Strong Categorical Datatypes II : A term logic for categorical programming

J. Robin B. Cockett
Dept. of Computer Science
University of Calgary
Calgary, Alberta T2N 1N4, Canada
cockett@cpsc.ucalgary.ca

Dwight Spencer
Dept. of Computer Science and Engineering
Oregon Graduate Institute
Beaverton, Oregon 97006-1999
dwights@cse.ogi.edu

May 12, 1992

Abstract

This paper lifts the category-theoretic results of [4] to the level of an abstract language suitable for basing categorical programming language implementations. The earlier work built a fibration-based strongly-normalizing categorical combinator reduction system based entirely on functorial strength that allows the distribution of context to the interior of a strong data structure. Strong type-forming functors accompanied by (1) a collection of constructor combinators (initial datatypes) or destructor combinators (final datatypes) and (2) a capability for building new state-transforming combinators that operate with structures of the datatypes formed by these functors can be abstractly declared in a Hagino-Wraith style to form a reasonably expressive computing environment.

However, the high complexity of programming exclusively in combinators warrants the development of categorical programming languages that are isomorphic to the distributive category settings in which strong datatypes can be declared. Towards this goal, a distributive term logic is developed and proven consistent with and equivalent to the combinator theory. Due to its basis in strength higher-order operations, such as generalized `map` operators, are directly expressible in this first-order language. The term logic forms the underlying base of the **Charity** categorical programming system. The resulting categorical programming paradigm of structurally inductive state transforms is illustrated by several examples that accompany the term logic reduction rules.

1 Introduction

We recall from Part I [4] that we may build a categorical setting for strength-based, or *strong*, initial and final datatypes by selecting a cartesian category theory (i.e. finite products) and successively adding to it collections of special combinators for each datatype as it is adjoined to the theory. The adjoining is accomplished by declaration; the two declarative formats are shown below with **Charity** [3] code. They portray the factorizer versions of the Hagino "right" and "left" forms of datatype declaration [5] done in a style similar to that suggested by Wraith [12]. Wraith's style allows translation of the resulting category theory into the polymorphic lambda calculus and thereby allowing the theory to inherit the calculus' strong normalization property.

$$
\begin{array}{llcll}
\textbf{data} & C \longrightarrow R(A) \quad = & \qquad & \textbf{data} & L(A) \longrightarrow C \quad = \\
 & d_1 : C \longrightarrow E_1(A, C) & & & c_1 : E_1(A, C) \longrightarrow C \\
 & \quad \ldots & & & \quad \ldots \\
 & d_n : C \longrightarrow E_n(A, C). & & & c_n : E_n(A, C) \longrightarrow C.
\end{array}
$$

The "left" declaration delivers a strong initial datatype-forming endofunctor L in which $L(A)$ is a datatype of structures whose basic, or structurally internal, data are elements of the parameter type A. L is strong because it arrives paired with a parametrized combinator (natural transformation) called a *strength*

$$\theta^L_{A,X} : L(A) \times X \longrightarrow L(A \times X)$$

that distributes the environment or context X throughout the data structure $L(A)$. This allows the context to be accessible for processing by actions on basic data in the structure. The labels "c_i" refer to combinators, termed *constructors*,

$$c_i : E_i(A, L(A)) \longrightarrow L(A)$$

that are canonical embeddings which build the data structure $L(A)$ from its component structures $E_i(A, L(A))$. In fact, $L(A)$ is the categorical coproduct of the $E_i(A, L(A))$ via the c_i.

The "left" declaration code itself asserts that any map from the new type $L(A)$ to a state or result type C is to be determined uniquely by a set of n parametric programmer-chosen maps or state-transformers from $E_i(A, C)$ to C. That is, a new combinator that operates on $L(A)$ data structures, called a *fold factorizer* and denoted fold^L, becomes available simply by specifying a collection of n state transformers that employ only the theory generated by earlier declarations.

Analogously, the "right" declaration format acts dually to also provide a type-forming functor possessing a strength. In this case, a set of canonical *destructors* $d_i : R(A) \longrightarrow E_i(A, L(A))$, by which $R(A)$ is the categorical product of the $E_i(A, L(A))$, is spawned. A destructor d_i essentially projects the i-th "field" of a right data structure having type $R(A)$. The declaration also allows building new combinators that produce $R(A)$ data structures from a state or input type C. Such an R data structure-producing combinator is called an *unfold factorizer* and denoted unfold^R. Each unfold factorizer is built by specifying a collection of n parametric programmer-chosen state de-transformers from C to $E_i(A, C)$.

The labels c_i and d_i name only the canonical operations and should not be confused with the programmer-chosen maps. These constructors and destructors are universally associated with their respective datatypes and in one-to-one correspondence with the programmer-chosen maps used for building each factorizer. By this discussion it is now clear that the "right" definition declares a final datatype and the "left" an initial datatype.

The chosen parameter type variable A usually ranges over a power of a given parameter datatype category $\mathbf{A}$. However, we shall take the interpretation that it simply ranges over some category and assume the typing $E_i : \mathbf{A} \times \mathbf{C} \longrightarrow \mathbf{C}$ where $\mathbf{C}$ serves as an approporiate category of state types.

The strong initial and final datatypes were required in Part I to satisfy respectively universal strong initiality diagrams

$$
\begin{tikzcd}
E_i(A, L(A)) \times X \arrow[r, "c_i \times id_X"] \arrow[d, "\langle \theta_2^{E_i}, p_1 \rangle"'] & L(A) \times X \arrow[dd, dotted, "fold^L\{hs\}"] \\
E_i(A, L(A) \times X) \times X \arrow[d, dotted, "E_i(A, fold^L\{hs\}) \times id_X"'] & \\
E_i(A, C) \times X \arrow[r, "h_i"'] & C
\end{tikzcd}
$$

and universal strong finality diagrams

$$
\begin{tikzcd}
C \times X \arrow[r, "\langle g_i, p_1 \rangle"] \arrow[dd, dotted, "unfold^R\{gs\}"'] & E_i(A, C) \times X \arrow[d, "\theta_2^{E_i}"] \\
& E_i(A, C \times X) \arrow[d, dotted, "E_i(A, unfold^R\{gs\})"] \\
R(A) \arrow[r, "d_i"'] & E_i(A, R(A))
\end{tikzcd}
$$

from which rewrite rules for the fold and unfold factorizers were directly derived.

Additional specializations of these factorizers were also defined to provide a reasonable combinator tool-kit for programming. The *case factorizer* $case^L$ carries out a single operation on an initial data structure to produce a state. The *record factorizer* $record^R$ does a single operation on a state to produce a final data structure. The *map factorizers* map^L and map^R, respectively for initial and final datatypes, perform the same operation on each basic datum of a data structure while being recursively driven by that same structure. Thus the special factorizers operate in the same way as the familiar like-named functions used in conventional functional programming systems.

With the assumptions that an initial datatype declaration provides for n constructors that build structures of a datatype $L(A)$ of parametric arity m, the set of generated rewriting rules that are selected to augment the previously declared theory become:

$$
\begin{array}{rcl}
c_i \times id_X \,; fold^L\{h_1, \ldots, h_n\} & \Longrightarrow & \langle \mathrm{map}^{E_i}\{p_0, fold^L\{h_1, \ldots, h_n\}\}, p_1 \rangle \,; h_i \\
c_i \times id_X ; case^L\{h_1, \ldots, h_n\} & \Longrightarrow & h_i \\
c_i \times id_X \,; \mathrm{map}^L\{f_1, \ldots, f_m\} & \Longrightarrow & map^{E_i}\{(f_1, \ldots, f_m), map^L\{f_1, \ldots, f_m\}\} \,; c_i
\end{array}
$$

Similarly, the rules brought forth for augmentation by a declaration of the datatype $R(A)$ are:

$$
\begin{array}{rcl}
\mathit{unfold}^R\{g_1,\ldots,g_n\}\,;d_i & \Longrightarrow & \langle g_i, p_1\rangle\,;\mathit{map}^{E_i}\{p_0, \mathit{unfold}^R\{g_1,\ldots,g_n\}\} \\
\mathit{record}^R\{g_1,\ldots,g_n\}\,;d_i & \Longrightarrow & g_i \\
\mathit{map}^R\{f_1,\ldots,f_m\}\,;d_i & \Longrightarrow & d_i \times \mathit{id}_X\,;\mathit{map}^{E_i}\{(f_1,\ldots,f_m), \mathit{map}^R\{f_1,\ldots,f_m\}\}
\end{array}
$$

As in the earlier paper, the combinator sequence $\theta^F\,;F(f_1,\ldots,f_k)$ is represented as the combinator $\mathit{map}^F\{f_1,\ldots,f_k\}$ for any strong functor F. The collections $\{h_i\}$ and $\{g_i\}$ represent the programmer-chosen maps that respectively parameterize the fold^L and unfold^R combinators. The collection $\{f_i\}$ are the programmer-chosen mapping actions.

The standard set of cartesian rewriting rules combined with the new rules accumulated by any datatype declarations form a confluent and terminating polymorphic reduction system for evaluating programs that compute with the declared datatypes.

Definitions of some useful strong datatypes are given below. The less familiar ones likely warrant some explanation. The DBTree datatype permits node data and leaf data to have different structures. Colists capture all non-empty finite lists as well as the infinite lists. In fact, the cotail destructor is partial: the tail of a length-1 list does not exist. Similarly, the Cotree type contains all finite binary trees as well as infinite binary trees, and its cobranch destructor is partial. Bushes are unbounded-branching-factor tree structures.

Booleans

$$
\begin{array}{l}
\textbf{data}\ \mathrm{Bool} \longrightarrow C\ = \\
\quad \mathrm{true} : 1 \longrightarrow C \\
\quad \mathrm{false} : 1 \longrightarrow C.
\end{array}
$$

Natural Numbers

$$
\begin{array}{l}
\textbf{data}\ \mathrm{Nat} \longrightarrow C\ = \\
\quad \mathrm{zero} : 1 \longrightarrow C \\
\quad \mathrm{succ} : C \longrightarrow C.
\end{array}
$$

Finite Lists

$$
\begin{array}{l}
\textbf{data}\ \mathrm{List}(A) \longrightarrow C\ = \\
\quad \mathrm{nil} : 1 \longrightarrow C \\
\quad \mathrm{cons} : A \times C \longrightarrow C.
\end{array}
$$

Finite Binary Trees

$$
\begin{array}{l}
\textbf{data}\ \mathrm{Btree}(A) \longrightarrow C\ = \\
\quad \mathrm{bleaf} : A \longrightarrow C \\
\quad \mathrm{bnode} : C \times C \longrightarrow C.
\end{array}
$$

$$
\begin{array}{l}
\textbf{data}\ \mathrm{DBTree}(A,B) \longrightarrow C\ = \\
\quad \mathrm{dbleaf} : A \longrightarrow C \\
\quad \mathrm{dbnode} : B \times (C \times C) \longrightarrow C.
\end{array}
$$

Infinite Binary Trees

$$
\begin{array}{l}
\textbf{data}\ C \longrightarrow \mathrm{Inftree}(A)\ = \\
\quad \mathrm{root} : C \longrightarrow A \\
\quad \mathrm{branch} : C \longrightarrow (C \times C).
\end{array}
$$

$$
\begin{array}{l}
\textbf{data}\ C \longrightarrow \mathrm{Cotree}(A,B)\ = \\
\quad \mathrm{coroot} : C \longrightarrow A \\
\quad \mathrm{cobranch} : C \longrightarrow 1 + (C \times C).
\end{array}
$$

Infinite Lists

$$
\begin{array}{l}
\textbf{data}\ C \longrightarrow \mathrm{Inflist}(A)\ = \\
\quad \mathrm{head} : C \longrightarrow A \\
\quad \mathrm{tail} : C \longrightarrow C.
\end{array}
$$

$$
\begin{array}{l}
\textbf{data}\ C \longrightarrow \mathrm{Colist}(A)\ = \\
\quad \mathrm{cohead} : C \longrightarrow A \\
\quad \mathrm{cotail} : C \longrightarrow 1 + C.
\end{array}
$$

Finite Products

$$
\begin{array}{l}
\textbf{data}\ C \rightarrow \mathrm{Nprod}(A_1,\ldots,A_n)\ = \\
\quad \mathrm{field1} : C \longrightarrow A_1 \\
\quad \ldots \\
\quad \mathrm{fieldn} : C \longrightarrow A_n.
\end{array}
$$

Finite Sums

$$
\begin{array}{l}
\textbf{data}\ \mathrm{Nsum}(A_1,\ldots,A_n) \rightarrow C\ = \\
\quad \mathrm{tag1} : A_1 \longrightarrow C \\
\quad \ldots \\
\quad \mathrm{tagn} : A_n \longrightarrow C.
\end{array}
$$

Bushes

$$
\begin{array}{l}
\textbf{data}\ \mathrm{Bush}(A,B) \longrightarrow C\ = \\
\quad \mathrm{fruit} : A \rightarrow C \\
\quad \mathrm{fork} : B \times (\mathrm{List}(C) \times \mathrm{List}(C)) \rightarrow C.
\end{array}
$$

$$
\begin{array}{l}
\textbf{data}\ C \longrightarrow \mathrm{InfBush}(A,B)\ = \\
\quad \mathrm{limb} : C \rightarrow A + B \times (\mathrm{List}(C) \times \mathrm{List}(C)).
\end{array}
$$

An infinitude of type-instantiations and hybrids of these datatypes are evidently available. For example, data structures similar to

$$
\begin{array}{ll}
\textbf{data} & C \longrightarrow \mathrm{Syntaxtree}(\mathrm{Binding},\mathrm{Token})\ = \\
& \mathrm{phraselook} : C \longrightarrow \mathrm{Binding} \times (\mathrm{Token} \times (C + \mathrm{list}(C))).
\end{array}
$$

have been employed as abstract syntax trees for language compilation.

A prototypical, yet versatile, categorical programming language (term logic) for all such strong datatypes whose computation semantic is isomorphic to these combinator rules is built in Section 2. It is intended to be an abstract language over which categorically-based functional languages supporting strong datatypes may be designed. In fact, one such language implementation — **Charity** — is currently an operational testbed for categorical programming using mixtures of eager initial and lazy final datatypes (Cockett and Fukushima [3]). The functional completeness and correctness of the term logic is exhibited in Section 3 as an equivalence between it and a cartesian category theory closed under the adjoining of strong datatypes. The equivalence is expressed as a pair of translations going in opposite directions. The categorical programming paradigm brought forth by the term logic is demonstrated by coding examples in Section 4.

2 Term Logic

Coding solely with categorical combinators is an intimidating and unintuitive style of programming. In this sense, raw combinators are an overly detailed mode of categorical computation. This section presents a higher-level programming language in the form of a term logic that eliminates the plenitude of projections occurring within combinator expressions. These projections appear naturally because of the distributivity properties of our categorical setting. In particular, the projections perform the distribution of the environment, or context, in this strong setting. The term logic will be shown to correspond exactly to these basic distributive structures.

We proceed by augmenting a "seed" cartesian theory that corresponds to a cartesian category, i.e. one possessing finite products, with terms and rules that reflect the new processing available by the addition of a finite sequence of strong datatypes to the original cartesian category. Our construction generalizes the term logic development presented for predistributive categories in Cockett [1].

The definition of the term logic is motivated by considering how programs in the logic should be translated or compiled into categorical combinators. The overall picture that will be arise from the forthcoming definitions is to treat a *program* t *with context* v as the unit of translation and represent it as a special binding operator

$$\{v \mapsto t\}$$

called a *closed abstract map* where t is a term and v is a variable-binding expression — a *variable base* — containing all the free variables of t. This form permits the binding of variables in t for substitution by the component variables of the variable base whenever the program is applied to a term t' having the same type as that possessed by the base.

2.1 The Cartesian Theory

The familiar manipulation of data built from products is generalized here to introduce the term notation and to provide a precise foundation for adding later some strong datatypes to a cartesian logic.

