Categorical Programming with Functorial Strength

Dwight L. Spencer
B.S., Bowling Green State University, 1965
M.S., University of Michigan, 1966

A dissertation submitted to the faculty of the
Oregon Graduate Institute of Science & Technology
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science and Engineering

January 1993

The dissertation "Categorical Programming with Functorial Strength" by Dwight L. Spencer has been examined and approved by the following Examination Committee:

James G. Hook
Assistant Professor
Thesis Research Advisor

Richard B. Kieburtz
Professor

Tim Sheard
Assistant Professor

J. Robin B. Cockett
Associate Professor, University of Calgary

Dedication

To the memory of my brother, Rick

Acknowledgements

It likely began that Ohio summer morning when I, an eighth-grader, climbed into my brother Rick's jalopy and rumbled to the University of Toledo eighteen miles away. Seeing differential equations and group theory wide-eyed on a backboard for the first time and watching Rick skillfully work in such worlds made an indelible mark. Rick's passion in teaching mathematics and his scholastic attentiveness towards me became the crucial kick-start for this farm boy to reach for college. Rick passed away in 1984 - I wish so much he could be reading this now.

Mathematics became sport, an action game, when I met Irwin Pressman, a Canadian algebraic topologist at Ohio State in the '60's. He was truly *excited* about his work; he passed much of that excitement on to me. To this day, he still offers his generous and frequent advice on working in academe.

And the electronic hallways to Australia and Canada brought me Robin Cockett - he need not have answered my network query about categorical combinators and to have offered me to join his work - *but he did*, and for that gracious act I will remain indebted to him. Robin schooled me on the gritty preciseness — the "nailing down" — needed for all mathematical formalization.

I have benefited also from the financial support of Tektronix Corporation and the Oregon Graduate Institute. But key people are responsible for the directions I eventually took. David Maier — and his 1987 one-of-a-kind complexity theory course — piqued my interest in theoretical computer science and in OGI. Steve Vegdahl's incredibly efficient and thorough functional language compiler design course remains with me as the model for classroom teaching mechanics, which included being a "night watchman" for answering our late-night e-mail questions on the spot. Richard Kieburtz was responsible for igniting my interest in functional languages and introducing me to programming language theory in its various forms. And I owe particular thanks to a special group: the superbly professional librarial staff headed by Maureen Sloan. Her team searched libraries everywhere to bring back those hard-to-find critical snippets of literature that often made my understanding more complete. Finally, my wife Devon and daughter Morgan have endured my old-dog adventure in solid spirit, for which I am quite thankful.

Contents

Abstract

Categorical Programming
with
Functorial Strength

Dwight L. Spencer, Ph.D.
Oregon Graduate Institute of Science & Technology, 1993

Supervising Professor: James G. Hook

We show that significant *categorical* programs can be computed applicatively using only a category's commutative diagrams: objects as datatypes and well-behaved arrows as combinators between datatypes. No assumptions of a underlying lambda calculus lie in wait to snare us with non-termination, non-confluence, or datatype constraints. Only the *structures of data* drive the reductions. The categorical datatypes of our computing universe — both initial and final structures — can be specified incrementally and flexibly by the programmer's declarations. These datatypes arrive bundled with structurally-natural constructive or destructive operators ready for programming purposes.

Tatsuya Hagino's 1987 thesis presented the first true categorically-based computing language. Yet the Hagino datatypes were restrictive (a declared datatype might not exist!) and without guaranty of program termination. Our work shows that by gently tightening the Hagino declaration style to force confluent termination, but then relaxing it by allowing environmental context to be distributed freely throughout the computation (what we call *strength*), an expressive categorical programming language of reasonable computing power is achievable. We further lift these category-theoretical results to a term logic precisely consistent with this categorical construction to form an abstract language base for erecting robust categorical languages.

A categorical pathway to our language computationally demands a basic *first-order* characterization of strength; it is achieved here by pairing ordinary functors with *explicit* strength or context-distribution morphisms. The conventional higher-order definition of strong endofunctor determines one by maps between exponentials; its functorial behavior is internalized within the category's objects — out of reach of ordinary first-order computation that equates morphisms with values. Within a cartesian closed category, this strength alternatively appears as $\theta_{A,X} : F(A) \times X \longrightarrow F(A \times X)$, a coherent natural transformation that distributes context X into the A-parametrized datatype-forming functor F. We capture the latter formulation for general cartesian settings via the 2-category of $\mathbf{X}$-strong categories and $\mathbf{X}$-strong functors relative to a fixed category $\mathbf{X}$ of admissible contexts. Only context-distributing $\mathbf{X}$-actions are consequently needed, eliminating exponential requirements and thereby providing "computable" strong functors.

Alas, the $\mathbf{X}$-strong categories lack the critical Eilenberg-Moore T-algebra construction often-used for building conventional categorical datatypes. The remedy is provided by equivalently framing strength in terms of fibrations using projections to $\mathbf{X}$-objects as display or context-extracting maps. The equivalence is portrayed as an embedding of $\mathbf{X}$-strong categories into the 2-category of split $\mathbf{X}$-indexed categories or split fibrations over $\mathbf{X}$. This embedding yields a coherent declarative implementation of parametrized datatypes-with-strength and their constructors/destructors in the categorical programming style of Hagino and Wraith.

The initiality and finality universal properties for strong datatype construction within the split fibration setting generates a strongly normalizing weak-head categorical combinator reduction engine that distributes context throughout data structure interiors. Both "eager" initial datatype usage and "lazy" final datatype usage permit coding of complex algorithms by using $\mathbf{X}$-actions over strong datatypes dictated by programmer-specified state transformations over component datatypes.

Chapter 1

Introduction

Our goal in this work is to move the promising categorical datatypes agenda of Hagino from the theory blackboard into a useful computing engine: *Can categorical programming become practical?* We significantly broaden categorical language's expressiveness, program proving, and computability aspects principally by allowing distribution of environmental context throughout the interior of any categorical datatype we wish to create. The underlying mechanism for distribution are the strong functors whose context-distributing strengths *explicitly* appear at the first-order level. We will carry our intuition along in our discussions by means of two canonically illustrative examples of strength and strong datatypes.

First, the strength for the finite list datatype-forming functor *List*, i.e.

$$\theta^{List}_{A,X} : List(A) \times X \longrightarrow List(A \times X) \quad ,$$

can be defined whereby pairs of finite lists of A-elements and context values (x) are mapped as

$$\langle [a_1, a_2, \ldots, a_n] \, , x \rangle \mapsto [\langle a_1, x \rangle, \langle a_2, x \rangle, \ldots, \langle a_n, x \rangle]$$

The conventional list notation is used, but our general experience already tells us that two constructions — forming the null list and "cons-ing" an A-element to the head of a list — are implicitly assumed within this notation.

Second, the strength for the binary tree datatype-forming functor *Btree*, i.e.

$$\theta^{Btree}_{A,X} : Btree(A) \times X \longrightarrow Btree(A \times X) \quad ,$$

can be illustrated for a particular tree with A-elements for leaf values. We note that explicit labeling of the nodes and leaves as constructed components of the tree are used since the syntactic structure is not as easily abbreviated as for lists. A typical strength mapping looks like

$$\begin{aligned}&\langle \mathit{bnode}(\mathit{bnode}(\mathit{bleaf}(a_1), \mathit{bleaf}(a_2)), \mathit{bleaf}(a_3)), x\rangle \mapsto \\ &\qquad \mathit{bnode}(\mathit{bnode}(\mathit{bleaf}(\langle a_1, x\rangle), \mathit{bleaf}(\langle a_2, x\rangle)), \mathit{bleaf}(\langle a_3, x\rangle))\end{aligned}$$

With such strengths, expressiveness is enhanced by consolidating both local and global context or scoping within programs. Proving programs correct or equivalent is simplified by disallowing contravariant constructions and thereby always providing an inductive structure within every program. Computability is maintained at a reasonably high level under the constraint of covariance by the availability of final datatypes that give infinitary resources for programs.

1.1 Philosophy

In many mathematical instances ordinary functors and natural transformations do not provide sufficient structure to successfully capture a particular phenomenon. A solution usually entails the enrichment (lifting sets of morphisms sharing common domains and codomains to exponential objects) of type-forming functors and natural transformations. When the phenomena are programs, we recognize a critical structure-capture requirement: *the calculation of a functor should be doable at the same level as the calculation of maps.* i.e. both should be first-order concepts. This has been reinforced by Moggi [39, 38, 40] with his demonstration of the direct relation of strong monads to the semantics of computation.

For cartesian closed categories, this forces us to focus on the obvious enrichment over the internal exponentiation. In this setting an endofunctor $F : \mathbf{X} \longrightarrow \mathbf{X}$ is enriched if it is represented as a natural transformation

$$f_{A,B} : A \Rightarrow B \longrightarrow F(A) \Rightarrow F(B)$$

which behaves correctly with respect to the composition as internally given. This natural transformation provides the means for building an "explicit strength" for the ordinary functor via the following adjoint transpose correspondence:

$$\frac{X \xrightarrow{\mathit{curry}(i_{A\times X})} A \Rightarrow (A \times X) \xrightarrow{f_{A,A\times X}} F(A) \Rightarrow F(A \times X)}{F(A) \times X \xrightarrow{\theta_{A,X}} F(A \times X)}.$$

Conversely, one may formally define the notion of a functor with strength and show that the notion also implies enrichment in a similar manner by

$$\frac{F(A) \times (A \Rightarrow B) \xrightarrow{\theta_{A,A\Rightarrow B}} F(A \times (A \Rightarrow B)) \xrightarrow{F(ev)} F(B)}{A \Rightarrow B \xrightarrow{f_{A,B}} F(A) \Rightarrow F(B)}.$$

These equivalences suggest the redundancy — within cartesian closed categories — of exponentiation in playing a computational role. We therefore simply consider extending the non-requirement of exponentials to general cartesian categories where only datatype-forming functors paired with explicit strengths are presumed available for computing. This work fully develops the consequences in adhering to this philosophy. In particular, we show that a recursive computation on an ordinary CCC categorical datatype always corresponds to one using explicit strengths.

1.2 Touring

The penetration of category theory into the programming language theory curriculum (even down to the collegiate freshman level — see Lawvere [32]), has made us sensitive to many readers' needs to be categorically equipped before venturing further. Fortunately, introductory sources on elementary category theory that are satisfying to computing researchers and adequate for this work now exist: Pierce's tutorial [44], Walter's text [53], or the early chapters of either Barr and Wells' text [2] or Asperti and Longo's monograph [1]. An additional well-written elementary reference that is distinguished by its thorough attention to explaining equality within categories in terms of the behaviour of morphisms operating on generalized elements, i.e. generalizations of set-elements, is McLarty's recent text [37]. The best sources for uncovering 2-categories still remain the works of Kelly and Street [29], and Gray [15].

We augment these publications by offering in Chapters 2 and 3 brisk tutorials on the particular categorical structures we concentrate upon: categorical datatypes, as Hagino saw them, and on fibrations, a new setting for *building* datatypes.

Those mainly interested in language development and programming issues may gather some categorical intuition from the synopsis below and then travel non-stop to the term logic of Chapter 8. The categorically aware may skip ahead to Chapter 4 and its tensored settings.

1.3 Synopsis

Chapter 4 defines the 2-category of $\mathbf{X}$–strong categories which serve, we believe, as the appropriate categorical foundation of context-distribution or strength. The definition departs from previous usage, i.e. as enrichment, by adopting the generality of a non-closed setting while concentrating on the categorical product as the source and effect of a tensor action.

Datatypes are built with limits and colimits, yet the resulting 2-category of $\mathbf{X}$–strong categories disappoints us by lacking arbitrary finite limits — in categorical terms, it is not finitely complete. In particular, the category of F-algebras of an endofunctor F on an $\mathbf{X}$-strong category cannot be constructed. More precisely, the category of objects of the form

$$\begin{tikzcd}
F(A) \arrow[d, "\alpha"'] \\
A
\end{tikzcd}$$

where α is a morphism in the concerned $\mathbf{X}$-strong category and algebra morphisms are commuting squares of the form

$$\begin{tikzcd}
F(A) \arrow[r, "F(\eta)"] \arrow[d, "\alpha_1"'] & F(B) \arrow[d, "\alpha_2"] \\
A \arrow[r, "\eta"'] & B
\end{tikzcd}$$

cannot be guaranteed to be an $\mathbf{X}$-strong category. Thus there is no immediate guidance for obtaining the "correct" notion of a free monad or, more generally, a datatype. For instance, the well-accepted "correct" notion of a free monad on an endofunctor F is that it is the monad generated by the left adjoint to the underlying functor from the category of F–algebras.

Chapter 5 shows how our venue of strength can be moved to a complete setting. There we define and explain an embedding, attributed by Moggi [38, 39] to Gordon Plotkin, of the 2-category of $\mathbf{X}$-strong categories into the 2-category of split fibrations over $\mathbf{X}$. Here $\mathbf{X}$ serves as a prescribed category of contexts-of-computation. The split fibrations category, by virtue of its completeness, has the algebraic constructions and the guidance we seek.

Chapter 6 explains the limit constructions of the category of algebras for a strong initial datatype and the category of coalgebras for a strong final datatype, both within fibrations. The limits are unraveled into the universal initiality and finality diagrams from which all our combinators and computation rules for strong datatypes are derived, as detailed in Chapter 7. The fibration-based datatype notion we propose is correct in terms of its context parametrization of *all* constructors and destructors, a feature neglected in other approaches towards recursive datatype development but vital to practical programming practices.

These early technical chapters show 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** [11] code.

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

They each introduce the concept of *factorizer* that intuitively describes a program that processes a data structure by invoking sub-programs (the "factoring") that in turn work upon the structure's less complex components. The displayed formats illustrate the factorizer versions of the Hagino "right" (datatype appears as a codomain) and "left" (datatype appears as a domain) forms of datatype declaration [17] in a style similar to that suggested by Wraith [56]. Wraith's technique allows translation of the resulting category theory into the polymorphic lambda calculus and thereby passing along to the theory the calculus' strong normalization property.

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)

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

— a strength — 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. More specifically, 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 combinator theory — the accumulated collection of morphisms and datatype objects — generated by earlier declarations. The state transformers essentially supply *recursive behavorial actions* that the fold factorizer is to perform *sequentially* on components of an initial data structure.

Analogously, the "right" declaration format acts dually to also provide a type-forming functor possessing a strength. This situation spawns 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 $E_i(A, L(A))$. A destructor d_i essentially projects out the i-th "field" of type $E_i(A, L(A))$ from 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 de-transformers separately and *concurrently* determine how each record field is recursively built.

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. And by this discussion of factorizers coming from or going to a datatype, 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 m of a given parameter datatype category $\mathbf{A}$; m is called the parametric arity of the datatype we are defining. However, we take the simplifying 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 appropriate category of state types.

The strong initial and final datatypes are finally discovered within the fibrational view to require definition, respectively, by the universal strong initiality diagrams

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

and the universal strong finality diagrams

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

from which rewrite rules for the fold and unfold factorizers are directly derived. The programmer specifies the *fold* (*unfold*) action from (to) the datatype $L(A)$ ($R(A)$) with the h_i (g_i) actions on the simpler component datatypes. The mysterious $\theta_2^{E_i}$ handles the selective distribution of context (X) into only the component structures that are themselves recursively built. Recursion is exhibited in the diagrams by the additional occurrences of the factorizer within the component E_i-functored maps.

Additional specializations of these factorizers are 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*, respectively map^L and map^R 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 assumption that an initial datatype declaration creates n constructors for building 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:

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

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

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

For any strong functor F, the combinator $\mathit{map}^F\{f_1,\ldots,f_k\}$ can be represented as the combinator sequence $\theta^F\,; F(f_1,\ldots,f_k)$. 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 sequence of datatype declarations form a confluent and terminating polymorphic reduction system for evaluating programs that compute with the declared datatypes.

A prototypical, yet versatile, categorical programming language (term logic) for all such strong datatypes whose computation semantics is isomorphic to these combinator rules is built in Chapter 8. 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 [11]).

In Chapter 9 the primitive induction technique for proving programs equal is illustrated in detail for several examples. The major lesson of this chapter is witnessing the interplay between the term logic and the categorical frame work in devloping equivalence proofs.

The functional completeness and correctness of the term logic is exhibited in Chapter 10 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 Chapter 11.

Chapter 12 closes this work by describing the on-going work towards extending our methods as well as suggestions for valuable investigation towards broadening the categorical focus of the **Charity** system.

Considerable progress in this effort resulted from extensive collaboration with Robin

Cockett in “thrashing and wringing” these ideas into their present form. Nevertheless, the author been principally charged with organizing, editing, extending, and writing the joint research papers that underlie this thesis, and has specially contributed the definitions of the specialized combinators (Chapter 7), the development of term logic (Chapter 8), the illustrations of the equivalence-proof methodology (Chapter 9), and the proof of the term logic's correctness (Chapter 10).

Chapter 2

Hagino Datatypes

Hagino's thesis [17] marks the advent of expressive categorical programming. Prior to his work categories found their most successful utility concerning computation as computational *models*. Categorical programming broadens their use to acting as *mediums* of computation. The categorical focus is expanded to develop abstract machines that use only a category's commutative diagrams for reducing a program built from compositions of morphisms to a useful value in canonical form. An equivalent perspective is to treat the collection of well-formed programs to be available to the user as the categorical setting in which to do computation.

This chapter reviews in outline form the Hagino strategy and its resulting declarative language. Any study of categorical programming must, we feel, travel carefully through his milestone development. Yet, we will eventually point out that strategy's impoverished view of the use of context and environment within computation, and its apparent over-generality of datatype declaration. Thus Hagino's work serves well as a baseline reference for our improvements and extensions in creating categorical data types.

2.1 The Categorical Programming Viewpoint

The foundational view of categorical datatypes and their associated categorical reductions is introduced conceptually with the aid of the "recursion" diagrams below.

$$\begin{tikzcd}
\text{COMPONENTS} \arrow[r, "\textbf{construct}"] \arrow[d, dotted, "\textbf{recurse}"'] & \text{INITIAL DATATYPE} \arrow[d, dotted, "\textbf{operate}"] \\
\text{PARTIAL RESULT} \arrow[r, "\textbf{process}"'] & \text{OUTPUT}
\end{tikzcd}$$

$$\begin{tikzcd}
\text{INPUT} \arrow[r, "\textbf{process}"] \arrow[d, dotted, "\textbf{operate}"'] & \text{PARTIAL RESULT} \arrow[d, dotted, "\textbf{recurse}"] \\
\text{FINAL DATATYPE} \arrow[r, "\textbf{destruct}"'] & \text{COMPONENTS}
\end{tikzcd}$$

To read these diagrams, the arrows should be considered as combinator operations that transform elements of one datatype to elements of another datatype. The **construct** morphism, i.e. the collection of "constructors", represents an embedding of all the component datatypes into the parent initial data type.

The first diagram states that in order to define *any* program operation that accepts a data structure of an initial datatype and generates a value of the output datatype, the programmer must *specify* the processing on the component data structures (necessarily less complex) of the initial data structure. Moreover we insist that our category is rich enough to insure that specification is *universal*: every specification determines exactly one operation. This property is called *initiality* because the determined operation in turn satisfies the commutativity of the recursion diagram and thereby gives a structural homomorphism from the constuctor to the specification. That is, there exists a unique (up to categorical isomorphism) homomorphism from the construction to every specification. To attain reasonable computational power, we should also allow the possibility that an initial data structure may be recursively defined; components may be expressed in terms of the parent initial datatype. The specified processing recursively descends through a succession of partial results into the components of its input data structure.

Ignoring for the present questions of confluence and coherence, the initiality diagram expresses a rewrite rule. The combinator composition of the top and right sides can be rewritten as the composition of the left and bottom sides. The second path traverses datatypes whose outermost structure is less complex than the initial datatype. Reduction may continue within the second path, successively producing combinator compositions that involve simpler and simpler datatypes, until some pre-assumed primitive datatypes, e.g. simple products and sums, are reached. For recursive datatypes, it is often appropriate to re-apply the same rewrite rule. Once the primitive datatypes have been reached, the computation can be explicitly carried out.

It is well-known that in a category with sufficient limits (1) the existence of an initial (final) datatype with its constructor (destructor) is unique up to categorical isomorphism and (2) the constructor (destructor) itself is an isomorphism. The latter result in the case of recursive initial datatypes should be treated with conservatism. Wraith [56] and others have quite properly opined that the additional equational fixpoint aspect of an isomorphic constructor for a recursive datatype is mere happenstance. It also obscures the understanding of the second recursion diagram which delineates recursive final datatypes. Rather, the initiality of the constructor and the finality of the destructor become the key contributions to our computational development.

The second diagram expresses the dual of the first: starting from a "seed" input value, a data structure can be built, or expanded, by specifying the processing needed to build each of its components. The universal property involved here is *finality*: every specification for building the components generates a unique building operation for the final data structure. Again the components may be permitted to be recursively defined in terms of the parent final datatype. Successive expansions may consequently be required to create the final structure, or at least a useful close-enough or as-needed approximation. From this discussion, the reader should correctly anticipate the possibility of final data structures that are infinitary.

The computation engine just described is built — as required — by the categorical programmer in stages by first assuming existence of the primitive datatypes for which categorical computation is explicit and easy to implement and then successively adding new recursion diagrams that are expressed with earlier-specified datatypes.

2.2 Recursion Examples from the Past

Let us review some classical recursion diagrams and particularize our concepts to them. The natural numbers datatype can be defined by splitting the constructor into two component constructors, *zero* and *succ*:

$$\begin{tikzcd}
1 + Nat \arrow[r, "\langle zero \mid succ \rangle"] \arrow[d, dotted, "id + op"'] & Nat \arrow[d, dotted, "op"] \\
1 + C \arrow[r, "\langle f \mid g \rangle"'] & C
\end{tikzcd}$$

The initiality property says that given any $f : 1 \longrightarrow C$ and $g : C \longrightarrow C$, we can form precisely one operation $op : Nat \longrightarrow C$ that behaves as f on *zero* and as $g(op((n)))$ on $succ(n)$. The operator's behavior reflects its recursive occurrence within the **recurse** morphism of the diagram (i.e $id + op$). In the particular case that C equals Nat, it becomes immediate that all primitive recursive functions are expressible in this style, e.g. see Lambek and Scott [31]. The categorical sums and co-pairs in the diagram allow it to be equivalently described as the more familiar pair of squares:

$$\begin{tikzcd}
1 \arrow[r, "zero"] \arrow[d, dotted, "id"'] & Nat \arrow[d, dotted, "op"] & Nat \arrow[l, "succ"'] \arrow[d, dotted, "op"] \\
1 \arrow[r, "f"'] & C & C \arrow[l, "g"]
\end{tikzcd}$$

We move on to a more complex datatype that is additionally *parametrized*: lists containing elements of a parameter datatype A:

$$\begin{tikzcd}
1 + A \times \mathit{List}(A) \arrow[r, "\langle \mathit{nil} \mid \mathit{cons} \rangle"] \arrow[d, dotted, "\mathit{id} + (\mathit{id} \times \mathit{op})"'] & \mathit{List}(\mathrm{A}) \arrow[d, dotted, "\mathit{op}"] \\
1 + A \times C \arrow[r, "\langle f \mid g \rangle"'] & C
\end{tikzcd}$$

The object $\mathit{List}(A)$ is defined by initiality for each choice of the datatype A. The constructors are likewise defined and should technically be annotated with "A". Also, this object assignment can be extended to a covariant endofunctor by specifying the morphism $\mathit{List}(f)$ for $f : A_1 \longrightarrow A_2$ via initiality of the recursion diagram

$$\begin{tikzcd}
1 + A_1 \times \mathit{List}(A_1) \arrow[rr, "\langle \mathit{nil} \mid \mathit{cons} \rangle"] \arrow[d, dotted, "\mathit{id} + (\mathit{id} \times \mathit{List}(f))"'] & & \mathit{List}(\mathrm{A}_1) \arrow[d, dotted, "\mathit{List}(f)"] \\
1 + A_1 \times \mathit{List}(A_2) \arrow[r, "\mathit{id} + (f \times \mathit{id})"'] & 1 + A_2 \times \mathit{List}(A_2) \arrow[r, "\langle \mathit{nil} \mid \mathit{cons} \rangle"'] & \mathit{List}(A_2)
\end{tikzcd}$$

and showing functoriality via an elementary diagram chase. This diagram also exposes the constructors as natural transformations between the functors $1 + (\_ \times \mathit{List}(\_))$ and $\mathit{List}(\_)$. Thus *List* is rightfully a datatype-forming functor.

2.3 Sums-and-Products Generalization

Our observations with these well-studied initial datatypes can be patterned into the general diagram

$$\begin{tikzcd}
\oplus_1^n E_i(A, L(A)) \arrow[r, "\langle c_1 \mid \ldots \mid c_n \rangle"] \arrow[d, dotted, "\oplus_1^n E_i(id, h)"'] & L(A) \arrow[d, dotted, "h"] \\
\oplus_1^n E_i(A, C) \arrow[r, "\langle f_1 \mid \ldots \mid f_n \rangle"'] & C
\end{tikzcd}$$

where each E_i is a covariant bifunctor with its first argument reserved for the parameter datatype and $\oplus_1^n$ represents the n-fold categorical sum. The occurrences of $L(A)$ in the E_i expressions can be thought of as another variable "L" into which only one value, viz. $L(A)$, can be substituted to effect initiality. The c_i, $i = 1, \ldots, n$ represent the *constructors* of the parametrized initial datatype which has been expressed as the functor L. And, as we noted in the list datatype example, with parametrization constructors and destructors become polymorphic in A; they form natural transformations $E_i(\_, L(\_)) \xrightarrow{\bullet} L(\_)$ and $L(\_) \xrightarrow{\bullet} E_i(\_, L(\_))$, respectively.

