp. 19-31 (1967).

[32] P. Freyd and A. Scedrov, "Some semantic aspects of polymorphic lambda calculus", *Proc. symposium on logic in computer science*, IEEE , New York, pp. 315-319 (1987).

[33] M.J. Gordon, R. Milner, and C.P. Wadsworth, *Edinburgh LCF* ,Lecture Notes in Computer Science, No. 78, Springer-Verlag, New York, 1979.

[34] J.W. Goguen, J.W. Thatcher, and E.G. Wagner, "An initial algebra semantics and continuous algebras", *JACM* , Vol 24, No. 1, pp. 68-95 (1977).

[35] J.A. Goguen, J.W. Thatcher, and E.G. Wagner, " An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types", *Current Trends in Programming Methodology*, R. Yeh (ed.) Prentice-Hall, 1978.

[36] J.A. Goguen and R.M. Burstall, "Institutions: Abstract Model Theory for Computer Science". Technical Report CSLI-85-30, Center for the Study of Language and Information, Stanford University, 1985.

[37] J.A. Goguen and J. Messeguer "Completeness of Many-sorted Logic". *Houston Journal of Mathematics*, Volume 11, No. 3, pp. 307-334 (1985).

[38] Goldblatt, R., *Topoi: The Categorical Analysis of Logic. Study in Logic and The Foundations of Mathematics*, Edited by J. Barwise *et.al.*, Vol. 98, North-Holland, 1984.

[39] R.L. Goodstein, *Development of mathematical logic*, Springer-Verlag (1971).

[40] M. Gordon, *The Denotational Description of Programming Languages*, Springer-Verlag, Berlin, (1979).

[41] J.V. Guttag, "Abstract Data Types and the Development of Data Structures", *Communications ACM*, Vol. 20, No. 6, (June 1977), pp. 396-404.

[42] T. Hagino, *A Categorical Programming Language*, PhD thesis, University of Edinburgh, 1987.

[43] P. Henderson, *Functional Programming: Application and Implementation*, Prentice/Hall International, Englewood Cliffs, (1980).

[44] Herrlich, H. and Strecker, G.E., *Category Theory.* Allyn and Bacon Inc., Boston, 1973.

[45] G. Huet, J. Levy, *Computations in nonambiguous linear rewriting systems*, INRIA, 1979.

[46] C.A.R. Hoare, "An axiomatic basis for Computer Programming", *Communications ACM*, Vol. 12, No. 10, pp. 576-583 (October 1969).

[47] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley Publishing Company, Reading Massachusetts, (1979).

[48] P. Hudak and D. Kranz, "A combinator based compiler for a functional language", *Proceedings of the 11th ACM Symposium on Principles of Programming Languages* , pp. 122-32, (1984).

[49] P. Hudak and L. Smith, "Para-functional Programming - a Paradigm for Programming Multiprocessor Systems," *Conference on Functional Programming and Computer Architecture, Nancy*, LNCS 201, Springer-Verlag, (1985).

[50] T. Johnsson, "Efficient Compilation of Lazy Evaluation" *Proceeding of the ACM Symposium on Compiler Construction*, Montreal, (June 1984).

[51] P. Jones, "Using Futurebus in a fifth generation computer", *Microprocessors and Microsystems*, Vol. 10, no. 2, (1986).

[52] S.P. Jones, *The Implementation of Functional Programming Languages*, Prentice/Hall International, Englewood Cliffs, 1988.

[53] H. Klaeren, "On parameterized abstract software modules using inductively specified operations", *afcet: Mathematics for Computer Science*, Paris, 1982.

[54] J.W. Klop, *Term rewriting systems: a tutorial* , Center for Mathematics and Computer Science, Department of Mathematics and Computer Science, Free University, de Boelelaan, (1987).

[55] J. Lambek,"From λ-calculus to cartesian closed categories, *To H.B. Curry: Essays on Combinatory Logic, λ-Calculus and Formalism*, Academic Press, pp. 375-402 (1980).

[56] Lambek, J. and Scott P.J., *Introduction to Higher Order Categorical Logic.* Cambridge University Press, Cambridge, 1986.

[57] P.J. Landin, "The Next 700 Programming Languages", *Communications ACM*, Vol. 9, No. 3, (March 1966), pp. 157-166.

[58] MacLane, *Categories for the Working Mathematician.* Springer-Verlag Graduate Texts in Mathematics, Berlin Heidelberg, 1971.

[59] B.J. MacLennan, *Principles of Programming Languages: Design, Evaluation, and Implementation*, second edition, Holt, Rinchart and Winston, New York, 1987.

[60] B.J. MacLennan, *Functional Programming - Practice and Theory*, Addison-Wesley Publishing Company, Inc. 1990.

[61] P. Martin-Lof, "Constructive Mathematics and Computer Programming." *6th International Congress for Logic, Methodology and Philosophy of Science*, (1979).

[62] P. Martin-Lof, "Constructive logic and computer programming *Logic, methodology and Philosophy of science VI,L*, J. Cohen et al, North-Holland, Amsterdam, The Netherlands, pp. 153-175, 1982.

[63] D.C.J. Matthews,"Poly Manual", *SIGPLAN NOTICE* , Vol. 20, No. 9, September 1985.

[64] J. McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I", *Communications ACM* , Vol. 3, No. 4, (April 1960), pp. 184-195.

[65] H.D. Mills, R.C. Linger, and A.R. Hevner, *Principles of Information Systems Analysis and Design*, Academic Press, Orlando, Florida, 1986.

[66] R. Milner,"A theory of Type Polymorphism in Programming", *Journal of Computer and System Sciences*, Vol. 17, No. 3, December 1978.

[67] M. O'Donnel, "Computing in systems described by equations," *Lecture Notes in Computer Science 58*, Springer-Verlag, 1977.