A specification of a cartesian theory, $\mathcal{D}_0 = (T_0, F_0, S_0, E_0)$, consists of a set T_0 of *primitive types*, a set F_0 of *function symbols*, a *signature* S_0 for the function symbols, and a set E_0 of *equations* between programs with like-typed contexts. The signature is a map

$$S_0 : F_0 \to T_0 \times T_0 \quad .$$

that provides, via projections, the domain and codomain primitive types for each function symbol. Using the sets of rules below, the full cartesian theory can be generated.

Types:

The collection of *types* for the cartesian theory is defined inductively starting from the primitive types:

(i) If $\tau \in T_0$ then τ is a type.

(ii) 1 is a type.

(iii) If τ_0 and τ_1 are types then $\tau_0 \times \tau_1$ is a type.

Variables and Variable Bases:

With every type τ there is assumed to be a countable collection of variables:

$$v_\tau, v_\tau^{(1)}, v_\tau^{(2)}, v_\tau^{(3)}, \ldots, v_\tau^i, \ldots$$

For clarity the superscripting will often be omitted and the typing abbreviated, with both being inferable from discussion context. For example, v_i will typically mean v_{τ_i} for some assumed indexed collection of types $\{\tau_i\}$.

In addition to isolated variables, *variable bases* are specially provided as a well-needed programming convenience to carry out two oft-desired tasks: (1) to bind more than one distinct variable at one time within a single expression and (2) to provide direct binding access (i.e. avoid use of projections) to component data of a structure having a product datatype. These tasks are often combined and treated as the problem of *pattern matching* bound variables to an operand in a function application.

Variable bases with their type associations are defined as follows:

(i) () is variable base of type 1.

(ii) A variable v_τ^i is a variable base for its type τ.

(iii) If v_{τ_0} and v_{τ_1} are variable bases for τ_0 and τ_1, respectively, *having no variables in common* then (v_{τ_0}, v_{τ_1}) is a variable base for $\tau_0 \times \tau_1$.

A variable base is also said to have the type for which it is a base.

Terms:

The variables, function symbols, and all projections of all product types producible in the theory are available for building terms. The signature mapping is assumed to be extended inductively to encompass all term and type constructions. The inductive definition of terms and their associated types are given below:

(i) Singleton Term: () is a term of type 1.

(ii) Variable Term: A variable v_τ is a term of type τ.

(iii) Pair Term: If t_0 and t_1 are terms of type τ_0 and τ_1, respectively, then (t_0, t_1) is a term of type $\tau_0 \times \tau_1$.

(iv) Projection Terms: If t is a term of the product type $\tau_0 \times \tau_1$ then $p_0(t)$ is a *first projection* term of type τ_0 and $p_1(t)$ is a *second projection* term of type τ_1, where the symbols p_0 and p_1 *are added to the collection of function symbols* and are given the signatures of the projections associated with $\tau_0 \times \tau_1$.

(v) Function Terms: If f is any non-projection function symbol with domain of type τ and codomain of type τ', and t is a term of type τ, then $f(t)$ is a term of type τ'.

(vi) Application Terms: If t is a term of type τ, t' a term of type τ', and v a variable base of type τ' then the *abstract map t applied to t'*, i.e.

$$\{v \mapsto t\}(t')$$

is a term of type τ. The term t is referred to as an *abstracted term.*

Some cautionary words are appropriate here. An abstract map is *not* a term by itself because it cannot be evaluated. Its strong suggestion as a map or function should be tempered by the fact that maps and functions are not values in the first-order logic we will develop. Also, an abstract map need not necessarily be closed. A closed abstract map is called a *program.*

Secondly, variable bases are *not* terms as well. Nevertheless, we will employ identical notation for both a variable base within an abstracted term and the term formed from the same variables under the same sequence of pairings. For example, $((v_1, (v_2, (v_3, v_4))), v_5)$ is usable as either a (non-term) variable base or as a term built from variables. The usage will be clear from the expression context.

Also, the occurences of pairs in expressions will frequently be abbreviated by omitting the outermost parentheses: e.g $f((a, b))$ becomes $f(a, b)$.

Free Variables:

Since we have pronounced a binding operator, viz. the abstract map, free and bound variables are consequently definable. The *free* variables of terms are determined inductively by the rules below.

(i) $fvars(()) = \emptyset$.

(ii) If v is a variable, then $fvars(v) = \{v\}$.

(iii) $fvars(t_0, t_1) = fvars(t_0) \cup fvars(t_1)$.

(iv) If f is any function symbol (including any amended projections), then $fvars(f(t)) = fvars(t)$.

(v) $fvars(\{v \mapsto t\}(t')) = fvars(t') \cup (fvars(t) - fvars(v))$.

If an abstract map expression $\{v \mapsto t'\}$ occurs in a term t, the occurrence of a free variable in t' that occurs also in v is said to be *bound in t'* or *local in t'*. Clearly the determination of bound-ness can be carried on throughout all variable occurrences of t. A variable occurrence not judged bound is said to be *unbound* or *global.* It is easy to show that the collection of unbound variables of a term within the empty context forms its set of free variables.

The reader should note that our brevity causes v in rule (v) to be treated on the left side as a variable base and on the right as a term.

We shall conventionally refer to a variable as being *in* a term when that variable occurs freely in the term. Bound variables will be treated as invisible variables that can be renamed without changing the semantics of the term. We hereafter assume, again for brevity, that all variables in a concerned expression have been previously determined to be either bound or unbound.

Substitution:

With the definitions of terms and free variables, we can inductively define the meaning of applying a simultaneous substitution σ to a term t'. More precisely, writing

$$\sigma_{v:=t}(t')$$

where v is a variable base for the type of the term t, means that the component variables of v be pattern-matched with the appropriate component values of t and then substituted simultaneously into like-named free variables of t'. The result has its type equal to the type of t'. The pattern-matching is performed inductively on the structure of the substitution operand by essentially invoking projections precisely according to the substitution rules below. Since variables may substituted by terms containing variables, we hereafter implicitly assume for all substitution rules that *renaming is performed in parallel with the substitution.* The renaming prevents any variable clashes that would cause substituted free variables to become bound; doing it in parallel allows substitution proofs to be correctly built as structurally inductive ones (see [8]).

(i) $\sigma_{v:=t}(()) = ()$.

(ii) For each variable x occuring in t', then

  (1) If x does not occur in v, then $\sigma_{v:=t}(x) = x$,

  (2) else if x is bound and $x = v$ then $\sigma_{v:=t}(x) = x$,

  (3) else if x is unbound and $x = v$ then $\sigma_{v:=t}(x) = t$,

  (4) else if $v = (v_{\tau_0}, v_{\tau_1})$ and x occurs in v_{τ_i}, then $\sigma_{v:=t}(x) = \sigma_{v_{\tau_i}:=p_i(t)}(x)$.

(iii) $\sigma_{v:=t}(t_0, t_1) = (\sigma_{v:=t}(t_0), \sigma_{v:=t}(t_1))$

(iv) If f is any function symbol, then $\sigma_{v:=t}(f(t')) = f(\sigma_{v:=t}(t'))$

(v) $\sigma_{v:=t}(\{v' \mapsto t'\}(t'')) = \{v' \mapsto \sigma_{v:=t}(t')\}(\sigma_{v:=t}(t''))$

The last sub-rule of rule (ii) is used to complete the pattern matching of the variable base v with the term t by breaking down t, if necessary, with projections to fit, type-wise, into the variables of v. Thus

$$\sigma_{(x,y):=(t_0,t_1)}(x, y) = (p_0(t_0, t_1), p_1(t_0, t_1))$$

differs substitutively from

$$\sigma_{x:=t_0}(\sigma_{y:=t_1}(x, y)) = (t_0, t_1) \quad .$$

due to the latter's better-fitting pattern-match.

Axioms and Inference Rules:

A set of axioms and rules are used to construct a variable-base-indexed family of equivalence relations $(=_{v_\tau})$ among programs (closed abstract maps) of the general form $\{v_\tau \mapsto t\}$ where v_τ is a variable base of type τ such that $\mathit{fvars}(v_\tau) \supset \mathit{fvars}(t)$. We say that v_τ *is a variable base for* t. The equivalences will hold *up to renaming* via the substitution rules above. This assumption will be implemented by the following axiom:

- **Equivalence of Identities:**
  For a fixed type τ, all programs of the form $\{v_\tau \mapsto v_\tau\}$ are equal.

With respect to all the other rules already presented and to come, this axiom can be shown as equivalent to an explicit formalization of renaming the variables of a program, as well as to extensionality properties for abstract maps and programs. This axiom also yields the surjective pairing property for programs, making the theory really "cartesian". Nevertheless, we select this axiomization in lieu of all these alternatives to avoid distracting from the critical role of substitution in the cartesian theory and its augmentations.

By virtue of its equivalence to extensionality, this axiom justifies the use of the notation

$$t_0 \ =_{v_\tau} \ t_1$$

to signify equality between any program renaming-equivalent to $\{v_\tau \mapsto t_0\}$ and any program renaming-equivalent to $\{v_\tau \mapsto t_1\}$ where v_τ is an explicit variable base for both t_0 and t_1.

The remaining rules for $=_{v_\tau}$ are as follows:

- **Unit:**

$$\frac{t \text{ is of type } 1 \qquad v_\tau \text{ is a variable base for } t}{t \ =_{v_\tau} \ ()}$$

- **Projection:**

$$\frac{v_\tau \text{ is a variable base for } p_0(t_0, t_1)}{p_0((t_0, t_1)) \ =_{v_\tau} \ t_0}$$

$$\frac{v_\tau \text{ is a variable base for } p_0(t_0, t_1)}{p_1(t_0, t_1) \ =_{v_\tau} \ t_1}$$

- **Application:**

$$\frac{v_\tau \text{ is a variable base for } \{v' \mapsto t\}(t')}{\{v' \mapsto t\}(t') \ =_{v_\tau} \ \sigma_{v':=t'}(t)}$$

- **Congruence:**

$$\frac{t_1 \ =_{v_\tau} \ t_2 \qquad t_1' \ =_{v_\tau'} \ t_2' \qquad v_1' \text{ and } v_2' \text{ are } \tau'\text{-typed variable bases of } t_1' \text{ and } t_2' \qquad \mathrm{type}(t_1) = \tau'}{\sigma_{v_1':=t_1}(t_1') \ =_{v_\tau} \ \sigma_{v_2':=t_2}(t_2')}$$

We now form the equivalence relation $=_{v_\tau}$ for each variable base v_τ by letting it be the the symmetric transitive closure of this congruence relation generated by these rules.

With the eventual goal of showing an isomorphism of the cartesian theory with the standard combinator theory for a cartesian category in mind, the associative composition of categorical combinators must be reflected in the cartesian theory by the properties of substitution. The concept of *composing programs* can be expressed by the composition definition where the types of v_1 and t_0 match:

$$\{v_0 \mapsto t_0\} ; \{v_1 \mapsto t_1\} \quad \equiv \quad \{v_0 \mapsto \sigma_{v_1:=t_0}(t_1)\} \quad .$$

Its well-definedness up to $=_{v_0}$-equivalence is straightforwardly derivable from the equivalence-of-identities axiom.

The requirement of associativity up to equivalence then becomes

$$(\{v_0 \mapsto t_0\} ; \{v_1 \mapsto t_1\}) ; \{v_2 \mapsto t_2\} \quad = \quad \{v_0 \mapsto t_0\} ; (\{v_1 \mapsto t_1\} ; \{v_2 \mapsto t_2\})$$

where substitution is now forced by the composition rule to satisfy:

$$\sigma_{v_1:=t_0}(\sigma_{v_2:=t_1}(t_2)) \ =_{v_0} \ \sigma_{v_2:=\sigma_{v_1:=t_0}(t_1)}(t_2) \quad .$$

This requirement is fulfulled as a corollary of the following elementary substitution property of the cartesian theory:

Lemma 2.1 Associativity of Substitution:

Let t_2 be any term whose bound variables have been fully determined by its expression context. Then

$$\sigma_{v_1:=t_0}(\sigma_{v_2:=t_1}(t_2)) \quad =_{v_0} \quad \sigma_{v_2:=\sigma_{v_1:=t_0}(t_1)}(t_2)$$

whenever the variables of v_2 contain the unbound variables of t_2.

With associativity we can also quickly derive

Lemma 2.2 Simultaneity of Substitution Composition:

Let t_2 be any term whose bound variables have been fully determined by its expression context. Then

$$\sigma_{v_1:=t_0}(\sigma_{v_2:=t_1}(t_2)) \quad =_{v_0} \quad \sigma_{(v_2,v_1):=(\sigma_{v_1:=t_0}(t_1),t_0)}(t_2)$$

whenever the variables of v_2 contain the unbound variables of t_2.

In the following subsections, the cartesian theory will undergo augmentations that consist of adding new $=_{v_\tau}$-relation rules. The corresponding renaming-transparent congruent equivalences are then generated as above using the enlarged $=_{v_\tau}$-relations. The associativity of substitution is routinely extendable for each augmentation.

2.2 Augmenting a Theory with a Strong Initial Datatype

Expanding a theory to include a new strong initial datatype requires the addition of new term logic machinery to reason about terms of that type. This section explains the incremental aspects of adding a sequence of initial datatypes to a cartesian theory.

We will assume that the parametrized datatype $L(A)$ being added is built using earlier-specified datatypes E_i, $i = 1, \ldots, n$, with its components in the $E_i(A, L(A))$, and that the parametric arity of L equals m, that is, $A = (A_1, \ldots, A_m)$. The precise augmentation to the previously built theory is itemized below:

Types:

If $\tau_1, \ldots, \tau_m$ are types then $L(\tau_1, \ldots, \tau_m)$ is a type.

Variables and Variable Bases:

$L(\tau_1, \ldots, \tau_m)$ has a countable collection $\{v^i_{L(\tau_1,\ldots,\tau_m)}\}$ of variables available. Because there are no mechanisms via projections or destructors to ease the general binding access to initial datatypes in the manner described earlier for product types, the collection of variable bases for the datatype $L(\tau_1, \ldots, \tau_m)$ will be constituted only with isolated variables.

Terms:

(i) If $c_1, \ldots, c_n$ are the constructors associated with $L(\tau_1, \ldots, \tau_m)$ then if t is a term of type $E_i((\tau_1, \ldots, \tau_m), L(\tau_1, \ldots, \tau_m))$ then $c_i(t)$ is a *constructor term* of type $L(\tau_1, \ldots, \tau_m)$.

(ii) If t is a term of type $L(\tau_1, \ldots, \tau_m)$, v_i is a variable base of type $E_i((\tau_1, \ldots, \tau_m), \tau)$ for $i = 1, \ldots, n$ and all the t_i, $i = 1, \ldots, n$, are terms of a common type τ then

$$\left\{\!\!\left|\begin{array}{c} c_1 : v_1 \mapsto t_1 \\ \cdots \\ c_n : v_n \mapsto t_n \end{array}\right|\!\!\right\}(t)$$

is a *fold-from-L term* of type τ.

(iii) If t is a term of type $L(\tau_1, \ldots, \tau_m)$, v_i is a variable base of the component type $E_i((\tau_1, \ldots, \tau_m), L(\tau_1, \ldots, \tau_m))$ for $i = 1, \ldots, n$ and all the t_i, $i = 1, \ldots, n$, are terms of a common type τ then

$$\left\{\begin{array}{c} c_1(v_1) \mapsto t_1 \\ \cdots \\ c_n(v_n) \mapsto t_n \end{array}\right\}(t)$$

is a *case-from-L term* of type τ.