The same generalization with familiar final datatypes, such as infinite lists, additionally gives

$$\begin{tikzcd}
C \arrow[r, "\langle g_1, \ldots, g_n \rangle"] \arrow[d, dotted, "k"'] & \otimes_1^n E_i(A, C) \arrow[d, dotted, "\otimes_1^n E_i(id, k)"] \\
R(A) \arrow[r, "\langle d_1, \ldots, d_n \rangle"'] & \otimes_1^n E_i(A, R(A))
\end{tikzcd}$$

where the d_i represent the *destructors* of the parametrized final datatype which has been denoted by the functor (and variable within the E_i expressions) R. And expectedly, $\otimes_1^n$ is the n-fold categorical product. In both cases this generalization compacts or tuples n commutative squares, one for each datatype component, into a single diagram. The compacted diagrams are also immmediately seen as special cases of universality for $\oplus^n E(A, \_)$-algebras and $\otimes^n E(A, \_)$-coalgebras. Our initial datatype is thusly an initial algebra, our final datatype a final coalgebra. The specification of a parametrized datatype in this style is therefore complete once (1) the initiality or the finality condition has been chosen and (2) the component covariant functors $E_i(A, \_)$ have been selected from the collection of previously specified datatype functors. The general structure of the datatypes possible under this regime are sums and products of component (possibly recursive) datatypes, i.e. only those of first-order.

2.4 Dialgebra Generalization

Hagino probed a far-reaching generalization of this specification method by defining initial and final parametrized *dialgebras*:

$$\begin{tikzcd}
\vec{E}(A, L(A)) \arrow[r, "\vec{c}"] \arrow[d, dotted, "\vec{E}(id, op)"'] & \vec{F}(A, L(A)) \arrow[d, dotted, "\vec{F}(id, op)"] \\
\vec{E}(A, C) \arrow[r, "\vec{f}"'] & \vec{F}(A, C)
\end{tikzcd}$$

$$\begin{tikzcd}
\vec{E}(A, C)) \arrow[r, "\vec{g}"] \arrow[d, dotted, "\vec{E}(id, op)"'] & \vec{F}(A, C) \arrow[d, dotted, "\vec{F}(id, op)"] \\
\vec{E}(A, R(A)) \arrow[r, "\vec{d}"'] & \vec{F}(A, R(A))
\end{tikzcd}$$

We quickly notice the appearance of general functors in *both* the domains and the codomains, blurring somewhat the intuitive roles of constructors ($\vec{c}$), destructors ($\vec{d}$), and recurse morphisms. Additionally, all functors and morphisms have been finitely tupled (as indicated by the notation), similar to how component functors, constructors, destructors, and specified processing had been tupled — paired — in the *Nat* and *List* examples. Thus $\vec{E}(\_, \_)$ means $(E_1(\_, \_), \ldots, E_n(\_, \_))$ is a tupled collection of functors and $\vec{f}$ portrays $(f_1, \ldots, f_n)$ as a morphism between objects in the n-th power of some parent category. But the major generalizing aspect lies in functorial *varity*: in order to attain higher-order datatypes and their concommitant computing strength, the Hagino parametrized dialgebra permits the domain functors (E_i) to be of arbitrary variance in each of the parameter arguments (i.e. say for $A = (A_1, \ldots, A_n)$) and the codomain functors to be of opposing variance (in a monoidal sense - see Hagino [17] for details) in the corresponding parameter arguments. His complicated definition anticipates the possibility of defining exponential or function-space datatype bifunctors that typically mix covariance and contravariance.

By using second-projection functors either for all the domains in the finality case or for all the codomains in the initiality case, along with covariant-only functors, the reader can easily establish that the dialgebra generalizes the simpler earlier-mentioned categorical datatypes. Furthermore, the ascent to higher-order can be illustrated by the following final dialgebra definition (singlely tupled, $n = 1$) of an exponential datatype:

$$\begin{tikzcd}
Pr_0(A) \otimes C \arrow[r, "g"] \arrow[d, dotted, "Pr_0(id) \otimes op"'] & Pr_1(A) \arrow[d, dotted, "id"] \\
Pr_0(A) \otimes Exp(A) \arrow[r, "eval"'] & Pr_1(A)
\end{tikzcd}$$

Here Pr_i are projection functors, so A must be in the squared parameter category, i.e. A has the form (A_1, A_2). Rephrased, the defined Exp datatype is parametrized with two arguments from the parameter category. This diagram is fitted to the dialgebra definition by letting $E_1((A_1, A_2), C) = Pr_0((A_1, A_2)) \otimes C$ and $F_1((A_1, A_2), C) = Pr_1((A_1, A_2))$. It asserts that for every process morphism $g : A_1 \otimes C \longrightarrow A_2$ there exists a unique mediating morphism $op : C \longrightarrow Exp(A_1, A_2)$ that behaves compatibly as the "curry" operator. Clearly the particular diagram provides unique existence in the opposite direction, giving the familar adjunction for defining exponentials.

The extreme generality of the dialgeba leads immediately to pathology and the question of characterizing the "computable" dialgebras that guarantee existence of initial or final datatypes. For example, consider the following dialgebras, excerpted from the Hagino thesis, which are assumed to lie within a cartesian closed category:

$$\begin{tikzcd}
A \arrow[r, "c"] \arrow[d, dotted, "id"'] & \mathit{List}(L(A)) \arrow[d, dotted, "\mathit{List}(op)"] \\
A \arrow[r, "f"'] & \mathit{List}(C)
\end{tikzcd}$$

$$\begin{tikzcd}
A \arrow[r, "g"] \arrow[d, dotted, "id"'] & \mathit{List}(C) \arrow[d, dotted, "\mathit{List}(op)"] \\
A \arrow[r, "d"'] & \mathit{List}(R(A))
\end{tikzcd}$$

The implicit domain functor in both examples is Pr_0. Hagino pointed out that the first specification (an initiality condition) yields $L(A)$ to be the category's initial object, and the second specification (a finality condition) would force degenerate collapse! Still other diagrams could specify adjoints that frequently do not exist in reasonable computational settings such as sets, e.g. left adjoints of $\_ \otimes A$ and $\_ \oplus A$.

2.5 Computable Dialgebras

Hagino expended considerable effort in constructing a computational semantics that takes the mixed varity into account (viz. his functorial calculus for specification, Categorical Specification Language or CSL) in order to reason formally about computability. He worked backwards from the pathological examples to establish appropriate restrictions on the forms of datatype specifications. These restrictions generate reduction rules that would converge to weak-head canonical forms having heads consisting only of constructors and final datatype building operations.

The resulting definitions of "computable" dialgebra-based datatypes are reported below to purposely give perspective in later discussions of work by Wraith, Cockett and the author that relaxed these requirements to obtain simpler requirements for datatypes to be "computable".

Definition 2.5.1 *All functors* **productive in the variable** X *are inductively generated by the following:*

- *identity functor and projection functors are productive in each of their argument variables,*
- *if* $F(X_1, \ldots, X_n)$ *is productive in its i-th variable* X_i *and* G *is productive in the variable* Y*, then for any collection of functors* $G_1, \ldots, G_{i-1}, G_{i+1}, \ldots, G_n$ *that do not contain* Y*,* $F(G_1, \ldots, G_{i-1}, G, G_{i+1}, \ldots, G_n))$ *is productive in* Y*.*

Definition 2.5.2 *A final datatype* $R(A)$ *is* **productive in the parameter argument** A_i *if*

- R *does not occur in any of the codomain functors* F_j*,*
- A_i *does not occur in any of the domain functors* E_j*,*
- *the i-th destructor,* d_i *has the typing* $d_i : R(A) \longrightarrow F_i(A)$*,*
- F_i *is productive in the variable* A_i

Definition 2.5.3 *A final datatype R(A) is* **computable** *if all of its domain functors* E_i *are productive in the variable R.*

Definition 2.5.4 *An initial datatype L(A) is* **computable** *if all of its codomain functors are last-coordinate projections (i.e.* $F_i(A, C) = C$*).*

Hagino designed and implemented a declaration format by which programmers can build a *succession* of computable categorical datatypes. Each declaration automatically (1) generates the associated constructors/destructors that are available for coding specifications of programs which, in turn, become uniquely determined by the initiality/finality properties and (2) extends the reduction system of the predecessor datatypes that derived directly from the basic re-writing properties of their respective recursion diagrams. His principal result is

Theorem 2.5.5 *Any succession of declarations of computable initial and final dialgeba-based datatypes produces a strongly-normalizing reduction system.*

It should be noted that Hagino's reductions took place only within a deduction system (8 inference rules) and an associated simply-typed lambda calculus. To improve the understanding of the Hagino datatypes, Wraith [56] undertook to give a polymorphic lambda calculus translation for a significant subset of the possible computable Hagino declarations: build initial datatypes with only last-coordinate projections as codomain functors ($F_i(A, C) = Pr_1(A, C) = C$) and final datatypes with only last-coordinate projections as domain functors ($E_i(A, C) = Pr_1(A, C) = C$). Wraith's recursion diagrams looked exactly as the simpler ones discussed prior to the dialgebras, but the categorical setting remained equivalent to that of Hagino's: a bicartesian-closed category possessing initial algebras and final coalgebras.

The contribution of Wraith to this work is his insight to restricting the ungainly definitions of computable dialgebras to much simpler and more polymorphically intuitive forms without losing strong normalization and without sacrificing any fundamentally vital datatypes that programmers use daily. His restricted declaration style is adopted in our development of categorical datatypes, but *without the presumption that exponentials exist.*

2.6 Shortcomings

Looking back, the inclusion of exponentials forces complications in combining covariance with contravariance that have, so far, been attacked with only drastic and partially successful means. Even Wraith had to labor in his paper to overcome the mixed-varity problems in his smaller system. We therefore resolve to remain at the first-order level throughout this thesis by exploring how far datatypes can be categorically produced without the assumption of exponentials.

Another major shortcoming of *all* the recursion diagrams that have been presented so far still awaits our concern: the absence of environmental scope or context. All expressive programming demands the capability to compute within the current scope of context values, but these diagrams compute only globally, i.e. in isolation. Certain datatypes can be re-expressed equivalently with context to some degree, but the general means to allow the distribution of context *thoughout* a computation will be accomplished in our work by moving into the richer framework of a first-order categorical setting that has had historical success in capturing correctly the semantics of contexts — fibrations.

Chapter 3

Fibrations

This introduction to the theory of fibrations will be constrained to definitions, examples, and properties that are pertinent for understanding the particular fibrational developments presented in later chapters. We ease our way to the formal definitions by preceding them with intuitive descriptions. The reader who wishes to probe more deeply into the foundational world of fibrations is encouraged to inspect the critical (in fact, relatively few) sources: the introductory chapter of Bart Jacobs' thesis [24], Wesley Phoa's excellent lecture notes [43], the fibrations chapter of Barr and Wells' textbook [2], the classic English language source of Gray [14], and the seminal French language treatise of Benabou [3]. For understanding the basic relationships between fibrations and constructive logic, the advanced reader may consult Pavlović's thesis [42].

Fibrations are explained below principally as mathematical entities that possess important indexing properties. But we can motivate straightaway why these abstractions should be employed in computational work:

- Categories of fibrations are abundant in categorical limits and colimits needed to define a reasonable variety of categorical datatypes.
- The "fibers" of fibrations (defined later in this chapter) provide a conceptually tractable universe-of-computation of small categories, along with their objects, intra-fiber morphisms, and inter-fiber morphisms. The alternative is often to work

at the "metacategory" level using, for example, the category of all categories as our foundation.

- Forming datatypes with functors frequently becomes computationally ambiguous by the presence of pseudo-functoriality: the preservation of morphism composition only up to isomorphism. Fibrations, specifically the "split" variety, nail down the precise reduction rules for each datatype.
- Fibrations have been historically good at classifying computations by the context or assumption lists under which the computations are to be performed.

With this buttressing of fibrations' computational worth, we now proceed to tackle the mathematical formalities.

3.1 Indexed Covers

The class of problem to be addressed by fibrations is to express a category $\mathbf{E}$ as a disjoint union or covering of specified collections of objects, whereby all member objects of each collection is are *identically* "indexed" in some fashion by an object in a category $\mathbf{B}$. That is, within a collection every member holds in common an indexing property reflected in some way by a single $\mathbf{B}$-object. Moreover, every object of $\mathbf{B}$ must be an indexing object for a collection of objects of $\mathbf{E}$. $\mathbf{E}$ is often called the *total* category and $\mathbf{B}$ is usually termed an *indexing* or *base* category for $\mathbf{E}$. Here are some common general situations of abstract indexing that have been well-modeled by fibrations:

contexts: $\mathbf{E}$ is a category of terms with ordered sets of free typed variables (for example, sequents) with morphisms represented as substitutions to transform a term in one context into a term having another context. $\mathbf{B}$ is a category of datatypes formed from products of primitive types and morphisms built from tuplings of a primitive set of morphisms between the primitive types. Each object in $\mathbf{B}$ indexes the collection of terms in **E** whose free variables, when tupled, are of the type represented by that object.

type dependency: **E** is a collection of dependent types with its morphisms as proof derivations between types. **B** is the collection of quantifier types that serve as the bounds of quantification. Each object in **B** indexes all the dependent types whose quantification bound is that object.

Variations of these situations can be studied in the literature. For example, fibrations for Martin-Löf type theory and the calculus of constructions are presented by Hyland and Pitts [23], and fibrations for general theories of contexts are thoroughly developed by Cartmell [6].

3.2 Two Equivalent Views

Two principal categorical alternatives for capturing such indexed phenomena are available: fibrations and indexed categories. First, an indexed category is a functor $F : \mathbf{B}^{op} \longrightarrow \mathbf{Cat}$, that takes an indexing object I to a *category* of objects that form the object family to be indexed by I. In general, indexed categories preserve compositions contravariantly only up to isomorphism, i.e. they are actually pseudo-functors. A fibration, on the other hand, goes roughly in the opposite direction but requires a more complicated definition. It has the additional virtue of being a true functor, i.e. perserves compositions on the nose, and is of the form $\mathcal{P} : \mathbf{E} \longrightarrow \mathbf{B}$ where special reindexing properties hold and an **E**-object is mapped to its indexing object. In the latter view all of the families of like-indexed objects are thought of as belonging to a sub-category of a single category **E**, thereby often making fibrations more tractible for technical reasoning than indexed categories. We frequently follow this view by referring to the fibration by its total category **E** rather than the functor $\mathcal{P}$. Still, indexed categories are widely regarded as being the more intuitive and understandable option. This respect for indexed categories is exemplified in present research towards using them as vehicles for modular specification [13] and even specification of programs [20]. For a thorough development of pure indexed category theory, the curious reader is urged to consult Johnstone and Paré's work [26].

3.3 Some Friendly Fibrations

The set-theoretic intuition suggested by "indexing" remains appropriate for aiding our understanding of the earlier-mentioned abstract applications and many others in fibrational terms. We will therefore lean heavily upon some "canonical" fibrations to cleanly explain fibrations without creating unnecessary technical debris involving type theory, proof theory, and so on.

The first canonical example's base category $\mathbf{B}$ is the category of sets, **Sets**. Its total category is $\mathit{Fam}(\mathbf{Sets})$, defined as follows:

objects: families $\{X_i\}_{i \in I}$ where $I \in \mathbf{Sets}$ and $X_i \in \mathbf{Sets}$,

morphisms: $f : \{X_i\}_{i \in I} \longrightarrow \{Y_j\}_{j \in J}$ specified by a set function $\phi : I \longrightarrow J$ and a collection of set functions $\{f_i : X_i \longrightarrow Y_{\phi(i)}\}$.

The map $\mathcal{P} : \mathit{Fam}(\mathbf{Sets}) \longrightarrow \mathbf{Sets}$ that assigns $\{X_i\}_{i \in I}$ to its indexing set I and maps a $\mathit{Fam}(\mathbf{Sets})$-arrow (between possibly different-indexed families) to its underlying defining set function ϕ (between the corresponding index sets) is easily seen to be a functor. This functor will serve as a canonical fibration for building and illustrating the necessary definitions. Here each object collection in $\mathbf{B}$ whose members are identically indexed by I in **Sets** consists of all $\mathbf{B}$-objects, i.e. families, of cardinality equal to that of I.

The second example is the codomain functor from the arrow category $\mathbf{B}^{\rightarrow}$ of any category $\mathbf{B}$ that possesses pullbacks. That is, $\mathit{Cod} : \mathbf{B}^{\rightarrow} \longrightarrow \mathbf{B}$ where a $\mathbf{B}^{\rightarrow}$-morphism (f, g) maps to its underlying codomain morphism-as-object k:

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "h"'] & B \arrow[d, "k"] \arrow[rr, phantom, "\mapsto", shift right=4ex] & & B \arrow[d, "k"] \\
C \arrow[r, "g"] & D & & D
\end{tikzcd}$$

Here we see that $\mathbf{B}$ indexes its own slice categories, i.e. an object B in $\mathbf{B}$ is the index for the sub-category $\{\mathbf{B} \mid B\}$ of $\mathbf{B}^{\rightarrow}$. The slice category's objects share the common "indexing" property determined by B of simply being $\mathbf{B}$-morphisms having B as codomain.

The technical requirement (pullbacks) of the second example exposes our demand for something more than just indexing from fibrations: we also need the important contravariant substitutive property of *reindexing*. Such a property connotes a capability of "lifting" base category morphisms between index objects up to reindexing operations in the total category, as will be explained in the next section.

3.4 Fibrations as Cartesian Liftings

The influence of the more comfortable indexed category view is seen by requiring fibrations to lift morphisms in the base $\mathbf{B}$ to morphisms in the total category $\mathbf{E}$ in a manner so that the lifting provides an eventual means for building a contravariant functor upward from the base into **Cat**. Such a lifted morphism is called a *cartesian map* and is defined formally in terms of a factorization property:

Definition 3.4.1 *Suppose* $\mathbf{B}$ *and* $\mathbf{E}$ *are categories and* $\mathcal{P} : \mathbf{E} \longrightarrow \mathbf{B}$ *is a functor. An* $\mathbf{E}$*-morphism* $f : Y \longrightarrow X$ *is* **cartesian** *if, given any map* $g : Z \longrightarrow X$ *such that the* $\mathcal{P}$*-image of* g *factors through the* $\mathcal{P}$*-image of* f *(see diagram below), then there exists uniquely, up to isomorphism, a* $\mathbf{E}$*-map* $h : Z \longrightarrow Y$ *that factors* g *through* f *and whose* $\mathcal{P}$*-image completes the factorization in* $\mathbf{B}$*.*

$$\begin{tikzcd}
Z \arrow[rr, "g"] \arrow[dr, dotted, "\exists ! h"'] \arrow[dd, dotted, "\mathcal{P}"'] & & X \arrow[dd, dotted, "\mathcal{P}"] & \text{(in } \mathbf{E}\text{)} \\
 & Y \arrow[ur, "f"'] \arrow[dd, dotted, "\mathcal{P}"] & & \\
J \arrow[rr, "\phi"] \arrow[dr, "\psi"'] & & I & \text{(in } \mathbf{B}\text{)} \\
 & K \arrow[ur, "\alpha"'] & &
\end{tikzcd}$$

In the *Fam*(**Sets**) example, it is straightforward to show that the cartesian maps $f : \{X_i\}_{i \in I} \longrightarrow \{Y_j\}_{j \in J}$ are exactly those for which all the corresponding $f_i : X_i \longrightarrow Y_{\phi(i)}$ are set isomorphisms. That is, "only reindexing" or "only substitution" is really going on. In more abstract settings cartesian maps routinely serve as the reindexing, relabeling, inverse image, or substitution mechanism for the model.

For example, a canonical lifting in *Fam*(**Sets**) to a cartesian map begins by specifying (1) an index set map $\alpha : K \longrightarrow I$ in **Sets** and (2) a family $\{X_i\}_{i \in I}$ in the $\mathcal{P}$-preimage of I. A cartesian map fitted to this specification can be constructed as $f : \{X_{\alpha(k)}\}_{k \in K} \longrightarrow \{X_i\}_{i \in I}$ by the collection $\{f_k : X_{\alpha(k)} \longrightarrow X_{\alpha(k)}\}$ of identity morphisms.

For the *Cod* fibration, the cartesian maps are precisely the pullback diagrams of **B**. The canonical lifting of a **B**-morphism $k : B \longrightarrow D$ with respect to a $\mathbf{B}^{\rightarrow}$-object $f : A \longrightarrow B$ is pulling back f along k.

Because dual categorical situations arise often, we also include a *covariant* version of lifting:

Definition 3.4.2 *Suppose* **B** *and* **E** *are categories and* $\mathcal{Q} : \mathbf{E} \longrightarrow \mathbf{B}$ *is a functor. An* **E**-*morphism* $f : Y \longrightarrow X$ *is* **cocartesian** *(or* **opcartesian***) if, given any map* $g : Y \longrightarrow Z$ *such that the* $\mathcal{P}$*-image of* g *factors through the* $\mathcal{P}$*-image of* f *(see diagram below), then* *there exists uniquely, up to isomorphism, a* $\mathbf{E}$*-map* $k : X \longrightarrow Z$ *that factors* g *through* f *and whose* $\mathcal{P}$*-image completes the factorization in* $\mathbf{B}$.

$$\begin{tikzcd}
Y \arrow[rr, "g"] \arrow[dr, "f"'] \arrow[dd, dotted, "Q"'] & & Z \arrow[dd, dotted, "Q"] & \\
& X \arrow[ur, dotted, "\exists! k"'] \arrow[dd, dotted, "Q"] & & (\text{in } \mathbf{E}) \\
K \arrow[rr, "\gamma"] \arrow[dr, "\alpha"'] & & J & \\
& I \arrow[ur, "\zeta"'] & & (\text{in } \mathbf{B})
\end{tikzcd}$$

A fibration's preimage of an index set I constitutes the collection of objects that are indexed by I. This collection can be extended to a category by including the $\mathbf{E}$-morphisms that map to id_I. Designated as E_I, it is called the *fiber over I.* All of a fiber's morphisms are described as *vertical.* With a bit of reflection, two properties of vertical morphisms become apparent from the cartesian map definition: (1) every vertical morphism is an isomorphism and (2) cartesian liftings, if any exist, are all isomorphic within factorizations with vertical morphisms.

We are finally prepared for the principal definitions where we distinguish the major brands of fibrations according to their lifting mechanisms:

Definition 3.4.3 *A functor* $\mathcal{P} : \mathbf{E} \longrightarrow \mathbf{B}$ *is a* **fibration** *if for all pairs* (α, X) *where* $\alpha : K \longrightarrow I$ *is a* $\mathbf{B}$*-map and* X *is in* E_I*,* α *can be lifted to a cartesian map* $f_{\alpha,X} : Y \longrightarrow X$ *for some object* Y*:*

$$\begin{tikzcd}
Y \arrow[rr, dotted, "f_{\alpha,X}"] & {} \arrow[d, dotted, "\mathcal{P}"] & X \\
K \arrow[rr, "\alpha"'] & {} & I
\end{tikzcd}$$

Definition 3.4.4 *A functor $\mathcal{Q} : \mathbf{E} \longrightarrow \mathbf{B}$ is a* **cofibration** *(or* **opfibration***) if for all pairs (α, X) where $\alpha : I \longrightarrow K$ is a $\mathbf{B}$-map and X is in E_I, α can be lifted to a cartesian map $f_{\alpha,X} : X \longrightarrow Y$ for some object Y:*

$$\begin{tikzcd}
X \arrow[rr, dotted, "f_{\alpha,X}"] & {} \arrow[d, dotted, "\mathcal{Q}"] & Y \\
I \arrow[rr, "\alpha"'] & {} & K
\end{tikzcd}$$

Definition 3.4.5 *A functor that is both a fibration and a cofibration is termed a* **bifibration***.*

As we move on, we note that both of our examples are quickly recognized to be bifibrations. Hereafter, we will use the words "fibration" and "cartesian" generically except when we contextually specify a particular lifting or when a dual property does not hold.

3.5 Cleaving and Splitting

Clearly the possibility of certain fibrations having a choice of cartesian liftings for a particular pair (α, X) is plausible, even though all the choices are isomorphic to each other. This does happen, so an explicit choice is often made to facilitate manipulation and calculation with cartesian morphisms. It is precisely the Axiom of Choice that allows this apparent simplification to be made. The choice is usually appended to the specification of a fibration:

Definition 3.5.1 *For a fibration, a choice function for selecting a cartesian arrow for every pair* (α, X) *is called a* **cleavage**. *A fibration specified with a cleavage is called a* **cloven fibration**.

A cleavage allows the definition of a functor $\mathcal{F}$ from $\mathbf{B}^{op}$ to **Cat**, i.e. an indexed category, in the following way. First, each object $I \in \mathbf{B}$ can be mapped to the fiber category E_I. Next, a functor must be assigned to each index set map $\alpha : K \longrightarrow I$. To carry it out with clarity, we modify the notation for the cartesian lifting arising from the cleavage so that the cartesian arrow is explicitly annotated:

$$\begin{tikzcd}
\alpha^* X \arrow[rr, dotted, "\alpha^X"] & {} & X \\
& & \\
K \arrow[rr, "\alpha"'] & {} \arrow[uu, dotted, "\mathcal{F}"'] & I
\end{tikzcd}$$

The lifting diagram indicates that α is to be assigned by $\mathcal{F}$ to an functor $\alpha^* : \mathbf{E}_I \longrightarrow \mathbf{E}_K$ "induced" by the collection of chosen cartesian maps $\{\alpha^X\}_{X \in E_I}$. The object-mapping component of α^* maps the codomain X of the cartesian map α^X that was selected by the cleavage for the pair (α, X) to its domain, now represented as $\alpha^* X$. The morphism-mapping component is defined in the diagram below by the universal property of cartesian maps.

$$\begin{tikzcd}
\alpha^*Y \arrow[rr, "\alpha^Y"] \arrow[dr, dotted, "\alpha^*f"'] & & Y \arrow[dr, "f \in \mathbf{E}_I"] & \\
& \alpha^*X \arrow[rr, "\alpha^X"] & & X \\
& {} \arrow[u, dotted, "\mathcal{F}"] & & \\
K \arrow[rr, "\alpha"] \arrow[dr, "id_K"'] & & I \arrow[dr, "id_I"] & \\
& K \arrow[rr, "\alpha"] & & I
\end{tikzcd}$$

We recall that α^*f is determined only up to vertical isomorphism, which leads to an easy demonstration of the pseudo-functoriality of $\mathcal{F}$: $(\beta \circ \alpha)^* \cong \alpha^* \circ \beta^*$. In later chapters our work will focus on particular fibrations for which $(\beta \circ \alpha)^* = \alpha^* \circ \beta^*$, making derivations and computations non-ambiguous. This special type of fibration is called a *split fibration* and its cleavage is referred to as a *splitting*. Similarly, an indexed category that is a true functor is termed a *split indexed category*.

In the particular case of a cofibration, it should be clear that this construction can be imitated to produce a *covariant* form of an indexed category.