[68] L. Román, *Cartesian Categories with Natural Numbers Object*, Dept. Mathematics, McGill University, (submitted to J. Pure Appl. Algebra, 1986).

[69] F.j. Oles, Type algebras, functor categories and block structure *Algebraic methods in semantics*, M. Nivat, and J. C. Reynolds (Eds), Cambridge University Press, New York, pp. 543-573, 1985.

[70] J.C. Reynolds, "Toward a theory of type structure", *Proc. colloque sur la programmation*, LNCS 19, Springer-Verlag, Berlin, pp. 408-425, 1974.

[71] J.C. Reynolds, *The craft of programming*, Prentice-Hall, Englewood Cliffs, NJ, 1981, see CR 24, 1 (Jan. 1983), Rev. 39,967.

[72] J.C. Reynold, "Idealized Algol and its specification logic", *Tools and notions for program construction*, D. Neel (Ed.), Cambridge University Press, New York, pp. 121-161, 1982.

[73] J.C. Reynolds, Three Approaches to Type-Structure *Lecture Notes in Computer Science*, 185, Springer-Verlag, Berlin, 1986.

[74] J.A. Robinson, "A Machine Oriented Logic Based on the Resolution Principle, *Journal of the ACM*, Vol. 12, (1965), pp 23-31.

[75] H.P. Sanders, "Categorical combinators" Report 38, Programming Methodology Group, Department of Computer Science, Chelmers University of Technology, Sweden, June 1987.

[76] R.A.G. Seely, "Locally-cartesian closed categories and type theory", *Math. Proc. Camb. Philos. Soc.* , 95, (1984), 33-48.

[77] D. Scott, "Data Types as Lattices." *SIAM Journal of Computing*, Vol. 5, pp. 552-587 (1976).

[78] S. Stenlund, *Combinators, λ-Terms and Proof Theory*, D. Reidel, Dordrecht, 1972.

[79] J.E. Stoy, *The Scott-Strachey Approach to Programming Language Theory*, MIT Press, 1977.

[80] D.A. Turner, *Miranda - a non-strict functional language with polymorphic types*, In Conference on Functional Programming Language and Computer Architecture, Nancy. Jouannaud(ed.), Lecture Notes On Computer Science 201, Springer Verlag, (1985).

[81] R. Wilensky, *Common LISPcraft*, W.W.Norton & Company, New York, 1986.

[82] G. C. Wraith, *A note on categorical datatypes,* Lecture Notes in Computer Science 389, 1989.

APPENDIX

Preliminary User's Manual for CHARITY

J.R.B. Cockett H.G. Chen L.R.S. Smith

August 17, 1990

Abstract

Charity is a polymorphic functional style programming language with a categorical semantics in a *pre-locos*. The system is written in C and currently runs on SUN work stations.

This user manual is for the *pure* language which was developed initially as a class project (CS 683 Category theory). One of the main concerns was to achieve reasonable execution speeds employing categorical combinators. The manual describes the **charity** language (categorical combinators) and how to use the interpreter itself.

Currently the system is being expanded to include a number of features which will make it considerably more usable. In particular, a type checking, a method for creating data structures, and an equational style interface are all in the works. Further, down the road are input/output facilities and the utilization of hardware features directly from the language.

1 Preface

1.1 Manual objectives

This user's guide is intended as a preliminary support document for the **charity** interpreter which was developed as a class project in CS 683 Category Theory in the Computer Science Department at the University of Tennessee, Fall 1988.

At the time of writing the **charity** interpreter is a prototype system and lacks many of the features which would be expected of a full working system. For example the interpreter has no type checking or input/output facilities in the language. It is intended to add these, together with other features in the future.

1.2 Intended audience

This manual is intended for those who have a working knowledge of *category theory* and some understanding of *term rewriting.*

1.3 Structure of this document

The information in each section is presented in the following basic order: concept, format, example. The manual ends with a development of a small **charity** program to perform binary addition.

1.4 Conventions

The following conventions are observed in this manual:

- Boldface words, letters, and punctuation indicate keywords.
- Ellipses (...) indicate the preceding item(s) can be repeated one or more times.
- The nonprinting characters are denoted as follows:

carriage return	`<cr>`
tab	`<tb>`
control z	`<ctrl-z>`

All execution times are for a **SUN 3/50**.

2 Introduction

This manual describes the basic operations needed to invoke the services currently offered by the **charity** interpreter. The interpreter is written in C and can be made available on request to the authors.
The **charity** language is a functional, (weakly) polymorphic language, whose semantics is in the initial *pre-locos*. A *pre-locos* is a distributive category with a parameterized list constructor [1, 2, 3, 6, 7]. A distributive category has finite products and coproducts with the product distributing over the coproduct. The initial pre-locos is the term-category generated from these requirements.
Term-category based semantics are of considerable interest as they have the property that every program is guaranteed to terminate. There is also a complete equational style logic which totally describes logical equivalence between programs for each such term-category. It should be noted that this equivalence may be strictly weaker than *behavioral* equivalence: where two maps have the same behavior if and only if they have the same effect on elements (that is maps with domain the final object).

The **charity** language currently uses categorical notation. We refer to this language as *categorical combinators*. As the nomenclature suggests there is an *equational language* whose role parallels that of lambda calculus with its equivalence to the combinator algebra [5]. It is intended to add translation routines from equational notation to categorical combinators in the future. This will considerably enhance the usability of the system.
The **charity** interpreter acts as an interface to the *charity evaluation machine* which lazily rewrites elements of the initial locos to their canonical form using a technique which we have called chain rewriting. This charity evaluation machine currently only embodies a small number of rewriting rules associated with the product, coproduct, and list constructors. However, the chain rewriting technique is quite general and it is intended eventually to allow the user to determine his own rewriting rules by specifying the data structures he wishes to use (in the sense of [4]).
One of the main concerns of this initial implementation is to show that a chain rewriting system can be implemented efficiently. We are aware that the viability of extensions depends critically on the efficiency of the underlying implementation of the chain rewriting system. At the time of writing it is still difficult to determine whether this goal has been achieved: there are still significant implementation steps to be undertaken before the system can be realistically benchmarked. However, the indications we have from informal comparisons give reason for optimism.
These changes, however, will only effect the machine and its speed: the syntax of the interpreter is relatively stable. Thus, we felt it was appropriate to write a manual for the system in its current state.