(iv) If t is a term of type $L(\tau_1, \ldots, \tau_m)$, w_i is a variable base of the parameter type τ_i for $i = 1, \ldots, m$, and t_i is a term of type $\tau_i{}'$ for $i = 1, \ldots, m$, then

$$L\begin{bmatrix} w_1 \mapsto t_1 \\ \ldots \\ w_m \mapsto t_m \end{bmatrix}(t)$$

is a *map-on-L term* of type $L(\tau_1{}', \ldots, \tau_m{}')$

In the case/fold/map term definitions the variables in each variable base bind with scope equal only to the term assigned to the base. Each line in these three kinds of terms is called a *phrase* because it behaves, intuitively, as an abstract map that is "run" sequentially with respect to the others. The phrase may possibly have variables not bound by its variable base, allowing the entry of outside context into the abstracted term. Therefore phrases act as program subroutines with parameters.

Free Variables:

The free variable rules are below. The definition of bound variable is extended in the expected way for each of the abstracted terms in the phrases.

(i) $\mathit{fvars}(c_i(t)) = \mathit{fvars}(t)$ for $i = 1, \ldots, n$

(ii)

$$\mathit{fvars}(\left\{\!\!\left|\begin{matrix} c_1 : v_1 \mapsto t_1 \\ \ldots \\ c_n : v_n \mapsto t_n \end{matrix}\right|\!\!\right\}(t)) = \left(\bigcup_{i=1}^{n}(\mathit{fvars}(t_i) - \mathit{fvars}(v_i))\right) \cup \mathit{fvars}(t)$$

(iii)

$$\mathit{fvars}(\left\{\begin{matrix} c_1(v_1) \mapsto t_1 \\ \ldots \\ c_n(v_n) \mapsto t_n \end{matrix}\right\}(t)) = \left(\bigcup_{i=1}^{n}(\mathit{fvars}(t_i) - \mathit{fvars}(v_i))\right) \cup \mathit{fvars}(t)$$

(iv)

$$\mathit{fvars}(L\begin{bmatrix} w_1 \mapsto t_1 \\ \ldots \\ w_m \mapsto t_m \end{bmatrix}(t)) = \left(\bigcup_{i=1}^{m}(\mathit{fvars}(t_i) - \mathit{fvars}(w_i))\right) \cup \mathit{fvars}(t)$$

Substitution:

(i) $\sigma_{v:=t}(c_i(t')) = c_i(\sigma_{v:=t}(t'))$ for $i = 1, \ldots, n$

(ii)

$$\sigma_{v:=t}(\left\{\!\!\left|\begin{matrix} c_1 : v_1 \mapsto t_1 \\ \ldots \\ c_n : v_n \mapsto t_n \end{matrix}\right|\!\!\right\}(t')) = \left\{\!\!\left|\begin{matrix} c_1 : v_1 \mapsto \sigma_{v:=t}(t_1) \\ \ldots \\ c_n : v_n \mapsto \sigma_{v:=t}(t_n) \end{matrix}\right|\!\!\right\}(\sigma_{v:=t}(t'))$$

(iii)

$$\sigma_{v:=t}(\left\{\begin{matrix} c_1(v_1) \mapsto t_1 \\ \ldots \\ c_n(v_n) \mapsto t_n \end{matrix}\right\}(t')) = \left\{\begin{matrix} c_1(v_1) \mapsto \sigma_{v:=t}(t_1) \\ \ldots \\ c_n(v_n) \mapsto \sigma_{v:=t}(t_n) \end{matrix}\right\}(\sigma_{v:=t}(t'))$$

(iv)

$$\sigma_{v:=t}(L\begin{bmatrix} w_1 \mapsto t_1 \\ \ldots \\ w_m \mapsto t_m \end{bmatrix}(t')) = L\begin{bmatrix} w_1 \mapsto \sigma_{v:=t}(t_1) \\ \ldots \\ w_m \mapsto \sigma_{v:=t}(t_m) \end{bmatrix}(\sigma_{v:=t}(t'))$$

Axioms and Inference Rules:

The $=_v$ relations are amended by the rules below. Each rule is accompanied by its counterpart examples for both finite lists $\mathrm{List}(A)$ and database trees $\mathrm{DBTree}(A, B)$ as declared in the introduction's table of datatype examples.

The examples have been abstracted *only* on context to show clearly how the basic data (shown as a and b) of lists and database trees are processed. All variables are shown in the examples as annotations of the notation v; all other symbols are to be regarded as general terms whose only possible free variables are context variables.

(i) **Fold-from-L:**

$$\left\{\!\left|\begin{array}{c} c_1 : v_1 \mapsto t_1 \\ \dots \\ c_n : v_n \mapsto t_n \end{array}\right|\!\right\}(c_i(t)) \quad =_{v_X} \quad \{v_i \mapsto t_i\}(E_i \left[\begin{array}{l} w_A \mapsto w_A \\ w_L \mapsto \left\{\!\left|\begin{array}{c} c_1 : v_1 \mapsto t_1 \\ \dots \\ c_n : v_n \mapsto t_n \end{array}\right|\!\right\}(w_L) \end{array}\right](t))$$

List(A):

$$\left\{\!\left|\begin{array}{l} \mathrm{nil} : () \mapsto t_1 \\ \mathrm{cons} : (v_A, v_C) \mapsto t_2 \end{array}\right|\!\right\}(\mathrm{nil}()) \;\; =_{v_X} \;\; t_1$$

$$\left\{\!\left|\begin{array}{l} \mathrm{nil} : () \mapsto t_1 \\ \mathrm{cons} : (v_A, v_C) \mapsto t_2 \end{array}\right|\!\right\}(\mathrm{cons}(a, l)) \;\; =_{v_X} \;\; \{(v_A, v_C) \mapsto t_2\}(a, \left\{\!\left|\begin{array}{l} \mathrm{nil} : () \mapsto t_1 \\ \mathrm{cons} : (v_A, v_C) \mapsto t_2 \end{array}\right|\!\right\}(l))$$

DBTree(A,B):

$$\left\{\!\left|\begin{array}{l} \mathrm{dbleaf} : v_A \mapsto t_1 \\ \mathrm{dbnode} : (v_B, (v_C, v'_C)) \mapsto t_2 \end{array}\right|\!\right\}(\mathrm{dbleaf}(a)) \;\; =_{v_X} \;\; \{v_A \mapsto t_1\}(a)$$

$$\left\{\!\left|\begin{array}{l} \mathrm{dbleaf} : v_A \mapsto t_1 \\ \mathrm{dbnode} : (v_B, (v_C, v'_C)) \mapsto t_2 \end{array}\right|\!\right\}(\mathrm{dbnode}(b, (t, t'))) \;\; =_{v_X} \;\; \{(v_B, (v_C, v'_C)) \mapsto t_2\}(b, (\{\!|\dots|\!\}(t), \{\!|\dots|\!\}(t')))$$

(ii) **Case-from-L:**

$$\left\{\begin{array}{c} c_1(v_1) \mapsto t_1 \\ \dots \\ c_n(v_n) \mapsto t_n \end{array}\right\}(c_i(t)) \quad =_{v_X} \quad \{v_i \mapsto t_i\}(t)$$

List(A):

$$\left\{\begin{array}{l} \mathrm{nil}() \mapsto t_1 \\ \mathrm{cons}(v_A, v_{\mathrm{List(A)}}) \mapsto t_2 \end{array}\right\}(\mathrm{nil}()) \;\; =_{v_X} \;\; t_1$$

$$\left\{\begin{array}{l} \mathrm{nil}() \mapsto t_1 \\ \mathrm{cons}(v_A, v_{\mathrm{List(A)}}) \mapsto t_2 \end{array}\right\}(\mathrm{cons}(a, l)) \;\; =_{v_X} \;\; \{(v_A, v_{\mathrm{List(A)}}) \mapsto t_2\}(a, l)$$

DBTree(A,B):

$$\left\{\begin{array}{l} \mathrm{dbleaf}(v_A) \mapsto t_1 \\ \mathrm{dbnode}(v_B, (v_{\mathrm{DBTree}}, v'_{\mathrm{DBTree}})) \mapsto t_2 \end{array}\right\}(\mathrm{dbleaf}(a)) \;\; =_{v_X} \;\; \{v_A \mapsto t_1\}(a)$$

$$\left\{\begin{array}{l} \mathrm{dbleaf}(v_A) \mapsto t_1 \\ \mathrm{dbnode}(v_B, (v_{\mathrm{DBTree}}, v'_{\mathrm{DBTree}})) \mapsto t_2 \end{array}\right\}(\mathrm{dbnode}(b, (t, t'))) \;\; =_{v_X} \;\; \{(v_B, (v_{\mathrm{DBTree}}, v'_{\mathrm{DBTree}})) \mapsto t_2\}(b, (t, t'))$$

(iii) **Map-on-L:**

$$L\left[\begin{array}{c} w_1 \mapsto t_1 \\ \dots \\ w_m \mapsto t_m \end{array}\right](c_i(t)) \quad =_{v_X} \quad c_i(E_i\left[\begin{array}{l} \dots \\ w_i \mapsto t_i \\ \dots \\ w_L \mapsto L\left[\begin{array}{c} \dots \\ w_i \mapsto t_i \\ \dots \end{array}\right](w_L) \end{array}\right](t))$$

List(A):

$$\mathrm{List}\,[v_A \mapsto t](\mathrm{nil}()) \;=_{v_X}\; \mathrm{nil}()$$

$$\mathrm{List}\,[v_A \mapsto t](\mathrm{cons}(a,l)) \;=_{v_X}\; \mathrm{cons}(t(a), \mathrm{List}\,[v_A \mapsto t](l))$$

DBTree(A,B):

$$\mathrm{DBTree}\left[\begin{array}{c} v_A \mapsto t_1 \\ v_B \mapsto t_2 \end{array}\right](\mathrm{dbleaf}(a)) \;=_{v_X}\; \mathrm{dbleaf}(t_1(a))$$

$$\mathrm{DBTree}\left[\begin{array}{c} v_A \mapsto t_1 \\ v_B \mapsto t_2 \end{array}\right](\mathrm{dbnode}(b,(t,t'))) \;=_{v_X}\; \mathrm{dbnode}(t_2(b), (\mathrm{DBTree}[\dots](t), \mathrm{DBTree}[\dots](t')))$$

(iv) L **Fold Uniqueness:**

$$\frac{\forall_{i=1}^{n} \quad \{v_L \mapsto t\}(c_i(t')) \quad =_{v_X} \quad \{v_i \mapsto t_i\}(E_i\left[\begin{array}{l} w_A \mapsto w_A \\ w_L \mapsto \{v_L \mapsto t\}(w_L) \end{array}\right](t'))}{\{v_L \mapsto t\}(t'') \quad =_{v_X} \quad \left\{\begin{array}{c} c_1 : v_1 \mapsto t_1 \\ \dots \\ c_n : v_n \mapsto t_n \end{array}\right\}(t'')}$$

List(A):

$$\frac{t(\mathrm{nil}()) \;=_{v_X}\; t_1 \quad \text{and} \quad t(\mathrm{cons}(a,l)) \;=_{v_X}\; t_2(a, t(l))}{t(l) \;=_{v_X}\; \left\{\begin{array}{l} \mathrm{nil} : () \mapsto t_1 \\ \mathrm{cons} : (v_A, v_C) \mapsto t_2 \end{array}\right\}(l)}$$

DBTree(A,B):

$$\frac{t(\mathrm{dbleaf}(a)) \;=_{v_X}\; t_1(a) \quad \text{and} \quad t(\mathrm{dbnode}(b,(t',t''))) \;=_{v_X}\; t_2(b, t(t'), t(t''))}{t(t') \;=_{v_X}\; \left\{\begin{array}{l} \mathrm{dbleaf} : v_A \mapsto t_1 \\ \mathrm{dbnode} : (v_A, (v_C, v'_C)) \mapsto t_2 \end{array}\right\}(t')}$$

The axioms imitate exactly the corresponding rewrite rules listed in the introduction. The distribution, i.e. global scope, of context is reflected in the permitted occurrences of context variables in all phrases of the fold terms. This distribution of context will be exhibited by the strong final datatypes as well.

The reader may also compare the substitution rules and the non-recursive cases of the case and fold axioms with those of the *Sum* datatype shown in Cockett [1] to see more clearly the generalizations made here.

The reader may also easily verify that associativity of substitution easily extends to the augmentation.

The term logic analogy of the **X**-sum lemma [4] arises directly from the fold uniqueness rule:

Theorem 2.3 *The terms $(c_i(v_i), v_X)$ are "parametrized embedding terms" in the term logic, i.e. if there exists for each i a term t_i of type C such that the disjoint bases v_i and v_X together contain the free variables of t_i then there also exists uniquely (up to provable equality) a term t of type C such that*

$$\{v_L \mapsto t\}(c_i(v_i)) \;=_{(v_i,v_X)}\; t_i$$

and in fact

$$t \;=_{(v_L,v_X)}\; \left\{\begin{array}{c} c_1(v_1) \mapsto t_1 \\ \dots \\ c_n(v_n) \mapsto t_n \end{array}\right\}(v_L)$$

where v_L does not occur in v and any t_i.

Proof. It is routine to verify that the expression for t in the theorem satisfies the embedding property.

For uniqueness, assume t is any term that satisfies the embedding property. The L fold uniqueness rule can be shown applicable for "t" set to (v_L, t) and "t_i" set to

$$\{v_i \mapsto (c_i(v_i), t_i)\}(E_i\begin{bmatrix} w_A \mapsto w_A \\ (w_L, w_C) \mapsto w_L \end{bmatrix}(v_i))$$

Thus (v_L, t) is uniquely determined. So by second projection and congruence, t is unique. $\Box$

The results below show that the augmentation process could have been alternately expressed entirely in terms of the fold-from-L term. However, the separate presentation of case/fold/map terms is desirable both for clarity and for the reason that all are usefully expressive in programming.

Theorem 2.4 *The case-from-L is determined by the fold-from-L:*

$$\left\{\begin{array}{c} \dots \\ c_i(v_i) \mapsto t_i \\ \dots \end{array}\right\}(v_L) \;=_{(v_L,v_X)}\; p_1(\left\{\!\!\left|\begin{array}{c} \dots \\ c_i : v_i \mapsto \{v_i \mapsto (c_i(v_i), t_i)\}(E_i\begin{bmatrix} w_A \mapsto w_A \\ (w_L, w_C) \mapsto w_L \end{bmatrix}(v_i)) \\ \dots \end{array}\right|\!\!\right\}(v_L))$$

Proof. The L fold uniqueness rule is applicable where "t" is set to

$$(v_L \;,\; \left\{\begin{array}{c} \dots \\ c_i(v_i) \mapsto t_i \\ \dots \end{array}\right\}(v_L))$$

and "t_i" to

$$\{v_i \mapsto (c_i(v_i), t_i)\}(E_i\begin{bmatrix} w_A \mapsto w_A \\ (w_L, w_C) \mapsto w_L \end{bmatrix}(v_i))$$

$\Box$

Theorem 2.5 *The map-on-L is determined by the fold-from-L:*

$$L\begin{bmatrix} \dots \\ w_j \mapsto t_j \\ \dots \end{bmatrix}(v_L) \;=_{(v_L,v_X)}\; \left\{\!\!\left| \begin{array}{c} \dots \\ c_i : v_i \mapsto \{v_i \mapsto (c_i(v_i))\}(E_i\begin{bmatrix} \dots \\ w_j \mapsto t_j \\ \dots \\ w_L \mapsto w_L \end{bmatrix}(v_i)) \\ \dots \end{array} \right|\!\!\right\}(v_L)$$

Proof. The L fold uniqueness rule is applicable where "t" is set to

$$L\begin{bmatrix} \dots \\ w_j \mapsto t_j \\ \dots \end{bmatrix}(v_L)$$

and "t_i" to

$$\{v_i \mapsto (c_i(v_i))\}(E_i\begin{bmatrix} \dots \\ w_j \mapsto t_j \\ \dots \\ w_L \mapsto w_L \end{bmatrix}(v_i))$$

□

2.3 Augmenting a Theory with a Strong Final Datatype

Often computational complexity is drastically reduced by incorporating opportunities for lazy evaluation into a reduction system. This section quickly tours the process of adding strong final datatypes that parallels the development of the preceding section and discloses how the lazy evaluation of terms acting over such datatypes, e.g. infinite lists, is accomplished.