3.6 Indexed Categories to Fibrations

The foregoing construction of an indexed category from a fibration motivates us to look at the converse: the well-known *Grothendieck construction* of a cloven fibration from a given indexed category $\mathcal{F} : \mathbf{B}^{op} \longrightarrow \mathbf{Cat}$. The derived total category $\mathbf{E}$ is defined as follows:

objects: pairs (I, X) where $I \in \mathbf{B}$ and $X \in \mathcal{F}(I)$,

morphisms: $(I, X) \longrightarrow (J, Y)$ is a pair (α, f) where $\alpha : I \longrightarrow J$ and $f = \alpha^X : X \longrightarrow \mathcal{F}(\alpha)(Y)$.

composition: $(\beta, g) \circ (\alpha, f) = (\beta \circ \alpha, \mathcal{F}(\alpha)(g) \circ f)$

A morphism in $\mathbf{E}$ represents the "amount of miss" when the induced functor $\mathcal{F}(\alpha)$ is applied to Y to "shoot" at X, while composition "adds up" the misses. The first projection of the pair-objects and pair-maps of $\mathbf{E}$ provides the required fibration. A splitting is straightforwardly found to be

$$\begin{tikzcd}
(I, \mathcal{F}(\alpha)(Y)) \arrow[rr, dotted, "{(\alpha, id_{\mathcal{F}(\alpha)(Y)})}"] & {} & (J, Y) \\
I \arrow[rr, "\alpha"'] & {} \arrow[u, dotted] & J
\end{tikzcd}$$

The Grothendieck construction is also relevant to our later fibrational work concerning contexts. We can consider the object (I, X) as "the object X in the context I" and the fibration as a means for extracting context. This idea will reappear in Chapter 5.

3.7 Categories of Fibrations

Fibrations serve well in their own right as objects of categories. A *morphism between fibrations* $\mathcal{P}$ and $\mathcal{Q}$ is defined as a commuting square of functors

$$\begin{tikzcd}
\mathbf{E} \arrow[r, "H"] \arrow[d, "\mathcal{P}"'] & \mathbf{E}' \arrow[d, "\mathcal{Q}"] \\
\mathbf{B} \arrow[r, "K"'] & \mathbf{B}'
\end{tikzcd}$$

where H maps $\mathcal{P}$-fibers into $\mathcal{Q}$-fibers while preserving cartesian maps. Composition is defined by the abutting of such squares. We will narrow our view to letting $\mathbf{B} = \mathbf{B}'$ and K be the identity functor and thereby form the category ($\mathbf{SplFib}(\mathbf{B})$) $\mathbf{Fib}(\mathbf{B})$ of (split) fibrations over a fixed base category $\mathbf{B}$:

$$\begin{tikzcd}
\mathbf{E} \arrow[rr, "H"] \arrow[dr, "\mathcal{P}"'] & & \mathbf{E}' \arrow[dl, "\mathcal{Q}"] \\
& \mathbf{B} &
\end{tikzcd}$$

In this situation several common terms have been historically attached to H: a *fibration morphism*, a *cartesian functor*, or a *morphism over* $\mathbf{B}$.

Over on the indexed category side, we can define a morphism between indexed categories $\mathcal{F}, \mathcal{G} : \mathbf{B}^{op} \longrightarrow \mathbf{Cat}$ as simply any natural transformation. The indexed categories, and also just the split ones, respectively form their associated functor categories. From the constructions above and additional perfunctory detail, we have the following expected result:

Theorem 3.7.1 *The category of (split) fibrations over* $\mathbf{B}$ *is equivalent to the category of categories (split-)indexed by* $\mathbf{B}$.

3.8 Parametrization means Fiberization

Finally, there is a technical aspect of fibrations that allows capturing of not only context but also datatype parameterization. Datatypes are properly formed when they arise as left adjoints to underlying functors. In the fibration settings that we will encounter, the base category often supplies both context values and parameter values to the data structure terms sitting up in the fibers inside the total category. Thus the datatype-defining adjunctions must not be allowed to vary over parameter values. Rather, exactly one adjunction should be in effect for each parameter value. This is expressed by the following definition:

Definition 3.8.1 *Let $\mathcal{P} : \mathbf{E} \longrightarrow \mathbf{B}$ and $\mathcal{Q} : \mathbf{E}' \longrightarrow \mathbf{B}$ be fibrations. A* **fibered adjunction** *from $\mathcal{P}$ to $\mathcal{Q}$ is a pair of cartesian functors $F : \mathbf{E} \longrightarrow \mathbf{E}'$ and $G : \mathbf{E}' \longrightarrow \mathbf{E}$ that are adjoint by means of a vertical unit and counit:*

$$\begin{tikzcd}
\mathbf{E} \arrow[rr, shift right, "G"'] \arrow[rr, phantom, "\perp"] \arrow[dr, "\mathcal{P}"'] & & \mathbf{E}' \arrow[ll, shift right, "F"'] \arrow[dl, "\mathcal{Q}"] \\
& \mathbf{B} &
\end{tikzcd}$$

A vertical unit (or counit) means the particular natural transformation's components are vertical morphisms. In fact, the definition is slightly redundant: it is easy to derive from the adjunction's triangle identities that the unit is vertical if and only if the counit is vertical. The "fiberization" of the adjunction appears in the restrictions $F|_I : \mathbf{E}_I \longrightarrow \mathbf{E}'_I$ and $G|_I : \mathbf{E}'_I \longrightarrow \mathbf{E}_I$. Because the unit and counit are vertical, the triangle identities hold within the corresponding fibers over I and thereby yield an adjunction $F|_I \dashv G|_I$ over each $I \in \mathbf{B}$.

It is almost with reluctance that we now scurry on to the development of functorial strength, as the subject of fibrations has recently rapidly expanded in its general importance in erecting mathematics. So, as a closing aside, the author recommends to the reader the manifesto of Benabou [4] for gaining appreciation for fibrations' possible destiny in replacing set theory as the starting point for mathematics.

Chapter 4

X–strength

This section's purpose is to cast a foundational description of strength and then mold it into an elementary description that will be more intuitively recognizable. The molding will be performed quite independently of cartesian closedness.

4.1 Tensored Beginnings

A cartesian closed category's correspondence between explicit and internal strengths as outlined in the introduction chapter was demonstrated and generalized by Kock [30] to any $\mathbf{V}$-tensored category $\mathbf{Y}$ defined roughly as:

(i) $\mathbf{V}$ is a symmetric monoidal closed category,

(ii) $\mathbf{Y}$ is an arbitrary underlying category,

(iii) $\otimes : \mathbf{Y} \times \mathbf{V} \longrightarrow \mathbf{Y}$ is a monoidal structure-preserving bifunctor, or *tensor*, that provides a $\mathbf{V}$-action on $\mathbf{Y}$ (e.g. we would expect for a monoidal operation $\oplus$ in $\mathbf{V}$ to satisfy $Y \otimes (V_1 \oplus V_2) \cong (Y \otimes V_1) \otimes V_2$,

(iv) the tensor action is coherent with respect to the symmetric monoidal closed structure of $\mathbf{V}$, i.e. any monoidal structure-related diagram involving only (1) the defining morphisms of the SMCC structure of $\mathbf{V}$ and (2) the tensor actions by them upon the morphisms of $\mathbf{Y}$ will commute.

In this $\mathbf{V}$-tensored category setting, Kock showed that a correspondence between internal (implicit) functorial strengths of the form

$$\mathit{hom}(Y_1, Y_2) \longrightarrow \mathit{hom}(F(Y_1), F(Y_2))$$

and tensorial (explicit) strengths of the form

$$F(Y) \otimes V \longrightarrow F(Y \otimes V)$$

exists in two cases: (1) F is an ordinary functor between underlying categories and (2) F is a $\mathbf{V}$-strong functor between $\mathbf{V}$-tensored categories in which its tensorial strength satisfies special coherence conditions. Both correspondences are bijective and are natural in all parameter arguments.

Here the function *hom* yields the *internal hom object* in $\mathbf{V}$ that is pre-assigned (by definition of $\mathbf{V}$-tensored categories) to each pair of $\mathbf{Y}$-objects. The internal hom object is a direct generalization of the more familiar exponential object defined for each pair of objects within a cartesian closed category. In that example, a cartesian closed category simultaneously plays both roles of $\mathbf{Y}$ and $\mathbf{V}$.

A $\mathbf{V}$-strong functor is a functor between underlying categories that additionally exhibits its functoriality in an internal form within $\mathbf{V}$, as required in the two diagrams below. The strong functor's role is played by its internal strengths, shown as s morphisms. Intuitively, the first diagram states a strong functor maps the specified "identity" element i_Y of the "exponential" of Y with itself to the "identity" element $i_{F(Y)}$ of the "exponential" of the F-image of Y with itself. The $\mathbf{V}$-object I is the monoidal unit under the $\oplus$ operation. The second diagram makes use of the morphisms m that exist, again by accepted definition, to compose two internal hom objects together. That diagram expresses the preservation of internal composition by the functor.

$$\begin{tikzcd}
& I \arrow[dl, "i_Y"'] \arrow[dr, "i_{F(Y)}"] & \\
\mathit{hom}(Y, Y) \arrow[rr, "s"] & & \mathit{hom}(F(Y), F(Y))
\end{tikzcd}$$

$$\begin{tikzcd}
hom(Y_1, Y_2) \oplus hom(Y_2, Y_3) \arrow[r, "m"] \arrow[d, "s \oplus s"'] & hom(Y_1, Y_3) \arrow[d, "s"] \\
hom(F(Y_1), F(Y_2)) \oplus hom(F(Y_2), F(Y_3)) \arrow[r, "m"'] & hom(F(Y_1), F(Y_3))
\end{tikzcd}$$

The specific coherence conditions demanded of tensorial strengths to yield the V-strong version of the correspondence are the *unit condition* and the *associativity condition*, respectively shown in the next two diagrams below.

$$\begin{tikzcd}
F(Y) \otimes I \arrow[rr, "t"] \arrow[dr, "r_Y"'] & & F(Y \otimes I) \arrow[dl, "F(r_Y)"] \\
& F(Y) &
\end{tikzcd}$$

$$\begin{tikzcd}
(F(Y) \otimes V_1) \otimes V_2 \arrow[r, "t \otimes id"] \arrow[d, "ass"'] & F(Y \otimes V_1) \otimes V_2 \arrow[r, "t"] & F((Y \otimes V_1) \otimes V_2) \arrow[d, "F(ass)"] \\
F(Y) \otimes (V_1 \oplus V_2) \arrow[rr, "t"'] & & F(Y \otimes (V_1 \oplus V_2))
\end{tikzcd}$$

In them the instances of tensorial stengths are simply designated by t. By the **V**-category definition, the "right-side identity" morphisms, r_Y and $r_{F(Y)}$, are specified to be isomorphisms. The associativity isomorphism, also provided directly by the definition, is shown as ass. The associativity essentially specifies the mutual coherence of the two monoidal operations $\oplus$ and $\otimes$.

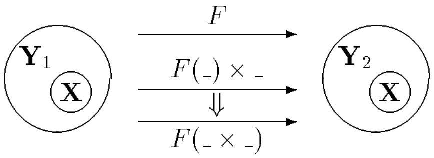
Kock's results suggest the redundancy of exponentials and the capability for translating functorial operations into first-order terms in many useful categories. We therefore take aim at Kock's second correspondence (the more powerful "**V**-strong" case) and proceed to erect a tensored category for which **V** need not be closed, i.e. exponentials are not required, and yet allow the functors to be strong in the explicit sense. A series of intuitions, described below, leads us to a reasonable starting point in generality.

First, we particularly wish to use ordinary categorical products as tensor products because categorical products of morphisms are computationally easy to manipulate. A possible notion of functorial strength that satisfies the coherence diagrams above and uses cartesian products, say in the general form of

$$F(A) \times X \longrightarrow F(A \times X) \quad ,$$

would immediately suggest that × play the role of the tensor. Moreover, this notion hints that A should exist as an object in a category $\mathbf{Y}$ of parameter and functor-built datatypes and that X should come from a cartesian sub-category $\mathbf{X}$ of $\mathbf{Y}$. $\mathbf{X}$ would represent the context datatypes that would serve as our monoidal but not-necessarily-closed category $\mathbf{V}$. In this way computation is kept within a consolidated setting and would allow for earlier building (via declaration) of datatypes that might be later used as parameters or context in building more complex datatypes. That is, the context datatypes would occur together with the parameter/functor datatypes within the category $\mathbf{Y}$.

To generalize this idea in a fashion akin to the $\mathbf{V}$-categories, we fix the cartesian category of context datatypes $\mathbf{X}$ and consider all datatype categories $\mathbf{Y}$ into which a tensor-preserving (i.e. cartesian product preserving) embedding of $\mathbf{X}$ exists. We describe any such $\mathbf{Y}$ as "$\mathbf{X}$-strong". A first-order strong functor could then be specified in two pieces: (1) it would be an ordinary functor between two $\mathbf{X}$-strong categories, say $\mathbf{Y}_1$ and $\mathbf{Y}_2$, that have the fixed context category $\mathbf{X}$ as a sub-category, and (2) a collection of strength morphisms of the form shown above that form a natural transformation. We have the following visualization:



With these intuitions in mind, we formally begin in the next section with a **V**-tensored category **Y** with a **V**–action $\oslash$ on it given by a monoidal structure-preserving functor from **V** into *Endo*(**Y**). That is, for each $V \in \mathbf{V}$, we have the endofunctor $\_ \oslash V$ that satisfies properties like the one mentioned in the definition of **V**-tensored categories.

Our practical interest there will then narrow to the special case of having **V**'s tensor action be realized as a cartesian product. We will thereafter designate such a category **V** as **X**. Furthermore, we will demand that **Y** have its own monoidal operation, viz. a cartesian product, and that the action, now an **X**–action, be representable by **Y**'s product.

4.2 A Tensored View of X-strong Categories

If **X** is a cartesian category (i.e. has a binary product and a terminal object) an **X**-*strong category* **Y** is a cartesian category **Y** with a bifunctor, called an **X**–*action*,

$$\_ \oslash \_ \quad : \quad \mathbf{Y} \times \mathbf{X} \longrightarrow \mathbf{Y}$$

that is compatible with the coherent natural isomorphisms

- *ass* $\;:\; Y \oslash (X_1 \times X_2) \longrightarrow (Y \oslash X_1) \oslash X_2,$
- *rid* $\;:\; (Y \oslash 1) \longrightarrow Y,$
- *rep* $\;:\; Y \oslash X \longrightarrow Y \times (1 \oslash X).$

The first two natural isomorphisms are the standard ones and must satisfy the coherence conditions expected for any monoidal action. The last allows the action to be represented by the product of the category **Y** on which **X** acts.

General coherence theorems — and well-accepted folklore — that apply to this setting and motivate the three natural isomorphisms above were established in the early 1970s by several workers. One pertinent reference that the reader may consult are the coherence theorems for natural transformations within closed **V**-categories developed by Kelly and Mac Lane [28]. It is their generalized method of showing coherence with only the basic definitions of **V**-category and **V**-functor, i.e. independently of using specific properties of closure (exponentials), that permits adapting and simplifying the standard tensored category coherence diagrams to our non-closed cartesian settings. The resulting simplification includes the following set of four equations that we require:

- $\mathit{ass}\,;\mathit{rid} = \mathit{id} \oslash p_0 \quad : \quad Y \oslash (X \times 1) \longrightarrow Y \oslash X,$
- $\mathit{rid} = \mathit{rep}\,; p_0 \quad : \quad (Y \oslash X) \oslash 1 \longrightarrow Y \oslash X,$
- $\mathit{ass}\,;(\mathit{rid} \oslash \mathit{id}) = \mathit{id} \oslash p_1 \quad : \quad Y \oslash (1 \times X) \longrightarrow Y \oslash X,$
- $\mathit{rid} \oslash \mathit{id} = \mathit{rep}\,; p_1 \quad : \quad (1 \oslash 1) \oslash X \longrightarrow 1 \oslash X.$

Notice that they principally ensure that the map *rid* behaves correctly, a property that is critically depended upon in several later proofs.

Proposition 4.2.1 *A cartesian category* **Y** *with an* **X***–action is* **X***–strong if and only if there is a cartesian functor*

$$J : \mathbf{X} \longrightarrow \mathbf{Y}$$

and a natural isomorphism $r_{Y,X} : Y \oslash X \longrightarrow Y \times J(X)$.

Proof. If we are given J and the natural isomorphism r then by setting

$$\mathit{rid} = r_{Y,1}\,; p_0$$

$$\mathit{ass} = r_{Y,X_1 \times X_2}\,; \langle\langle p_0, p_1\,; J(p_0)\rangle, p_1\,; J(p_1)\rangle\,; r^{-1}_{Y \times J(X_1),X_2}\,; r^{-1}_{Y,X_1} \oslash \mathit{id}$$

$$\mathit{rep} = r_{Y,X}\,; \mathit{id} \times \langle !, \mathit{id}\rangle\,; \mathit{id} \times r^{-1}_{1,X}$$

the (simplified) coherence requirements are straightforward to verify.

Conversely, given an **X**–action we may choose $J(X) = 1 \oslash X$. Clearly J is a functor and r is definable as *rep*. It remains to show J is cartesian. Because $\mathit{rid} : 1 \oslash 1 \longrightarrow 1$

is an isomorphism it follows that $1 \oslash 1$ is final. To show that products are preserved we consider

$$\begin{tikzcd}
1 \oslash A & 1 \oslash (A \times B) \arrow[l, "1 \oslash p_0"'] \arrow[r, "1 \oslash p_1"] & 1 \oslash B
\end{tikzcd}$$

and show that it is naturally isomorphic to

$$\begin{tikzcd}
1 \oslash A & 1 \oslash A \times 1 \oslash B \arrow[l, "p_0"'] \arrow[r, "p_1"] & 1 \oslash B
\end{tikzcd}$$

This is accomplished with the following two diagrams:

$$\begin{tikzcd}
1 \oslash (X \times Y) \arrow[r, "id \oslash (id \times !)"'] \arrow[d, "ass"'] & 1 \oslash (X \times 1) \arrow[d, "ass"'] \arrow[dr, "id \oslash p_0"] & \\
(1 \oslash X) \oslash Y \arrow[r, "(id \oslash id) \oslash !"'] \arrow[d, "rep"'] & (1 \oslash X) \oslash 1 \arrow[r, "rid"'] \arrow[d, "rep"'] & 1 \oslash X \\
1 \oslash X \times 1 \oslash Y \arrow[r, "id \times id \oslash !"'] & 1 \oslash X \times 1 \oslash 1 \arrow[ur, "p_0"'] &
\end{tikzcd}$$

and

$$\begin{tikzcd}
1 \oslash (X \times Y) \arrow[r, "id \oslash (! \times id)"'] \arrow[d, "ass"'] & 1 \oslash (1 \times Y) \arrow[d, "ass"'] \arrow[dr, "id \oslash p_1"] & \\
(1 \oslash X) \oslash Y \arrow[r, "(id \oslash !) \oslash id"'] \arrow[d, "rep"'] & (1 \oslash 1) \oslash Y \arrow[r, "rid \oslash id"'] \arrow[d, "rep"'] & 1 \oslash Y \\
1 \oslash X \times 1 \oslash Y \arrow[r, "id \oslash ! \times id"'] & 1 \oslash 1 \times 1 \oslash Y \arrow[ur, "p_1"'] &
\end{tikzcd}$$

in which the four identities are used in the triangles.

□

All the $\mathbf{X}$–strong categories of interest can be regarded as having a cartesian embedding J of $\mathbf{X}$ into it. The $\mathbf{X}$–action is, up to isomorphism, simply given by the product with the image of the objects of $\mathbf{X}$ under this embedding. This observation motivates our dropping the action notation $A \oslash X$ in favor of $A \times J(X)$ or just $A \times X$.

$\mathbf{X}$–*strong functors*, or simply *strong functors*, are functors between $\mathbf{X}$–strong categories equipped with a natural transformation

$$\theta^F_{Y,X} : F(Y) \oslash X \longrightarrow F(Y \oslash X),$$

called an $\mathbf{X}$–*strength*, or simply a *strength*, such that

$$\begin{tikzcd}
(F(A) \oslash B) \oslash C \arrow[d, "\theta^F_{A,B} \oslash id_C"'] & F(A) \oslash (B \times C) \arrow[l, "\mathrm{ass}"'] \arrow[dd, "\theta^F_{A,B \times C}"] \\
F(A \oslash B) \oslash C \arrow[d, "\theta^F_{A \oslash B,C}"'] & \\
F((A \oslash B) \oslash C) & F(A \oslash (B \times C)) \arrow[l, "F(\mathrm{ass})"]
\end{tikzcd}$$

and

$$\begin{tikzcd}
F(A) \oslash 1 \arrow[r, "\theta^F_{A,1}"] \arrow[dr, "rid"'] & F(A \oslash 1) \arrow[d, "F(rid)"] \\
& F(A)
\end{tikzcd}$$

commute. These diagrams are clearly recognizable adaptations of the $\mathbf{V}$-functor unit and associativity conditions to our situation.

To maintain our intuition, we consider again the canonical example of strength first presented in Chapter 1, viz. the strength of the *List* functor relative to the ordinary cartesian product (i.e. $\oslash = \times$). We should recall from our general previous experience from lists that *List* functoriality is exhibited by performing *map* operations on morphisms in the standard functional programming sense. Thus for $f : A \rightarrow B$, we have $\mathit{List}(f) = \mathit{map}(f) : \mathit{List}(A) \rightarrow \mathit{List}(B)$ which operates on lists of A-elements to create lists of B-elements by applying f list element-wise. And, with $g : B \rightarrow C$, we are familiar with the fact that $\mathit{List}(f\,;g) = \mathit{List}(f)\,; \mathit{List}(g)$. The following diagram chases of elements concretely supports the intuition that *List* is strong in the above sense:

$$\begin{tikzcd}
\langle\langle[a_1, a_2, \ldots, a_n], b\rangle, c\rangle \arrow[d, "\theta^{\mathit{List}}_{A,B} \oslash id_C"'] & \langle[a_1, a_2, \ldots, n], \langle b, c\rangle\rangle \arrow[l, "\mathit{ass}"'] \arrow[dd, "\theta^{\mathit{List}}_{A,B\times C}"] \\
\langle[\langle a_1, b\rangle, \langle a_2, b\rangle, \ldots, \langle a_n, b\rangle], c\rangle \arrow[d, "\theta^{\mathit{List}}_{A\oslash B,C}"'] & \\
[\langle\langle a_1, b\rangle, c\rangle, \langle\langle a_2, b\rangle, c\rangle, \ldots, \langle\langle a_n, b\rangle, c\rangle] & {[\langle a_1, \langle b, c\rangle\rangle, \langle a_2, \langle b, c\rangle\rangle, \ldots, \langle a_n, \langle b, c\rangle\rangle]} \arrow[l, "\mathit{List}(\mathit{ass})"]
\end{tikzcd}$$

$$\begin{tikzcd}
\langle[a_1, a_2, \ldots, a_n], e_0\rangle \arrow[r, "\theta^{\mathit{List}}_{A,1}"] \arrow[dr, "p_0"'] & {[\langle a_1, e_0\rangle, \langle a_2, e_0\rangle, \ldots, \langle a_n, e_0\rangle]} \arrow[d, "\mathit{List}(p_0)"] \\
 & {[a_1, a_2, \ldots, a_n]}
\end{tikzcd}$$

The sole element of the datatype 1 is designated e_0 in the last diagram. That diagram also exposes the right-identity *rid* morphism to be a first projection.

A strong functor, as used by Moggi for cartesian closed categories, is simply an $\mathbf{X}$-strong endofunctor on $\mathbf{X}$ itself with the natural $\mathbf{X}$–action given by product. It is quite reasonable to discuss “strong functors” for non-cartesian closed categories: the list datatype on a distributive category [8] is a strong functor even though its setting is not closed.

An $\mathbf{X}$-*strong natural transformation* between strong functors $F, G : \mathbf{Y}_1 \longrightarrow \mathbf{Y}_2$ is a parameterized natural transformation

$$\alpha_{A,X} : F(A) \oslash X \longrightarrow G(A \oslash X) : \mathbf{Y}_1 \times \mathbf{X} \longrightarrow \mathbf{Y}_2$$

such that

$$\begin{tikzcd}
(F(A) \oslash X) \oslash Y \arrow[r, "\theta^F_{A,X} \oslash id_Y"] \arrow[d, "\alpha_{A,X} \oslash id_Y"'] & F(A \oslash X) \oslash Y \arrow[d, "\alpha_{A \oslash X,Y}"] \\
G(A \oslash X) \oslash Y \arrow[r, "\theta^G_{A \oslash X,Y}"'] & G((A \oslash X) \oslash Y)
\end{tikzcd}$$

commutes.

Again we regain our intuition with our canonical examples. Consider the strong natural transformation

$$\mathit{flatten}_{A.X} : \mathit{Btree}(A) \times X \longrightarrow \mathit{List}(A \times X)$$

that simultaneously flattens a tree by forming the list of its leaves and merges the context with the leaf values. An instance of the element diagram chase for illustrating that this natural transformation is strong follows below:

$$\begin{tikzcd}
\langle\langle bnode(bnode(bleaf(a_1), bleaf(a_2)), bleaf(a_3)), x\rangle, y\rangle \arrow[d, "flatten_{A,X} \times id_Y"'] \arrow[dr, "\theta^{Btree}_{A,X} \times id_Y"] & \\
& \langle bnode(bnode(bleaf(\langle a_1, x\rangle), bleaf(\langle a_2, x\rangle)), bleaf(\langle a_3, x\rangle)), y\rangle \arrow[dd, "flatten_{A\times X,Y}"] \\
\langle[\langle a_1, x\rangle, \langle a_2, x\rangle, \langle a_3, x\rangle], y\rangle \arrow[dr, "\theta^{List}_{A\times X,Y}"'] & \\
& [\langle\langle a_1, x\rangle, y\rangle, \langle\langle a_2, x\rangle, y\rangle, \langle\langle a_3, x\rangle, y\rangle]
\end{tikzcd}$$

Let $\nu : X \xrightarrow{\cong} X_1 \times X_2$ be any decomposition of X into a product. Then the composition $\alpha_{A,X} \bullet \beta_{A,X}$ of two strong transformations $\alpha_{A,X}$ and $\beta_{A,X}$ is defined by the following sequence of maps:

$$\begin{tikzcd}
F(A) \oslash X \arrow[r, "id \oslash \nu"] \arrow[dd, "(\alpha \bullet \beta)_{A,X}"'] & F(A) \oslash (X_1 \times X_2) \arrow[r, "ass"] & (F(A) \oslash X_1) \oslash X_2 \arrow[d, "\alpha_{A,X_1} \oslash id"'] \\
& & G(A \oslash X_1) \oslash X_2 \arrow[d, "\beta_{A \oslash X_1, X_2}"'] \\
H(A \oslash X) & H(A \oslash (X_1 \times X_2)) \arrow[l, "H(id \oslash \nu^{-1})"'] & H((A \oslash X_1) \oslash X_2) \arrow[l, "H(ass^{-1})"']
\end{tikzcd}$$

The composition does not depend on the particular decomposition of X. That is, all decompositions result in the same parameterized composition. Furthermore, this composition straightforwardly commutes with the strengths in the required manner.