3 Getting started

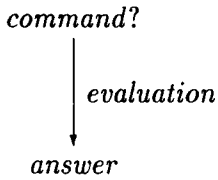
Before invoking the charity interpreter, the user should ensure that the charity interpreter has been installed and can be called from the current directory. Appendix A contains the charity files **arith.ch** and **arithex.ch**. These should be available with the distribution and should be placed in the current directory if these examples are to be followed.

To invoke the charity interpreter type **charity** and the charity prompt will appear:

```
char>
```

One can then read in the files or request evaluation of some expressions. By turning the trace on, one can also follow the evaluation of a charity combinator step by step.

The underlying command format of the interpreter is:



When the user inputs a command the interpreter evaluates the command to an answer. Every command to the interpreter must be terminated by:

```
?<cr>
```

the question mark followed by carriage return tells the interpreter to execute the preceding command. Until this termination sequence is encountered the interpreter will wait for more input.

The most common command is a categorical combinator, whose syntax is described in section 5, which the user wishes to have evaluated. In this case the expression will be translated into an internal representation and be passed to the charity evaluation machine. When the evaluation machine has finished rewriting, the resulting canonical expression is passed back to the interpreter and displayed.
The next most common commands are *specifications*, see section 4. The various specification modes are for interacting with the interpreter in a non-evaluation mode. For example the **set** mode is for setting switches in the interpreter, such as turning the trace on:

```
set:{trace}?
```

Finally, there is one rather important command:

```
halt?
```

which allows one to exit the interpreter.

Below is an example session which uses the charity files **arith.ch** and **arithex.ch**.

```
% charity

Charity Interpreter
version 0.4

char> file:{arith}?
reading arith.ch:
  add := _act(i,_cons)
  mult := _act(<i,[]>,_p1.<_p0,_add>).p1
  exp := _act(<i,[!]>,_p1.<_p0,_mult>).p1
  0 := []
  1 := [!]
  2 := [!,!]
  3 := [!,!,!]
  reading arithex.ch:
    32m := _2.<i,i>.mult.<i,_2>.mult.<i,_2>.mult.<i,_2>.mult
    64e := <<_2,_3>_mult,_2>_exp
    64m := _2.<i,i>.mult.<_2,i>.mult.<i,i>.mult
    128m := _2.<i,i>.mult.<_2,i>.mult.<i,i>.mult.<_2,i>.mult
    256m := _2.<i,i>.mult.<i,i>.mult.<i,i>.mult
    512e := <<_2,_3>_exp,_2>_exp
    1024m := <<_2,_3>_add,_2>_exp.<i,i>.mult
  read arithex.ch: 0.002 sec
  statistics is on
read arith.ch: 0.003 sec
char> set:{trace}?
trace is on
char> <1,1>.add?
<_1,_1>_add
(1) sub: <_1,_1>*add
(2) act: <_1,_1>*act(i,_cons)
   (3) sub: *1
   (4) done: [!]*
(2) act(cons): <[!]*,_1>_act(i,_cons)
(5) move: <!,<[],_1>_act(i,_cons)>*cons
(6) pr(p0): [!|<[],_1>_act(i,_cons)]*
   (7) canonical: !*
(6) pr(p1): [!|<[],_1>_act(i,_cons)]
   (8) act: <[],_1>*act(i,_cons)
      (9) done: []*
   (8) act(nil): <[],_1>_act(i,_cons)
      (10) sub: *1
      (11) done: [!]*
   (8) act(nil): <[],[!]*>_act(i,_cons)
   (12) canonical: [!]*
(6) pr: [!|[!]*]

[!,!]
```

```
user time used 0 sec. 20000 microsec
system time used 0 sec. 20000 microsec
0 page reclaims
0 page faults

char> set:{notrace}?
trace is off
char> 512e?
_512e

[!,!,!,
```

4 Specification modes

The specification modes are used for changing the state of the interpreter. This can be done by changing the value of one of its internal parameters, (**set**), by defining a macro, (**defn**), or by altering the input (**file**).

4.1 setting switches

The **set** specification mode turns on/off switches in the interpreter.
The format of a **set** specification is:

set:{switch1,switch2,...}?

Switches available at present:

- trace/notrace
- stats/nostats
- log/nolog

4.1.1 trace/notrace

The **trace** facility allows the user to step through the evaluation of a charity combinator. The dormant rewriting heads are indicated by "_", while the active rewriting head is denoted by "*".
An example is:

```
char> set:{trace}?
trace is on
char> <2,2>.add?
<_2,_2>_add
(1) sub: <_2,_2>*add
(2) act: <_2,_2>*act(i,_cons)
   (3) sub: *2
   (4) done: [!,!]*
(2) act(cons): <[!,!]*,_2>_act(i,_cons)
(5) move: <!,<[!],_2>_act(i,_cons)>*cons
(6) pr(p0): [!|<[!],_2>_act(i,_cons)]*
   (7) canonical: !*
(6) pr(p1): [!|<[!],_2>_act(i,_cons)]
   (8) act: <[!],_2>*act(i,_cons)
      (9) done: [!]*
   (8) act(cons): <[!],_2>_act(i,_cons)
   (10) move: <!,<[],_2>_act(i,_cons)>*cons
   (11) pr(p0): [!|<[],_2>_act(i,_cons)]*
      (12) canonical: !*
   (11) pr(p1): [!|<[],_2>_act(i,_cons)] skip
   (11) pr: [!|[!,!]*]
   (18) canonical: [!,!,!]*
```

```
(6) pr: [!|[!,!,!]*]

[!,!,!,!]

user time used 0 sec. 0 microsec
system time used 0 sec. 20000 microsec
0 page reclaims
0 page faults

char>
```