Types:

If $\tau_1, \dots, \tau_m$ are types then $R(\tau_1, \dots, \tau_m)$ is a type.

Variables and Variable Bases:

$R(\tau_1, \dots, \tau_m)$ has a countable collection $\{v^i_{R(\tau_1,\dots,\tau_m)}\}$ of variables available.

Just as for the initial datatypes. the variable bases for the datatype $R(\tau_1, \dots, \tau_m)$ are only the isolated variables. This is due to the subtlety in that (as will be explained) unlike ordinary product datatypes, general final data using a common set of destructors can be built either in one step or recursively. This means that the application of a destructor function to access a component of such a datum will yield different results according to how the datum was built. Consequently, easy access of "components" of final data values for binding purposes becomes problematic.

Terms:

(i) If $d_1, \dots, d_n$ are the destructors associated with $R(\tau_1, \dots, \tau_m)$ and t is a term of type $R(\tau_1, \dots, \tau_m)$ then $d_i(t)$ is a *destructor term* of type $E_i((\tau_1, \dots, \tau_m), R(\tau_1, \dots, \tau_m))$.

(ii) If t is a term of type C, v_C is a variable base of type C scoped simultaneously to all the t_i, and each t_i is a term of type $E_i((\tau_1, \dots, \tau_m), C)$ then

$$\left(\!\left| v_C \mapsto \begin{array}{c} d_1 : t_1 \\ \dots \\ d_n : t_n \end{array} \right|\!\right)(t)$$

is an *unfold-to-R term* of type $R(\tau_1, \ldots, \tau_m)$.

(iii) If t is a term of type C and each t_i is a term of type $E_i((\tau_1, \ldots, \tau_m), R(\tau_1, \ldots, \tau_m))$ then

$$\begin{pmatrix} d_1 : t_1(t) \\ \ldots \\ d_n : t_n(t) \end{pmatrix}$$

is a *record-to-R term* of type $R(\tau_1, \ldots, \tau_m)$.

(iv) If t is a term of type $R(\tau_1, \ldots, \tau_m)$, w_i is a variable base of the parameter type τ_i that is scoped only to the corresponding t_i for $i = 1, \ldots, m$, and t_i is a term of type $\tau_i{}'$ for $i = 1, \ldots, m$, then

$$R\begin{bmatrix} w_1 \mapsto t_1 \\ \ldots \\ w_m \mapsto t_m \end{bmatrix}(t)$$

is a *map-on-R term* of type $R(\tau_1{}', \ldots, \tau_m{}')$

The lines in the unfold and record terms can be considered as abstract maps, possibly not closed, that act simultaneously, or concurrently, on its copy of the data entering via the common variable base v_C. We call each line a *thread* since its processing is independent of the others. This independence is evident from the reduction rules to be presented later in this section.

Free Variables:

We abbreviate the rules by showing only those for the destructors and the unfold term. The rules for the remaining record and map terms are completely analogous. The definition of bound variable is also extended over the abstracted terms of the threads.

(i) $$\mathit{fvars}(\left(\left| v_C \mapsto \begin{matrix} d_1 : t_1 \\ \ldots \\ d_n : t_n \end{matrix} \right|\right)(t)) \;=\; (\textstyle\bigcup_{i=1}^{n} \mathit{fvars}(t_i) - \mathit{fvars}(v_C)) \cup \mathit{fvars}(t)$$

(ii) $$\mathit{fvars}(d_i(\left(\left| v_C \mapsto \begin{matrix} d_1 : t_1 \\ \ldots \\ d_n : t_n \end{matrix} \right|\right)(t))) \;=\; (\mathit{fvars}(t_i) - \mathit{fvars}(v_C)) \cup \mathit{fvars}(t)$$
for $i = 1, \ldots, n$

Substitution:

The substitution rules involving tuples imitate closely the rules for pairs in the cartesian theory and are consequently not listed here. Thus the rules are similarly abbreviated to only the destructor and unfold term cases.

(i) $\sigma_{v:=t}(d_i(t')) = d_i(\sigma_{v:=t}(t'))$ for $i = 1, \ldots, n$

(ii) $$\sigma_{v:=t}(\left(\left| v_C \mapsto \begin{matrix} d_1 : t_1 \\ \ldots \\ d_n : t_n \end{matrix} \right|\right)(t')) \;=\; \left(\left| v_C \mapsto \begin{matrix} d_1 : \sigma_{v:=t}(t_1) \\ \ldots \\ d_n : \sigma_{v:=t}(t_n) \end{matrix} \right|\right)(\sigma_{v:=t}(t'))$$

Axioms and Inference Rules:

The rules below are added to the $=_v$ relations. Each rule is accompanied by its corresponding examples for the infinite lists *Inflist*(A) and the co-lists *Colist*(A) as declared in the introduction.

As before for the initial datatype augmentation, the examples have been abstracted only on context to show clearly how the "seed" term, represented as c, are processed to produce infinite lists and colists.

(i) **Unfold-to-R:**

$$d_i(\left(\!\left|\, v_C \mapsto \begin{array}{c} d_1 : t_1 \\ \dots \\ d_n : t_n \end{array} \right|\!\right)(t)) \;=_{v_X}\; E_i\left[\begin{array}{l} w_A \mapsto w_A \\ w_C \mapsto \left(\!\left|\, v_C \mapsto \begin{array}{c} d_1 : t_1 \\ \dots \\ d_n : t_n \end{array} \right|\!\right)(w_C) \end{array}\right](\{v_C \mapsto t_i\}(t))$$

Inflist(A):

$$\mathrm{head}\,(\left(\!\left|\, v_C \mapsto \begin{array}{c} \mathrm{head} : t_1 \\ \mathrm{tail} : t_2 \end{array} \right|\!\right)(c)) \;=_{v_X}\; \{v_C \mapsto t_1\}(c)$$

$$\mathrm{tail}\,(\left(\!\left|\, v_C \mapsto \begin{array}{c} \mathrm{head} : t_1 \\ \mathrm{tail} : t_2 \end{array} \right|\!\right)(c)) \;=_{v_X}\; \left(\!\left|\, v_C \mapsto \begin{array}{c} \mathrm{head} : t_1 \\ \mathrm{tail} : t_2 \end{array} \right|\!\right)(\{v_C \mapsto t_2\}(c))$$

Colist(A):

$$\mathrm{cohead}\,(\left(\!\left|\, v_C \mapsto \begin{array}{c} \mathrm{cohead} : t_1 \\ \mathrm{cotail} : t_2 \end{array} \right|\!\right)(c)) \;=_{v_X}\; \{v_C \mapsto t_1\}(c)$$

$$\mathrm{cotail}\,(\left(\!\left|\, v_C \mapsto \begin{array}{c} \mathrm{cohead} : t_1 \\ \mathrm{cotail} : b_0(()) \end{array} \right|\!\right)(c)) \;=_{v_X}\; b_0(())$$

$$\mathrm{cotail}\,(\left(\!\left|\, v_C \mapsto \begin{array}{c} \mathrm{cohead} : t_1 \\ \mathrm{cotail} : b_1(t_2') \end{array} \right|\!\right)(c)) \;=_{v_X}\; b_1(\left(\!\left|\, v_C \mapsto \begin{array}{c} \mathrm{cohead} : t_1 \\ \mathrm{cotail} : b_1(t_2') \end{array} \right|\!\right)(\{v_C \mapsto t_2'\}(c)))$$

(ii) **Record-to-R:**

$$d_i(\left(\begin{array}{c} d_1 : t_1(t) \\ \dots \\ d_n : t_n(t) \end{array}\right)) \;=_{v_X}\; \{v_C \mapsto t_i\}(t)$$

Inflist(A):

$$\mathrm{head}\,(\left(\begin{array}{l} \mathrm{head} : t_1(c) \\ \mathrm{tail} : t_2(c) \end{array}\right)) \;=_{v_X}\; \{v_C \mapsto t_1\}(c)$$

$$\mathrm{tail}\,(\left(\begin{array}{l} \mathrm{head} : t_1(c) \\ \mathrm{tail} : t_2(c) \end{array}\right)) \;=_{v_X}\; \{v_C \mapsto t_2\}(c)$$

Colist(A):

$$\mathrm{cohead}\,(\left(\begin{array}{l} \mathrm{cohead} : t_1(c) \\ \mathrm{cotail} : t_2(c) \end{array}\right)) \;=_{v_X}\; \{v_C \mapsto t_1\}(c)$$

$$\mathrm{cotail}\,(\left(\begin{array}{l} \mathrm{cohead} : t_1(c) \\ \mathrm{cotail} : b_0(()) \end{array}\right)) \;=_{v_X}\; b_0(())$$

$$\mathrm{cotail}\,(\left(\begin{array}{l} \mathrm{cohead} : t_1(c) \\ \mathrm{cotail} : b_1(t_2'(c)) \end{array}\right)) \;=_{v_X}\; b_1(\{v_C \mapsto t_2'\}(c))$$

(iii) **Map-on-R:**

$$d_i(R\begin{bmatrix} w_1 \mapsto t_1 \\ \ldots \\ w_m \mapsto t_m \end{bmatrix}(t)) \;=_{v_X}\; E_i\begin{bmatrix} \ldots \\ w_i \mapsto t_i \\ \ldots \\ w_R \mapsto R\begin{bmatrix} w_1 \mapsto t_1 \\ \ldots \\ w_m \mapsto t_m \end{bmatrix}(w_R) \end{bmatrix}(d_i(t))$$

Inflist(A):

$$\text{head}\,(\text{Inflist}[v_A \mapsto t](\left(\!\left|\, v_C \mapsto \begin{array}{c} \text{head} : t_1 \\ \text{tail} : t_2 \end{array} \right|\!\right)(c))) \;=_{v_X}\; t(t_1(c))$$

$$\text{head}\,(\text{Inflist}[v_A \mapsto t](\begin{pmatrix} \text{head} : t_1(c) \\ \text{tail} : t_2(c) \end{pmatrix})) \;=_{v_X}\; t(t_1(c))$$

$$\text{tail}\,(\text{Inflist}[v_A \mapsto t](\left(\!\left|\, v_C \mapsto \begin{array}{c} \text{head} : t_1 \\ \text{tail} : t_2 \end{array} \right|\!\right)(c))) \;=_{v_X}$$
$$\text{Inflist}[\ldots](\left(\!\left|\, v_C \mapsto \begin{array}{c} \text{head} : t_1 \\ \text{tail} : t_2 \end{array} \right|\!\right)(t_2(c))$$

$$\text{tail}\,(\text{Inflist}[v_A \mapsto t](\begin{pmatrix} \text{head} : t_1(c) \\ \text{tail} : t_2(c) \end{pmatrix})) \;=_{v_X}\; \text{Inflist}[\ldots](t_2(c)))$$

Colist(A):

$$\text{cohead}\,(\text{Colist}\,[\,v_A \mapsto t\,]\,(\left(\!\left|\, v_C \mapsto \begin{array}{c} \text{cohead} : t_1 \\ \text{cotail} : t_2 \end{array} \right|\!\right)(c))) \;=_{v_X}\; t(t_1(c))$$

$$\text{cohead}\,(\text{Colist}\,[\,v_A \mapsto t\,]\,(\begin{pmatrix} \text{cohead} : t_1(c) \\ \text{cotail} : t_2(c) \end{pmatrix})) \;=_{v_X}\; t(t_1(c))$$

$$\text{cotail}\,(\text{Colist}\,[\,v_A \mapsto t\,]\,(\left(\!\left|\, v_C \mapsto \begin{array}{c} \text{cohead} : t_1 \\ \text{cotail} : b_0(()) \end{array} \right|\!\right)(c))) \;=_{v_X}\; b_0(())$$

$$\text{cotail}\,(\text{Colist}\,[\,v_A \mapsto t\,]\,(\begin{pmatrix} \text{cohead} : t_1(c) \\ \text{cotail} : b_0(()) \end{pmatrix})) \;=_{v_X}\; b_0(())$$

$$\text{cotail}\,(\text{Colist}\,[\,v_A \mapsto t\,]\,(\left(\!\left|\, v_C \mapsto \begin{array}{c} \text{cohead} : t_1 \\ \text{cotail} : b_1(t_2') \end{array} \right|\!\right)(c))) \;=_{v_X}$$
$$b_1(\text{Colist}\,[\ldots](\left(\!\left|\, v_C \mapsto \begin{array}{c} \text{cohead} : t_1 \\ \text{cotail} : b_1(t_2') \end{array} \right|\!\right)(t_2'(c))))$$

$$\text{cotail}\,(\text{Colist}\,[\,v_A \mapsto t\,]\,(\begin{pmatrix} \text{cohead} : t_1(c) \\ \text{cotail} : b_1(t_2'(c)) \end{pmatrix})) \;=_{v_X}\; b_1(\text{Colist}\,[\ldots](t_2'(c))$$

(iv) **R Unfold Uniqueness:**

$$\frac{\forall_{i=1}^{n}\;\; d_i(\{v_C \mapsto t\}(t')) \;=_{v_X}\; E_i\begin{bmatrix} w_A \mapsto w_A \\ w_C \mapsto \{v_C \mapsto t\}(w_C) \end{bmatrix}(\{v_C \mapsto t_i\}(t'))}{\{v_C \mapsto t\}(t'') \;=_{v_X}\; \left(\!\left|\, v_C \mapsto \begin{array}{c} d_1 : t_1 \\ \ldots \\ d_n : t_n \end{array} \right|\!\right)(t'')}$$

Inflist(A):

$$\frac{\text{head}\,(t(c)) \;=_{v_X}\; t_1(c) \quad \text{and} \quad \text{tail}\,(t(c)) \;=_{v_X}\; t(t_2(c))}{t(c) \;=_{v_X}\; \left(\!\left| v_C \mapsto \begin{array}{c} \text{head} : t_1 \\ \text{tail} : t_2 \end{array} \right|\!\right)(c)}$$

Colist(A):

$$\frac{\text{cohead}\,(t(c)) \;=_{v_X}\; t_1(c) \quad \text{and either} \quad \text{cotail}\,(t(c)) \;=_{v_X}\; b_0(()) \quad \text{or} \quad \text{cotail}\,(t(c)) \;=_{v_X}\; b_1(t(t_2'(c)))}{t(c) \;=_{v_X}\; \left(\!\left| v_C \mapsto \begin{array}{c} \text{cohead} : t_1 \\ \text{cotail} : t_2 \end{array} \right|\!\right)(c)}$$

By comparison with the axioms for initial datatypes, the opportunity for "lazy" usage of final datatypes can be intuitively inferred. In the substitution rule for the unfold-to-R, processing is essentially using the recursive mapping operation "on the outside" once to build the outermost base data of the R-structure while deferring the building of the remaining pieces of the R-structure. This allows the possible use of a destructor to extract this outermost data. On the other hand, the fold-from-L rule forces all recursion to complete before the data structure can be examined since the mapping occurs "inside" the data structure at the base data level.

As expected, most of the counterparts of the strong initial datatype properties are operative for the final strong datatypes: associativity of substitution and definability of the strong final datatype record and map terms by means of the unfold term. These results are left to the reader to be found by closely imitating the techniques used to establish the corresponding initial datatype results.

The notion analogous to **X**-sum (i.e. "parametrized projection terms" in the term logic) does not carry over to the strong final datatypes.

2.4 Example: parametricity

Reynolds' abstraction theorem [7] has been specialized by Wadler [9] as a "free parametricity result" to give a more direct tool for proving program equivalences. Due to the totally parametrized development of the term logic, it is expected that the abstraction theorem holds in our setting as well. The theorem is particularized below in the style of Wadler for the case of fold-from-*List* terms. What is instructive is the apparent need for structural induction in its proof, an indication that such instantiations of the abstraction theorem are not really "free". The general necessity for structural induction in proving parametric theorems was first pointed out by Mairson [6].