By noting that (1) the definition of strength implies $\theta^F_{A,1}$ is an isomorphism and (2) the definition of strong natural transformation trivially implies θ^F is strong and commutative with all other strong transformations, we see θ^F is actually the identity strong transformation on F under this composition. This establishes that **X**–strong categories, **X**–strong functors and **X**–strong transformations form a 2-category **Strong**(**X**).

4.3 A Cartesian View of X-strong Categories

This formulation of an **X**–strong transformation reveals the structural effect of making transformations strong. However, it fails to promote easy manipulation of these transformations. We offer instead a simpler yet equivalent view of **Strong**(**X**).

Notice that an ordinary transformation α gives rise to a related strong transformation provided

$$\begin{tikzcd}
F(A) \oslash X \arrow[r, "\theta^F_{A,X}"] \arrow[d, "\alpha_A \oslash id_X"'] & F(A \oslash X) \arrow[d, "\alpha_{A \oslash X}"] \\
G(A) \oslash X \arrow[r, "\theta^G_{A,X}"'] & G(A \oslash X)
\end{tikzcd}$$

commutes. The associated parameterized transformation is

$$\alpha_{A,X} \quad = \quad \alpha_A \oslash id_X \,; \theta^G_{A,X} \quad = \quad \theta^F_{A,X} \,; \alpha_{A \oslash X},$$

which is clearly strong. Conversely, a strong transformation is related to an ordinary natural transformation by

$$F(A) \xrightarrow{rid^{-1}} F(A) \oslash 1 \xrightarrow{\alpha_{A,1}} G(A \oslash 1) \xrightarrow{G(rid)} G(A)$$

and these associations are inverse. Thus we have reorganized our view of $\mathbf{X}$–strong categories into a more accessible form:

Theorem 4.3.1 *The 2-category* **Strong**$(\mathbf{X})$ *is isomorphic to the 2-category having as*

0-cells: *cartesian functors* $J : \mathbf{X} \longrightarrow \mathbf{Y}$,

1-cells: *pairs* (F, θ^F) *where* F *is a functor* $F : \mathbf{Y} \longrightarrow \mathbf{Y}'$ *and* θ^F *is a strength:*

$$\theta^F_{Y,X} : F(Y) \times J'(X) \longrightarrow F(Y \times J(X)),$$

satisfying

$$\begin{tikzcd}
(F(A) \times J'(X_1)) \times J'(X_2) \arrow[d, "\theta^F_{A,X_1} \times id"'] & F(A) \times J'(X_1 \times X_2) \arrow[l, "\mathrm{ass}"'] \arrow[dd, "\theta^F_{A,X_1 \times X_2}"] \\
F(A \times J(X_1)) \times J'(X_2) \arrow[d, "\theta^F_{A \times J(X_1),X_2}"'] & \\
F((A \times J(X_1)) \times J(X_2)) & F(A \times J(X_1 \times X_2)) \arrow[l, "F(\mathrm{ass})"]
\end{tikzcd}$$

and

$$\begin{tikzcd}
F(A) \times J'(X) \arrow[rr, "\theta^F_{A,X}"] \arrow[dr, "p_0"'] & & F(A \times J(X)) \arrow[dl, "F(p_0)"] \\
& F(A) &
\end{tikzcd}$$

2-cells: *natural transformations* $\alpha : F \longrightarrow G$ *which satisfy :*

$$\begin{tikzcd}
F(A) \times J'(X) \arrow[r, "\theta^F_{A,X}"] \arrow[d, "\alpha_A \times id"'] & F(A \times J(X)) \arrow[d, "\alpha_{A \times J(X)}"] \\
G(A) \times J'(X) \arrow[r, "\theta^G_{A,X}"'] & G(A \times J(X))
\end{tikzcd}$$

Therefore an **X**–strong category may be regarded without loss as a cartesian category with a cartesian functor from **X**, an **X**-strong functor as a functor with a strength, and an **X**–strong transformation as a transformation which commutes with the strengths as shown above. We shall henceforth identify $\oslash$ and $\times$ and avoid the use of the relatively unfamiliar coherence maps for $\_ \oslash \_$.

Chapter 5

Strength to Fibrations

A strong functor may equivalently be regarded as a morphism of fibrations [14] in which $\mathbf{X}$ is viewed as a category of contexts. Technically this is given by an embedding of the 2–category $\mathbf{Strong}(\mathbf{X})$ into the 2–category of split fibrations over $\mathbf{X}$, $\mathbf{SplFib}(\mathbf{X})$. The latter can, of course, be equivalently viewed as $\mathbf{X}$–indexed categories. This chapter derives the embedding; the next chapter attaches the importance of this embedding to our fabrication methodology for datatypes.

5.1 Fibrations with Context

Given an $\mathbf{X}$–strong category $\mathbf{Y}$ we may form the category $P_{\mathbf{X}}(\mathbf{Y})$ to be be thought of as $\mathbf{Y}$–maps with context from $\mathbf{X}$:

Objects: pairs of objects (Y, X) where $Y \in \mathbf{Y}$ and $X \in \mathbf{X}$,

Maps: pairs of maps $(f, x) : (Y, X) \longrightarrow (Y', X')$ where $f : Y \times X \longrightarrow Y'$ in $\mathbf{Y}$ and $x : X \longrightarrow X'$ in $\mathbf{X}$,

Identities: $(p_0, id) : (Y, X) \longrightarrow (Y, X)$,

Composition: $(f, h)\,;(g, k) = (\langle f, p_1; h\rangle; g, h; k)$.

A map (f, h) uses h to propagate the context in the second component and f to compute new data values from both data values in Y and the available context:

$$(f, h) : (Y, X) \longrightarrow (Y', X') \text{ where } (y, x) \mapsto (f(y, x), h(x)).$$

Composing this map with $(g, k) : (Y', X') \longrightarrow (Y'', X'')$ gives:

$$(f, h) ; (g, k) : (Y, X) \longrightarrow (Y'', X'') \text{ where } (y, x) \mapsto (g(f(y, x), h(x)), k(h(x))).$$

Thus g receives a propagated context and the computed value. This can be viewed as the Kleisli composition for the comonad of the $\mathbf{X}$–action (see Section 5.4).

Aside, the objects (Y, X) of this category may be equivalently considered as second projections $p_1 : Y \times X \longrightarrow X$ and the maps as naturality squares. The category, seen as arrows-as-objects, can be viewed as a fibration sitting over $\mathbf{X}$, i.e.

$$\delta_{\mathbf{Y}} : P_{\mathbf{X}}(\mathbf{Y}) \longrightarrow \mathbf{X}$$

where $(f, x) \mapsto x$. The second projections behave as display maps [49], i.e. context extractors. The map which carries $\mathbf{Y}$ to the fibration $\delta_{\mathbf{Y}}$ is the object map of a 2–functor from strong categories to split fibrations over $\mathbf{X}$, denoted $P_{\mathbf{X}}(\_)$.

We are now at the starting point for discussing the correspondence of strength to fibrations attributed to Plotkin by Moggi. The remainder of this section is dedicated to proving this correspondence:

Theorem 5.1.1 *The 2–functor* $P_{\mathbf{X}}(\_) : \mathbf{Strong}(\mathbf{X}) \longrightarrow \mathbf{SplFib}(\mathbf{X})$ *is a full and faithful embedding.*

5.2 Fibration Morphisms to Strong Functors

Let $\mathbf{StrSplFib}(\mathbf{X})$ denote the 2–category of split fibrations of the form $\delta_{\mathbf{Y}}$. The 1-cells are *morphisms of fibrations* (functors $\tilde{F} : P_{\mathbf{X}}(\mathbf{Y}) \longrightarrow P_{\mathbf{X}}(\mathbf{Z})$ that commute with the fibration legs and preserve the canonical splitting on the nose). The 2-cells are *natural transformations of morphisms of fibrations* ($\tilde{\alpha} : \tilde{F}_1 \overset{\bullet}{\longrightarrow} \tilde{F}_2$ such that $\delta_{\mathbf{Y}} = \delta_{\mathbf{Z}} * \tilde{\alpha}$ (horizontal composite)). We will show that $P_{\mathbf{X}}(\_)$ factored through this full sub–2–category of **SplFib**(**X**) is an isomorphism.

First we assure ourselves that the functors δ are indeed fibrations that individually possess a splitting. The underlying cleavage is canonically defined by

$$(x, (Y, X')) \mapsto (p_0, x) \quad .$$

This is illustrated by the diagram

$$\begin{tikzcd}
(Y,X) \arrow[r, "{(p_0,x)}"] \arrow[d, dotted, "\delta"'] & (Y,X') \arrow[d, dotted, "\delta"'] \\
X \arrow[r, "x"'] & X'
\end{tikzcd}$$

The fact that this cleavage generates cartesian arrows follows directly from the definitions and parallels the reasoning that

$$\begin{tikzcd}
Y \times X \arrow[r, "id \times x"] \arrow[d, "p_1"'] \arrow[dr, phantom, "pb"] & Y \times X' \arrow[d, "p_1"'] \\
X \arrow[r, "x"'] & X'
\end{tikzcd}$$

is a pullback. Because the **X**–action canonically provides the products used in defining the maps in $P_{\mathbf{X}}(\mathbf{Y})$, the cleavage is closed to composition, i.e. it is a splitting.

Suppose $\tilde{F} : P_{\mathbf{X}}(\mathbf{Y}) \longrightarrow P_{\mathbf{X}}(\mathbf{Z})$ is a morphism of fibrations. On the objects it must take the simple form

$$\tilde{F}(A, X) = (F(A), X)$$

for some object function F due jointly to fixing the second coordinate and preserving the splitting on the nose, thereby allowing the definition of F to be independent of X. On maps it must take the less precise form

$$\tilde{F}(f,x) = (\tilde{F}_0(f,x), x)$$

since the fibration morphism preserves fibers over morphisms as well.

In the fiber over 1 we may recapture an ordinary functor $F \ : \ \mathbf{Y} \longrightarrow \mathbf{Z}$ from $\tilde{F}$ by setting for $x : Y \longrightarrow Y'$

$$F(x) = \langle id_{F(Y)}, !_{F(Y)} \rangle \,; \tilde{F}_0(p_0 \,; x \ , \ id_1) \quad .$$

To see that this is a functor note first that

$$F(id) = \langle id, ! \rangle \,; \tilde{F}_0(p_0, id_1) = \langle id, ! \rangle \,; p_0 = id$$

where $\tilde{F}_0(p_0, id_1) = p_0$ since $\tilde{F}$, by preserving the splitting, also preserves identities. For composition we exploit the immediate fact

$$\tilde{F}(p_0 \,; x \,; y \ , \ id_1) = \tilde{F}(p_0 \,; x \ , \ id_1) \,; \tilde{F}(p_0 \,; y \ , \ id_1)$$

so that

$$\begin{aligned} \langle id, ! \rangle \,; \tilde{F}_0(p_0 \,; x \,; y \ , \ id_1) &= \langle id, ! \rangle \,; \langle \tilde{F}_0(p_0 \,; x \ , \ id_1) \ , \ p_1 \,; id_1 \rangle \,; \tilde{F}_0(p_0 \,; y \ , \ id_1) \\ &= \langle id, ! \rangle \,; \tilde{F}_0(p_0 \,; x \ , \ id_1) \,; \langle id, ! \rangle \,; \tilde{F}_0(p_0 \,; y \ , \ id_1) \end{aligned}$$

follows, showing composition is preserved.

Next we shall show that this functor has a strength given by $\theta^F = \tilde{F}_0(id, !)$, the first component of the morphism

$$\tilde{F}(id, !) \ : \ (F(A), X) \longrightarrow (F(A \times X), 1) \quad .$$

An intuitive interpretation of this strength-generating morphism is that $\tilde{F}(id, !)$ builds a "datatype closure" in which environment has been encapsulated within data structure, leaving an empty residue environment.

θ^F's naturality in each argument derives easily from the commutativity of the following square:

Lemma 5.2.1 *The diagram*

$$\begin{tikzcd}
(F(A),X) \arrow[r, "{(\theta^F,!)}"] \arrow[d, "{(p_0\,;F(a)\ ,\ x)}"'] & (F(A\times X),1) \arrow[d, "{(p_0\,;F(a\times x)\ ,\ id_1)}"] \\
(F(A'),X') \arrow[r, "{(\theta^F,!)}"'] & (F(A'\times X'),1)
\end{tikzcd}$$

commutes.

Proof. The lemma's diagram is the image in $P_{\mathbf{X}}(\mathbf{Z})$ of

$$\begin{tikzcd}
(A,X) \arrow[r, "{(id,!)}"] \arrow[d, "{(p_0\,;a\ ,\ x)}"'] & (A\times X,1) \arrow[d, "{(p_0\,;(a\times x)\ ,\ id_1)}"] \\
(A',X') \arrow[r, "{(id,!)}"'] & (A'\times X',1)
\end{tikzcd}$$

in $P_{\mathbf{X}}(\mathbf{Y})$ under $\tilde{F}$. To see this we observe that

$$\tilde{F}(p_0\,;a\ ,\ x) = \tilde{F}(p_0,x)\,;\tilde{F}(p_0\,;a\ ,\ id) = (p_0,x)\,;\tilde{F}(p_0\,;a\ ,\ id)$$

because the splitting over x, (p_0,x), must be taken by $\tilde{F}$ to the specified (i.e. same) splitting over x. Furthermore, any map $\tilde{F}(p_0\,;g\ ,\ id)$ is determined as the unique map over the splitting of $!\ :\ X \longrightarrow 1$ by applying $\tilde{F}$ to

$$\begin{tikzcd}
(A,X) \arrow[r, "{(p_0,!)}"] \arrow[d, dotted, "{(p_0\,;g\ ,\ id_X)}"'] & (A,1) \arrow[d, "{(p_0\,;g\ ,\ id_1)}"] \\
(A',X) \arrow[r, "{(p_0,!)}"'] & (A',1)
\end{tikzcd}$$

and observing that $(\tilde{F}_0(p_0\,;g\ ,\ id_X)\ ,\ id_X)$ is the unique solution in the resultant image

$$\begin{tikzcd}
(F(A),X) \arrow[r, "{(p_0,!)}"] \arrow[d, dotted] & (F(A),1) \arrow[d, "{(\tilde{F}_0(p_0\,;g\ ,\ id_1),id_1)}"] \\
(F(A'),X) \arrow[r, "{(p_0,!)}"'] & (F(A'),1)
\end{tikzcd}$$

Tracing the diagram's first coordinate yields $\tilde{F}_0(p_0\,;g\ ,\ id_X) = p_0\,;F(g)$, giving the commutativity of the lemma's diagram (and consequently the naturality of θ^F). □

A more precise form for the $\tilde{F}$–image of an arbitrary map $(a,x)\ :\ (A,X) \longrightarrow (A',X')$ can now be derived:

Lemma 5.2.2

$$\tilde{F}(a,x) = (\theta^F\,;F(a)\ ,\ x).$$

Proof. A splitting can be factored as a vertical-map/cartesian-map pair $(a,x) = (a,id)\,;(p_0,x)$, leaving the problem of finding $\tilde{F}(a,id)$. But we note that

$$\begin{tikzcd}
(A,X) \arrow[r, "{(id,!)}"] \arrow[d, "{(a,id)}"'] & (A\times X,1) \arrow[d, "{(p_0\,;a\ ,\ id_1)}"] \\
(A',X) \arrow[r, "{(p_0,!)}"'] & (A',1)
\end{tikzcd}$$

commutes so $\tilde{F}(a,id)$ is the unique solution (by the familiar splitting argument) in the diagram's $\tilde{F}$-image (created via the proof of Lemma 5.2.1)

$$\begin{tikzcd}
(F(A),X) \arrow[r, "{(\theta^F,!)}"] \arrow[d, dotted] & (F(A\times X),1) \arrow[d, "{(p_0\,;F(a)\ ,\ id_1)}"] \\
(F(A'),X) \arrow[r, "{(p_0,!)}"'] & (F(A'),1)
\end{tikzcd}$$

which is clearly given by $(\theta^F\,;F(a)\ ,\ id_X)$.

□

This observation is useful in showing how to reconstruct the functor $\tilde{F}$ from (F, θ^F). More immediately we see it provides

$$\begin{tikzcd}
F(A) \times 1 \arrow[rr, "\theta^F"] \arrow[dr, "p_0"'] & & F(A \times 1) \arrow[dl, "F(p_0)"] \\
& F(A) &
\end{tikzcd}$$

the first requirement for strength by examining the first coordinate of the image of

$$\begin{tikzcd}
(A, 1) \arrow[rr, "(id_A, !)"] \arrow[dr, "(p_0, id)"'] & & (A \times 1, 1) \arrow[dl, "(p_0\,; p_0\ ,\ id_1)"] \\
& (A, 1) &
\end{tikzcd}$$

under $\tilde{F}$. For verifying associativity of the strength we can now track the first coordinate in the $\tilde{F}$–image of

$$\begin{tikzcd}
(A, X \times Y) \arrow[r, "(\mathrm{ass}\,; p_0\ ,\ p_1)"] \arrow[d, "(p_0, id)"'] & (A \times X, Y) \arrow[r, "(id, !)"] & ((A \times X) \times Y, 1) \\
(A, X \times Y) \arrow[rr, "(id, !)"'] & & (A \times (X \times Y), 1) \arrow[u, "(p_0\,; \mathrm{ass}\ ,\ id_1)"']
\end{tikzcd}$$

This boils down to distilling from the fibration morphism a strong functor. In similar fashion we can extract from a natural transformation $\tilde{\alpha} : \tilde{F}_1 \xrightarrow{\bullet} \tilde{F}_2$ of fibration morphisms a strong transformation α between the respective extracted functors. The (A, X)-component of $\tilde{\alpha}$ has the general form

$$\tilde{\alpha}^{A,X} = (\tilde{\alpha}_0^{A,X}, id_X)$$

due to the horizontal composition requirement. In the fiber over 1 the A-component of α is defined as

$$\alpha_A = \langle id, ! \rangle \, ; \tilde{\alpha}_0^{A,1}$$

It is straightforward to verify the required transformational properties of α and to derive the relationship

$$\tilde{\alpha}^{A,X} = (p_0 \, ; \alpha_A \ , \ id_X)$$

directly as the unique comparison over a splitting arrow in the following diagram:

$$\begin{tikzcd}
\tilde{F}(A,X) \arrow[r, "\tilde{F}(p_0,!)"] \arrow[d, dotted] & \tilde{F}(A,1) \arrow[d, "{(\tilde{\alpha}_0^{A,1}, id_1)}"] \\
\tilde{G}(A,X) \arrow[r, "\tilde{G}(p_0,!)"'] & \tilde{G}(A,1)
\end{tikzcd}$$

This extraction process preserves composition of ordinary functors and transformations as it clearly does so in the fiber over 1. We must also check that it preserves the strong compositions. For functors this fact is given by breaking the defining map for the strength of F into a composition

$$(\theta^F, !) = (id, !) \ ; \ (p_0 \, ; \theta_F \ , \ id_1)$$

Under $\tilde{G}$ this brings forth in the first coordinate the Kock equation [30]

$$\theta^{G \circ F} = \theta^G \, ; G(\theta^F)$$

for the canonical composition of tensorial strengths as desired. For natural transformations of morphisms of fibrations the strengthening of the extracted natural transformations is found in the first coordinate of

$$\begin{tikzcd}
\tilde{F}(A,X) \arrow[r, "\tilde{F}(id,!)"] \arrow[d, "\tilde{\alpha}^{A,X}"'] & \tilde{F}(A \times X, 1) \arrow[d, "\tilde{\alpha}^{A \times X, 1}"] \\
\tilde{G}(A,X) \arrow[r, "\tilde{G}(id,!)"'] & \tilde{G}(A \times X, 1)
\end{tikzcd}$$

5.3 Strong Functors to Fibration Morphisms

This establishes a 2–functor

$$\mathbf{StrSplFib}(\mathbf{X}) \longrightarrow \mathbf{Strong}(\mathbf{X})$$

To show further it is an isomorphism it suffices to show bijectivity on objects, functors, and transformations. By construction, bijectivity immediately holds for objects.

Given a strong functor (F, θ^F) we define an opposing lifting map that sends F to $\tilde{F}$ defined by:

$$[(A, X) \xrightarrow{(a,x)} (A', X')] \quad \mapsto \quad [(F(A), X) \xrightarrow{(\theta^F ; F(a)\ ,\ x)} (F(A'), X')].$$

Note that if $\tilde{F}$ is a morphism of fibrations the strong functor recaptured from the fiber over 1 will be F since

$$\begin{aligned}
\langle id, ! \rangle ; \tilde{F}_0(p_0 ; a\ ,\ id_1) &= \langle id, ! \rangle ; \theta^F ; F(p_0 ; a) \\
&= \langle id, ! \rangle ; p_0 ; F(a) \\
&= F(a).
\end{aligned}$$

So $F \longrightarrow \tilde{F} \longrightarrow F$ will be the identity. Conversely, Lemma 5.2.2 will tell us that the transition $\tilde{F} \longrightarrow F \longrightarrow \tilde{F}$ is also the identity. This means the 2–functor will be an isomorphism on functors.

$\tilde{F}$ is indeed a morphism of fibrations as it is a functor $P_{\mathbf{X}}(\mathbf{Y}) \longrightarrow P_{\mathbf{X}}(\mathbf{Z})$ by

$$\begin{aligned}
\langle \theta^F ; F(g)\ ,\ x \rangle ; \theta^F ; F(g') &= \langle \theta^F, p_1 \rangle ; (F(g) \times x) ; \theta^F ; F(g') \\
&= \langle \theta^F, p_1 \rangle ; \theta^F ; F(g \times x) ; F(g') \\
&= \langle id, p_1 \rangle ; (\theta^F \times id) ; F(g \times x) ; F(g') \\
&= (id \times \Delta) ; ass ; (\theta^F \times id) ; F(g \times x) ; F(g')
\end{aligned}$$

$$
\begin{aligned}
&= (id \times \Delta) ; \theta^F ; F(ass) ; F(g \times x) ; F(g') \\
&= \theta^F ; F(id \times \Delta) ; F(ass) ; F(g \times x) ; F(g') \\
&= \theta^F ; F(\langle g \ , \ p_1 ; x \rangle ; g')
\end{aligned}
$$

and it preserves the splitting because $\theta^F ; F(p_0) = p_0$.

Finally, we must show that strong transformations can be lifted back to transformations of fibrations. We now know that a strong transformation is determined by its underlying ordinary transformation. From this fact the process of extracting the strong transformation from a lifted strong transformation is clearly seen to behave as the identity.

If the lifting of the strong transformation extracted from a transformation of fibrations can also be proved to be the identity then we will have established an isomorphism of transformations. To confirm this, we first remark that a strong transformation α may be lifted by using the following diagram as a "naturality square":

$$\begin{tikzcd}
(F(A), X) \arrow[r, "(\theta^F ; F(a) \ , \ x)"] \arrow[d, "(p_0 ; \alpha_A \ , \ id_X)"'] & (F(A'), X') \arrow[d, "(p_0 ; \alpha_{A'} \ , \ id_{X'})"] \\
(G(A), X) \arrow[r, "(\theta^G ; G(a) \ , \ x)"'] & (G(A'), X')
\end{tikzcd}$$

Furthermore, we discovered above that transformations of fibrations also have this form. So the lifting of an extracted transformation is truly the identity. This completes the proof of Theorem 5.1.1.

We shall hereafter refer to the full and faithful embedding of **Strong**(**X**) into **SplFib**(**X**) that follows from this development as the *strong–fibration embedding*.

5.4 An Indexed Category View of Strength

A very compact perspective of strength leverages the equivalence between the categories of split fibrations and of split indexed categories. Because we can view the $\mathbf{X}$–action $\oslash$ of the $\mathbf{X}$–strong category $\mathbf{Y}$ as an ordinary product in $\mathbf{Y}$, each functor $\_ \oslash X$ for $X \in \mathbf{X}$ is a comonad of $\mathbf{Y}$. The unit and multiplier are:

$$\eta : \_ \oslash X \xrightarrow{p_0} \_$$

$$\mu : \_ \oslash X \xrightarrow{\langle id, p_1 \rangle} (\_ \oslash X) \oslash X$$

The Kleisli category of this comonad is also Lambek's polynomial category obtained by adjoining an element [31].

To each $X \in \mathbf{X}$ we may associate the Kleisli category $\mathbf{Y}_X$. Since any morphism $f : X_1 \rightarrow X_2$ in $\mathbf{X}$ clearly induces a morphism of the respective comonads, then it also induces a functor $f^* : \mathbf{Y}_{X_2} \rightarrow \mathbf{Y}_{X_1}$. Our association is consequently an indexed category $\mathcal{F} : \mathbf{X}^{op} \rightarrow \mathbf{Cat}$. The morphisms of fibrations in $\mathbf{StrSplFib}(\mathbf{X})$ restricted to the fiber over 1, i.e. the strong functors, can now be seen to be exactly the morphisms of the Grothendieck construction of the corresponding split fibration for $\mathcal{F}$.

Chapter 6

Datatypes and Strength

Tatsuya Hagino's thesis [17] proposed a categorical programming language that introduced two sorts of datatype declarations: initial datatypes and final datatypes. Gavin Wraith [56] somewhat simplified the original system and declarations by assuming explicitly that the underlying setting was a bicartesian closed category. In our treatment we take another direction: assume that the underlying category is a cartesian category and replace the closed requirement by the assumption that all datatypes derivable from our declarations are strong in the sense of being parametrized images of first-order strong functors. Because our strong functors of Chapter 5 are always covariant, we avoid the well-known nuisance of moving contravariant functors to new venues where covariance can be reestablished for taking colimits. For example, the mixed-variant exponentials and hom-functors can be transported into O-categories [47], where they can also behave covariantly — the limit-colimit coincidence [41] — and thereby become eligible for computing ω-colimits as solutions of recursive domain equations for datatypes.