Each step of the trace represents one cycle in the charity evaluation machine. At the end of each line of the trace the system waits for an interaction with the user. The simplest response of the user is to give a <cr> which moves the trace to the next step. In the above interaction note that the **skip** option was used: the possible options are:

continue: Used to prompt the trace to display the effect of the next cycle of the machine. This is specified by a <cr>.

leap: Used to leap to the answer when the user no longer wants to step through evaluation. The trace is still on after a leap. For example:

```
char> set:{trace}?
trace is on
char> <2,2>.add?
<_2,_2>_add
(1) sub: <_2,_2>*add
(2) act: <_2,_2>*act(i,_cons)
   (3) sub: *2
   (4) done: [!,!]*
(2) act(cons): <[!,!]*,_2>_act(i,_cons) leap

[!,!,!,!]

char>
```

abort Abort ends the current evaluation and returns directly to the charity prompt. The trace is still on after an abort.

skip This allows one to skip over intermediate rewritings. It is best used when a right active is at the evaluation head (i.e. act, d_0, d_1, p_0, p_1, ...). In these cases the indented steps which follow in the trace are suppressed until the same level of indentation is reached again.

The first letter of all trace commands is sufficient to obtain the desired action.

4.1.2 stats/nostats

Displays the time and the number of page faults used during the execution of a command. For example:

```
char> set:{trace,stats}? <cr>
trace is on
stats are on

char> 512e?
_512e

[!,!,!,!,!,!,!,!,!,!,!,!,!,!,!,!,!,!,!,!,!,.....,
 !,!,!,!,!,!,!,!,!,!,!,!,!,!,!,!,!,!,!,!,!,!,!,!]

user time used 0 sec. 820000 microsec
system time used 0 sec. 40000 microsec
0 page reclaims
0 page faults

char> set:{notrace,nostats}?
trace is off
stats are off
```

4.1.3 log/nolog

Opens a file named **log. ch** and records the current session until the switch is turned off.

4.2 Defining functions

Macro definitions can be specified to the system using the **defn** mode. This device allows composite maps to be given names. The command has the following format:

defn:{map-name := expression}?

or for two or more maps:

defn:{ map-name$_1$:= exp$_1$, map-name$_2$:= exp$_2$, ... }?

A macro function can only use previously defined functions. If two definitions are given the same name, the last one given will take precedence over the previous declared macro definition and *all macro definitions made after the previous definition will be forgotten.* In this way the system prevents cyclic definitions.
An example of a macro definition is as follows:

```
char> defn:{1 := [!]}? <cr>
1 := [!]

char> defn: {2 := <!,1>.cons, <cr>
```

```
            3 := <!,2>.cons
           }?<cr>
2 := [!|_1]
3 := [!|_2]

char>
```

This example also illustrates the list syntax which is supported in **charity**.

4.3 Reading from files

The **file** facility allows all **charity** commands to be read from a file. The interpreter does this by switching the input channel to the file until the end of the file is encountered. This means that the syntax the file must conform to is exactly the same as that described for the interpreter generally. If an evaluation is initiated from a file while the trace is on then that interaction will default to the standard device (rather than the file).
Charity files should have the extension **.ch**. When specifying a file name, if the extension **.ch** is missing, the extension will be added by the interpreter. If the user does not desire this to happen it is necessary to quote the name of the file (as in `'strange.file'`).
The format for the command is as follows:

$$\textbf{file:}\{\text{file}_1, \text{file}_2, \text{file}_3, \ldots\}?$$

For example to read the file `data.ch` one would write:

```
char> file:{data}? <cr>
reading data:
  1 := []
  trace is off
  log is off
read data: 0.002 sec.
```

Where `data.ch` is the following sequence of commands:

```
defn:{one := <!,nil>.cons}?
set:{notrace, nolog}?
```

It is quite permissible to open files from within files. There is no checking that a circular series of opening request is not present. The onus is on the user to ensure that the facility is used sensibly. However there is a (small) finite limit to the number of files one can hold open so that such abuses should quickly be apparent!

5 Syntax of charity combinators

The **charity** combinators are maps in categorical notation of the initial pre-locos. The evaluation machine can only evaluate elements of the pre-locos. However, maps in general are useful when defining macros. The syntax of the **charity** combinators can be specified as follows:

5.1 Types

Combinators are polymorphic transformations from one type shape to another type shape, composition of combinators and other methods of composite combinator construction are only legal if the types shapes of the combinators unify in the appropriate manner.
The types of the **charity** language may be defined recursively as follows:

(i) 1 is a type

(ii) if A and B are types, $A + B$ is a type (co-product)

(iii) if A and B are types, $A \times B$ is a type (product)

(iv) if A is a type, $list(A)$ is a type (lists of A)

Thus, for example

- $(1 + 1) \times list(1)$,
- $list((1 + 1) \times (1 + 1))$,
- $(list(1) + list(1 + 1)) \times list(1)$

are all legal types.

5.2 Combinators

Every combinator has a most general form for the domain and codomain. This form reflects the polymorphic nature of the combinator. When a combinator is used in an evaluation a more specific domain and codomain will be implied by the context of the calculation.
In the recursive definitions below we let $X, Y, Z, \ldots$ be type variables and the notation $f : X \longrightarrow Y$ mean that f is a combinator with most general domain X and most general codomain Y.