Theorem 2.6 Fold-from-List Parametricity

Suppose for datatypes A, A', B, and B' that $t_{A'}$, $t_{B'}$, u_B, $u_{B'}$, s_B, and $s_{B'}$ are terms of their respective annotated types. Assume also the coherence conditions

(i) $\{v_B \mapsto u_B\}\,;\{v_B \mapsto t_{B'}\} \;=\; \{v_B \mapsto u_{B'}\}$

(ii) $\{(v_A, v_B) \mapsto s_B\}\,;\{v_B \mapsto t_{B'}\} \;=\; \{(v_A, v_B) \mapsto (t_{A'}, t_{B'})\}\,;\{(v_{A'}, v_{B'}) \mapsto s_{B'}\}$

Then for any term t_L of List(A) we have

$$\begin{aligned}&\left\{\begin{array}{l} nil : () \mapsto u_{B'} \\ cons : (v_{A'}, v_{B'}) \mapsto s_{B'} \end{array}\right\}(List[v_A \mapsto t_{A'}](t_L)) \\ = \quad & \{v_B \mapsto t_{B'}\}(\left\{\begin{array}{l} nil : () \mapsto u_B \\ cons : (v_A, v_B) \mapsto s_B \end{array}\right\}(t_L))\end{aligned}$$

Proof. We proceed by structural induction on t_L and assume without loss that t_L is of the form $c_i(t_*)$ where $c_1 = nil$ and $c_2 = cons$.

(i) Let $t_L = nil()$. Then both sides quickly reduce to showing

$$u_{B'} \quad =_{v_B} \quad \sigma_{v_B := u_B}(t_{B'})$$

which merely rephrases assumption (i).

(ii) Let $t_L = cons(a, l)$. Thus

$$\begin{aligned}
&\left\{\begin{array}{l} nil : () \mapsto u_{B'} \\ cons : (v_{A'}, v_{B'}) \mapsto s_{B'} \end{array}\right\}(List[v_A \mapsto t_{A'}](cons(a, l))) \\
= \quad & \left\{\begin{array}{l} nil : () \mapsto u_{B'} \\ cons : (v_{A'}, v_{B'}) \mapsto s_{B'} \end{array}\right\}(cons(\sigma_{v_A := a}(t_{A'}), List[v_A \mapsto t_{A'}](l))) \\
= \quad & \{(v_{A'}, v_{B'}) \mapsto s_{B'}\}(Prod\left[\begin{array}{l} w_A \mapsto w_A \\ w_L \mapsto \left\{\begin{array}{l} nil : () \mapsto u_{B'} \\ cons : (v_{A'}, v_{B'}) \mapsto s_{B'} \end{array}\right\}(w_L) \end{array}\right](\sigma_{v_A := a}(t_{A'}), List[v_A \mapsto t_{A'}](l))) \\
= \quad & \{(v_{A'}, v_{B'}) \mapsto s_{B'}\}(\sigma_{v_A := a}(t_{A'}), \left\{\begin{array}{l} nil : () \mapsto u_{B'} \\ cons : (v_{A'}, v_{B'}) \mapsto s_{B'} \end{array}\right\}(List[v_A \mapsto t_{A'}](l))) \\
= \quad & \{(v_{A'}, v_{B'}) \mapsto s_{B'}\}(\sigma_{v_A := a}(t_{A'}), \{v_B \mapsto t_{B'}\}(\left\{\begin{array}{l} nil : () \mapsto u_B \\ cons : (v_A, v_B) \mapsto s_B \end{array}\right\}(l))) \\
= \quad & \{(v_{A'}, v_{B'}) \mapsto s_{B'}\}(\{(v_A, v_B) \mapsto (t_{A'}, t_{B'})\}(a, \left\{\begin{array}{l} nil : () \mapsto u_B \\ cons : (v_A, v_B) \mapsto s_B \end{array}\right\}(l))) \\
= \quad & \{v_B \mapsto t_{B'}\}(\{(v_A, v_B) \mapsto s_B\}(a, \left\{\begin{array}{l} nil : () \mapsto u_B \\ cons : (v_A, v_B) \mapsto s_B \end{array}\right\}(l))) \\
= \quad & \{v_B \mapsto t_{B'}\}(\{(v_A, v_B) \mapsto s_B\}(Prod\left[\begin{array}{l} w_A \mapsto w_A \\ w_L \mapsto \left\{\begin{array}{l} nil : () \mapsto u_B \\ cons : (v_A, v_B) \mapsto s_B \end{array}\right\}(w_L) \end{array}\right](a, l))) \\
= \quad & \{v_B \mapsto t_{B'}\}(\left\{\begin{array}{l} nil : () \mapsto u_B \\ cons : (v_A, v_B) \mapsto s_B \end{array}\right\}(cons(a, l)))
\end{aligned}$$

The required induction occurs in the fourth step and the application of assumption (ii) in the sixth.

□

3 The Equivalence of Combinators and Term Logic

An equivalence between a categorical combinator theory and an equational programming logic is expressed by a pair of mutually inverse consistent, or well-defined, translations: combinators-to-programs and programs-to-combinators. Consistency simply means that a translation preserves equality.

A cartesian combinator theory $\mathcal{C}$ has the specification $(\mathcal{T}, \mathcal{F}, \mathcal{S}, \mathcal{E})$ where $\mathcal{T}$ is a collection of primitive types, $\mathcal{F}$ is a collection of pre-determined maps or combinators, $\mathcal{S}$ is the set of type signatures of the combinators, and $\mathcal{E}$ is a set of equations between combinators.

The types are given by the same rules used earlier for generating the cartesian theory types.

The combinators are generated inductively by

- For every type τ there is an identity combinator $id_\tau : \tau \longrightarrow \tau$.
- For every type τ there is a final combinator $!_\tau : \tau \longrightarrow 1$.
- If $f \in \mathcal{F}$ has signature (τ_1, τ_2), then $f : \tau_1 \longrightarrow \tau_2$ is a combinator.
- For every pair of types τ_0 and τ_1, there are projection combinators $p_0^{\tau_0,\tau_1} : \tau_0 \times \tau_1 \longrightarrow \tau_0$ and $p_1^{\tau_0,\tau_1} : \tau_0 \times \tau_1 \longrightarrow \tau_1$.
- If $c_0 : \tau \longrightarrow \tau_0$ and $c_1 : \tau \longrightarrow \tau_1$ are combinators, then their pairing $\langle c_0, c_1 \rangle : \tau \longrightarrow \tau_0 \times \tau_1$ is a combinator.
- If $c_0 : \tau_0 \longrightarrow \tau_1$ and $c_1 : \tau_1 \longrightarrow \tau_2$ are combinators, then their composition $c_0 ; c_1 : \tau_0 \longrightarrow \tau_2$ is a combinator.

The symmetric transitive closure of the relation defined by the fundamental axioms and inference rules below give the congruence relation among combinators possessing the same domain and codomain types:

- $(c_0 ; c_1) ; c_2 \ \equiv \ c_0 ; (c_1 ; c_2)$.
- $id ; c \ \equiv \ c \ \equiv \ c ; id$.
- $\langle c_0, c_1 \rangle ; p_0 \ \equiv \ c_0$.
- $\langle c_0, c_1 \rangle ; p_1 \ \equiv \ c_1$.
- $\langle c ; p_0 \ , \ c ; p_1 \rangle \ \equiv \ c$.
- $c ; ! \ \equiv !$.
- If $c_0 \ \equiv \ c_0'$ and $c_1 \ \equiv \ c_1'$ then $c_0 ; c_1 \ \equiv \ c_0' ; c_1'$.
- If $c_0 = c_1$ in $\mathcal{E}$ then $c_0 \ \equiv \ c_1$.

This relation constitutes the minimal set of equations to be contained in $\mathcal{E}$ for a cartesian theory.

3.1 Translating Programs to Combinators

A translation, herein denoted $\mathcal{C}$, of closed abstracted terms, or programs, to combinators and a proof of its consistency is presented in this section. We first define below the translation of programs in the cartesian theory. The notation

$$\mathcal{C}[\![v \mapsto t]\!]$$

represents the application of the translation $\mathcal{C}$ to any member of the $=_{v_\tau}$-equivalence class of the program $\{v \mapsto t\}$. The definition proceeds inductively on the construction of the abstracted term representing the program:

(i) $\mathcal{C}[\![v_\tau \mapsto ()]\!] \ = !_\tau$

(ii) If v_τ is a variable then $\mathcal{C}[\![v_\tau \mapsto v_\tau]\!] \ = \ id_\tau$

(iii) If v_0 and v_1 are variable bases then

$$\mathcal{C}[\![(v_0, v_1) \mapsto v]\!] \;=\; p_0\,;\mathcal{C}[\![v_0 \mapsto v]\!] \text{ if } v \in \mathit{fvars}(v_0))$$

and

$$\mathcal{C}[\![(v_0, v_1) \mapsto v]\!] \;=\; p_1\,;\mathcal{C}[\![v_1 \mapsto v]\!] \text{ if } v \in \mathit{fvars}(v_1)$$

(iv) $\mathcal{C}[\![v \mapsto \{v' \mapsto t\}(t')]\!] \;=\; \langle \mathcal{C}[\![v \mapsto t']\!], id\rangle\,;\mathcal{C}[\![(v', v) \mapsto t]\!]$

(v) If f is a function symbol (including any amended projections) then $\mathcal{C}[\![v \mapsto f(t)]\!] \;=\; \mathcal{C}[\![v \mapsto t]\!]\,;f$

(vi) $\mathcal{C}[\![v \mapsto (t_0, t_1)]\!] \;=\; \langle \mathcal{C}[\![v \mapsto t_0]\!], \mathcal{C}[\![v \mapsto t_1]\!]\rangle$

The rest of this section is devoted to showing the well-defineness of this translation. Appropriately, the generating set of given equations $\mathcal{E}$ in the target cartesian combinator theory should extend the $\equiv$-relation defined above and be exactly the translation images of those in E_0, the equations of the cartesian theory.

Whenever strong datatypes are appended to the theory, the translation requires an extension to the new terms generated by the addition of the associated constructors and factorizers. It is technically necessary to define the extension for only the constructed terms and either the fold terms (for an initial datatype) or the unfold terms (for a final datatype) since the remaining terms are expressible in terms of folds and unfolds, respectively. Yet it is instructive to see the development for the case and map terms as well and therefore appropriate to be presented here. Shown below is the incremental extension to be used for adding a strong initial datatype L:

(i) $\mathcal{C}[\![v \mapsto c_i(t)]\!] = \mathcal{C}[\![v \mapsto t]\!]\,;c_i$

(ii)

$$\mathcal{C}\left[\!\!\left[v_X \mapsto \left\{ \begin{array}{c} c_1 : v_1 \mapsto t_1 \\ \cdots \\ c_n : v_n \mapsto t_n \end{array} \right\}(t) \right]\!\!\right] \;=\; \langle \mathcal{C}[\![v_X \mapsto t]\!], id\rangle\,;\mathit{fold}^L\{\mathcal{C}[\![(v_1, v_X) \mapsto t_1]\!], \ldots, \mathcal{C}[\![(v_n, v_X) \mapsto t_n]\!]\}$$

(iii)

$$\mathcal{C}\left[\!\!\left[v_X \mapsto \left\{ \begin{array}{c} c_1(v_1) \mapsto t_1 \\ \cdots \\ c_n(v_n) \mapsto t_n \end{array} \right\}(t) \right]\!\!\right] \;=\; \langle \mathcal{C}[\![v_X \mapsto t]\!], id\rangle\,;\mathit{case}^L\{\mathcal{C}[\![(v_1, v_X) \mapsto t_1]\!], \ldots, \mathcal{C}[\![(v_n, v_X) \mapsto t_n]\!]\}$$

(iv)

$$\mathcal{C}\left[\!\!\left[v_X \mapsto L\left[\begin{array}{c} w_1 \mapsto t_1 \\ \cdots \\ w_m \mapsto t_m \end{array} \right](t) \right]\!\!\right] \;=\; \langle \mathcal{C}[\![v_X \mapsto t]\!], id\rangle\,;\mathit{map}^L\{\mathcal{C}[\![(w_1, v_X) \mapsto t_1]\!]), \ldots, \mathcal{C}[\![(w_m, v_X) \mapsto t_m]\!]\}$$

Next are the translation rules to be added when adjoining a new strong final datatype R:

(i) $\mathcal{C}[\![v \mapsto d_i(t)]\!] = \mathcal{C}[\![v \mapsto t]\!]\,;d_i$

(ii)

$$\mathcal{C}\left[\!\!\left[v_X \mapsto \left(\left| v_C \mapsto \begin{array}{c} d_1 : t_1 \\ \cdots \\ d_n : t_n \end{array} \right) (t) \right]\!\!\right] \;=\; \langle \mathcal{C}[\![v_X \mapsto t]\!], id\rangle\,;\mathit{unfold}^R\{\mathcal{C}[\![(v_C, v_X) \mapsto t_1]\!], \ldots, \mathcal{C}[\![(v_C, v_X) \mapsto t_n]\!]\}$$

(iii)

$$\mathcal{C}\left[\!\!\left[v_X \mapsto \left(\begin{array}{c} d_1 : t_1(t) \\ \cdots \\ d_n : t_n(t) \end{array} \right) \right]\!\!\right] \;=\; \langle \mathcal{C}[\![v_X \mapsto t]\!], id\rangle\,;\mathit{record}^R\{\mathcal{C}[\![(v_C, v_X) \mapsto t_1]\!], \ldots, \mathcal{C}[\![(v_C, v_X) \mapsto t_n]\!]\}$$

(iv)

$$\mathcal{C}\left[\!\!\left[v_X \mapsto R\left[\begin{array}{c} w_1 \mapsto t_1 \\ \cdots \\ w_m \mapsto t_m \end{array}\right](t) \right]\!\!\right] = \langle \mathcal{C}[\![v_X \mapsto t]\!], id\rangle\,;\mathit{map}^R\{\mathcal{C}[\![(w_1, v_X) \mapsto t_1]\!], \ldots, \mathcal{C}[\![(w_m, v_X) \mapsto t_m]\!]\}$$

The following lemma is helpful in showing the consistency of this translation:

Lemma 3.1 *For the logic-to-combinator translation $\mathcal{C}$,*

(i) *If v_L does not occur freely in any of the t_i then*

$$\mathcal{C}\left[\!\!\left[(v_L, v_X) \mapsto \left\{\!\!\left|\begin{array}{c} c_1 : v_1 \mapsto t_1 \\ \cdots \\ c_n : v_n \mapsto t_n \end{array}\right|\!\!\right\}(v_L) \right]\!\!\right] = \mathit{fold}^L\{\mathcal{C}[\![(v_1, v_X) \mapsto t_1]\!], \ldots, \mathcal{C}[\![(v_n, v_X) \mapsto t_n]\!]\}$$

(ii) *If v_L does not occur freely in any of the t_i then*

$$\mathcal{C}\left[\!\!\left[(v_L, v_X) \mapsto \left\{\begin{array}{c} c_1(v_1) \mapsto t_1 \\ \cdots \\ c_n(v_n) \mapsto t_n \end{array}\right\}(v_L) \right]\!\!\right] = \mathit{case}^L\{\mathcal{C}[\![(v_1, v_X) \mapsto t_1]\!], \ldots, \mathcal{C}[\![(v_n, v_X) \mapsto t_n]\!]\}$$

(iii) *If v_L does not occur freely in any of the t_i then*

$$\mathcal{C}\left[\!\!\left[(v_L, v_X) \mapsto L\left[\begin{array}{c} w_1 \mapsto t_1 \\ \cdots \\ w_m \mapsto t_m \end{array}\right](v_L) \right]\!\!\right] = \mathit{map}^L\{\mathcal{C}[\![(w_1, v_X) \mapsto t_1]\!]), \ldots, \mathcal{C}[\![(w_m, v_X) \mapsto t_m]\!]\}$$