6.1 Declarations of Factorizers

Factorizer versions of the Hagino "right" and "left" forms of datatype declaration done in a style similar to that suggested by Wraith are:

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

The first declaration asserts that any map to the new type $R(A)$ from a state type C is to be determined by a set of programmer-chosen maps from C to the component types $E_1(A,C), \ldots, E_n(A,C)$. The second definition forms exactly the dual. The labels d_i and c_i name the collection of canonical operations (not to be confused with the programmer-chosen maps) that are universally associated with their respective datatypes and in one-to-one correspondence with the programmer-chosen maps in a particular declaration. Clearly the first definition declares a final datatype, the second an initial datatype.

The chosen parameter type variable A usually ranges over a power of the given category. However, we shall take the interpretation that it simply ranges over some category and assume the typing $E_i : \mathbf{A} \times \mathbf{C} \longrightarrow \mathbf{C}$. The object $R(A)$ comes equipped with canonical maps

$$d_i : R(A) \longrightarrow E_i(A, R(A))$$

which Wraith called **destructors** as they play the dual role to the canonical maps associated with $L(A)$

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

which are traditionally called **constructors**. With respect to these maps an initial datatype satisfies simultaneously a constructor-indexed collection of universality squares of the form of the right diagram below, and a final datatype satisfies a destructor-indexed collection of diagrams of the left form:

$$\begin{tikzcd}
C \arrow[r, "g_i"] \arrow[d, dotted, "f_R(gs)"'] & E_i(A,C) \arrow[d, dotted, "E_j(A, f_R(gs))"] \\
R(A) \arrow[r, "d_i"'] & E_i(A, R(A))
\end{tikzcd}
\qquad
\begin{tikzcd}
E_j(A, L(A)) \arrow[r, "c_j"] \arrow[d, dotted, "E_i(A, f_L(hs))"'] & L(A) \arrow[d, dotted, "f_L(hs)"] \\
E_j(A,C) \arrow[r, "h_j"'] & C
\end{tikzcd}$$

These diagrams are Wraith's original simple versions for defining categorical datatypes. The datatype actions, i.e. the mediating factorizers f_L and f_R, are respectively determined uniquely by providing the tuples of g_i and h_j morphisms. Here the collections $\{g_i\}$ and $\{h_j\}$ are designated as "gs" and "hs".

We shall denote the datatypes as (R, d) and (L, c), respectively, to emphasize the definitional consolidation of destructors and constructors with their respective datatypes. Additionally, these finality and initiality diagrams often constitute recursive definitions; they shall be referred to as the "recursion" diagrams for their datatypes.

6.2 2-Categorical Initiality and Finality

Potentially the category of coalgebras for the final datatypes may be constructed as an **inserter** (see Kelly [27]), a 2–categorical limit, if it exists. The inserter resembles the comma category except that the functors giving the domain and codomain originate on the same object. In fact it is a lax equalizer: for general functors $G, H : \mathbf{A} \rightarrow \mathbf{B}$, the inserter is the universal pair (U, β) where $U : \mathbf{Insert}(G, H) \rightarrow \mathbf{A}$ is a functor and $\beta : \; U \, ; G \xrightarrow{\bullet} U \, ; H$ is a natural transformation:

$$\begin{tikzcd}
\mathbf{Insert}(G,H) \arrow[r] & \mathbf{A} \arrow[r, bend left, "G"] \arrow[r, bend right, "H"'] \arrow[r, phantom, "\Downarrow\beta"] & \mathbf{B} \quad .
\end{tikzcd}$$

For example, in the 2-category **Cat** the objects of the inserter category $\mathbf{Insert}(G, H)$ are the pairs (A, b) where A is in $\mathbf{A}$ and $b : G(A) \rightarrow H(A)$. A morphism $(A_1, b_1) \rightarrow (A_2, b_2)$ corresponds to the existence of a commuting "naturality square" generated by an $\mathbf{A}$-morphism $f : A_1 \rightarrow A_2$:

$$\begin{tikzcd}
G(A_1) \arrow[r, "G(f)"] \arrow[d, "b_1"'] & G(A_2) \arrow[d, "b_2"] \\
H(A_1) \arrow[r, "H(f)"'] & H(A_2)
\end{tikzcd}$$

The inserter functor U is realized as the underlying (forgetful) functor from $\mathbf{Insert}(G,H)$. The required natural transformation β consequently arises with the components $\beta_{(A,b)} = b$.

In our final datatype situation this means that we seek an underlying functor to the category of origin

$$U_R : \mathbf{Insert}(Pr_1\,;\Delta_n\ ,\ \langle E_1,\ldots,E_n\rangle) \longrightarrow \mathbf{A}\times\mathbf{C}$$

and a natural transformation α, regarded as the universal destructor of the final datatype, where

$$\begin{tikzcd}
\mathbf{Insert}(Pr_1\,;\Delta_n\ ,\ \langle E_1,\ldots,E_n\rangle) \arrow[r, shift left] \arrow[r, shift right] \arrow[r, phantom, "\Downarrow\alpha"] & \mathbf{C}^n
\end{tikzcd}$$

in which $\alpha : U_R\,; Pr_1\,;\Delta_n \overset{\bullet}{\longrightarrow} U_R\,;\langle E_1,\ldots,E_n\rangle$. Here objects of the inserter are algebras within $\mathbf{C}^n$ exactly of the form

$$\begin{tikzcd}
(C,C,\ldots,C) \arrow[d, "{(f_1,f_2,\ldots,f_n)}"] \\
(E_1(A,C),E_2(A,C),\ldots,E_n(A,C))
\end{tikzcd}$$

which are equivalent to n-tuplings of ordinary $E_i(A,\_)$-coalgebras "simultaneously" over the same datatype C. As we soon explain, our goal will be to locate a terminal object among those tupled co-algebras *parametrized by a common datatype A* by which a final datatype functor can be defined.

Similarly the opposite parametric structure, viz.

$$(\mathbf{Insert}(\langle E_1,\ldots,E_n\rangle\ ,\ Pr_1\,;\Delta_n), U_L, \alpha') \quad ,$$

constructs the simultaneous tupled $E_i(A,\_)$-algebras for the initial datatype and its universal constructor. In this case we deal with algebras of the form

$$\begin{tikzcd}
(E_1(A,C), E_2(A,C), \ldots, E_n(A,C)) \arrow[d, "{(g_1, g_2, \ldots, g_n)}"] \\
(C, C, \ldots, C)
\end{tikzcd}$$

To find what the universal properties expressed by the destructor squares translate to in this construction, we must take heed that these datatypes are *parametrized* by objects of $\mathbf{A}$. To naively demand that the underlying functor followed by the first projection, i.e. $U_R\,;\,Pr_0$, have a right adjoint for extracting the final coalgebra and the associated destructor squares is inadequate: such an adjunction would permit the parameter to vary within it! More precisely, this adjunction would appear, with possibly different parameter values A and A', as

$$\frac{A \longrightarrow A'}{((A,C), \alpha_{(A,C)}) \longrightarrow ((A', R(A')), \alpha_{(A',R(A'))})}$$

The codomain of the lower morphism expresses the would-be $R(A)$-defining final coalgebra object of the inserter.

Parametrization is more adequately viewed in fibrational terms. It is elementary to show that $\mathcal{P} = U_R\,;\,Pr_0 : \mathbf{Insert}(Pr_1\,;\,\Delta_n, \langle E_1, \ldots, E_n\rangle) \longrightarrow \mathbf{A}$ that maps each inserter morphism to its underlying $\mathbf{A}$-morphism is a cofibration. The cocartesian arrows in the inserter category are the naturality squares of the form $((A,C),\alpha) \to ((A',C),\alpha')$ generated by $\mathbf{A}\times\mathbf{C}$-morphisms $(h_{\mathbf{A}}, h_{\mathbf{C}}) : (A,C) \to (A',C)$ where $h_{\mathbf{C}} = id_C$.

The final coalgebra for each A-object can now be viewed as picking out final objects in each fiber of $\mathcal{P}$. This is formalized by asserting the existence of a *fibered* right adjoint R of $U_R\,;\,Pr_0(=\mathcal{P})$ between the identity cofibration $\mathcal{I} : \mathbf{A} \to \mathbf{A}$ and $\mathcal{P}$:

$$\begin{tikzcd}
\mathbf{A} \arrow[rr, shift right, "R"'] \arrow[rr, phantom, "\perp"] \arrow[dr, "\mathcal{I}"'] & & \mathbf{Insert}(\ldots) \arrow[ll, shift right, "\mathcal{P}"'] \arrow[dl, "\mathcal{P}"] \\
& \mathbf{A} &
\end{tikzcd}$$

The unit and counit of this adjunction are required by definition to be vertical arrows (i.e. those mapped by a fibration to an identity), thereby allowing the restrictions of the functors $\mathcal{P}$ and R to the respective fibers over any object A to form an adjunction where the parameter is constant. Each fiber-restricted right adjoint of $\mathcal{P}$ does the selection of the final coalgebra for the corresponding base in $\mathbf{A}$ exactly according to the universal destructor square. Furthermore, this definition is slightly redundant: it is easy to show that the unit is vertical if and only if the counit is vertical. Since $\mathcal{I}$-verticals can only be identities, the fundamental requirement becomes having a right adjoint to $\mathcal{P}$ that possesses an identity counit.

The adjunction bijection displayed earlier now reduces to usage of a single parameter value A, i.e.

$$\frac{A \longrightarrow A}{((A,Y),\alpha_{(A,Y)}) \longrightarrow ((A,R(A)),\alpha_{(A,R(A))})} \quad ,$$

where the top morphism must be the identity. Thus for any coalgebra (the domain of the bottom morphism, which lies in the inserter) there uniquely exists an inserter morphism from that coalgebra to the final coalgebra (the codomain of the bottom morphism).

Moreover, with R being a fibration morphism, this view of parametrization converts back to the general 2-categorical setting where we essentially insist $\mathbf{A}$ to be a reflective subcategory of $\mathbf{Insert}(\ldots)$ via an inclusion that is the right adjoint to $\mathcal{P}$. Under the expanded requirement, we can now let $\mathcal{P}$ act as a (fiber-wise) underlying functor in order to follow the "correct" notion for building final datatypes.

Likewise, the initial datatype algebra for each A-object can be "correctly" specified by noting that $\mathcal{Q} = U_L\,;Pr_0 : \mathbf{Insert}(\langle E_1,\ldots,E_n\rangle, Pr_1\,;\Delta_n) \longrightarrow \mathbf{A}$ is a fibration and providing a fibered left adjoint L that has an identity unit as follows:

$$\begin{tikzcd}
\mathbf{Insert}(\ldots) \arrow[rr, shift right, "\mathcal{Q}"'] \arrow[rr, phantom, "\perp"] \arrow[dr, "\mathcal{Q}"'] & & \mathbf{A} \arrow[ll, shift right, "L"'] \arrow[dl, "\mathcal{I}"] \\
& \mathbf{A} &
\end{tikzcd}$$

Analogously, here $\mathbf{A}$ can also be seen as a coreflective subcategory of **Insert**$(\ldots)$ under the inclusion L.

The motivation for laying out such abstract formalization is that once datatype building can be understood in this way we may ship it into any 2–category with finite limits in order to install the correct notion in that setting. The most natural candidate for receiving shipment is, of course, the 2–category **Strong**$(\mathbf{X})$. Alas, **Strong**$(\mathbf{X})$ is not finitely complete and even fails to have inserters. This situation is rescued by the strong–fibration embedding. With it we can make datatypes using **Strong**$(\mathbf{X})$ inside of **SplFib**$(\mathbf{X})$ which, by its equivalence to the 2–category of $\mathbf{X}$–indexed categories, is finitely complete.

6.3 Embedded Recursion Diagrams

The abstract construction of a strong initial or strong final datatype using the strong–fibration embedding, i.e. respectively the initial object or the final object of an inserter, can be unwound and the computationally unimportant components of morphism pairs then removed. The resulting elementary recursion diagrams expressing the two relevant fibered universal properties are displayed below. We express the collection of functions, $g_1, \ldots, g_n$ as "gs". For a *strong final datatype* over a particular parameter type A the diagram is

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

in which $\theta_2^{E_i}$, as $\theta^{E_i}\,; E_i(p_0, id_{C \times X})$ via the naturality of θ^{E_i}, is the strength applied in the second argument only. Similarly, the diagram for a *strong initial datatype* over a particular parameter datatype A is

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

These diagrams are explored further in the next section, but we should note here the serendipitous inclusion of the context parameter X in the domain of each h_i and g_i, the components of actions — *fold* and *unfold*, respectively — that are in turn determined by these components.

It makes moral intuitional sense to allow the context to be processed with the state by the components of an action using that same context. This feature contrasts with other often-used examples, e.g. concerning lists and natural number objects, where this morality has not been kept.

6.4 Embedding Corollaries

An immediate corollary of the strong-fibration embedding is :

Corollary 6.4.1 *In the datatypes (R, d) and (L, c), R and L are strong functors.*

It is long-held folklore that if a category has coproducts and admits the construction of final datatypes then the destructors form a product cone. Dually, if the category has products then the constructors of initial datatypes form coproduct cocones. For strong initial datatypes we have the even stronger result:

Corollary 6.4.2 *If (L, c) is an initial $\mathbf{X}$–strong datatype then c forms an $\mathbf{X}$–sum.*

Here we use the term $\mathbf{X}$–*sum* to indicate that the $\mathbf{X}$–action distributes over the coproduct, i.e. the $c_i \times id_X$ morphisms are embeddings.

Proof. Suppose $h_i : E_i(A, L(A)) \times X \longrightarrow C$ then any comparison map

$$h : L(A) \times X \longrightarrow C$$

with $(c_i \times id_X)\,; h = h_i$ satisfies the following recursion diagram

$$\begin{tikzcd}
E_i(A, L(A)) \times X \arrow[r, "c_i \times id_X"] \arrow[d, "\langle \theta_2^{E_i}, p_1 \rangle"'] & L(A) \times X \arrow[dd, "\langle p_0, h \rangle"] \\
E_i(A, L(A) \times X) \times X \arrow[d, "E_i(A, \langle p_0, h \rangle) \times id_X"'] & \\
E_i(A, L(A) \times C) \times X \arrow[r, "\nu_i"'] & L(A) \times C
\end{tikzcd}$$

where $\nu_i = \langle p_0\,; E_i(id_A, p_0)\,; c_i\;,\; (E_i(id_A, p_0) \times id_X)\,; h_i \rangle$. Therefore it exists and is unique.

□

6.5 Choosing a Recursion Scheme

Many initial datatypes have been studied in first-order settings. For example, Leopoldo Román [46] proposed various recursion schemes for the natural numbers, including the one suggested by the above analysis. Cockett [8] studied the list datatype by using a slight variant of the list recursion diagram. For initial datatypes based on functors whose strengths are isomorphisms (these we call *linear functors*) the exact form of the recursion diagrams is not critical. As an example, the recursion suggested by Cockett is correct — but morally "wrong" — and is shown by

$$\begin{tikzcd}
A \times (\mathit{list}(A) \times X) \arrow[r, "\mathit{ass}\,;\,\mathit{cons} \times \mathit{id}_X"] \arrow[d, "\mathit{id}_A \times a"] & \mathit{list}(A) \times X \arrow[d, "a"] & 1 \times X \arrow[l, "\mathit{nil} \times \mathit{id}_X"'] \arrow[dl, "f"] \\
A \times C \arrow[r, "g"'] & C &
\end{tikzcd}$$

It is quickly seen to be equivalent to the morally correct version

$$\begin{tikzcd}
(A \times \mathit{list}(A)) \times X \arrow[r, "\mathit{cons} \times \mathit{id}_X"] \arrow[d, "\langle \mathit{ass}^{-1}\,;\,\mathit{id}_A \times a\,,\,p_2 \rangle"] & \mathit{list}(A) \times X \arrow[d, "a"] & 1 \times X \arrow[l, "\mathit{nil} \times \mathit{id}_X"'] \arrow[dl, "f"] \\
(A \times C) \times X \arrow[r, "g'"'] & C &
\end{tikzcd}$$

that combines the context X with the state C for processing by the programmer-chosen maps f and g'.

For more complex datatypes based on functors whose strengths are not isomorphisms the equivalence between the datatypes implied by the diagrams above fails.

Choosing the correct diagram in these cases becomes critical, but the proper choice is made evident by our general construction.

6.6 CCC Datatypes are Strong

A common asserted justification for stronger recursion diagrams are their validity in a cartesian closed category. This justification's weakness is revealed by its not confirming whether we have obtained all the strength it is natural to have.

Yet, admittedly, knowing that our strength-based diagrams are valid in a cartesian closed category certainly provides assurance:

Proposition 6.6.1 *In a cartesian closed category ordinary datatypes are necessarily strong datatypes.*

Proof. We shall demonstrate this for the initial datatypes. Suppose (L, c) is an ordinary initial datatype, i.e. it satisfies the simple algebra initiality property firstly-mentioned in this section.

Let $h_i : E_i(A, C) \times X \longrightarrow C$ be a collection of programmer-chosen maps. Define h_i^* as a particular transpose via

$$\frac{E_i(A, X \Rightarrow C) \times X \xrightarrow{\langle \theta_2^{E_i}, p_1 \rangle} E_i(A, (X \Rightarrow C) \times X) \times X \xrightarrow{E_i(id_A, \mathrm{eval}) \times id_X \,;\, h_i} C}{E_i(A, X \Rightarrow C) \xrightarrow{h_i^*} X \Rightarrow C}$$

Using the h_i^* as programmer-chosen maps for the ordinary initial datatype L, we construct the diagram below to obtain a unique arrow $a : L(A) \longrightarrow X \Rightarrow C$ which gives commutativity in its top square. The bottom pentagon commutes by adjointness.

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

By rearranging the left vertical composite, the outermost polygon becomes the strong recursion diagram for $\mathit{fold}\{hs\} = (a \times id_X); \mathit{eval}$. Since any *fold* map can be factored in the presence of cartesian closure into the form of the right vertical composite, the comparison map is unique. Thus (L, c) is strong as well.

□

Chapter 7

Datatypes to Combinator Reduction

We now transition from specifying strong datatypes to establishing a categorical combinator reduction system that is directly derivable from our brand of Hagino-Wraith style declarations.

The strong initiality and strong finality diagrams of the previous chapter provide immediately the rewrite rules for the $\mathit{fold}\{h_1, \ldots, h_n\}$ and $\mathit{unfold}\{g_1, \ldots, g_n\}$ actions by pasting the diagrams in the direction of decreasing data structure complexity, i.e. towards earlier declaration. These respective actions are uniquely built from a programmer's specification of context-parameterized state transformers h_i coming from components of a datatype and of context-parametrized state de-transformers g_i going to components of a datatype. If components of a datatype are intuitively regarded both as its generalized "subterms" and as "instances" of the domains (codomains) of the transformers (de-transformers), the rewriting rules can be seen to correspond closely to folding and unfolding program transformations as first cataloged by Burstall and Darlington [5].

Thus the *fold* rewrite rule for the initial datatype L is easily derived from the embedded initiality diagram of Chapter 6 to be:

$$c_i \times id_X \,;\mathit{fold}^L\{h_1, \ldots, h_n\} \quad \Longrightarrow \quad \langle \theta^{E_i}\{p_0, \mathit{fold}^L\{h_1, \ldots, h_n\}\}, p_1\rangle \,; h_i$$

and similarly the *unfold* rewrite rule for the final datatype R is found from the embedded finality diagram to be:

$$\mathit{unfold}^R\{g_1, \ldots, g_n\} \,; d_i \quad \Longrightarrow \quad \langle g_i, p_1\rangle \,; \theta^{E_i}\{p_0, \mathit{unfold}^R\{g_1, \ldots, g_n\}\}$$

where $\theta^F\{f_1,\ldots,f_m\}$ will hereafter abbreviate $\theta^F\,;F(f_1,\ldots,f_m)$ for any strong datatype F.

We call fold^L the *L fold factorizer* or informally the *fold from L*. Likewise we term unfold^R the *R unfold factorizer* or informally the *unfold to R*.

7.1 Specialized Factorizers

These two factorizers can form a minimal utilizable set of categorical programming primitives within this setting, but such a severe constraint in expressiveness leads expectedly to over-complicated combinator sequences.

This problem is alleviated by defining two specializations, respectively, of both folding and unfolding that correspond to familiar and vital programming tools. The definitions leverage the fact that, although we are working in a cartesian setting, this setting is closed over *all* declarations of strong initial and final datatypes. We first state the specializations in terms of their equivalence to either a fold or unfold operator as a group, and then explain their individual derivations and associated rewriting rules.

First, the respective analogies of case analysis and higher-order transformation for initial datatypes can be specified with the use of the available binary products, which can be considered as the strong final datatype Prod of parametric arity 2 ($\mathbf{A}$ is actually $\mathbf{A}\times\mathbf{A}$) where the component functors E_1 and E_2 are respectively $\mathit{Pr}_0\circ\mathit{Pr}_0$ and $\mathit{Pr}_1\circ\mathit{Pr}_0$ (i.e. $E_1((A_1,A_2),C)=A_1$ and $E_2((A_1,A_2),C)=A_2$):

(i) *L case factorizer* :

$$\begin{aligned}&\mathrm{case}^L\{h_1,\ldots,h_n\}\equiv\\&\quad \mathit{fold}^L\{\ldots,\langle p_0\,;E_i(id_A,p_0)\,;c_i\ ,\ (E_i(id_A,p_0)\times id_X)\,;h_i\rangle,\ldots\}\,;p_1\end{aligned}$$

(ii) *L map factorizer* :

$$\mathit{map}^L\{h_1,\ldots,h_n\}\equiv\mathit{fold}^L\{\ldots,\theta^{E_i}\{(h_1,\ldots,h_n),p_0\}\,;c_i,\ldots\}$$

Second, the analogies of record formation and higher-order transformation for final datatypes are definable by using the available binary sums, which can be treated as the strong initial datatype *Sum* of arity 2 with the same component functors as *Prod*:

(i) *R record factorizer* :

$$\begin{aligned}&\mathit{record}^R\{g_1,\ldots,g_n\} \equiv \\ &\quad b_1 \times \mathit{id}_X \,;\, \mathit{unfold}^R\{\ldots,\langle \mathit{case}^{\mathit{Sum}}\{p_0\,;d_i\,,\,g_i\}\,;E_i\{\mathit{id}_A,b_0\}\,,\,p_1\rangle,\ldots\}\end{aligned}$$

(ii) *R map factorizer* :

$$\begin{aligned}&\mathit{map}^R\{g_1,\ldots,g_n\} \equiv \\ &\quad \mathit{unfold}^R\{\ldots,d_i \times \mathit{id}_X\,;\langle \theta^{E_i}\{(g_1,\ldots,g_n),p_0\},p_1\rangle,\ldots\}\end{aligned}$$

The L case factorizer is precisely the comparison map h of the diagram in the proof of Lemma 6.4.2. The rewrite rule derives directly from the **X**–sum factoring property:

$$c_i \times \mathit{id}_X ; \mathit{case}^L\{h_1,\ldots,h_n\} \quad \Longrightarrow \quad h_i$$

The L map factorizer is defined by the diagram below in which the the collection of passed functions, $h_1,\ldots,h_n$ is abbreviated as "hs":

$$\begin{tikzcd}
E_i(A,L(A)) \times X \arrow[rr, "c_i \times id_X"] \arrow[d, "\langle\theta_2^{E_i}{,}p_1\rangle"] & & L(A) \times X \arrow[dd, dotted, "\theta^L"] \\
E_i(A,L(A) \times X) \times X \arrow[d, dotted, "E_i(id{,}\theta^L) \times id_X"] & & \\
E_i(A,L(A \times X)) \times X \arrow[r, "\theta_1^{E_i}"] \arrow[d, dotted, "E_i(id{,}L(hs)) \times id_X"] & E_i(A \times X, L(A \times X)) \arrow[r, "c_i"] \arrow[d, "E_i(hs{,}L(hs))"] & L(A \times X) \arrow[d, dotted, "L(hs)"] \\
E_i(A,L(B)) \times X \arrow[r, "\theta^{E_i}\{hs{,}p_0\}"'] & E_i(B,L(B)) \arrow[r, "c_i"'] & L(B)
\end{tikzcd}$$

The top hexagon defines L *strength* as a fold operator and the outermost polygon defines the L map factorizer as a fold operator. That is, $map^L\{hs\} = \theta^L\{hs\}$. The diagram's polygons are essentially pasted from upper-right-to-lower-left to derive the following rewrite rule:

$$c_i \times id_X \,;\, \mathrm{map}^L\{h_1, \ldots, h_n\} \quad \Longrightarrow \quad \theta^{E_i}\{(h_1, \ldots, h_n), \mathrm{map}^L\{h_1, \ldots, h_n\}\} \,;\, c_i$$

The R record factorizer can be defined in a manner dual to the definition of the L case factorizer:

$$\begin{tikzcd}
(R(A)+C) \times X \arrow[r, "\nu_i"] \arrow[dd, dotted, "case^{Sum}\{p_0, g\}"'] & E_i(A, R(A)+C) \times X \arrow[d, "\theta_2^{E_i}"] \\
 & E_i(A, (R(A)+C) \times X) \arrow[d, dotted, "E_i(id_A, case^{Sum}\{p_0, g\})"] \\
R(A) \arrow[r, "d_i"'] & E_i(A, R(A))
\end{tikzcd}$$

where

$$\nu_i = \langle case^{Sum}\{p_0 \,;\, d_i \;,\; g_i\} \;,\; p_1\rangle \,;\, E_i(id_A, b_0) \times id_X$$

and $g \;:\; C \times X \;\to\; R(A)$ is any comparison map such that $g\,; d_i \;=\; g_i$. Here $record^R\{g_1, \ldots, g_n\}$ is defined to be exactly g which is, in turn, defined by the unique definition of $case^{Sum}\{p_0, g\}$ as an unfold operator. By Lemma 6.4.2 g is uniquely specified. From the definition's factorization condition we have the rewrite rule:

$$record^R\{g_1, \ldots, g_n\} \,;\, d_i \quad \Longrightarrow \quad g_i$$

The duality is seen clearly by noting that the L case factorizer definition can now be expressed in terms of record factorizers, viz. by using the relation $record^{Prod}\{f_1, f_2\} = \langle f_1, f_2\rangle$ for f_1 and f_2 having a common domain.