(i) $i : X \longrightarrow X$ is a combinator,

(ii) $! : X \longrightarrow 1$ is a combinator,

(iii) $b_0 : X \longrightarrow X + Y$, $b_1 : X \longrightarrow Y + X$ are combinators,

(iv) $p_0 : X \times Y \longrightarrow X$, $p_1 : X \times Y \longrightarrow Y$ are combinators,

(v) $d_0 : (X + Y) \times Z \longrightarrow (X \times Z) + (Y \times Z)$, $d_1 : Z \times (X + Y) \longrightarrow (Z \times X) + (Z \times Y)$ are combinators,

(vi) $\text{cons} : X \times list(X) \longrightarrow list(X)$ is a combinator,

(vii) $\text{split} : list(X) \longrightarrow 1 + X \times list(X)$ is a combinator,

(viii) if $f : B_1 \longrightarrow Z_1$ and $g : A_2 \times Z_2 \longrightarrow Z_2$ are combinators, then $\text{act}(f, g) : list(A) \times B \longrightarrow Z$ is a combinator if (Z_1, B_1, A_2) and (Z_2, B_1, A_2) unify to (Z, A, B).

(ix) if $f : X_1 \longrightarrow Y_1$ and $g : X_1 \longrightarrow Z_1$ are combinators then $\langle f, g \rangle : X \longrightarrow Y \times Z$ is a combinator if (X_1, Y_1, Z_1) and (X_2, Y_1, Z_1) unify to (X, Y, Z).

(x) if $f : X_1 \longrightarrow Z_1$ and $g : Y_1 \longrightarrow Z_2$ are combinators, then $\langle f; g\rangle : X + Y \longrightarrow Z$ is a combinator if (X_1, Y_1, Z_1) and (X_1, Y_1, Z_2) unify to (X, Y, Z).

(xi) if $f : X_1 \longrightarrow Y_1$ and $g : Y_2 \longrightarrow Z_2$ are combinators then $f.g : X \longrightarrow Z$ is a combinator if $(X_1, Y1, Z_2)$ and (X_1, Y_2, Z_2) unify to (X, Y, Z).

The following is an example of a combinator:

$$\mathrm{act}(i, \mathrm{cons}) : \mathit{list}(X) \times \mathit{list}(X) \longrightarrow \mathit{list}(X).$$

In fact, it is the standard "append" map. The typing given is the most general typing information which can be associated with this map. When this map is combined in the following way:

$$\langle\langle !, \langle !, !.\,\mathrm{nil}\rangle.\,\mathrm{cons}\rangle.\,\mathrm{cons}, \langle !, !.\,\mathrm{nil}\rangle.\,\mathrm{cons}\rangle.\,\mathrm{act}(i, \mathrm{cons})$$

A more specific typing is forced:

$$\mathrm{act}(i, \mathrm{cons}) : \mathit{list}(1) \times \mathit{list}(1) \longrightarrow \mathit{list}(1).$$

Here $\mathit{list}(1)$ is the natural numbers (base 1). Thus, in this carnation the map becomes addition.

It is intended to implement type checking in later versions of charity.

5.3 Canonical elements

the charity evaluation machine can only evaluate *elements*. These are maps of the form:

$$g : 1 \longrightarrow X,$$

that is whose domain is 1. The following are examples of elements:

- $\langle !, \mathrm{nil}\rangle.\,\mathrm{cons}.\langle i, i\rangle.\,\mathrm{act}(i, \mathrm{cons})$,
- $\langle !, \langle !, !.\,\mathrm{nil}\rangle.\,\mathrm{cons}\rangle.\,\mathrm{cons}$,
- $\langle !.b_0, \langle !, !.\,\mathrm{nil}\rangle.\,\mathrm{cons}\rangle.d_0$.

charity evaluates elements to a *canonical form* which is defined as follows (where the type information is respected as above):

- $! : 1 \longrightarrow 1$ is canonical,
- $\langle f, g\rangle : 1 \longrightarrow X \times Y$ is canonical if $f : 1 \longrightarrow X$ and $g : 1 \longrightarrow Y$ are canonical,
- $f.\,\mathrm{cons}$ and $g.\,\mathrm{nil}$ are canonical if f and g are canonical,
- $f.b_0$ and $g.b_1$ are canonical if f and g are canonical.

For example the following are canonical:

- $\langle !.b_1, \langle !.b_0, \langle !.b_1, !.\,\mathrm{nil}\rangle.\,\mathrm{cons}\rangle.\,\mathrm{cons}\rangle : 1 \longrightarrow (X + 1) \times \mathit{list}(1)$
- $\langle !.\,\mathrm{nil}, !.b_1\rangle : 1 \longrightarrow \mathit{list}(X) \times (Y + 1)$

5.4 List syntax

The basic data structure of the **charity** language is the list. For this reason a special syntax for the list construct has been supplied. The syntax is similar to that used in prolog. Its computational significance, however, is rather different as it indicates a certain level of evaluation has already been performed.
We have the following semantic equivalences:

(i) $[] \leftarrow !.\,\mathrm{nil}$,

(ii) $[t_0|t_1] \leftarrow \langle t_0, t_1\rangle.\,\mathrm{cons}$,

(iii) $[t_0, ..., t_r] \leftarrow [t_0, ..., t_r|[]]$,

(iv) $[s_0, ..., s_m, t_1, ...t_r|t] \leftarrow [s_0, ..., s_m|[t_1, ..., t_r|t]]$.

Semantically there is an exact equivalence between $\langle !.b_0, (!.b1, \mathrm{nil}).cons\rangle.cons$ and $[!.b_0, !.b_1]$. However, computationally the second form is advantageous as movement and compacting of the data structures can be done at parse time rather than evaluation time. The syntax itself is, of course, much more succinct and is therefore used extensively in the demonstration programs.