(iv) *If v'_C does not occur freely in any of the t_i then*

$$\mathcal{C}\left[\!\!\left[(v'_C, v_X) \mapsto \left(\!\!\left| v_C \mapsto \begin{array}{c} d_1 : t_1 \\ \cdots \\ d_n : t_n \end{array}\right|\!\!\right)(v'_C) \right]\!\!\right] = \mathit{unfold}^R\{\mathcal{C}[\![(v_C, v_X) \mapsto t_1]\!], \ldots, \mathcal{C}[\![(v_C, v_X) \mapsto t_n]\!]\}$$

(v) *If v'_C does not occur freely in any of the t_i then*

$$\mathcal{C}\left[\!\!\left[(v'_C, v_X) \mapsto \left(\begin{array}{c} d_1 : t_1(v'_C) \\ \cdots \\ d_n : t_n(v'_C) \end{array}\right) \right]\!\!\right] = \mathit{record}^R\{\mathcal{C}[\![(v_C, v_X) \mapsto t_1]\!], \ldots, \mathcal{C}[\![(v_C, v_X) \mapsto t_n]\!]\}$$

(vi) *If v_R does not occur freely in any of the t_i then*

$$\mathcal{C}\left[\!\!\left[(v_R, v_X) \mapsto R\left[\begin{array}{c} w_1 \mapsto t_1 \\ \cdots \\ w_m \mapsto t_m \end{array}\right](v_R) \right]\!\!\right] = \mathit{map}^R\{\mathcal{C}[\![(w_1, v_X) \mapsto t_1]\!], \ldots, \mathcal{C}[\![(w_m, v_X) \mapsto t_m]\!]\}$$

Proof. The lemma simply states the effect of freely expanding the abstraction of the major augmentation terms. The derivation for only the fold-from-L term is given below where the key step, distributing $id \times p_1$, arises from the naturality of strength transformations. The remaining cases have analogous proofs: the derivation for the unfold-to-R term also depends on the naturality of strength, the case-from-L on the naturality of $c_i \times id$, the record-to-R directly on the universality property, and both map terms on the naturality of strength.

$$\mathcal{C}\left[\!\!\left[(v_L, v_X) \mapsto \left\{ \begin{array}{c} c_1 : v_1 \mapsto t_1 \\ \dots \\ c_n : v_n \mapsto t_n \end{array} \right\} (v_L) \right]\!\!\right] =$$

$$\begin{aligned}
&= \langle \mathcal{C}[\![(v_L, v_X) \mapsto v_L]\!], id\rangle \,; fold^L\{\dots, \mathcal{C}[\![(v_i, (v_L, v_X)) \mapsto t_i]\!], \dots\} \\
&= \langle p_0, id\rangle \,; fold^L\{\dots, \mathcal{C}[\![(v_i, (v_L, v_X)) \mapsto t_i]\!], \dots\} \\
&= \langle p_0, id\rangle \,; fold^L\{\dots, (id \times p_1) \,; \mathcal{C}[\![(v_i, v_X) \mapsto t_i]\!], \dots\} \\
&= \langle p_0, id\rangle \,; (id \times p_1) \,; fold^L\{\dots, \mathcal{C}[\![(v_i, v_X) \mapsto t_i]\!], \dots\} \\
&= fold^L\{\dots, \mathcal{C}[\![(v_i, v_X) \mapsto t_i]\!], \dots\}
\end{aligned}$$

□

We establish the consistency of this translation for any finite sequence of augmentations of the cartesian theory whereby each augmentation made be built for either a strong initial datatype or a strong final datatype. More precisely, since each augmentation depends on the prior specification of datatypes that produced earlier augmentations, we want to establish consistency for (1) the cartesian theory $\mathcal{T}_0$ having no strong datatypes (other than products) and (2) any theory $\mathcal{T}_n$ resulting from an augmentation of any consistently translatable theory built from a sequence of $n-1$ augmentations starting from $\mathcal{T}_0$.

The consistency result depends heavily on the following lemma that shows substitution in the programming logic corresponds exactly to composition in the category.

Lemma 3.2 *Composition is substitution:*

$$\mathcal{C}[\![v \mapsto t]\!] \,; \mathcal{C}[\![v' \mapsto t']\!] = \mathcal{C}[\![v \mapsto \sigma_{v':=t}(t')]\!]$$

Proof. The proof proceeds by a structural induction on the complexity of the term t'. The base case, i.e. the terms of the cartesian theory, has been practically shown in [1]. However, due to the fact that the cartesian theory presented here is a slight generalization of the original one in [1], we complete the base case proof by (1) noting renaming-equality is preserved by the straightforward result

$$\mathcal{C}[\![v_\tau \mapsto v_\tau]\!] = id_\tau$$

for any variable base v_τ and (2) by demonstrating the application term case:

Let $t' = \{v_0 \mapsto t_0\}(t_1)$. Then

$$\mathcal{C}[\![v \mapsto t]\!] \,; \mathcal{C}[\![v' \mapsto \{v_0 \mapsto t_0\}(t_1)]\!]$$

$$\begin{aligned}
&= \mathcal{C}[\![v \mapsto t]\!] \,; \langle \mathcal{C}[\![v' \mapsto t_1]\!], id\rangle \,; \mathcal{C}[\![(v_0, v') \mapsto t_1]\!] \\
&= \langle \mathcal{C}[\![v \mapsto t]\!] \,; \mathcal{C}[\![v' \mapsto t_1]\!] \,,\ \mathcal{C}[\![v \mapsto t]\!]\rangle \,; \dots \\
&= \langle \mathcal{C}[\![v \mapsto \sigma_{v':=t}(t_1)]\!] \,,\ \mathcal{C}[\![v \mapsto t]\!]\rangle \,; \dots \\
&= \mathcal{C}[\![v \mapsto (\sigma_{v':=t}(t_1) \,,\ t)]\!] \,; \dots \\
&= \mathcal{C}[\![v \mapsto \sigma_{(v_0,v'):=(\sigma_{v':=t}(t_1)\ ,\ t)}(t_0)]\!] \\
&= \mathcal{C}[\![v \mapsto \sigma_{v':=t}(\sigma_{(v_0,v'):=(t_1,v')}(t_0))]\!] \\
&= \mathcal{C}[\![v \mapsto \sigma_{v':=t}(\sigma_{v_0:=t_1}(t_0))]\!] \\
&= \mathcal{C}[\![v \mapsto \sigma_{v':=t}(\{v_0 \mapsto t_0\}(t_1))]\!]
\end{aligned}$$

where associativity of substitution provides the critical third-from-last step.

Composition is now assumed to be substitution in the augmented theory $\mathcal{T}_{n-1}$. For the case of an initial datatype augmentation, it suffices to show consistency for the new constructer terms and fold terms presented by $\mathcal{T}_n$:

(i) Let $t' = c_i(t_0)$. Then

$$\begin{array}{cl} & \mathcal{C}[\![v \mapsto t]\!] \,;\mathcal{C}[\![v' \mapsto c_i(t_0)]\!] \\ = & \mathcal{C}[\![v \mapsto t]\!] \,;\mathcal{C}[\![v' \mapsto t_0]\!] \,; c_i \\ = & \mathcal{C}[\![v \mapsto \sigma_{v':=t}(t_0)]\!] \,; c_i \\ = & \mathcal{C}[\![v \mapsto c_i(\sigma_{v':=t}(t_0))]\!] \\ = & \mathcal{C}[\![v \mapsto \sigma_{v':=t}(c_i(t_0))]\!] \end{array}$$

(ii) Let

$$t' = \left\{\!\!\left| \begin{array}{c} c_1 : v_1 \mapsto t_1 \\ \dots \\ c_n : v_n \mapsto t_n \end{array} \right|\!\!\right\}(t_0)$$

and assume without loss of generality that $t_0 = c_j(t_*)$. Then

$$\begin{array}{cl} & \mathcal{C}[\![v \mapsto t]\!] \,;\mathcal{C}\left[\!\!\left[v' \mapsto \left\{\!\!\left| \begin{array}{c} c_1 : v_1 \mapsto t_1 \\ \dots \\ c_n : v_n \mapsto t_n \end{array} \right|\!\!\right\}(t_0) \right]\!\!\right] \\ = & \mathcal{C}[\![v \mapsto t]\!] \,; \langle \mathcal{C}[\![v' \mapsto t_0]\!], id\rangle \,; fold^L\{\mathcal{C}[\![(v_1, v') \mapsto t_1]\!], \dots, \mathcal{C}[\![(v_n, v') \mapsto t_n]\!]\} \\ = & \mathcal{C}[\![v \mapsto t]\!] \,; \langle \mathcal{C}[\![v' \mapsto t_*]\!], id\rangle \,; \langle map^{E_j}\{p_0, fold^L\{\dots\}\}, p_1\rangle \,; \mathcal{C}[\![(v_j, v') \mapsto t_j]\!] \\ = & \langle \mathcal{C}[\![v \mapsto \sigma_{v':=t}(t_*)]\!] \;,\; \mathcal{C}[\![v \mapsto t]\!]\rangle \,; \langle map^{E_j}\{p_0, fold^L\{\dots\}\}, p_1\rangle \,; \mathcal{C}[\![(v_j, v') \mapsto t_j]\!] \\ = & \mathcal{C}[\![v \mapsto (\sigma_{v':=t}(t_*), t)]\!] \,; \langle map^{E_j}\{p_0, fold^L\{\dots\}\}, p_1\rangle \,; \mathcal{C}[\![(v_j, v') \mapsto t_j]\!] \\ = & \dots; \langle map^{E_j}\{\mathcal{C}[\![(w_A, v') \mapsto w_A]\!], \mathcal{C}\left[\!\!\left[(w_L, v') \mapsto \left\{\!\!\left| \begin{array}{c} c_1 : v_1 \mapsto t_1 \\ \dots \\ c_n : v_n \mapsto t_n \end{array} \right|\!\!\right\}(w_L)\right]\!\!\right]\}, p_1\rangle \,; \dots \\ = & \dots; \langle \mathcal{C}[\![(v_j', v') \mapsto E_j\left[\begin{array}{l} w_A \mapsto w_A \\ w_L \mapsto \{\!|\dots|\!\}(w_L)\end{array}\right](v_j')]\!], p_1\rangle \,; \dots \\ = & \dots; \mathcal{C}[\![(v_j', v') \mapsto (E_j\left[\begin{array}{l} w_A \mapsto w_A \\ w_L \mapsto \{\!|\dots|\!\}(w_L)\end{array}\right](v_j'), v')]\!] \,; \dots \\ = & \mathcal{C}[\![v \mapsto \sigma_{(v_j', v'):=(\sigma_{v':=t}(t_*), t)}(E_j\left[\begin{array}{l} w_A \mapsto w_A \\ w_L \mapsto \{\!|\dots|\!\}(w_L)\end{array}\right](v_j'), v')]\!] \,; \dots \\ = & \mathcal{C}[\![v \mapsto \sigma_{v':=t}(\sigma_{(v_j', v'):=(t_*, v')}(E_j\left[\begin{array}{l} w_A \mapsto w_A \\ w_L \mapsto \{\!|\dots|\!\}(w_L)\end{array}\right](v_j'), v'))]\!] \,; \dots \\ = & \mathcal{C}[\![v \mapsto \sigma_{v':=t}(E_j\left[\begin{array}{l} w_A \mapsto w_A \\ w_L \mapsto \{\!|\dots|\!\}(w_L)\end{array}\right](t_*), t)]\!] \,; \dots \\ = & \mathcal{C}[\![v \mapsto \sigma_{(v_j, v'):=(\sigma_{v':=t}(E_j[\dots](t_*), t)}(t_j)]\!] \\ = & \mathcal{C}[\![v \mapsto \sigma_{v':=t}(\sigma_{v_j:=E_j[\dots](t_*)}(t_j))]\!] \\ = & \mathcal{C}\left[\!\!\left[v \mapsto \sigma_{v':=t}(\left\{\!\!\left| \begin{array}{c} c_1 : v_1 \mapsto t_1 \\ \dots \\ c_n : v_n \mapsto t_n \end{array} \right|\!\!\right\}(t_0)) \right]\!\!\right] \end{array}$$

For augmenting with final datatypes, we analogously show consistency for the destructor terms and the unfold terms:

(i) For $t' = d_i(t_0)$, the proof parallels exactly the one for constructor terms.

(ii) Let

$$t' \;=\; \Bigg(\!\!\Bigg|\, v_C \mapsto \begin{matrix} d_1 : t_1 \\ \dots \\ d_n : t_n \end{matrix} \Bigg|\!\!\Bigg)\,(t_0)$$

Then

$$
\begin{aligned}
&\mathcal{C}\llbracket v \mapsto t\rrbracket \,;\mathcal{C}\left\llbracket v' \mapsto \Bigg(\!\!\Bigg|\, v_C \mapsto \begin{matrix} d_1 : t_1 \\ \dots \\ d_n : t_n \end{matrix} \Bigg|\!\!\Bigg)\,(t_0)\right\rrbracket \;; d_j \\
&= \mathcal{C}\llbracket v \mapsto t\rrbracket \,; \langle \mathcal{C}\llbracket v' \mapsto t_0\rrbracket, id\rangle \,; \mathit{unfold}^R\{\mathcal{C}\llbracket (v_C, v') \mapsto t_1\rrbracket, \dots, \mathcal{C}\llbracket (v_C . v') \mapsto t_n\rrbracket\} \,; d_j \\
&= \langle \mathcal{C}\llbracket v \mapsto \sigma_{v':=t}(t_0)\rrbracket, \mathcal{C}\llbracket v \mapsto t\rrbracket\rangle \,; \langle \mathcal{C}\llbracket (v_C, v') \mapsto t_j\rrbracket, p_1\rangle \,; \mathit{map}^{E_j}\{p_0, \mathit{unfold}\{\dots\}\} \\
&= \mathcal{C}\llbracket v \mapsto (\sigma_{v':=t}(t_0), t)\rrbracket \,; \mathcal{C}\llbracket (v_C, v') \mapsto (t_j, v')\rrbracket \,; \dots \\
&= \mathcal{C}\llbracket v \mapsto \sigma_{(v_C, v'):=(\sigma_{v':=t}(t_0), t)}(t_j, v')\rrbracket \,; \dots \\
&= \mathcal{C}\llbracket v \mapsto \sigma_{v':=t}(\sigma_{v_C:=t_0}(t_j), v')\rrbracket \,; \dots \\
&= \dots \,; \mathit{map}^{E_j}\left\{\mathcal{C}\llbracket (w_A, v') \mapsto w_A\rrbracket, \mathcal{C}\llbracket (w_L, v') \mapsto \Bigg(\!\!\Bigg|\, v_C \mapsto \begin{matrix} d_1 : t_1 \\ \dots \\ d_n : t_n \end{matrix} \Bigg|\!\!\Bigg)\,(w_L)\rrbracket\right\} \\
&= \dots \,; \mathcal{C}\left\llbracket (v_j, v') \mapsto E_j\begin{bmatrix} w_A \mapsto w_A \\ w_L \mapsto (\!|\dots|\!)(w_L) \end{bmatrix} (v_j)\right\rrbracket \\
&= \mathcal{C}\llbracket v \mapsto \sigma_{(v_j, v'):=(\sigma_{v':=t}(\sigma_{v_C:=t_0}(t_j)), t)}(E_j[\dots](v_j))\rrbracket \\
&= \mathcal{C}\llbracket v \mapsto \sigma_{v':=t}(\sigma_{v_j:=\sigma_{v_C:=t_0}(t_j)}(E_j[\dots](v_j)))\rrbracket \\
&= \mathcal{C}\left\llbracket v \mapsto \sigma_{v':=t}\left(E_j\begin{bmatrix} w_A \mapsto w_A \\ w_C \mapsto \Bigg(\!\!\Bigg|\, v_C \mapsto \begin{matrix} d_1 : t_1 \\ \dots \\ d_n : t_n \end{matrix} \Bigg|\!\!\Bigg)\,(w_C) \end{bmatrix} (\sigma_{v_C:=t_0}(t_j))\right)\right\rrbracket \\
&= \mathcal{C}\left\llbracket v \mapsto \sigma_{v':=t}\left(d_j\left(\Bigg(\!\!\Bigg|\, v_C \mapsto \begin{matrix} d_1 : t_1 \\ \dots \\ d_n : t_n \end{matrix} \Bigg|\!\!\Bigg)\,(t_0)\right)\right)\right\rrbracket \\
&= \mathcal{C}\left\llbracket v \mapsto \sigma_{v':=t}\left(\Bigg(\!\!\Bigg|\, v_C \mapsto \begin{matrix} d_1 : t_1 \\ \dots \\ d_n : t_n \end{matrix} \Bigg|\!\!\Bigg)\,(t_0)\right)\right\rrbracket \;; d_j
\end{aligned}
$$

□

The major consistency properties of the translation can now be stated and proven. The first theorem below is a specialization of the second, but it is stated due to its importance in studying basic distributive computation and explaining how the three primary combinators fold^{Sum}, case^{Sum}, and map^{Sum} for the strong initial datatype *Sum*, simply the coproduct of two datatypes, merge into essentially a *common* operation in the simplest of all predistributive theories.