Finally, the R map factorizer is defined by the following diagram in which the collection of passed functions $g_1, \ldots, g_n$ is abbreviated as "gs":

$$\begin{tikzcd}
R(A) \times X \arrow[r, "d_i \times id_X"] \arrow[ddd, dotted, "\theta^R"] & E_i(A, R(A)) \times X \arrow[d, "\langle \theta_1^{E_i}, p_1 \rangle"] & \\
& E_i(A \times X, R(A)) \times X \arrow[r, "E_i(gs, id) \times id_X"] \arrow[d, "\theta_2^{E_i}"] & E_i(B, R(A)) \times X \arrow[d, "\theta_2^{E_i}"'] \\
& E_i(A \times X, R(A) \times X) \arrow[r, "E_i(gs, id)"] \arrow[d, dotted, "E_i(id, \theta^R)"] & E_i(B, R(A) \times X) \arrow[dd, dotted, "E_i(id, \theta^R\{gs\})"'] \\
R(A \times X) \arrow[r, "d_i"'] \arrow[d, dotted, "R(gs)"] & E_i(A \times X, R(A \times X)) \arrow[dr, "E_i(gs, R(gs))"'] & \\
R(B) \arrow[rr, "d_i"'] & & E_i(B, R(B))
\end{tikzcd}$$

The top left hexagon defines R *strength* as an unfold operator and the outermost polygon defines the R map factorizer as an unfold operator. Again, this means $map^R\{gs\} = \theta^R\{gs\}$. The polygons of the diagram are pasted lower-left-to-upper-right to obtain the rewrite rule:

$$\mathrm{map}^R\{g_1, \ldots, g_n\}\,; d_i \quad \Longrightarrow \quad d_i \times id_X\,; \theta^{E_i}\{(g_1, \ldots, g_n), \mathrm{map}^R\{g_1, \ldots, g_n\}\}$$

Conjoining these six rewriting rules with the well-known basic rewriting rules for cartesian categories forms a strongly-normalizing polymorphic combinatory reduction system in which an expansive variety of initial and final datatypes can be declared in a cartesian setting and their terms computed. This system forms the abstract machine backbone of the **Charity** categorical programming language implemented by Cockett and Fukushima [11].

7.2 Recursion Examples

To balance our discussions of the more familiar initial datatypes, we close this section with two concrete strong final datatype examples: infinite lists and colists, two forms of "lazy lists". Their utility often lies in reducing complexity in list manipulation by providing infinite stacks to hold intermediate results. Final datatypes are often infinitary and and can be made to correspond to "greatest fixpoints" of colimit-preserving functors in many set-theoretic situations (e.g. see the greatest fixpoint definition and accompanying state transform examples of Manes and Arbib [36]). Our first example, infinite lists, are declared with the component functors Pr_0 and Pr_1 and with the destructors that extract respectively the head and the (infinite) tail of an infinite list. Essentially we are solving the domain equation $\mathit{Inflist}(A) = A \times \mathit{Inflist}(A)$ maximally, i.e. finding all truly infinite lists of the parameter type A. The strong infinite list recursion diagram is presented below:

$$\begin{tikzcd}
A \times X \arrow[d, "p_0"'] & C \times X \arrow[l, "\langle f, p_1 \rangle"'] \arrow[r, "\langle g, p_1 \rangle"] \arrow[d, "\mathit{unfold}^{\mathit{Inflist}}\{f, g\}"'] & C \times X \arrow[d, "\mathit{unfold}^{\mathit{Inflist}}\{f, g\}"'] \\
A & \mathit{Inflist}(A) \arrow[l, "\mathit{head}"] \arrow[r, "\mathit{tail}"'] & \mathit{Inflist}(A)
\end{tikzcd}$$

The colist datatype includes the non-empty finite lists as well as the infinite lists. Here Pr_0 and $Sum\{1, C\}$ are the declaration functors and consequently we envision the equation $\mathit{Colist}(A) = A + (A \times \mathit{Colist}(A)) = A \times (1 + \mathit{Colist}(A))$. Recall that the second equality follows from Lemma 6.4.2. The strong colist recursion diagram reduces to

$$\begin{tikzcd}
A \times X \arrow[d, "p_0"'] & C \times X \arrow[l, "\langle f, p_1 \rangle"'] \arrow[r, "\langle g, p_1 \rangle"] \arrow[d, "\mathit{unfold}^{\mathit{Colist}}\{f, g\}"'] & (1 + C) \times X \arrow[d, "\theta^{\mathit{Sum}}\{p_0, \mathit{unfold}^{\mathit{Colist}}\{f, g\}\}"'] \\
A & \mathit{Colist}(A) \arrow[l, "\mathit{cohead}"] \arrow[r, "\mathit{cotail}"'] & 1 + \mathit{Colist}(A)
\end{tikzcd}$$

The partiality of the colist *cotail*, revealed by the appearance of an error type (1) in its codomain, contrasts with the totality of *cohead*.

Chapter 8

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 types — which will be shown in Chapter 10 to correspond directly with strong datatypes — to the original cartesian category. Often the terms "type" and "datatype" will be abused: technically we are building types within our theory, but we may also refer to them by their associated datatype (functor). Our construction generalizes the term logic development presented for predistributive categories in Cockett [7].

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 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.

8.1 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

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

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 type. 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 a 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 amended 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) $\mathit{fvars}(()) = \emptyset$.

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

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

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

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

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 identically-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 be 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 as demonstrated by Stoughton [48]. The rules are:

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

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

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

  (2) else if x is bound 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 proven 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:**

$$\frac{t_1 \ =_{v_\tau} \ t_2 \qquad t'_1 \ =_{v'_\tau} \ t'_2 \qquad v'_1, v'_2 \text{ are type}(t_1)-\text{typed variable bases of } t'_1, t'_2}{\sigma_{v'_1:=t_1}(t'_1) \ =_{v_\tau} \ \sigma_{v'_2:=t_2}(t'_2)}$$

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 8.1.1 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 8.1.2 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 sections, 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.

8.2 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

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

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

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

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 having parameters and running within scopes of environmental variables.

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 $i = 1, \dots, n$

(ii)

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

(iii)

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

(iv)

$$\sigma_{v:=t}(L\left[\begin{array}{c} w_1 \mapsto t_1 \\ \dots \\ w_m \mapsto t_m \end{array}\right](t')) = L\left[\begin{array}{c} w_1 \mapsto \sigma_{v:=t}(t_1) \\ \dots \\ w_m \mapsto \sigma_{v:=t}(t_m) \end{array}\right](\sigma_{v:=t}(t'))$$

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, List(A), and database trees, DBTree(A, B), as declared within the display of datatype examples in the opening paragraphs of Chapter 11.

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:**

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

List(A):

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

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

DBTree(A,B):

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

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

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

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

List(A):

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

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

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:**

$$L\begin{bmatrix} w_1 \mapsto t_1 \\ \dots \\ w_m \mapsto t_m \end{bmatrix}(c_i(t)) \;=_{v_X}\; c_i(E_i\begin{bmatrix} \dots \\ w_i \mapsto t_i \\ \dots \\ w_L \mapsto L\begin{bmatrix} \dots \\ w_i \mapsto t_i \\ \dots \end{bmatrix}(w_L) \end{bmatrix}(t))$$

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):

$$\mathrm{DBTree}\begin{bmatrix} v_A \mapsto t_1 \\ v_B \mapsto t_2 \end{bmatrix}(\mathrm{dbleaf}(a)) \;=_{v_X}\; \mathrm{dbleaf}(t_1(a))$$

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

(iv) L **Fold Uniqueness:**

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

List(A):

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

DBTree(A,B):

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

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 [7] 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 [12] arises directly from the fold uniqueness rule:

Theorem 8.2.1 *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

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

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.

□

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

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

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

$$(v_L\ ,\ \left\{\begin{matrix} \cdots \\ c_i(v_i) \mapsto t_i \\ \cdots \end{matrix}\right\}(v_L))$$

and "t_i" to

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

□

Theorem 8.2.3 *The map-on-L is determined by the fold-from-L:*

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

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

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

and "t_i" to

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

□

8.3 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 $\tau_1, \ldots, \tau_m$ are types then $R(\tau_1, \ldots, \tau_m)$ is a type.

Variables and Variable Bases:

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

Just as for the initial datatypes, the variable bases for the datatype $R(\tau_1, \ldots, \tau_m)$ 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 $d_1, \ldots, d_n$ are the destructors associated with $R(\tau_1, \ldots, \tau_m)$ and t is a term of type $R(\tau_1, \ldots, \tau_m)$ then $d_i(t)$ is a *destructor term* of type $E_i((\tau_1, \ldots, \tau_m), R(\tau_1, \ldots, \tau_m))$.

(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 $E_i((\tau_1, \ldots, \tau_m), C)$ then

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

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

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

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

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

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

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

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

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:**

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

Inflist(A):

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

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

Colist(A):

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

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

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

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

$$d_i\left(\begin{pmatrix} d_1 : t_1(t) \\ \ldots \\ d_n : t_n(t) \end{pmatrix}\right) =_{v_X} \{v_C \mapsto t_i\}(t)$$

Inflist(A):

$$\text{head}\,(\begin{pmatrix}\text{head} : t_1(c)\\ \text{tail} : t_2(c)\end{pmatrix}) \ =_{v_X}\ \{v_C \mapsto t_1\}(c)$$

$$\text{tail}\,(\begin{pmatrix}\text{head} : t_1(c)\\ \text{tail} : t_2(c)\end{pmatrix}) \ =_{v_X}\ \{v_C \mapsto t_2\}(c)$$

Colist(A):

$$\text{cohead}\,(\begin{pmatrix}\text{cohead} : t_1(c)\\ \text{cotail} : t_2(c)\end{pmatrix}) \ =_{v_X}\ \{v_C \mapsto t_1\}(c)$$

$$\text{cotail}\,(\begin{pmatrix}\text{cohead} : t_1(c)\\ \text{cotail} : b_0(())\end{pmatrix}) \ =_{v_X}\ b_0(())$$

$$\text{cotail}\,(\begin{pmatrix}\text{cohead} : t_1(c)\\ \text{cotail} : b_1(t_2'(c))\end{pmatrix}) \ =_{v_X}\ b_1(\{v_C \mapsto t_2'\}(c))$$

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

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

Inflist(A):

$$\mathrm{head}\,(\mathrm{Inflist}[v_A \mapsto t](\left(\left|\; v_C \mapsto \begin{array}{c}\mathrm{head} : t_1\\[1ex] \mathrm{tail} : t_2\end{array}\right|\right)(c))) \;=_{v_X}\; t(t_1(c))$$

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

$$\begin{aligned}\mathrm{tail}\,(\mathrm{Inflist}[v_A \mapsto t](\left(\left|\; v_C \mapsto \begin{array}{c}\mathrm{head} : t_1\\[1ex] \mathrm{tail} : t_2\end{array}\right|\right)(c))) \;=_{v_X}&\\ \mathrm{Inflist}[\ldots](\left(\left|\; v_C \mapsto \begin{array}{c}\mathrm{head} : t_1\\[1ex] \mathrm{tail} : t_2\end{array}\right|\right)(t_2(c))&\end{aligned}$$

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

Colist(A):

$$\mathrm{cohead}\,(\mathrm{Colist}\left[v_A \mapsto t\right](\left(\left|\; v_C \mapsto \begin{array}{c}\mathrm{cohead} : t_1\\[1ex] \mathrm{cotail} : t_2\end{array}\right|\right)(c))) \;=_{v_X}\; t(t_1(c))$$

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

$$\mathrm{cotail}\,(\mathrm{Colist}\left[v_A \mapsto t\right](\left(\left|\; v_C \mapsto \begin{array}{c}\mathrm{cohead} : t_1\\[1ex] \mathrm{cotail} : b_0(())\end{array}\right|\right)(c))) \;=_{v_X}\; b_0(())$$

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

$$\begin{aligned}\mathrm{cotail}\,(\mathrm{Colist}\left[v_A \mapsto t\right](\left(\left|\; v_C \mapsto \begin{array}{c}\mathrm{cohead} : t_1\\[1ex] \mathrm{cotail} : b_1(t_2')\end{array}\right|\right)(c))) \;=_{v_X}&\\ b_1(\mathrm{Colist}\,[\ldots](\left(\left|\; v_C \mapsto \begin{array}{c}\mathrm{cohead} : t_1\\[1ex] \mathrm{cotail} : b_1(t_2')\end{array}\right|\right)(t_2'(c))))&\end{aligned}$$

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

(iv) R **Unfold Uniqueness:**

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

Inflist(A):

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

Colist(A):

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

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.

Chapter 9

Program Equivalences

The term logic's validation with respect to the strong categorical datatype setting now lies in wait. Yet with the logic's inference rules afresh, this juncture is timely for thoroughly demonstrating an associated methodology for proving equivalences among programs. The reader should note the use of categorical diagrams for motivating various proofs in the examples. The to-be-proved equivalence of term logic with the categorical setting allows us to move freely back and forth between these two arenas to see more clearly the paths towards solutions. Nevertheless, every diagram in these examples represents a compacted rewriting sequence in the term logic and examples could have thusly been expressed entirely with term equations.

9.1 Structural versus Primitive Induction

The only "induction" principles available are the primitive forms instantiated as the fold-uniqueness and unfold-uniqueness rules. These should not be confused with the more familiar stronger form — structural induction and coinduction — that categorically demands the existence of equalizers and coequalizers. For example, the *structural equality* of two functions $f, g : \mathit{List}(A) \longrightarrow C$ in an unstrengthened (without context) setting is in hand if the equalizer object E of the equalizer (E, e) below is isomorphic to $\mathit{List}(A)$:

$$\begin{tikzcd}
E \arrow[r, "e"] & \mathit{List}(A) \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & C
\end{tikzcd}$$

Of course, we must deal with context, which suggests generalizing our diagram to

$$\begin{tikzcd}
E \times X \arrow[r, "e \times id"] & \mathit{List}(A) \times X \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & C
\end{tikzcd}$$

However, a universality property for defining "strong" equalizers that survives adequately among the strong datatype recursion diagrams appears to need higher levels of fibrational hardware that are beyond reasonable bounds of discussion here. It suffices to state only that, at the time of writing, tentative investigations by Robin Cockett point towards a possible useful characterization of structurally inductive reasoning within this term logic. We will therefore constrain our induction tools to the uniqueness rules for the coming equivalence examples.

9.2 Methodological Examples

The game to be played is proving two programs, or closed abstract maps, to be equal under the congruence of the term logic. The basic steps are:

(i) Initially attempt to reduce and/or transform both programs to a common definitionally equal program via factorizer reductions and previously-proven program equivalences.

(ii) Otherwise construct the recursion diagram for one of the programs to discover its defining specification or processing morphisms t_i that, in turn, occur in the associated uniqueness inference rule (e.g. the uniqueness rule examples in Chapter 8).

(iii) Allow the remaining program to play the role of t in the rule and thereby use t and the t_i to construct the program equivalence sub-problems that represent the rule's premises.

(iv) Repeat this method for all remaining generated sub-problems.

Solving the sub-problems produced by a pair of programs is equivalent to showing that one program behaves exactly the same as the other on common component structures of the datatype, i.e. both programs *share the same specified actions.*

9.2.1 Associativity of Appending Lists

Of course, we expect and do get a bit more: our lists are monoidal under append. Our method can easily establish it with the empty list as the unit. Nevertheless the associativity requirement is the more pedagogical one and is demonstrated below. The unit property is left as a reader's exercise.

With the definition

$$append(x, y) \equiv \left\{\!\!\left|\begin{array}{l} nil : () \mapsto y \\ cons : (a, c) \mapsto cons(a, c) \end{array}\right|\!\!\right\}(x)$$

the equivalence to be shown is

$$append(x, append(y, z)) = append(append(x, y), z) \quad .$$

Clearly reductions are not applicable and previously-proved transforms/equivalences are not yet available at this point, so we turn to constructing the recursion diagram for the program on the left, principally because it conveniently has a factorizer form:

$$append(\_, append(y, z)) = \left\{\!\!\left|\begin{array}{l} nil : () \mapsto append(y, z) \\ cons : (a, c) \mapsto cons(a, c) \end{array}\right|\!\!\right\}(\_)$$

We denote it below as pgm_l. The reader should note the strategic selection of the x-variable as the datatype variable, making the remaining variables serve as context variables. Abbreviating *List*(A) as $L(A)$, we have

$$\begin{tikzcd}
1 \times (L(A) \times L(A)) \arrow[r, "nil \times id"] \arrow[d, dotted, "id"'] & L(A) \times (L(A) \times L(A)) \arrow[d, dotted, "pgm_l"'] & (A \times L(A)) \times (L(A) \times L(A)) \arrow[l, "cons \times id"'] \arrow[d, dotted, "\langle \mathrm{ass}^{-1}; id \times pgm_l\ ,\ p_1 \rangle"'] \\
1 \times (L(A) \times L(A)) \arrow[r, "p_1; append(\_,\_)"'] & L(A) & (A \times L(A)) \times (L(A) \times L(A)) \arrow[l, "p_0; cons"]
\end{tikzcd}$$

Using the program on the right as "$t(\_) = append(append(\_, y), z)$", we construct the premises of the fold uniqueness rule for *List*(A). The two sub-program equivalence problems become:

(i) $append(append(nil(), y), z) = append(y, z)$

(ii) $append(append(cons(a, l), y), z) = cons(a, append(append(l, y), z))$

The proof of (i) is immediate with a single factorizer reduction. The proof of (ii) requires two reductions:

$$\begin{aligned}
append(append(cons(a, l), y), z) &= append(cons(a, append(l, y)), z) \\
&= cons(a, append(append(l, y), z))
\end{aligned}$$

Thus the associativity property is established.

9.2.2 Self-Inversion of List Reversal

With the definition

$$reverse(x, y) \equiv \left\{ \begin{array}{|l|} nil : () \mapsto nil() \\ cons : (a, c) \mapsto append(c, [a]) \end{array} \right\} (x) \quad ,$$

where $[a]$ abbreviates $cons(a, nil())$, we wish to show $reverse(reverse(x)) = x$. This time we draw the recursion diagram for the right hand side, i.e. the identity:

$$\begin{tikzcd}
1 \arrow[r, "\mathit{nil}"] \arrow[d, dotted, "\mathit{id}"'] & L(A) \arrow[d, dotted, "\mathit{id}"'] & A \times L(A) \arrow[l, "\mathit{cons}"'] \arrow[d, dotted, "\mathit{id}"'] \\
1 \arrow[r, "\mathit{nil}"'] & L(A) & A \times L(A) \arrow[l, "\mathit{cons}"]
\end{tikzcd}$$

So to use fold-uniqueness, we let $t(\_) = \mathit{reverse}(\mathit{reverse}(\_))$, $t_1 = \mathit{nil}()$ and $t_2(a, y) = \mathit{cons}(a, y)$, and thereby create the following sub-problems:

(i) $\mathit{reverse}(\mathit{reverse}(\mathit{nil}())) = \mathit{nil}()$

(ii) $\mathit{reverse}(\mathit{reverse}(\mathit{cons}(a, l)) = \mathit{cons}(a, \mathit{reverse}(\mathit{reverse}(l)))$

The proof of (i) follows directly with two *reverse*-factorizer reductions. The example is completed with the proof of (ii) as

$$\begin{aligned}
\mathit{reverse}(\mathit{reverse}(\mathit{cons}(a, l))) &= \mathit{reverse}(\mathit{append}(\mathit{reverse}(l), [a])) \\
&= \mathit{append}(\mathit{reverse}([a]), \mathit{reverse}(\mathit{reverse}(l))) \\
&= \mathit{append}([a], \mathit{reverse}(\mathit{reverse}(l))) \\
&= \mathit{cons}(a, \mathit{reverse}(\mathit{reverse}(l)))
\end{aligned}$$

where the second step uses the following lemma:

Lemma 9.2.1

$$\mathit{reverse}(\mathit{append}(x, y)) = \mathit{append}(\mathit{reverse}(y), \mathit{reverse}(x))$$

Proof. The recursion diagram (with $L(A) \equiv \mathit{List}(A)$) for $\mathit{reverse}(\mathit{append}(\_, y))$ can be built as

$$\begin{tikzcd}
1 \times L(A) \arrow[r, "nil \times id"] \arrow[d, dotted, "id"] & L(A) \times L(A) \arrow[d, dotted, "reverse(append(\_, y))"] & (A \times L(A)) \times L(A) \arrow[l, "cons \times id"'] \arrow[d, dotted, "\langle ass^{-1}; id \times reverse(append(\_, y)) \ , \ p_1 \rangle"] \\
1 \times L(A) \arrow[r, "p_1; reverse(l)"'] & L(A) & (A \times L(A)) \times L(A) \arrow[l, "p_0; append(l, [a])"]
\end{tikzcd}$$

The sub-problems become:

(i) $append(reverse(y), reverse(nil)) = reverse(y)$

(ii) $append(reverse(y), reverse(cons(a, l))) = append(append(reverse(y), reverse(l)), [a])$

Sub-problem (i) proceeds as

$$\begin{aligned}
append(reverse(y), reverse(nil)) &= append(reverse(y), nil) \\
&= reverse(y)
\end{aligned}$$

with the aid of a lemma ($append(x, nil) = x$) for the second step. The lemma quickly follows from a simple *id*-recursion diagram and is left to the reader. The derivation for (ii) comes via *append*'s associativity:

$$\begin{aligned}
append(reverse(y), reverse(cons(a, l)) &= append(reverse(y), append(reverse(l), [a])) \\
&= append(append(reverse(y), reverse(l)), [a])
\end{aligned}$$

□

9.2.3 Reverse is Identity in $List(1)$

We try the obvious recursion diagram to attack the example, $reverse(x) = x$:

$$\begin{tikzcd}
1 \arrow[r, "nil"] \arrow[d, dotted, "id"] & List(1) \arrow[d, dotted, "id"] & 1 \times List(1) \arrow[l, "cons"'] \arrow[d, dotted, "id"] \\
1 \arrow[r, "nil"'] & List(1) & 1 \times List(1) \arrow[l, "cons"]
\end{tikzcd}$$

Therefore the following equivalences must be proved within $List(1)$:

(i) $reverse(nil) = nil$

(ii) $reverse(cons(1, l)) = cons(1, reverse(l))$

The definition of *reverse* immediately yields (i). However, (ii) is considerably trickier — a plausible recursion diagram "candidate" for the right hand side program is offered below:

$$\begin{tikzcd}
1 \arrow[r, "nil"] \arrow[d, dotted, "id"] & List(1) \arrow[d, dotted, "{cons(1, reverse(\_))}"] & 1 \times List(1) \arrow[l, "cons"'] \arrow[d, dotted, "{id \times cons(1, reverse(\_))}"] \\
1 \arrow[r, "{[\_]}"'] & List(1) & 1 \times List(1) \arrow[l, "cons"]
\end{tikzcd}$$

The left square commutes by using one *reverse*-factorizer reduction. The right square commutes by the derivation below with the aid of a lemma for its third step:

$$\begin{aligned}
cons(1, cons(1, reverse(l))) &= cons(1, append([1], reverse(l))) \\
&= cons(1, reverse(append(l, [1]))) \\
&= cons(1, reverse(append([1], l))) \\
&= cons(1, reverse(cons(1, l)))
\end{aligned}$$

The following lemma that generates the concerned step is the point at which our excursion depends precisely on all the elements of our lists being equal:

Lemma 9.2.2 *For any $l \in List(1)$*

$$append(l, [1]) = append([1], l)$$

Proof. An appropriate recursion diagram for the left-hand program is

$$\begin{tikzcd}
1 \arrow[r, "nil"] \arrow[d, dotted, "id"] & \mathit{List}(1) \arrow[d, dotted, "append(\_, [1])"] & 1 \times \mathit{List}(1) \arrow[l, "cons"'] \arrow[d, dotted, "id \times append(\_, [1])"] \\
1 \arrow[r, "{[\_]}"'] & \mathit{List}(1) & 1 \times \mathit{List}(1) \arrow[l, "cons"]
\end{tikzcd}$$

Both squares are easily seen to be commuting by a single *append*-factorizer reduction. Thus we need the equivalences

(i^*) $append([1], nil) = [1]$

(ii^*) $append([1], cons(1, l)) = cons(1, append([1], l))$

which are both also quickly attainable by *append*-factorizer reductions.

□

Therefore the candidate diagram is appropriate for showing (ii). It brings forth the following sub-problems:

(i') $reverse(cons(1, nil)) = cons(1, nil)$

(ii') $reverse(cons(1, cons(1, l))) = cons(1, reverse(cons(1, l)))$

Completing the example, we see that that (i') is immediate and (ii') is proved by our latest lemma:

$$\begin{aligned}
reverse(cons(1, cons(1, l))) &= append(reverse(cons(1, l)), [1]) \\
&= append([1], reverse(cons(1, l))) \\
&= cons(1, reverse(cons(1, l)))
\end{aligned}$$

It is straightforward to show a categorical isomorphism between the natural numbers datatype *Nat* and *List*(1). This relationship warrants treating $append(x, y)$ for $x, y \in$ *List*(1) as the arithmetic addition of the natural numbers corresponding to x and y.

With this interpretation it is interesting to note the consequent result, which does *not* hold for the polymorphic lambda calculus:

Corollary 9.2.3 *Arithmetic addition is commutative:*

$$\mathit{append}(x, y) = \mathit{append}(y, x)$$

Proof. In $\mathit{List}(1)$ we have, by the *reverse* acting as the identity,

$$\begin{aligned}\mathit{append}(x, y) &= \mathit{reverse}(\mathit{append}(x, y)) \\ &= \mathit{append}(\mathit{reverse}(y), \mathit{reverse}(x)) \\ &= \mathit{append}(y, x)\end{aligned}$$

□

9.2.4 Zipping Streams makes a Stream

This example illustrates primitive coinduction with an infinite list form of zipping. The problem is to show

$$\mathit{zip}(\mathit{stream}_1(c_1), \mathit{stream}_2(c_2)) = \mathit{stream}_{12}(c_1, c_2)$$

where the following definitions are applied:

$$\mathit{stream}_i(c_i) \equiv \left(\left| x \mapsto \begin{array}{r} \mathit{head} : h_i(x) \\ \mathit{tail} : t_i(x) \end{array} \right|\right) (c_i) \qquad (i = 1, 2)$$

$$\mathit{stream}_{12}(c_1, c_2) \equiv \left(\left| (x, y) \mapsto \begin{array}{r} \mathit{head} : (h_1(x), h_2(y)) \\ \mathit{tail} : (t_1(x), t_2(y)) \end{array} \right|\right) (c_1, c_2)$$

$$\mathit{zip}(s_1, s_2) \equiv \left(\left| (x, y) \mapsto \begin{array}{r} \mathit{head} : (\mathit{head}(x), \mathit{head}(y)) \\ \mathit{tail} : (\mathit{tail}(x), \mathit{tail}(y)) \end{array} \right|\right) (s_1, s_2)$$

The action of stream_i on a seed value c_i creates an infinite list $\{h(c_i),\ h(t(c_i)),\ h(t^2(c_i)),\ h(t^3(c_i)),\ \ldots\}$. (For clarity, the field label and nested-parenthesis syntax requirements in the infinite list expressions are eased here in favor of list-like notation.)