6 Programming binary addition

As an example of a program development we shall show how a small program to do binary addition can be developed. The full program is in appendix B.

The fundamental component of a binary adding operation is the one bit adder. This operation takes as arguments two bits and a carry bit, and outputs one bit and a carry bit. Now the order of the input arguments does not matter but this is not so for the output. We shall call this map carry; its specification is as follows.

$$\mathrm{carry} : 2 \times (2 \times 2) \longrightarrow 2 \times 2;\quad \begin{array}{rcl} (!.b_0, (!.b_0, !.b_0)) & \mapsto & (!.b_0, !.b_0) \\ (!.b_0, (!.b_0, !.b_1)) & \mapsto & (!.b_0, !.b_1) \\ (!.b_0, (!.b_1, !.b_0)) & \mapsto & (!.b_0, !.b_1) \\ (!.b_0, (!.b_1, !.b_1)) & \mapsto & (!.b_1, !.b_0) \\ (!.b_1, (!.b_0, !.b_0)) & \mapsto & (!.b_0, !.b_1) \\ (!.b_1, (!.b_0, !.b_1)) & \mapsto & (!.b_1, !.b_0) \\ (!.b_1, (!.b_1, !.b_0)) & \mapsto & (!.b_1, !.b_0) \\ (!.b_1, (!.b_1, !.b_1)) & \mapsto & (!.b_1, !.b_1) \end{array}$$

Here $2 = 1 + 1$ and the elements $!.b_0$ and $!.b_1$ represent respectively 1 and 0 (the necessity for this clumsy representation will be removed in later versions).
The problem is to turn this into a combinator form (it is intended this will be automatic in later versions). The process is, in fact, relatively easy: start by distributing the first coordinate, then the second coordinate, then the third coordinate. This gives a tree representation of this map:

```
carry:= d0.
        <p1.d0.
              <p1.
```

```
                <<!.b0,!.b0>
                ;<!.b0,!.b1>
                >
        ;p1.
                <<!.b0,!.b1>
                ;<!.b1,!.b0>
                >
        >
;p1.d0.
        <p1.
                <<!.b0,!.b1>
                ;<!.b1,!.b0>
                >
        ;p1.
                <<!.b1,!.b0>
                ;<!.b1,!.b1>
                >
        >
>
```

The next stage is to use this map to add numbers. To do this we must build a state machine the state of which will contain the answer of the addition as a component. This machine will be driven by the bits of one of the binary numbers its initial state will be determined by the other number. As each bit comes in it will build up a partial answer.
The state will consist of three components: the other binary number (with bits in reverse order), a carry bit, and the partial answer. The incremental step is to introduce the next bit from the first number and update the partial answer. This involves striping the top bit from the other number and doing a one bit addition using carry and updating the carry bit and the partial answer. This map we shall call cadd and it may be specified as follows:

$$\text{cadd} : 2 \times (\mathit{list}(2) \times (2 \times \mathit{list}(2))) \longrightarrow \mathit{list}(2) \times (2 \times \mathit{list}(2));$$

$$\begin{aligned} (x, (\text{nil}, (y, z))) &\mapsto (\text{nil}, (p_0(\text{carry}(b_0, (x, y))), \text{cons}(p_1(\text{carry}(b_0, (x, y)), z))) \\ (x, (\text{cons}(v, w), (y, z))) &\mapsto (w, (p_0(\text{carry}(v, (x, y))), \text{cons}(p_1(\text{carry}(v, (x, y)), z))) \end{aligned}$$

Again the problem is to turn this into combinators. To do this we split the first list and write down the remaining parts of the map. This defines an action badd_1 (which is initialized by reversing the second number):

```
cadd1:= <<p1.p1.p0,<p0,!.b0>>.carry,p1.p1.p1>.
        <!.nil,<p0.p0,<p0.p1,p1>.cons>>,

cadd2:= <<p1.p1.p0,<p0,p1.p0.p0>>.carry,<p1.p0.p1,p1.p1.p1>>.
        <p1.p0,<p0.p0,<p0.p1,p1.p1>.cons>>,

cadd:=
       <p0,<p1.p0.split,p1.p1>.d0>.d1.
       <cadd1
```

```
        ;cadd2
        >,

badd1:= act(<reverse,<!.b0,!.nil>>,cadd)
```

If the first number has more bits than the second number we are almost done. However, if the converse is the case we will still have the most significant bits of the second number embedded in the state. Thus, we need to add a further operation which adds any significant bits left of the second number into the partial answer. This we shall call $badd_2$ and is defined as:

```
badd2:= <p0,i>.act(i,<!.b0,p1>.cadd),
```

Binary addition is performed by composing these maps and then appending the carry bit when it is set:

```
cca:= d0.
        <p1
        ;<!.b1,p1>.cons
        >,

baddr:= baddr1.badd2.p1.cca,

badd:= badd1.badd2.p1.cca
```