Proposition 3.3 *Suppose a cartesian theory is augmented with the Sum initial datatype to form a basic predistributive theory. Then C is a consistent translation from the predistributive theory into distributive categorical combinators.*

Proof. This is a mild extension of the consistency result proven by Cockett [1] that included only the case-from-*Sum* axiom. It remains only to append that proof by showing translation consistency for the fold-from-*Sum* and map-on-*Sum* axioms.

The fold-from-*Sum* axiom is preserved because its translation differs from the case-from-*Sum* axiom translation only on the left side by their respective occurrences of the morphisms of the form $\mathit{fold}^{Sum}\{f_1, f_2\}$ and $\mathit{case}^{Sum}\{f_1, f_2\}$. Both morphisms are easily shown to be equal in a predistributive category.

The only difference between the translation of the map-on-*Sum* axiom using the assigned terms t_0 and t_1 and the case-from-*Sum* axiom using the assigned terms $b_0(t_0)$ and $b_1(t_1)$ (where b_0 and b_1 are the *Sum* constructors) are the respective occurrences on the left side of morphisms of the form $\mathit{map}^{Sum}\{f_1, f_2\}$ and $\mathit{case}^{Sum}\{f_1 \,;b_0, f_2 \,;b_1\}$. These two morphisms can be quickly verified to be equal in a predistributive category.
$\square$

We now state the major theorem of this section:

Proposition 3.4 *For any theory built from a cartesian theory by a finite sequence of augmentations of strong initial datatypes and strong final datatypes, $\mathcal{C}$ is a consistent translation.*

Proof. First we cover the initial datatype situation. Because the case-from-L and map-on-L terms can be expressed in terms of a fold-from-L term for any initial datatype L, it suffices to discuss consistency only for the fold-from-L axiom and the L fold uniqueness rule. An induction on augmentations has been obviated by the inductive proofs of Lemma 3.2. In fact, the proofs below follow those inductive proofs closely, indicating that consistency in this direction is reaffirming that *composition-is-substitution*.

(i) Fold-from-L axiom:

$$
\begin{aligned}
&\mathcal{C}\left[\!\!\left[v_X \mapsto \left\{\begin{matrix} c_1 : v_1 \mapsto t_1 \\ \dots \\ c_n : v_n \mapsto t_n \end{matrix}\right\}(c_i(t)) \right]\!\!\right] \\
&= \langle \mathcal{C}[\![v_X \mapsto c_i(t)]\!], id\rangle \,; \mathit{fold}^L\{\mathcal{C}[\![(v_1, v_X) \mapsto t_1]\!], \dots, \mathcal{C}[\![(v_n, v_X) \mapsto t_n]\!]\} \\
&= \langle \mathcal{C}[\![v_X \mapsto t]\!] \,; c_i \;,\; id\rangle \,; \mathit{fold}^L\{\mathcal{C}[\![(v_1, v_X) \mapsto t_1]\!], \dots, \mathcal{C}[\![(v_n, v_X) \mapsto t_n]\!]\} \\
&= \langle \mathcal{C}[\![v_X \mapsto t]\!], id\rangle \,; \langle \mathit{map}^{E_i}\{p_0, \mathit{fold}^L\{\dots\}\}, p_1\rangle \,; \mathcal{C}[\![(v_i, v_X) \mapsto t_i]\!] \\
&= \mathcal{C}[\![v_X) \mapsto (t, v_X)]\!] \,; \langle \mathit{map}^{E_i}\{p_0, \mathit{fold}^L\{\dots\}\}, p_1\rangle \,; \mathcal{C}[\![(v_i, v_X) \mapsto t_i]\!] \\
&= \dots \,; \langle \mathit{map}^{E_i}\{\mathcal{C}[\![(w_A, v_X) \mapsto w_A]\!], \mathcal{C}\left[\!\!\left[(w_L, v_X) \mapsto \left\{\begin{matrix} c_1 : v_1 \mapsto t_1 \\ \dots \\ c_n : v_n \mapsto t_n \end{matrix}\right\}(w_L) \right]\!\!\right]\}, p_1\rangle \,; \dots \\
&= \dots \,; \langle \mathcal{C}\left[\!\!\left[v_i', v_X) \mapsto E_i \begin{bmatrix} w_A \mapsto w_A \\ w_L \mapsto \{\!|\dots|\!\}(w_L) \end{bmatrix} (v_i') \right]\!\!\right], p_1\rangle \,; \dots \\
&= \dots \,; \mathcal{C}[\![v_i', v_X) \mapsto (E_i[\dots](v_i'), v_X)]\!] \,; \dots \\
&= \mathcal{C}[\![v_X \mapsto (E_i[\dots](t), v_X)]\!] \,; \dots \\
&= \mathcal{C}[\![v_X \mapsto \sigma_{v_i, v_X) := (E_i[\dots](t), v_X)}(t_i)]\!] \\
&= \mathcal{C}\left[\!\!\left[v_X \mapsto \{v_i \mapsto t_i\}(E_i \begin{bmatrix} w_A \mapsto w_A \\ w_L \mapsto \left\{\begin{matrix} c_1 : v_1 \mapsto t_1 \\ \dots \\ c_n : v_n \mapsto t_n \end{matrix}\right\}(w_L) \end{bmatrix} (t)) \right]\!\!\right]
\end{aligned}
$$

(ii) L fold uniqueness:

In straightforward fashion the left-hand and right-hand sides of the antecedent translate into the equation of the universal initiality diagram. The consequent translates directly into the assertion that the unique action is the combinator fold^L.

Now we examine the essential final datatype cases.

(i) Unfold-to-R axiom:

$$
\begin{aligned}
& \mathcal{C}\left\llbracket v_X \mapsto d_i\left(\left(\!\left| v_C \mapsto \begin{array}{c} d_1 : t_1 \\ \dots \\ d_n : t_n \end{array} \right|\!\right)(t)\right)\right\rrbracket \\
= \quad & \mathcal{C}\left\llbracket v_X \mapsto \left(\!\left| v_C \mapsto \begin{array}{c} d_1 : t_1 \\ \dots \\ d_n : t_n \end{array} \right|\!\right)(t)\right\rrbracket \; ; d_i \\
= \quad & \langle \mathcal{C}\llbracket v_X \mapsto t \rrbracket, id \rangle \, ; \mathit{unfold}^R \{\mathcal{C}\llbracket (v_C, v_X) \mapsto t_1 \rrbracket, \dots, \mathcal{C}\llbracket (v_C, v_X) \mapsto t_n \rrbracket\} \, ; d_i \\
= \quad & \mathcal{C}\llbracket v_X \mapsto (t, v_X) \rrbracket \, ; \mathit{unfold}^R \{\mathcal{C}\llbracket (v_C, v_X) \mapsto t_1 \rrbracket, \dots, \mathcal{C}\llbracket (v_C, v_X) \mapsto t_n \rrbracket\} \, ; d_i \\
= \quad & \mathcal{C}\llbracket v_X \mapsto (t, v_X) \rrbracket \, ; \langle \mathcal{C}\llbracket (v_i, v_X) \mapsto t_i \rrbracket, p_1 \rangle \, ; \mathit{map}^{E_i} \{p_0, \mathit{unfold}^R \{\dots\}\} \\
= \quad & \mathcal{C}\llbracket v_X \mapsto (t, v_X) \rrbracket \, ; \langle \mathcal{C}\llbracket (v_i, v_X) \mapsto (t_i, v_X) \rrbracket \, ; \mathit{map}^{E_i} \{p_0, \mathit{unfold}^R \{\dots\}\} \\
= \quad & \mathcal{C}\llbracket v_X \mapsto (\sigma_{v_i := t}(t_i), v_X) \rrbracket \, ; \mathit{map}^{E_i} \{p_0, \mathit{unfold}^R \{\dots\}\} \\
= \quad & \dots ; \mathit{map}^{E_i} \left\{\mathcal{C}\llbracket (w_A, v_X) \mapsto w_A \rrbracket, \mathcal{C}\left\llbracket (w_R, v_X) \mapsto \left(\!\left| v_C \mapsto \begin{array}{c} d_1 : t_1 \\ \dots \\ d_n : t_n \end{array} \right|\!\right)(w_R)\right\rrbracket\right\} \\
= \quad & \dots ; \mathcal{C}\left\llbracket (v_i', v_X) \mapsto E_i\begin{bmatrix} w_A \mapsto w_A \\ w_R \mapsto (\!|\dots|\!)(w_R) \end{bmatrix}(v_i')\right\rrbracket \\
= \quad & \mathcal{C}\left\llbracket v_X \mapsto \sigma_{v_i' := \sigma_{v_i := t}(t_i)}\left(E_i\begin{bmatrix} w_A \mapsto w_A \\ w_R \mapsto (\!|\dots|\!)(w_R) \end{bmatrix}(v_i')\right)\right\rrbracket \\
= \quad & \mathcal{C}\left\llbracket v_X \mapsto E_i\begin{bmatrix} w_A \mapsto w_A \\ w_R \mapsto (\!|\dots|\!)(w_R) \end{bmatrix}(\sigma_{v_i := t}(t_i))\right\rrbracket \\
= \quad & \mathcal{C}\left\llbracket v_X \mapsto E_i\begin{bmatrix} w_A \mapsto w_A \\ w_R \mapsto (\!|\dots|\!)(w_R) \end{bmatrix}(\{v_i \mapsto t_i\}(t))\right\rrbracket
\end{aligned}
$$

(ii) R unfold uniqueness:

Just as for the L fold uniqueness, the translation directly yields the universal finality diagram from the antecedent and the uniqueness of the unfold operator from the consequent.

□

3.2 Translating Combinators to Programs

We now present a consistent translation in the opposite direction. The translation of a combinator f will be denoted $\mathcal{P}\llbracket f \rrbracket$. Because we demand consistency, the definition of the translation must preserve, first of all, the definition of a cartesian category.

Naively, the translation of a combinator expression requires a choice of variable base up to variable-renaming for the resulting program. Thus the translation is first presented by using the following notational definitions of composition (";") and pairing (",") of programs to allow for combinations of variable bases that are type-equivalent but possibly syntactically unequal:

(i) $\mathcal{P}\llbracket f \rrbracket \equiv \{v \mapsto \mathcal{P}_v\llbracket f \rrbracket\}$ where v is *any* variable base for the domain of a combinator f

(ii) $\{v \mapsto t\} \, ; \{v' \mapsto t'\} \equiv \{v \mapsto \sigma_{v' := t}(t')\}$

(iii) $\langle \{v \mapsto t_0\}, \{v' \mapsto t_1\} \rangle \equiv \{v \mapsto (t_0, \sigma_{v' := v}(t_1))\}$

Using this notation on the right-hand sides of the definitions below, the translation $\mathcal{P}$ for a cartesian category is defined as follows:

(i) $\mathcal{P}[\![f;g]\!] = \mathcal{P}[\![f]\!]\,;\mathcal{P}[\![g]\!]$

(ii) $\mathcal{P}[\![\langle f,g\rangle]\!] = \langle \mathcal{P}[\![f]\!], \mathcal{P}[\![g]\!]\rangle$

(iii) $\mathcal{P}[\![id_\tau]\!] = \{v_\tau \mapsto v_\tau\}$

(iv) $\mathcal{P}[\![!_\tau]\!] = \{v_\tau \mapsto ()\}$

(v) If v is a variable base for the domain of the combinator whose symbol is f (including projection combinators) then $\mathcal{P}[\![f]\!] = \{v \mapsto f(v)\}$. That is, for a function symbol f we have $\mathcal{P}_v[\![f]\!] = f(v)$.

However, the notational definition immediately provides $\mathcal{P}[\![f]\!] = \{v \mapsto \mathcal{P}[\![f]\!](v)\}$ from the extensionality property of the target cartesian theory. This allows the equivalent re-expression of the $\mathcal{P}$ translation rules (i) and (ii) as

(i') $\mathcal{P}[\![f;g]\!] = \{v \mapsto \mathcal{P}[\![g]\!](\mathcal{P}[\![f]\!](v))\}$

(ii') $\mathcal{P}[\![\langle f,g\rangle]\!] = \{v \mapsto (\mathcal{P}[\![f]\!](v), \mathcal{P}[\![g]\!](v))\}$

where v is any variable base of the appropriate domain type. We thereby remove the dependency of the translation on the explicit choice of variable base and make the new notation behave exactly the same as its conventional usage in the cartesian theory.

With the addition of a strong initial datatype to the category, the translation is extended to the new generating morphisms — the constructors and the fold factorizers. For clarity, the translation is redundantly presented below for all three factorizers. It is straightforward to establish the translation rules for the case factorizer and the map factorizer from the translation rule defined for the fold factorizer.

(i) $\mathcal{P}[\![c_i]\!] = \{v \mapsto c_i(v)\}$ where v is a variable base for the domain of c_i

(ii) $$\mathcal{P}[\![\mathit{fold}^L\{h_1,\ldots,h_n\}]\!] = \left\{(v_L, v_X) \mapsto \left\{\!\!\left\{ \begin{array}{c} c_1 : v_1 \mapsto \mathcal{P}[\![h_1]\!](v_1, v_X) \\ \cdots \\ c_n : v_n \mapsto \mathcal{P}[\![h_n]\!](v_n, v_X) \end{array} \right\}\!\!\right\} (v_L)\right\}$$

(iii) $$\mathcal{P}[\![\mathit{case}^L\{h_1,\ldots,h_n\}]\!] = \left\{(v_L, v_X) \mapsto \left\{ \begin{array}{c} c_1(v_1) \mapsto \mathcal{P}[\![h_1]\!](v_1, v_X) \\ \cdots \\ c_n(v_n) \mapsto \mathcal{P}[\![h_n]\!](v_n, v_X) \end{array} \right\} (v_L)\right\}$$

(iv) $$\mathcal{P}[\![\mathit{map}^L\{f_1,\ldots,f_m\}]\!] = \left\{(v_L, v_X) \mapsto L\left[\begin{array}{c} w_1 \mapsto \mathcal{P}[\![f_1]\!](w_1, v_X) \\ \cdots \\ w_m \mapsto \mathcal{P}[\![f_m]\!](w_m, v_X) \end{array} \right] (v_L)\right\}$$

Similarly, the extension of the combinator translation to include strong final datatypes is (redundantly) presented below:

(i) $\mathcal{P}[\![d_i]\!] = \{v \mapsto d_i(v)\}$ where v is a variable base for the domain of d_i

(ii) $$\mathcal{P}[\![\mathit{unfold}^R\{g_1,\ldots,g_n\}]\!] = \left\{(v'_C, v_X) \mapsto \left(\!\!\left| v_C \mapsto \begin{array}{c} d_1 : \mathcal{P}[\![g_1]\!](v_C, v_X) \\ \cdots \\ d_n : \mathcal{P}[\![g_n]\!](v_C, v_X) \end{array} \right|\!\!\right) (v'_C)\right\}$$

(iii) $\mathcal{P}[\![record^R\{g_1,\ldots,g_n\}]\!] = \{(v'_C, v_X) \mapsto \begin{pmatrix} d_1 : \mathcal{P}[\![g_1]\!](v'_C, v_X) \\ \ldots \\ d_n : \mathcal{P}[\![g_n]\!](v'_C, v_X) \end{pmatrix}\}$

(iv) $\mathcal{P}[\![map^R\{f_1,\ldots,f_m\}]\!] = \{(v_R, v_X) \mapsto R\begin{bmatrix} w_1 \mapsto \mathcal{P}[\![f_1]\!](w_1, v_X) \\ \ldots \\ w_m \mapsto \mathcal{P}[\![f_m]\!](w_m, v_X) \end{bmatrix}(v_R)\}$

Theorem 3.5 *For any cartesian category closed under strong initial datatypes and strong final datatypes, $\mathcal{P}$ is a consistent translation of combinators into programs.*

Proof. The consistency for the cartesian combinator axioms was previously demonstrated in Cockett [1] using the naive form of the translation definition. The consistency of translating constructor combinators is immediate. Since all factorizers are expressible by fold or unfold factorizers, it remains only to show consistency for the rewriting reduction and the uniqueness of the fold and unfold factorizers.