The program stream_{12} operates on a pair of seed values (c_1, c_2) to build an infinite list of derived pairs: $\{(h_1(c_1), h_2(c_2)),\ (h_1(t_1(c_1)), h_2(t_2(c_2))),\ (h_1(t_1^2(c_1)), h_2(t_2^2(c_2))),\ \ldots\}$. The *zip* utility combines two infinite lists, or here as "streams", of the form $\{c_1,\ c_2,\ c_3,\ \ldots\}$ and $\{c'_1,\ c'_2,\ c'_3, \ldots\}$ into an infinite list of associated pairs: $\{(c_1, c'_1),\ (c_2, c'_2),,\ (c_3, c'_3),\ \ldots\}$.

A reasonable recursion diagram for stream_{12} is

$$\begin{tikzcd}
A \times A & C \times C \arrow[l, "h_1 \times h_2"'] \arrow[r, "t_1 \times t_2"] \arrow[d, dotted, "\mathit{stream}_{12}(\_,\_)"] & C \times C \arrow[d, dotted, "\mathit{stream}_{12}(\_,\_)"] \\
A \times A \arrow[from=u, dotted, "id"] & \mathit{Inflist}(A \times A) \arrow[l, "h_1 \times h_2"] \arrow[r, "t_1 \times t_2"'] & \mathit{Inflist}(A \times A)
\end{tikzcd}$$

The resulting program equivalences that must be verified are:

(i) $\mathit{head}(\mathit{zip}(\mathit{stream}_1(c_1), \mathit{stream}_2(c_2))) = (h_1(c_1), h_2(c_2))$

(ii) $\mathit{tail}(\mathit{zip}(\mathit{stream}_1(c_1), \mathit{stream}(c_2))) = \mathit{zip}(\mathit{stream}_1(t_1(c_1)), \mathit{stream}_2(t_2(c_2)))$

Unfold factorizer reductions easily provide both verifications. For (i) we have

$$\begin{aligned}
\mathit{head}(\mathit{zip}(\mathit{stream}_1(c_1), \mathit{stream}_2(c_2))) &= (\mathit{head}(\mathit{stream}_1(c_1)), \mathit{head}(\mathit{stream}_2(c_2))) \\
&= (h_1(c_1), h_2(c_2))
\end{aligned}$$

and for (ii) we see

$$\begin{aligned}
\mathit{tail}(\mathit{zip}(\mathit{stream}_1(c_1), \mathit{stream}_2(c_2))) &= \mathit{zip}(\mathit{tail}(\mathit{stream}_1(c_1)), \mathit{tail}(\mathit{stream}_2(c_2))) \\
&= \mathit{zip}(\mathit{stream}_1(t_1(c_1)), \mathit{stream}_2(t_2(c_2)))
\end{aligned}$$

9.2.5 Indexing Infinite Lists

Two equivalent ways for indexed-access of any infinite list built from a *fixed* pair $(h : C \times X \longrightarrow A\ ,\ t : C \times C \longrightarrow C)$ of head and tail field-generating functions is presented.

The example is interesting by its mixture of initiality induction with unfold-factorizer reductions.

First, we define a set of accessing routines relative to (h, t):

$$\mathit{fetch}(n, l) \equiv \mathit{head}(\left\{\left| \begin{array}{l} \mathit{zero} : () \mapsto l \\ \mathit{succ} : l' \mapsto \mathit{tail}(l') \end{array} \right|\right\}(n))$$

$$\mathit{stream}(c) \equiv \left(\left| x \mapsto \begin{array}{r} \mathit{head} : h(x) \\ \\ \mathit{tail} : t(x) \end{array} \right|\right)(c)$$

$$\mathit{expand}(n, c) \equiv h(\left\{\left| \begin{array}{l} \mathit{zero} : () \mapsto c \\ \mathit{succ} : c' \mapsto t(c') \end{array} \right|\right\}(n))$$

The variables are typed with $n \in \mathit{Nat}$, $l, l' \in \mathit{Inflist}(A)$, and $c, c' \in C$. The program equality to be targeted here is

$$\mathit{fetch}(n, \mathit{stream}(c)) = \mathit{expand}(n, c) \quad .$$

The left-hand program accesses the n-th entry of an infinite list built from the seed value c via the field-builders h and t. The right-hand program first builds from c the tail-field value by applying t in succession the proper number of times and then applies the final head-field build operation. Essentially *expand* accomplishes its work in ignorance of the $\mathit{Inflist}(A)$ datatype.

Some special observations are needed to find a path to the solution. First, the outermost functions of both programs (*fetch* on the left, *expand* on the right) are not quite in factorizer form due to the extra internal application of *head* and h, respectively, within them. So we define two simplications that *are* factorizers:

$$\mathit{fetch}^{-}(n, l) \equiv \left\{\left| \begin{array}{l} \mathit{zero} : () \mapsto l \\ \mathit{succ} : l' \mapsto \mathit{tail}(l') \end{array} \right|\right\}(n)$$

$$\mathit{expand}^{-}(n, c) \equiv \left\{\left| \begin{array}{l} \mathit{zero} : () \mapsto c \\ \mathit{succ} : c' \mapsto t(c') \end{array} \right|\right\}(n)$$

Now we re-phrase our problem into a "factorizer" version. By observing that $\mathit{head}(\mathit{stream}(c)) = h(c)$ (an unfold factorizer reduction), we can strip off an outermost application of *head* from *both* sides of our proposed equivalence to obtain an alternative problem:

$$\mathit{fetch}^{-}(n, \mathit{stream}(c)) = \mathit{stream}(\mathit{expand}^{-}(n, c))$$

The recursion diagram for the left-hand program with context variable c is a *Nat*-recursion as shown below:

$$\begin{tikzcd}
1 \times C \arrow[r, "\mathit{zero} \times \mathit{id}"] \arrow[d, dotted, "\mathit{id}"] & \mathit{Nat} \times C \arrow[d, dotted, "\mathit{id} \times \mathit{stream}; \mathit{fetch}^{-}"] & \mathit{Nat} \times C \arrow[l, "\mathit{succ} \times \mathit{id}"'] \arrow[d, dotted, "\langle \mathit{id} \times \mathit{stream}; \mathit{fetch}^{-}, p_1 \rangle"] \\
1 \times C \arrow[r, "p_1; \mathit{stream}(\_)"] & \mathit{Inflist}(A) & \mathit{Inflist}(A) \times C \arrow[l, "p_0; \mathit{tail}"']
\end{tikzcd}$$

It remains only to show that $\mathit{stream}(\mathit{expand}^{-}(n, c))$ is defined by the same specification:

(i) $\mathit{stream}(\mathit{expand}^{-}(\mathit{zero}, c)) = \mathit{stream}(c)$

(ii) $\mathit{stream}(\mathit{expand}^{-}(\mathit{succ}(n), c)) = \mathit{tail}(\mathit{stream}(\mathit{expand}^{-}(n, c)))$

Sub-problem (i) is a direct unfold factorizer reduction. The second one follows by applying to the left-hand program a fold factorizer reduction to the expand^{-} sub-term and a reverse unfold factorizer reduction to the result.

9.2.6 Primitive Recursion

We now employ the term logic's primitive induction for demonstrating the well-known equivalence of two popular definitions of primitive recursion over *Nat*. In fact, the proof imitates within our strong categorical setting the conventional one for obtaining this result.

Each recursion definition requires closure under composition (substitution) of the basic natural number functions *zero*, *succ*, projections, and recursive functions f built inductively from other primitive recursive functions h and k in a precise form. The two recursive forms, with context, are respectively

(i)

$$f(zero, x) = k(x)$$

$$f(succ(n), x) = h(f(n, x), x)$$

(ii)

$$f(zero, x) = k(x)$$

$$f(succ(n), x) = h'(f(n, x), n, x)$$

We first assume f to be (i)-primitive recursive and proceed to show its (ii)-primitive recursiveness entirely in categorical terms. The derivation starts by extending the definition of (i)-recursion outward into a larger commutative diagram:

$$\begin{tikzcd}
{} & {} \arrow[d, "\vdots"] & {} \arrow[d, "\vdots"] & {} \arrow[d, "\vdots"] & {} \\
{} \arrow[r, "\cdots"] & 1 \times X \arrow[r, "zero \times id"] \arrow[d, dotted, "id"] & N \times X \arrow[d, dotted, "f"] & N \times X \arrow[l, "succ \times id"'] \arrow[d, dotted, "\langle f, p_1 \rangle"] & {} \arrow[l, "\cdots"'] \\
{} \arrow[r, "\cdots"] & 1 \times X \arrow[r, "p_1; k"] & N & N \times X \arrow[l, "h"'] & {} \arrow[l, "\cdots"']
\end{tikzcd}$$

The diagram expansion for the *zero*-case portion is shown below:

$$\begin{tikzcd}
1 \times (N \times X) \arrow[r, "id \times p_1"] \arrow[dd, dotted, "id"] & 1 \times X \arrow[r, "\langle p_0; zero, zero \times id \rangle"] \arrow[d, "id"] & N \times (N \times X) \arrow[d, dotted, "id \times p_1"] & {} \arrow[l, "\cdots"'] \\
 & 1 \times X \arrow[r, "zero \times id"] \arrow[d, dotted, "id"] & N \times X \arrow[d, dotted, "f"] & {} \arrow[l, "\cdots"'] \\
1 \times (N \times X) \arrow[r, "id \times p_1"] & 1 \times X \arrow[r, "p_1; k"] & N & {} \arrow[l, "\cdots"']
\end{tikzcd}$$

The diagram's *succ*-case portion looks like:

$$\begin{tikzcd}
{} \arrow[r, "\cdots"] & N \times (N \times X) \arrow[d, dotted, "id \times p_1"] & N \times X \arrow[l, "\langle p_0; succ, id \rangle"'] \arrow[d, "id"] & N \times (N \times X) \arrow[l, "id \times p_1"'] \arrow[dd, dotted, "\langle id \times p_1; f, p_1 \rangle"] \\
{} \arrow[r, "\cdots"] & N \times X \arrow[d, dotted, "f"] & N \times X \arrow[l, "succ \times id"'] \arrow[d, dotted, "\langle f, p_1 \rangle"] & \\
{} \arrow[r, "\cdots"] & N & N \times X \arrow[l, "h"'] & N \times (N \times X) \arrow[l, "id \times p_1"']
\end{tikzcd}$$

The outermost square along with the center vertical composition forms the recursion diagram for $id \times p_1; f$ in the context $N \times X$:

$$\begin{tikzcd}
1 \times (N \times X) \arrow[r, "zero \times id"] \arrow[d, dotted, "id"] & N \times (N \times X) \arrow[d, dotted, "id \times p_1; f"] & N \times (N \times X) \arrow[l, "succ \times id"'] \arrow[d, dotted, "\langle id \times p_1; f, p_1 \rangle"] \\
1 \times (N \times X) \arrow[r, "p_1; p_1; k"] & N & N \times (N \times X) \arrow[l, "h'"']
\end{tikzcd}$$

i.e. f is of the (ii)-recursive form with $h' = id \times p_1; h$.

Going the reverse direction, we insist that f be of the (ii)-recursive form and proceed to show it consequently also (i)-recursive. We start in similar manner with the diagram below, an augmented version of the preceding diagram, to construct a larger recursion diagram of the (i) form:

$$\begin{tikzcd}
& {} \arrow[d, "\vdots"] & {} \arrow[d, "\vdots"] & {} \arrow[d, "\vdots"] & \\
{} \arrow[r, "\cdots"] & 1 \times (N \times X) \arrow[r, "\mathit{zero} \times \mathit{id}"] \arrow[d, dotted, "\mathit{id}"] & N \times (N \times X) \arrow[d, dotted, "\langle \mathit{id} \times p_1; f, p_1; p_0 \rangle"] & N \times (N \times X) \arrow[l, "\mathit{succ} \times \mathit{id}"'] \arrow[d, dotted, "\langle \langle \mathit{id} \times p_1; f, p_0 \rangle, p_1 \rangle"] & {} \arrow[l, "\cdots"'] \\
{} \arrow[r, "\cdots"] & 1 \times (N \times X) \arrow[r, "\langle p_1; p_1; k, p_1; p_0 \rangle"] & N \times N & (N \times N) \times (N \times X) \arrow[l, "\langle p_0 \times id; h, p_1; p_0 \rangle"'] & {} \arrow[l, "\cdots"']
\end{tikzcd}$$

The final overall diagram's form forces extra projections to be paired with the bottom morphisms.

The *zero*-case expansion of this diagram is

$$\begin{tikzcd}
1 \times X \arrow[r, "\mathit{id}"] \arrow[dd, dotted, "\mathit{id}"] & 1 \times X \arrow[r, "\mathit{zero} \times \mathit{id}"] \arrow[d, "\mathit{id} \times \langle \mathit{zero}, \mathit{id} \rangle"] & N \times X \arrow[d, dotted, "\langle p_0, \mathit{id} \rangle"] & {} \arrow[l, "\cdots"'] \\
& 1 \times (N \times X) \arrow[r, "\mathit{zero} \times \mathit{id}"] \arrow[d, dotted, "\mathit{id}"] & N \times (N \times X) \arrow[d, dotted, "\langle \mathit{id} \times p_1; f, p_1; p_0 \rangle"] & {} \arrow[l, "\cdots"'] \\
1 \times X \arrow[r, "\langle p_0, \mathit{zero} \times \mathit{id} \rangle"] & 1 \times (N \times X) \arrow[r, "\langle p_1; p_1; k, p_1; p_0 \rangle"] & N \times N & {} \arrow[l, "\cdots"']
\end{tikzcd}$$

The *succ*-expansion becomes

$$\begin{tikzcd}
{} \arrow[r, "\cdots"] & N \times X \arrow[d, dotted, "\langle p_0, \mathit{id} \rangle"] & N \times X \arrow[l, "\mathit{succ} \times \mathit{id}"'] \arrow[d, "\langle p_0, \mathit{id} \rangle"] & N \times X \arrow[l, "\mathit{id}"'] \arrow[dd, dotted, "\langle \langle f, p_0 \rangle, p_1 \rangle"] \\
{} \arrow[r, "\cdots"] & N \times (N \times X) \arrow[d, dotted, "\langle \mathit{id} \times p_1; f, p_1; p_0 \rangle"] & N \times (N \times X) \arrow[l, "\mathit{succ} \times \mathit{id}"'] \arrow[d, dotted, "\langle \langle \mathit{id} \times p_1; f, p_0 \rangle, p_1 \rangle"] & \\
{} \arrow[r, "\cdots"] & N \times N & (N \times N) \times (N \times X) \arrow[l, "\langle p_0 \times \mathit{id}; h, p_1; p_0 \rangle"'] & (N \times N) \times X \arrow[l, "\langle p_0, p_1 \times \mathit{id} \rangle"']
\end{tikzcd}$$

Again, the outermost square along with the center vertical composition conforms to a recursion diagram, this time for $\langle f, p_0 \rangle$ in the context X:

$$\begin{tikzcd}
1 \times X \arrow[r, "\mathit{zero} \times \mathit{id}"] \arrow[d, dotted, "\mathit{id}"] & N \times X \arrow[d, dotted, "\langle f, p_0 \rangle"] & N \times X \arrow[l, "\mathit{succ} \times \mathit{id}"'] \arrow[d, dotted, "\langle \langle f, p_0 \rangle, p_1 \rangle"] \\
1 \times X \arrow[r, "\langle p_1; k, p_0; \mathit{zero} \rangle"] & N \times N & (N \times N) \times X \arrow[l, "\langle \langle p_0; p_0, p_1 \times \mathit{id} \rangle; h, p_0; p_1 \rangle"']
\end{tikzcd}$$

Thus $\langle f, p_0 \rangle$ is (i)-recursive. When it is post-composed by the projection p_0, f is produced as a (i)-recursive map.

9.2.7 Co-appending Augmented Colists is Monoidal

The *Colist*(A) datatype representing all infinite lists together with all non-empty finite lists was introduced at the end of Chapter 7 and utilized for rewrite rule examples in Chapter 8. Its definition is extended here to an augmented form, *Colist*$^+(A)$, that additionally includes the empty colist. This example shows that augmented colists are monoidal under an "append" operation that has the empty colist as its unit.

This new datatype is defined with a single destructor and a field datatype functor $E_1(A, C) = A \times C + 1$:

$$\begin{aligned} &\textbf{data } C \longrightarrow \mathit{Colist}^{+}(A) \ = \\ &\quad \mathit{split} : C \longrightarrow A \times C + 1. \end{aligned}$$

To gather our intuition on these lists, we recall that a conventional colist $\{a_1, a_2, a_3, \ldots\}$ is formed precisely as

$$(\mathit{cohead} : a_1, \mathit{cotail} : b_1(\mathit{cohead} : a_2, \mathit{cotail} : b_1(\mathit{cohead} : a_3, \mathit{cotail} : \ldots$$

and, if finite, would terminate as

$$\ldots \mathit{cotail} : b_0())\ldots) \quad .$$

An augmented colist can either be empty,

$$(\mathit{split} : b_1()) \quad ,$$

or begin as

$$(\mathit{split} : b_0(a_1, (\mathit{split} : b_0(a_2, (\mathit{split} : b_0(a_3, (\mathit{split} : \ldots$$

and, if finite, end as

$$\ldots (\mathit{split} : b_1())\ldots) \quad .$$

An append operation for colists (augmented or not) can be expressed as follows:

$$\mathit{coappend}(x, y) \equiv$$

$$(\!|\ (l_1, l_2) \mapsto \mathit{split} : \left\{ \begin{array}{l} b_0(a, l) \mapsto b_0(a, (l, l_2)) \\ b_1() \mapsto \left\{ \begin{array}{l} b_0(a, l) \mapsto b_0(a, (l_1, l)) \\ b_1() \mapsto b_1() \end{array} \right\} (\mathit{split}(l_2)) \end{array} \right\} (\mathit{split}(l_1))\ |\!)(x, y)$$

The nested *Sum*-case within *coappend* drives the colist generation firstly to replicate the first argument colist. If its replication completes, i.e. the first colist is finite, then replication continues by replicating the second colist and attaching it to the first. In essence, the first-occurring colist argument that is infinite dominates the *coappend* processing. If both arguments are finite colists, the *coappend* produces results isomorphic to those of the familiar *List*-append operation.

The monoidal properties to be proved are:

(i) $\mathit{coappend}(x, (\mathit{split} : b_1())) = x$

(ii) $\mathit{coappend}((\mathit{split} : b_1()), x) = x$

(iii) $\mathit{coappend}(x, \mathit{coappend}(y, z)) = \mathit{coappend}(\mathit{coappend}(x, y), z)$

The unfold uniqueness rule for $\mathit{Colist}^+(A)$ to be leveraged for this purpose is:

Colist$^+$(A):

$$\frac{\begin{array}{c} \text{if } \ t_1(c) \ =_{v_X} b_0(t_1'(c), t_1''(c)) \ \text{ then } \ \text{split}(t(c)) \ =_{v_X} b_0(t_1'(c), t(t_1''(c))) \\ \text{and} \\ \text{if } \ t_1(c) \ =_{v_X} b_1() \ \text{ then } \ \text{split}(t(c)) \ =_{v_X} b_1() \end{array}}{t(c) \ =_{v_X} \ (\!| \, v_C \mapsto \text{split} : t_1 \, |\!) \, (c)}$$

The right-unit property (i) is shown by using the finality diagram for the identity on augmented colists:

$$\begin{tikzcd}
\mathit{Colist}^+(A) \arrow[r, "\mathit{split}"] \arrow[d, dotted, "\mathit{id}"] & A \times \mathit{Colist}^+(A) + 1 \arrow[d, dotted, "(\mathit{id} \times \mathit{id}) + \mathit{id}"] \\
\mathit{Colist}^+(A) \arrow[r, "\mathit{split}"'] & A \times \mathit{Colist}^+(A) + 1
\end{tikzcd}$$

Consequently, employing unfold uniqueness for “t” as the left program of (i) and “$t_1(c)$” as $\mathit{split}(c)$ with respect to this diagram requires solving the sub-problems:

(1) if $\mathit{split}(l) = b_0(t_1'(l), t_1''(l))$ then

$$\mathit{split}(\mathit{coappend}(l, (\mathit{split} : b_1()))) = b_0(t_1'(l), \mathit{coappend}(t_1''(l), (\mathit{split} : b_1())))$$

(2) if $\mathit{split}(l) = b_1()$ then

$$\mathit{split}(\mathit{coappend}(l, (\mathit{split} : b_1()))) = b_1()$$

For showing (1), we make the indicated assumption for $\mathit{split}(l)$. Then we derive:

$\mathit{split}(\mathit{coappend}(l, (\mathit{split} : b_1())))$

$$\begin{aligned}
&= E_1\begin{bmatrix} w_A \mapsto w_A \\ (w_C, w'_C) \mapsto \mathit{coappend}(w_C, w'_C) \end{bmatrix} \left(\left\{ \begin{array}{l} b_0(a,l) \mapsto b_0(a,(l,l_2)) \\ b_1() \mapsto \left\{ \begin{array}{l} b_0(a,l) \mapsto b_0(a,(l_1,l)) \\ b_1() \mapsto b_1() \end{array} \right\} (b_1()) \end{array} \right\} (\mathit{split}(l)) \right) \\
&= E_1\begin{bmatrix} w_A \mapsto w_A \\ (w_C, w'_C) \mapsto \mathit{coappend}(w_C, w'_C) \end{bmatrix} \left(\left\{ \begin{array}{l} b_0(a,l) \mapsto b_0(a,(l,l_2)) \\ b_1() \mapsto b_1() \end{array} \right\} (b_0(t'_1(l), t''_1(l)) \right) \\
&= E_1\begin{bmatrix} w_A \mapsto w_A \\ (w_C, w'_C) \mapsto \mathit{coappend}(w_C, w'_C) \end{bmatrix} (b_0(t'_1(l), (t''_1(l), (\mathit{split} : b_1())))) \\
&= b_0(t'_1(l), \mathit{coappend}(t''_1(l), (\mathit{split} : b_1())))
\end{aligned}$$

The derivation for (2) is closely analogous to this one; it is left as an easy exercise. This completes the verification of the right-unit property. As expected, the proof of the left-unit property (*ii*) follows the same line of argument and is also left to the reader.

We finally arrive at the demonstration of associativity for *coappend*. The underlying *Colist*$^+$-recursion diagram for the left-hand program of (*iii*) that motivates the proof is pictured as the outermost square of the diagram below:

$$\begin{tikzcd}
\mathit{Colist}^+(A) \times (\mathit{Colist}^+(A) \times \mathit{Colist}^+(A)) \arrow[r, "t_1^*"] \arrow[d, dotted, "id \times \mathit{coappend}"] & A \times (\mathit{Colist}^+(A) \times (\mathit{Colist}^+(A) \times \mathit{Colist}^+(A))) + 1 \arrow[d, dotted, "id \times (id \times \mathit{coappend}) + id"] \\
\mathit{Colist}^+(A) \times \mathit{Colist}^+(A) \arrow[r, "t_1"] \arrow[d, dotted, "\mathit{coappend}"] & A \times (\mathit{Colist}^+(A) \times \mathit{Colist}^+(A)) + 1 \arrow[d, dotted, "id \times \mathit{coappend} + id"] \\
\mathit{Colist}^+(A) \arrow[r, "\mathit{split}"] & A \times \mathit{Colist}^+(A) + 1
\end{tikzcd}$$

The bottom square commutes since it is the defining finality square for *coappend* where t_1 is the morphism corresponding to

$$\{(l_1, l_2) \mapsto \left\{ \begin{array}{l} b_0(a,l) \mapsto b_0(a,(l,l_2)) \\ b_1() \mapsto \left\{ \begin{array}{l} b_0(a,l) \mapsto b_0(a,(l_1,l)) \\ b_1() \mapsto b_1() \end{array} \right\} (\mathit{split}(l_2)) \end{array} \right\} (\mathit{split}(l_1))\}$$

The t_1^* morphism in the top square is the specification morphism for the left-hand program of (iii) and corresponds to

$$\{(l_1,(l_2,l_3)) \mapsto \left\{\begin{array}{l} b_0(a,l) \mapsto b_0(a,(l,(l_2,l_3))) \\ b_1() \mapsto \left\{\begin{array}{l} b_0(a,l) \mapsto b_0(a,(l_1,(l,l_3))) \\ b_1() \mapsto \left\{\begin{array}{l} b_0(a,l) \mapsto b_0(a,(l_1,(l_2,l))) \\ b_1() \mapsto b_1() \end{array}\right\} (\mathit{split}(l_3)) \end{array}\right\} (\mathit{split}(l_2)) \end{array}\right\} (\mathit{split}(l_1))\}$$

A straightforward diagram chase by cases confirms the commutativity of the top square. The cases are:

1. $l_1 = b_0(a,l)$
2. $l_1 = b_1()$ and $l_2 = b_0(a,l)$
3. $l_1 = l_2 = b_1()$ and $l_3 = b_0(a,l)$
4. $l_1 = l_2 = l_3 = b_1()$

For example, the chase for case 3 looks like:

$$\begin{tikzcd}
(l_1,(l_2,l_3)) \arrow[r, "t_1^*"] \arrow[d, "id \times coappend"] & b_0(a,(l_1,(l_2,l))) \arrow[d, "id \times (id \times coappend) + id"] \\
(l_1, coappend(l_2,l_3)) = (l_1, b_0(a, coappend(l_2,l))) \arrow[r, "t_1"] & b_0(a,(l_1, coappend(l_2,l)))
\end{tikzcd}$$

So, at this point, the diagram has been confirmed to be a recursion diagram for $coappend(x, coappend(y,z))$. Applying the unfold uniqueness rule for "$t(x,(y,z))$" equal to $coappend(coappend(x,y),z)$ and "t_1" equal to t_1^* yields the sub-problems:

(i') if $t_1^*(l_1,(l_2,l_3)) = b_0(t_1'(l_1,(l_2,l_3)), t_1''(l_1,(l_2,l_3)))$ then

$$split(coappend(coappend(l_1,l_2),l_3)) = b_0(t_1'(l_1,(l_2,l_3)), t(t_1''(l_1,(l_2,l_3))))$$

(ii') if $t_1^*(l_1,(l_2,l_3)) = b_1()$ then

$$split(coappend(coappend(l_1,l_2),l_3)) = b_1()$$

For proving (i'), we see that taking the indicated assumption for t_1^* forces consideration of the first three cases (1-3) for l_1, l_2, and l_3 used earlier in showing commutativity of the recursion diagram's top square. Because the devivations in these cases are alike, only the derivation for case 2 ($l_1 = b_0(), l_2 = b_1(a,l)$) is shown below. This case yields $t_1^*(l_1,(l_2,l_3)) = b_0(a,(l_1,(l,l_3)))$, which in turn says that $t'_1(l_1,(l_2,l_3)) = a$ and $t''_1(l_1(l_2,l_3)) = (l_1,(l,l_3))$. Proceeding, we have

$$
\begin{aligned}
& \mathit{split}(\mathit{coappend}(\mathit{coappend}(l_1,l_2),l_3)) \\
={}& E_1[\ \ldots\ ]\left(\left\{\begin{array}{l} b_0(a,l) \mapsto b_0(a,(l,l_2)) \\ b_1() \mapsto \left\{\begin{array}{l} b_0(a,l) \mapsto b_0(a,(l_1,l)) \\ b_1() \mapsto b_1() \end{array}\right\}(\mathit{split}(l_3)) \end{array}\right\}(\mathit{split}(\mathit{coappend}(l_1,l_2)))\right) \\
={}& E_1[\ \ldots\ ]\left(\left\{\begin{array}{l} b_0(a,l) \mapsto b_0(a,(l,l_2)) \\ b_1() \mapsto \left\{\begin{array}{l} b_0(a,l) \mapsto b_0(a,(l_1,l)) \\ b_1() \mapsto b_1() \end{array}\right\}(\mathit{split}(l_3)) \end{array}\right\}(b_0(a,(\mathit{coappend}(l_1,l),l_3)))\right) \\
={}& E_1\left[\begin{array}{l} w_A \mapsto w_A \\ (w_C,w'_C) \mapsto \mathit{coappend}(w_C,w'_C) \end{array}\right](b_0(a,(\mathit{coappend}(l_1,l),l_3))) \\
={}& b_0(a,\mathit{coappend}(\mathit{coappend}(l_1,l),l_3))
\end{aligned}
$$

Establishing (ii') requires case 4 in which all three arguments are empty, producing a simple case and map reduction. Thus $Colist^+(A)$ is monoidal under *coappend*.