Appendix B contains the completed program together with a version of binary multiplication.
We may now run these routines on some examples:

```
Charity Interpreter
version 0.4

char> file:{binadd,binmult}?
reading binadd.ch:
  carry := _d0.<_p1.d0.<_p1.<<!.b0,!.b0>;<!.b0,!.b1>>
               ;_p1.<<!.b0,!.b1>;<!.b1,!.b0>>>
           ;_p1.d0.<_p1.<<!.b0,!.b1>;<!.b1,!.b0>>
               ;_p1.<<!.b1,!.b0>;<!.b1,!.b1>>>>
  cadd1 := <<_p1.p1.p0,<_p0,!.b1>>_carry,_p1.p1.p1>.
               <[],<_p0.p0,[_p0.p1|_p1]>>
  cadd2 := <<_p1.p1.p0,<_p0,_p1.p0.p0>>_carry,<_p1.p0.p1,_p1.p1.p1>>.
               <_p1.p0,<_p0.p0,[_p0.p1|_p1.p1]>>
  cadd := <_p0,<_p1.p0.split,_p1.p1>_d0>_d1.<_cadd1;_cadd2>
  reverse := <i,i>_act(<i,[]>,_p1.<_p0.split,_p1>.d0.
```

```
                <i;<_p0.p1,[_p0.p0|_p1]>>).p1
  baddr1 := _act(<i,<!.b1,[]>>,_cadd)
  badd1 := _act(<_reverse,<!.b1,[]>>,_cadd)
  badd2 := <_p0,i>_act(i,<!.b1,_p1>_cadd)
  cca := _d0.<[!.b0|_p1];_p1>
  baddr := _baddr1.badd2.p1.cca
  badd := _badd1.badd2.p1.cca
read binadd.ch .005 sec
reading binmult.ch:
  bmult1 := _d0.<<_p1.baddr,[_b1|_p1.p1]>;<_p1.p0,[!.b1|_p1.p1]>>
  bmult := _act(<[],_reverse>,_bmult1).p0
read binmult.ch: .001 sec
char> set:{stats}?
statistics is on
char> <[!.b0,!.b0,!.b1],[!.b0,!.b0]>.bmult?
<[!.b0,!.b0,!.b1],[!.b0,!.b0]>_bmult

[!.b0,!.b1,!.b1,!.b0,!.b1]

user time used 0 sec. 120000 microsec
system time used 0 sec. 0 microsec
0 page reclaims
0 page faults

char> <[!.b0,!.b1,!.b1,!.b0,!.b1],[!.b0,!.b0,!.b1,!.b0]>.badd?
<[!.b0,!.b1,!.b1,!.b0,!.b1],[!.b0,!.b0,!.b1,!.b0]>_badd

[!.b0,!.b0,!.b0,!.b0,!.b0]

user time used 0 sec. 120000 microsec
system time used 0 sec. 0 microsec
0 page reclaims
0 page faults

char> [!.b0,!.b0,!.b0,!.b0,!.b1,!.b1,!.b1,!.b1,!.b1,!.b0].<i,i>.badd?
[!.b0,!.b0,!.b0,!.b0,!.b1,!.b1,!.b1,!.b1,!.b1,!.b0].<i,i>_badd

[!.b0,!.b0,!.b0,!.b0,!.b1,!.b1,!.b1,!.b1,!.b1,!.b0,!.b1]

user time used 0 sec. 180000 microsec
system time used 0 sec. 0 microsec
0 page reclaims
0 page faults

char>
```

7 Discussion of extensions

There are a number of respects in which **charity** falls short of a full blown programming language. Not only are there no interface routines to the outside world but the available types do not allow one to make use of the architecture of the underlying machine. For example, both binary addition and multiplication are implemented in hardware for sixteen bit words: our own routines above are a poor substitute!

There is no reason why the **charity** machine should not take advantage of hardware implemented functions. Indeed, it appears to be quite simple to consistently add such features (especially if the carry bit is made available and multiplication is to a double word). Similarly, adding primitive types for character strings and adding writing and reading commands, which would facilitate interactive programs, seems a relatively straightforward step.

However, this was not the main purpose behind the initial development. There was a worrying question to be answered: could this categorical paradigm be made to perform at a reasonable speed? This implementation, to some extent, has answered this questions. It has yet to be benchmarked carefully against comparable systems but the informal work we have done in this direction is cause for some encouragement.

The next steps in the development of the interpreter include adding a type checker (this is relatively simple), a facility for creating data structures, and an equational interface to the combinator language. These will greatly enhance its usability. These will not address the problems raised above. As those issues are "ad hoc" we shall probably delay them until there is a fairly complete pure system.

When all these improvements are made we will be in a much better position to benchmark the system sensibly. It is our *hope* that we can achieve performances comparable if not better than **lisp** and **prolog**.

References

[1] J.R.B. Cockett, *List-arithmetic open categories: basic properties.* University of Tennessee, Department of Computer Science, Technical Report. CS-88-76 (1988).

[2] J.R.B. Cockett, *On the decidability of objects in a locos.* Proceedings of the Conference in Computer Science and Logic, June 14-20, 1987, University of Colorado, Boulder. Editors: J.W. Gray and A. Scedrov (to appear 1989)

[3] A. Burroni, *Recursivite graphique (premier partie): catégorie des fonctions récursive primitive formelles.* Cahier de Topologie et Geométrie Différentielle Catégorique, Vol XXVII - 1 (1986) 49-79.

[4] T. Hagino, *A categorical programming language* PhD. dissertation University of Edinburgh.

[5] J. Lambek, P. Scott, *Introduction to higher order categorical logic,* Cambridge studies in advanced mathematics 7, Cambridge University Press, London (1986).

[6] L. Román, *Cartesian Categories with Natural Numbers Object,* Dept. Mathematics, McGill University, (Submitted to J. of Pure Appl. Algebra, 1986).

[7] L. Román, *Categories with Finite Limits and a Natural Numbers Object,* Dept. Mathematics, McGill University, (Submitted to J. of Pure Appl. Algebra, 1987).