The proofs employ an induction on the augmentation of strong datatypes. As before, all variables are assumed to have been renamed beforehand to avoid clashes.

L fold factorizer reduction:

$\mathcal{P}[\![c_i \times id_X\,; fold^L\{h_1,\ldots,h_n\}]\!]$

$$\begin{array}{cl}
= & \langle \mathcal{P}[\![p_0]\!]\,; \mathcal{P}[\![c_i]\!]\ ,\ \mathcal{P}[\![p_1]\!]\rangle\,; \mathcal{P}[\![fold^L\{h_1,\ldots,h_n\}]\!] \\
= & \{(v_i, v_X) \mapsto (c_i(p_0(v_i, v_X)), p_1(v_i, v_X))\}\,; \{(v_L, v_X) \mapsto \left\{\begin{array}{c} \ldots \\ c_i : v_i \mapsto \mathcal{P}[\![h_i]\!](v_i, v_X) \\ \ldots \end{array}\right\}(v_L)\} \\
= & \{(v_i, v_X) \mapsto \left\{\begin{array}{c} \ldots \\ c_i : v_i \mapsto \mathcal{P}[\![h_i]\!](v_i, v_X) \\ \ldots \end{array}\right\}(c_i(v_i))\} \\
= & \{(v_i, v_X) \mapsto \sigma_{v_i := E_i\left[\begin{array}{l} w_A \mapsto w_A \\ w_L \mapsto \left\{\begin{array}{c} \ldots \\ c_i : v_i \mapsto \mathcal{P}[\![h_i]\!](v_i, v_X) \\ \ldots \end{array}\right\}(w_L) \end{array}\right](v_i)} (\mathcal{P}[\![h_i]\!](v_i, v_X))\} \\
= & \{(v_i, v_X) \mapsto (E_i\left[\begin{array}{l} w_A \mapsto \mathcal{P}[\![p_0]\!](w_A, v_X) \\ w_L \mapsto \mathcal{P}[\![fold^L\{h_1,\ldots,h_n\}]\!](w_L, v_X) \end{array}\right](v_i), v_X)\}\,; \{(v_i, v_X) \mapsto \mathcal{P}[\![h_i]\!](v_i, v_X)\} \\
= & \langle \mathcal{P}[\![map^{E_i}\{p_0, fold^L\{h_1,\ldots,h_n\}\}]\!], \mathcal{P}[\![p_1]\!]\rangle\,; \mathcal{P}[\![h_i]\!] \\
= & \mathcal{P}[\![\langle map^{E_i}\{p_0, fold^L\{h_1,\ldots,h_n\}\}, p_1\rangle\,; h_i]\!]
\end{array}$$

L fold factorizer uniqueness:

Suppose that for $i = 1,\ldots,n$ that

$$c_i \times id_X\,; h \;=\; \langle map^{E_i}\{p_0, h\}, p_1\rangle\,; h_i\ .$$

By induction on augmentations the translations of both combinator expressions are equal. The left-hand side translates exactly to the same form as the left-hand side of the antecedent of the L fold uniqueness rule, and likewise for the right-hand side. Applying that rule, it immediately follows that the translations of h and $fold^L\{h_1,\ldots,h_n\}$ are equal.

R unfold factorizer reduction:

$$
\begin{aligned}
&\mathcal{P}[\![\mathit{unfold}^R\{g_1,\ldots,g_n\}\,;d_i]\!] \\
&= \mathcal{P}[\![\mathit{unfold}^R\{g_1,\ldots,g_n\}]\!]\,;\mathcal{P}[\![d_i]\!] \\
&= \{(v_C',v_X)\mapsto \left(\left|\begin{array}{c}\ldots\\ v_C\mapsto d_j:\mathcal{P}[\![g_j]\!](v_C,v_X)\\ \ldots\end{array}\right|\right)(v_C')\}\,;\{v_R\mapsto d_i(v_R)\} \\
&= \{(v_C',v_X)\mapsto d_i(\left(\left|\begin{array}{c}\ldots\\ v_C\mapsto d_j:\mathcal{P}[\![g_j]\!](v_C,v_X)\\ \ldots\end{array}\right|\right)(v_C'))\} \\
&= \{(v_C',v_X)\mapsto E_i\left[\begin{array}{l} w_A\mapsto w_A\\ w_R\mapsto \left(\left|\begin{array}{c}\ldots\\ v_C\mapsto d_j:\mathcal{P}[\![g_j]\!](v_C,v_X)\\ \ldots\end{array}\right|\right)(w_C)\end{array}\right](\mathcal{P}[\![g_i]\!](v_C',v_X)\} \\
&= \{(v_C',v_X)\mapsto(\mathcal{P}[\![g_i]\!](v_C',v_X),v_X)\}\,;\{(v_i,v_X)\mapsto E_i\left[\begin{array}{l} w_A\mapsto\mathcal{P}[\![p_0]\!](w_A,v_X)\\ w_R\mapsto\mathcal{P}[\![\mathrm{unfold}^R\{g_1,\ldots,g_n\}]\!](w_C,v_X)\end{array}\right](v_i)\} \\
&= \langle\mathcal{P}[\![g_i]\!],\mathcal{P}[\![p_1]\!]\rangle\,;\mathcal{P}[\![\mathrm{map}^{E_i}\{p_0,\mathrm{unfold}^R\{g_1,\ldots,g_n\}\}]\!] \\
&= \mathcal{P}[\![\langle g_i,p_1\rangle\,;\mathrm{map}^{E_i}\{p_0,\mathrm{unfold}^R\{g_1,\ldots,g_n\}\}]\!]
\end{aligned}
$$

R unfold factorizer uniqueness:

The argument proceeds by induction on augmentations in exactly the same manner as the one given for L fold uniqueness.

□

We are finally positioned for proving the equivalence theorem upon which the translation and type-checking subsystems of **Charity** rest:

Theorem 3.6 *$\mathcal{C}$ and $\mathcal{P}$ form an equivalence between a cartesian category that is closed under strong initial/final datatypes and the corresponding term logic containing all strong initial/final datatype augmentations.*

Proof. Consider the translational composition $\mathcal{C}\circ\mathcal{P}$. The argument proceeds by structural induction of combinator expressions. The cartesian cases were proved in [1]. The remaining non-trivial cases — the fold and unfold factorizers — both follow exactly the pattern of proof for the fold factorizer:

$$
\begin{aligned}
&\mathcal{C}[\![\mathcal{P}[\![\mathit{fold}^L\{h_1,\ldots,h_n\}]\!]]\!] \\
&= \mathcal{C}\left[\!\!\left[(v_L,v_X)\mapsto\left\{\begin{array}{l} c_1:v_1\mapsto\mathcal{P}[\![h_1]\!](v_1,v_X)\\ \qquad\ldots\\ c_n:v_n\mapsto\mathcal{P}[\![h_n]\!](v_n,v_X)\end{array}\right\}(v_L)\right]\!\!\right] \\
&= \mathit{fold}^L\{\ldots,\mathcal{C}[\![(v_i,v_X)\mapsto\mathcal{P}[\![h_i]\!](v_i,v_X)]\!],\ldots\} \\
&= \mathit{fold}^L\{\ldots,\mathcal{C}[\![\mathcal{P}[\![h_i]\!]]\!],\ldots\} \\
&= \mathit{fold}^L\{\ldots,h_i,\ldots\}
\end{aligned}
$$

Now consider the composition $\mathcal{P} \circ \mathcal{C}$. With comments analogous to those concerning the preceding derivation, we need to seriously look only at the proof pattern for the fold factorizer term:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{P}[\![\mathcal{C}[\![v_X \mapsto \left\{\begin{array}{c} c_1 : v_1 \mapsto t_1 \\ \cdots \\ c_n : v_n \mapsto t_n \end{array}\right\}(t)]\!]]\!]} \\
= & \mathcal{P}[\![\langle \mathcal{C}[\![v_X \mapsto t]\!], id\rangle \,;\, fold^L\{\ldots, \mathcal{C}[\![(v_i, v_X) \mapsto t_i]\!], \ldots\}]\!] \\
= & \mathcal{P}[\![\langle \mathcal{C}[\![v_X \mapsto t]\!], id\rangle]\!] \,;\, \mathcal{P}[\![fold^L\{\ldots, \mathcal{C}[\![(v_i, v_X) \mapsto t_i]\!], \ldots\}]\!] \\
= & \langle \mathcal{P}[\![\mathcal{C}[\![v_X \mapsto t]\!]]\!], \mathcal{P}[\![id]\!]\rangle \,;\, \{(v_L, v_X) \mapsto \left\{\begin{array}{c} \cdots \\ c_i : v_i \mapsto \mathcal{P}[\![\mathcal{C}[\![(v_i, v_X) \mapsto t_i]\!]]\!](v_i, v_X) \\ \cdots \end{array}\right\}(v_L)\} \\
= & \langle \{v_X \mapsto t\}, \{v_X \mapsto v_X\}\rangle \,;\, \{(v_L, v_X) \mapsto \left\{\begin{array}{c} \cdots \\ c_i : v_i \mapsto \{(v_i, v_X) \mapsto t_i\}(v_i, v_X) \\ \cdots \end{array}\right\}(v_L)\} \\
= & \{v_X \mapsto \left\{\begin{array}{c} \cdots \\ c_i : v_i \mapsto \{(v_i, v_X) \mapsto t_i\}(v_i, v_X) \\ \cdots \end{array}\right\}(t)\} \\
= & \{v_X \mapsto \left\{\begin{array}{c} \cdots \\ c_i : v_i \mapsto t_i \\ \cdots \end{array}\right\}(t)\}
\end{array}
$$

□

Remark 3.7 For the sake of compactness, the equivalence proof has been presented as an implicit amalgam of three levels of formal structure: sketches, theories, and categories. An alternative method for showing equivalences between theories using these levels explicitly is briefly outlined here.

By reviewing the proof, the reader can discover that our programs and combinators are generated from essentially the *same* primitive collections of sorts, function symbols, signatures, and equations. This common underlying base structure can be expressed in isolation as a *precartesian sketch* where finite "products" of sorts are available for forming signatures. We now consider starting the equivalence proof from a given precartesian sketch.

The *(strong) equational type theory of a precartesian sketch* can then be obtained as the cartesian theory built as according to Section 2.1 and augmented by the equations and inference rules for all strong initial and final datatypes.

The *(strong) cartesian combinator theory of a precartesian sketch* can also be standardly built as the combinator theory defined in Section 3, similarly augmented by the equations and inference rules corresponding to the initiality and finality diagrams explained in the introduction.

Either theory can be lifted to its *syntactic category.* For an equational type theory, sorts become objects, closed abstract maps become morphisms, and composition under congruence is defined as our program composition. For a combinator theory, the function symbols inductively generate the morphisms to include identity maps, final maps, projections, pairs, and composition as congruent juxtaposition of combinators. In both situations the equations are lifted directly to the category level.

Our alternative view can now be pictured as

$$
\begin{tikzcd}
\text{syntactic category} & & \text{syntactic category} \\
\text{equational type theory} \arrow[u] & \text{precartesian sketch} \arrow[l] \arrow[r] & \text{cartesian combinator theory} \arrow[u]
\end{tikzcd}
$$

where the *syntactic equivalence* of the two theories, defined as the categorical equivalence of the respective syntactic categories, becomes our new focus.

We particularly note that half of the equivalence proof now falls immediately from observing that the syntactic category of a combinator theory is actually a *generic model* of the underlying precartesian sketch. That is, there is (1) an evident interpretation or model of the sketch in the combinator syntactic category via its construction, and (2) any other model of the sketch in *any other category* determines a unique model morphism from the combinator syntactic category to the other category. In our situation, the translation composition $\mathcal{C} \circ \mathcal{P}$ is a model morphism of the combinator syntactic category into itself and thus, by uniqueness, can only be the identity translation.

Further details and explanations of sketches, their models, and model morphisms are available to the reader in [11, 10]. The complete development of this particular method is presented in [2].

4 Conclusions and Future Work

The **Charity** project at Calgary has verified that significant algorithms can be coded, compiled, and run with its categorical programming language compiler built atop the term logic. Such programs include Ackermann's function, eager and lazy versions of Quicksort, type unification, and an algorithmic approach for the CCS bisimulation relation. We expect to carry out considerable effort to design a robust user-level language that minimizes the programmer's adjustment required to master the categorical programming paradigm. For example, a "monad" syntax for coding monadic computation programs easily is currently being investigated.

The categorical programming design approach combines top-down modular design at the routine level with bottom-up structurally inductive state-transform design at the code level. Our experience suggests this design strategy quickly becomes natural for newcomers to categorical programming. For example, freely mixing both term logic expressions and categorical combinators is a future implementation goal. The mutual presence of declarative eager and lazy datatypes should also be explored from a programming practicality point of view.

References

[1] J. R. B. Cockett. Distributive logic. Technical Report CS-89-01, University of Tennessee, 1989.

[2] J. R. B. Cockett. The term logic of precartesian categories. Draft manuscript, February 1992.

[3] J. R. B. Cockett and T. Fukushima. About Charity. Unpublished manuscript, 1991.

[4] J. R. B. Cockett and D. Spencer. Strong categorical datatypes. In R. A. G. Seely, editor, *International Meeting on Category Theory 1991*, Canadian Mathematical Society Proceedings. AMS, Montreal, 1992.

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

[6] H. G. Mairson. Outline of a proof theory of parametricity. In *6th International Symposium on Functional Programming Languages and Computer Architecture*, pages 313–327, 1991.

[7] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523. North-Holland, 1983.

[8] A. Stoughton. Substitution revisited. *Theoretical Computer Science*, 59:317–325, 1988.

[9] P. Wadler. Theorems for free! In *4th International Symposium on Functional Programming Languages and Computer Architecture*, London, U.K., September 1989.

[10] C. Wells. A generalization of the concept of sketch. *Theoretical Computer Science*, 70:159–178, 1990.

[11] C. Wells and M. Barr. The formal description of data types using sketches. In *Mathematical Foundations of Programming Language Semantics*, volume 298 of *Lecture Notes in Computer Science*, Tulane University, April 1987. Springer-Verlag.

[12] G. C. Wraith. *A note on categorical datatypes*, volume 389 of *Lecture Notes in Computer Science*, pages 118–127. Springer-Verlag, 1989.