A Arithmetic Using Lists

There are two files which contain the basic functions of arithmetic over lists: `arith.ch` and `arithex.ch`.

arith.ch:

```
%       Basic arithmetic operations (base 1)

defn:{ add := act(i,cons),                %       also append!
       mult:= act(<i,[]>,p1.<p0,add>).p1,
       exp := act(<i,[!]>,p1.<p0,mult>).p1
              % note argument are reversed on exponentiation
     }?

%       some small numbers

defn:{  0 := [],
        1 := [!],
        2 := [!,!],
        3 := [!,!,!]
     }?

file:{arithex}?

set:{stats}?
```

arithex.ch:

```
defn:{ 32m := 2.<i,i>.mult.<i,2>.mult.<i,2>.mult.<i,2>.mult,
       64e := <<2,3>.mult,2>.exp,
       64m := 2.<i,i>.mult.<2,i>.mult.<i,i>.mult,
       128m := 2.<i,i>.mult.<2,i>.mult.<i,i>.mult.<2,i>.mult,
       256m := 2.<i,i>.mult.<i,i>.mult.<i,i>.mult,
       512e := <<2,3>.exp,2>.exp,
       1024m := <<2,3>.add,2>.exp.<i,i>.mult
      }?
```

B Binary Arithmetic

Here are the two files for implementing binary addition and multiplication:

binadd.ch:

```
defn:{
%       carry: 2 x (2 x 2) --> 2 x 2
% is a one add bit adder with a carry bit.

       carry:= d0.<p1.d0.<p1.<<!.b0,!.b0>;<!.b0,!.b1>>;
                         p1.<<!.b0,!.b1>;<!.b1,!.b0>>>;
                    p1.d0.<p1.<<!.b0,!.b1>;<!.b1,!.b0>>;
                         p1.<<!.b1,!.b0>;<!.b1,!.b1>>>
                   >,

%      cadd: 2 x (list(2) x (2 x list(2))) --> list(2) x (2 x list(2))
%      this strips the head from the first list and uses it in one
%      bit addition  to build the second list.

        cadd1:= <<p1.p1.p0,<p0,!.b1>>.carry,p1.p1.p1>.
                <[],<p0.p0,<p0.p1,p1>.cons>>,

        cadd2:= <<p1.p1.p0,<p0,p1.p0.p0>>.carry,<p1.p0.p1,p1.p1.p1>>.
                <p1.p0,<p0.p0,<p0.p1,p1.p1>.cons>>,

        cadd:= <p0,<p1.p0.split,p1.p1>.d0>.d1.
               <cadd1
               ;cadd2
               >,

        reverse:= <i,i>.act(<i,[]>,p1.<p0.split,p1>.
                      d0.<i;<p0.p1,<p0.p0,p1>.cons>>).p1,

        baddr1:= act(<i,<!.b1,!.nil>>,cadd),

        badd1:= act(<reverse,<!.b1,[]>>,cadd),

        badd2:= <p0,i>.act(i,<!.b1,p1>.cadd),

        cca:= d0.<<!.b0,p1>.cons;p1>,

        baddr:= baddr1.badd2.p1.cca,

        badd:= badd1.badd2.p1.cca
       }?
```

binmult.ch:

```
defn:{bmult1:= d0.<<p1.baddr,<b1,p1.p1>.cons>;
                      <p1.p0,<!.b1,p1.p1>.cons>>,
```

```
 bmult:= act(<!.nil,reverse>,bmult1).p0
}?
```

C An Initial Benchmark Test against QUINTUS Prolog

The **charity** system as it stands was compared in its simple list processing operations to QUINTUS prolog. This was done by comparing both "naive" reverse and shift reverse on a list with 100, 200, and 300 elements as it is interpreted in both systems. Next the QUINTUS compiler was used and the systems compared again.
In **prolog** the programs used were:

```
reverse(L,RL):-
reverse(L,[],RL).

reverse([],L,L).
reverse([H|T],L1,RL):-
reverse(T,[H|L1],RL).


naive_reverse([],[]).
naive_reverse([H|T],R):-
naive_reverse(T,R1),append(R1,[H],R).

append([],X,X).
append([X|T],Y,[X|YT]):-
append(T,Y,YT).
```

First the predicates were consulted and executed, next the predicates were compiled and executed. Execution time was obtained using the built-in predicate `statistics`. Each test was run several times and an average rounded execution time taken. The interpreted naive reverse on two hundred elements forced excessive paging: on three hundred elements we did not even wait for the response. Once the programs were compiled the running times seemed to be almost linear for both techniques!
The **charity** the programs are:

```
charity/test.ch
defn:{naivereverse := <i,!>.act([],<p1,[p0]>.act(i,cons)),
      reverse:= <i,i>.act(<i,[]>,p1.<p0.split,p1>.d0.
                  <i;<p0.p1,[p0.p0,p1]>>).p1,
      }?
```

The performance of the **charity** system was comparable to the prolog interpreter. The naive reverse seemed to be performing marginally faster (although the **charity** interpreter ran out of storage on the lists of length 200: this was no surprise as the limits are set very low while the system is still under development).

prolog	NR	3.8	211.9	
	R	0.5	0.9	1.3
prolog	NR	0.45	1.1	1.8
(compiled)	R	0.36	0.8	1.1
charity	NR	3.2		
	R	0.48	0.96	1.36

These results are reasonably encouraging as we are well aware that we have not squeezed all the speed which can be obtained out of this implementation of **charity**.

VITA

Hong G. Chen was born in Taichung, Taiwan on September 18, 1952. He attended elementary schools in that city and graduated from First Provincial High School in June 1970. In September 1971 he entered National Taiwan University, and he received a Bachelor of Art degree in Sociology in June 1975. In the fall of 1978 he entered The University of Tennessee, Knoxville and began study towards a Master of Business Administration concentrating in Accounting. This degree was awarded in June 1982.

After taking several undergraduate computer science courses, he re-entered the master's program in computer science at the University of Tennessee, Knoxville, in March 1983. He received a Master of Science degree in Computer Science in December 1984.

After working with Tektronix as a systems analyst for a year, he was accepted to the Ph.D. program in computer science at the University of Tennessee in September 1986. He received the Doctor of Philosophy degree with a major in Computer Science in December 1990.