9.2.8 Generalized Naturality: Fold Parametricity

Reynolds' abstraction theorem [45] essentially states, under a precise definition of logical relations among sets, that related polymorphic functions produce related output values from related input values. This result has been specialized by Wadler [51] as a "free parametricity result" to give a more direct tool for proving program equivalences. Wadler's assertion that parametric theorems are "free" due to the apparent removal of any need for structural induction in their proofs has been challenged by Mairson [33]. The point of contention is Wadler's inclusion of a structurally inductive datatype (lists) into the definition of logical relations by which Wadler expresses his version of the abstraction theorem. In Mairson's mind, the question of whether an object in hand is actually a list always requires answering by using a list structural induction principle.

We take an intermediate tack on this issue. Our categorical setting allows many of the free theorems of Wadler to arise directly from categorical naturality. For others, we need only the weaker tool of primitive induction rather than structural induction. Such an example, list-fold parametricity in the sense of Wadler, is demonstrated here and shown to be another manifestation of naturality.

For reference we re-state Wadler's fold parametricity result for lists:

Theorem 9.2.4 List-Fold Parametricity *Assume within the category of* **Sets** *that* $a : A \longrightarrow A'$ *and* $c : C \longrightarrow C'$ *are functions expressing a functional relation between the indicated domains and codomains. Let* $\oplus : A \times C \longrightarrow C$ *and* $\oplus' : A' \times C' \longrightarrow C'$ *be, respectively, a* C*-action on* A *with unit* $u : C$ *and a* C'*-action on* A' *with unit* $u' : C'$*. Allow the two actions to be* (a, c)*-related in the sense that the following diagrams commute:*

$$\begin{tikzcd}
1 \arrow[r, "id"] \arrow[d, "u"] & 1 \arrow[d, "u'"] \\
C \arrow[r, "c"] & C'
\end{tikzcd}
\qquad
\begin{tikzcd}
A \times C \arrow[r, "a \times c"] \arrow[d, "\oplus"] & A' \times C' \arrow[d, "\oplus'"] \\
C \arrow[r, "c"] & C'
\end{tikzcd}$$

Then the liftings of the two actions into folding operations on lists are also (a, c)*-related, i.e. the diagram below commutes:*

$$\begin{tikzcd}
List(A) \arrow[r, "List(a)"] \arrow[d, "fold(u, \oplus)"] & List(A') \arrow[d, "fold(u', \oplus')"] \\
B \arrow[r, "c"] & B'
\end{tikzcd}$$

The "folds" expressed in the theorem are the conventional right-sided collapsing operations well-used within functional programming. We now repeat and then generalize this result in our strong setting. Two commutative box diagrams provide visual perspective for the reasoning. The triangular box below

$$\begin{tikzcd}
 & 1+A\times \mathit{List}(A) \arrow[d, "\langle \mathit{nil} \mid \mathit{cons}\rangle"] \arrow[ddl, "id+id\times \mathit{fold}(u,\oplus)"'] \arrow[ddr, dotted, "id+id\times \mathit{fold}(u',a\times id;\oplus')"] & \\
 & \mathit{List}(A) \arrow[ddl, "\mathit{fold}(u,\oplus)"] \arrow[ddr, dotted, "\mathit{fold}(u,a\times id;\oplus')"'] & \\
1+A\times C \arrow[rr, "id+id\times c"'] \arrow[d, "\langle u \mid \oplus\rangle"'] & & 1+A\times C' \arrow[d, "\langle u' \mid a\times id;\oplus'\rangle"] \\
C \arrow[rr, "c"'] & & C'
\end{tikzcd}$$

shows the left rear facet as a *List*-recursion diagram with u and $\oplus$ serving as the specification for $\mathit{fold}(u,\oplus)$. (Now "fold" is used in the generalized term logic sense.) The front facet is a compact restatement of the (a,c)-relationship between the units and actions. The combining of these two facets forms on the third facet a *List*-recursion diagram with u' and $a\times id;\oplus'$ specifying the list factorizer $\mathit{fold}(u',a\times id;\oplus')$.

The rectangular box below provides an alternative form of the triangular box's third facet, from which springs the desired program equivalence of Wadler's theorem.

$$\begin{tikzcd}
1+A\times \mathit{List}(A) \arrow[rr, "id+a\times \mathit{List}(a)"] \arrow[d, "\langle \mathit{nil} \mid \mathit{cons}\rangle"'] \arrow[ddr, dotted, "id+id\times(\mathit{List}(a);\mathit{fold}(u',\oplus'))"] & & 1+A'\times \mathit{List}(A') \arrow[d, "\langle \mathit{nil} \mid \mathit{cons}\rangle"'] \arrow[ddr, "id+id\times \mathit{fold}(u',\oplus')"] & \\
\mathit{List}(A) \arrow[rr, "\mathit{List}(a)"] \arrow[ddr, dotted, "\mathit{List}(a);\mathit{fold}(u',\oplus')"'] & & \mathit{List}(A') \arrow[ddr, "\mathit{fold}(u',\oplus')"'] & \\
 & 1+A\times C' \arrow[rr, "id+a\times id"'] \arrow[d, "\langle u' \mid a\times id;\oplus'\rangle"] & & 1+A'\times C' \arrow[d, "\langle u' \mid \oplus'\rangle"] \\
 & C' \arrow[rr, "id"'] & & C'
\end{tikzcd}$$

The front facet commutes by merely expressing the post-composing of a sum with a co-pair. The right facet is a *List*-recursion diagram specified by u' and $\oplus'$. The back facet is also a *List*-recursion diagram for the special specification — the constructors themselves — that yields the map operator for lists. Finally, the left facet can be shown to be commutative by a lengthy diagram chase using the other three facets.

Comparing the rectangular box's left facet with the triangular box's right rear facet yields, by primitive induction (fold uniqueness), our goal:

$$\mathit{List}(a); \mathit{fold}(u', \oplus') \;=\; \mathit{fold}(u, \oplus); c$$

.

The existence of the left facet to make the rectangular box commute was not serendipitous. Looking again, we see both boxes represent diagrams of morphisms among the $\langle 1, \mathit{Prod}\rangle$-algebras (the vertically-aligned arrows) in the inserter category for the functor pair $(\langle 1, \mathit{Prod}\rangle, \Delta \circ \mathit{Pr}_1)$, i.e. diagrams of *List*-algebras. Recall from Chapter 6 that $\mathcal{Q} = \mathit{Pr}_0 \circ U_L : \mathbf{Insert}(\langle 1, \mathit{Prod}\rangle, \Delta_2 \circ \mathit{Pr}_1) \longrightarrow \mathbf{A}$ is a fibration that maps each facet to its underlying parameter-changing $\mathbf{A}$-morphism.

There is also an underlying state-changing $\mathbf{C}$-morphism in each algebra morphism. Focusing on the rectangular box, we note that the $\mathbf{C}$-morphism for the front facet is an identity (on C'), making that facet a *cartesian arrow in the fibration* $\mathcal{Q}$. Since the composition of the underlying A-morphisms in the back and right faces factors through the underlying A-morphism ($a : A \longrightarrow A'$) of the front facet, then the fibration's image of the back and right facets factors through its image of the front facet. Thus, by the cartesian property, the left facet must uniquely exist to make the box commute. In particular, the fibration and the other three side facets directly determine how the left facet can be constructed.

We make two final observations. First, by instantiating the example with "c" as $\mathit{List}(c)$, "C" as $\mathit{List}(C)$, and "C'" as $\mathit{List}(C')$, we obtain the proof of naturality for list-formation. Second, the entire preceeding argument is directly transferrable to *any* initial datatype. Thus we are witnessing a generalization of naturality by fold parametricity.

9.2.9 Tree Summations

Does the composition of two fold (unfold) factorizers yield an fold (unfold) datatype factorizer? In the context-less definition of categorical datatypes using standard T-algebras, the question is affirmatively answered by a simple initiality (finality) argument. An instance of this occurs as Malcolm's "catamorphism promotability" theorem [34]. But the pervasive inclusion of context in our theory makes a similarly simple result problematic. Still, we show in this example that composing factorizers may lead to discovering a composition factorizer and thereby help establish a program equivalence.

Our example manipulates binary trees having natural numbers as leaves. The problem is comparing two ways of summing the leaf values. The program *addtree* does it by direct recusive decent:

$$addtree(t) \equiv \left\{\!\!\left|\begin{array}{l} bleaf : n \mapsto n \\ bnode : (n,m) \mapsto add(n,m) \end{array}\right|\!\!\right\}(t)$$

A second way is to flatten the tree into a list of natural numbers and then sum the list. The appropriate routines are presented below:

$$flatten(t) \equiv \left\{\!\!\left|\begin{array}{l} bleaf : a \mapsto [a] \\ bnode : (l_1,l_2) \mapsto append(l_1,l_2) \end{array}\right|\!\!\right\}(t)$$

$$sum(l) \equiv \left\{\!\!\left|\begin{array}{l} nil : () \mapsto zero() \\ cons : (n,m) \mapsto add(n,m) \end{array}\right|\!\!\right\}(l)$$

The standard addition of natural numbers is defined by:

$$add(n,m) \equiv \left\{\!\!\left|\begin{array}{l} zero : () \mapsto m \\ succ : v \mapsto succ(v) \end{array}\right|\!\!\right\}(n)$$

Thus the example problem under consideration is

$$sum(flatten(t)) = addtree(t) \quad .$$

From *addtree* being a fold factorizer, our problem translates to establishing the commutativity of the following diagram:

$$\begin{tikzcd}
Nat \arrow[r, "bleaf"] \arrow[d, "id"] & Btree(Nat) \arrow[d, "sum(flatten(\_))"] & Btree(Nat) \times Btree(Nat) \arrow[l, "bnode"'] \arrow[d, "sum(flatten(\_)) \times sum(flatten(\_))"] \\
Nat \arrow[r, "id"] & Nat & Nat \times Nat \arrow[l, "add"']
\end{tikzcd}$$

A *Btree*-fold reduction followed by a *List*-fold reduction validates the left square. For the right square, we note that

$$sum(flatten(bnode(t_1, t_2))) = sum(append(flatten(t_1), flatten(t_2)))$$

by a *Btree*-fold reduction. The desired result becomes immediate by showing the lemma,

$$sum(append(l_1, l_2)) = add(sum(l_1), sum(l_2)) \quad ,$$

holds for any lists l_1 and l_2.

The left side of the lemma is the composition of two *List* fold factorizers: $append(\_, l_2)$ and $sum(\_)$. We apply the composition to the *List* cases to see if it is also a *List* fold factorizer:

$$sum(append(nil(), l_2)) = sum(l_2)$$

$$\begin{aligned}
sum(append(cons(n, l), l_2)) &= sum(cons(n, append(l, l_2))) \\
&= add(n, sum(append(l, l_2)))
\end{aligned}$$

Indeed, the composition is in fold factorizer form with *sum* occurring recursively exactly in accordance with its associated initiality diagram, viz. the composition is specified by the two maps $p_1; sum(\_)$ and $p_0; add(\_, sum(\_))$. We are left with applying the right side of the lemma to the same cases to see whether it satifies the same specification. Using the easy-to-check associativity of add, we confirm below that the specification holds and thusly the right square commmutes:

$$\begin{array}{rcl} add(sum(nil()), sum(l_2)) & = & add(zero(), sum(l_2)) \\ & = & sum(l_2) \end{array}$$

$$\begin{array}{rcl} add(sum(cons(n, l)), sum(l_2)) & = & add(add(n, sum(l)), sum(l_2)) \\ & = & add(n, add(sum(l), sum(l_2))) \end{array}$$

The next chapter finally reveals that our successful examples are true rewards from this approach towards computable data structures: the term logic is shown to have exactly the strong-functor foundation that was assumed and leveraged here for establishing program equivalences.

Chapter 10

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.

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

(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)

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

(iii)

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

(iv)

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

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)

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

(iii)

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

(iv)

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

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

Lemma 10.1.1 *For the logic-to-combinator translation $\mathcal{C}$,*

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

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

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

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

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

$$\mathcal{C}\left\llbracket (v_L, v_X) \mapsto L \begin{bmatrix} w_1 \mapsto t_1 \\ \dots \\ w_m \mapsto t_m \end{bmatrix} (v_L) \right\rrbracket \; = \; map^L\{\mathcal{C}\llbracket (w_1, v_X) \mapsto t_1 \rrbracket), \dots, \mathcal{C}\llbracket (w_m, v_X) \mapsto t_m \rrbracket\}$$

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

$$\mathcal{C}\left\llbracket (v'_C, v_X) \mapsto \left(\left| v_C \mapsto \begin{array}{c} d_1 : t_1 \\ \dots \\ d_n : t_n \end{array} \right|\right) (v'_C) \right\rrbracket \; = \; unfold^R\{\mathcal{C}\llbracket (v_C, v_X) \mapsto t_1 \rrbracket, \dots, \mathcal{C}\llbracket (v_C, v_X) \mapsto t_n \rrbracket\}$$

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

$$\mathcal{C}\left\llbracket (v'_C, v_X) \mapsto \begin{pmatrix} d_1 : t_1(v'_C) \\ \dots \\ d_n : t_n(v'_C) \end{pmatrix} \right\rrbracket \; = \; record^R\{\mathcal{C}\llbracket (v_C, v_X) \mapsto t_1 \rrbracket, \dots, \mathcal{C}\llbracket (v_C, v_X) \mapsto t_n \rrbracket\}$$

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

$$\mathcal{C}\left\llbracket (v_R, v_X) \mapsto R \begin{bmatrix} w_1 \mapsto t_1 \\ \dots \\ w_m \mapsto t_m \end{bmatrix} (v_R) \right\rrbracket \; = \; map^R\{\mathcal{C}\llbracket (w_1, v_X) \mapsto t_1 \rrbracket, \dots, \mathcal{C}\llbracket (w_m, v_X) \mapsto t_m \rrbracket\}$$

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.

$$\begin{aligned}
&\mathcal{C}\left\llbracket (v_L, v_X) \mapsto \left\{\left| \begin{array}{c} c_1 : v_1 \mapsto t_1 \\ \dots \\ c_n : v_n \mapsto t_n \end{array} \right|\right\} (v_L) \right\rrbracket \; = \\
&= \; \langle \mathcal{C}\llbracket (v_L, v_X) \mapsto v_L \rrbracket, id \rangle \,; fold^L\{\dots, \mathcal{C}\llbracket (v_i, (v_L, v_X)) \mapsto t_i \rrbracket, \dots\} \\
&= \; \langle p_0, id \rangle \,; fold^L\{\dots, \mathcal{C}\llbracket (v_i, (v_L, v_X)) \mapsto t_i \rrbracket, \dots\} \\
&= \; \langle p_0, id \rangle \,; fold^L\{\dots, (id \times p_1) \,; \mathcal{C}\llbracket (v_i, v_X) \mapsto t_i \rrbracket, \dots\} \\
&= \; \langle p_0, id \rangle \,; (id \times p_1) \,; fold^L\{\dots, \mathcal{C}\llbracket (v_i, v_X) \mapsto t_i \rrbracket, \dots\} \\
&= \; fold^L\{\dots, \mathcal{C}\llbracket (v_i, v_X) \mapsto t_i \rrbracket, \dots\}
\end{aligned}$$

□

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 10.1.2 *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 [7]. However, due to the fact that the cartesian theory presented here is a slight generalization of the original one in [7], 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

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

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

$$\begin{aligned}
&= \langle \mathcal{C}\llbracket v \mapsto \sigma_{v':=t}(t_1)\rrbracket\ ,\ \mathcal{C}\llbracket v \mapsto t\rrbracket\rangle\,;\ldots \\
&= \mathcal{C}\llbracket v \mapsto (\sigma_{v':=t}(t_1)\ ,\ t)\rrbracket\,;\ldots \\
&= \mathcal{C}\llbracket v \mapsto \sigma_{(v_0,v'):=(\sigma_{v':=t}(t_1)\ ,\ t)}(t_0)\rrbracket \\
&= \mathcal{C}\llbracket v \mapsto \sigma_{v':=t}(\sigma_{(v_0,v'):=(t_1,v')}(t_0))\rrbracket \\
&= \mathcal{C}\llbracket v \mapsto \sigma_{v':=t}(\sigma_{v_0:=t_1}(t_0))\rrbracket \\
&= \mathcal{C}\llbracket v \mapsto \sigma_{v':=t}(\{v_0 \mapsto t_0\}(t_1))\rrbracket
\end{aligned}$$

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

$$\mathcal{C}\llbracket v \mapsto t\rrbracket\,;\mathcal{C}\llbracket v' \mapsto c_i(t_0)\rrbracket$$

$$\begin{aligned}
&= \mathcal{C}\llbracket v \mapsto t\rrbracket\,;\mathcal{C}\llbracket v' \mapsto t_0\rrbracket\,;c_i \\
&= \mathcal{C}\llbracket v \mapsto \sigma_{v':=t}(t_0)\rrbracket\,;c_i \\
&= \mathcal{C}\llbracket v \mapsto c_i(\sigma_{v':=t}(t_0))\rrbracket \\
&= \mathcal{C}\llbracket v \mapsto \sigma_{v':=t}(c_i(t_0)))\rrbracket
\end{aligned}$$

(ii) Let

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

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

$$\mathcal{C}\llbracket v \mapsto t\rrbracket\,;\mathcal{C}\left\llbracket v' \mapsto \left\{\left|\begin{array}{c} c_1 : v_1 \mapsto t_1 \\ \ldots \\ c_n : v_n \mapsto t_n \end{array}\right|\right\}(t_0)\right\rrbracket$$

$$\begin{aligned}
&= \mathcal{C}\llbracket v \mapsto t\rrbracket\,;\langle \mathcal{C}\llbracket v' \mapsto t_0\rrbracket, id\rangle\,;fold^L\{\mathcal{C}\llbracket (v_1,v') \mapsto t_1\rrbracket,\ldots,\mathcal{C}\llbracket (v_n,v') \mapsto t_n\rrbracket\} \\
&= \mathcal{C}\llbracket v \mapsto t\rrbracket\,;\langle \mathcal{C}\llbracket v' \mapsto t_*\rrbracket, id\rangle\,;\langle map^{E_j}\{p_0, fold^L\{\ldots\}\}, p_1\rangle\,;\mathcal{C}\llbracket (v_j,v') \mapsto t_j\rrbracket \\
&= \langle \mathcal{C}\llbracket v \mapsto \sigma_{v':=t}(t_*)\rrbracket\ ,\ \mathcal{C}\llbracket v \mapsto t\rrbracket\rangle\,;\langle map^{E_j}\{p_0, fold^L\{\ldots\}\}, p_1\rangle\,;\mathcal{C}\llbracket (v_j,v') \mapsto t_j\rrbracket \\
&= \mathcal{C}\llbracket v \mapsto (\sigma_{v':=t}(t_*),t)\rrbracket\,;\langle map^{E_j}\{p_0, fold^L\{\ldots\}\}, p_1\rangle\,;\mathcal{C}\llbracket (v_j,v') \mapsto t_j\rrbracket
\end{aligned}$$

$$\begin{aligned}
&= \quad \ldots\,;\langle \mathrm{map}^{E_j}\{\mathcal{C}\llbracket(w_A,v')\mapsto w_A\rrbracket,\mathcal{C}\left[\!\!\left[(w_L,v')\mapsto \left\{\!\!\left|\begin{array}{c} c_1\,:v_1\mapsto t_1\\ \ldots\\ c_n\,:v_n\mapsto t_n\end{array}\right|\!\!\right\}(w_L)\right]\!\!\right]\},p_1\rangle\,;\ldots\\
&= \quad \ldots\,;\langle\mathcal{C}\llbracket(v_j',v')\mapsto E_j\begin{bmatrix} w_A\mapsto w_A\\ w_L\mapsto \{\!|\ldots|\!\}(w_L)\end{bmatrix}(v_j')\rrbracket,p_1\rangle\,;\ldots\\
&= \quad \ldots\,;\mathcal{C}\llbracket(v_j',v')\mapsto (E_j\begin{bmatrix} w_A\mapsto w_A\\ w_L\mapsto \{\!|\ldots|\!\}(w_L)\end{bmatrix}(v_j'),v')\rrbracket\,;\ldots\\
&= \quad \mathcal{C}\llbracket v\mapsto \sigma_{(v_j',v'):=(\sigma_{v':=t}(t_*),t)}(E_j\begin{bmatrix} w_A\mapsto w_A\\ w_L\mapsto \{\!|\ldots|\!\}(w_L)\end{bmatrix}(v_j'),v')\rrbracket\,;\ldots\\
&= \quad \mathcal{C}\llbracket v\mapsto \sigma_{v':=t}(\sigma_{(v_j',v'):=(t_*,v')}(E_j\begin{bmatrix} w_A\mapsto w_A\\ w_L\mapsto \{\!|\ldots|\!\}(w_L)\end{bmatrix}(v_j'),v'))\rrbracket\,;\ldots\\
&= \quad \mathcal{C}\llbracket v\mapsto \sigma_{v':=t}(E_j\begin{bmatrix} w_A\mapsto w_A\\ w_L\mapsto \{\!|\ldots|\!\}(w_L)\end{bmatrix}(t_*),t)\rrbracket\,;\ldots\\
&= \quad \mathcal{C}\llbracket v\mapsto \sigma_{(v_j,v'):=(\sigma_{v':=t}(E_j[\ldots](t_*),t)}(t_j)\rrbracket\\
&= \quad \mathcal{C}\llbracket v\mapsto \sigma_{v':=t}(\sigma_{v_j:=E_j[\ldots](t_*)}(t_j))\rrbracket\\
&= \quad \mathcal{C}\left[\!\!\left[v\mapsto \sigma_{v':=t}(\left\{\!\!\left|\begin{array}{c} c_1\,:v_1\mapsto t_1\\ \ldots\\ c_n\,:v_n\mapsto t_n\end{array}\right|\!\!\right\}(t_0))\right]\!\!\right]
\end{aligned}$$

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

$$t' \;=\; \left(\!\!\left|\, v_C\mapsto \begin{array}{c} d_1\,:t_1\\ \ldots\\ d_n\,:t_n\end{array}\right|\!\!\right)(t_0)$$

Then

$$\mathcal{C}\llbracket v\mapsto t\rrbracket\,;\mathcal{C}\left[\!\!\left[v'\mapsto \left(\!\!\left|\, v_C\mapsto \begin{array}{c} d_1\,:t_1\\ \ldots\\ d_n\,:t_n\end{array}\right|\!\!\right)(t_0)\right]\!\!\right]\;;d_j$$

$$\begin{aligned}
&= \quad \mathcal{C}\llbracket v\mapsto t\rrbracket\,;\langle\mathcal{C}\llbracket v'\mapsto t_0\rrbracket,id\rangle\,;\mathit{unfold}^R\{\mathcal{C}\llbracket(v_C,v')\mapsto t_1\rrbracket,\ldots,\mathcal{C}\llbracket(v_C.v')\mapsto t_n\rrbracket\}\,;d_j\\
&= \quad \langle\mathcal{C}\llbracket v\mapsto \sigma_{v':=t}(t_0)\rrbracket,\mathcal{C}\llbracket v\mapsto t\rrbracket\rangle\,;\langle\mathcal{C}\llbracket(v_C,v')\mapsto t_j\rrbracket,p_1\rangle\,;\mathrm{map}^{E_j}\{p_0,\mathit{unfold}\{\ldots\}\}
\end{aligned}$$

$$
\begin{aligned}
&= \mathcal{C}\llbracket v \mapsto (\sigma_{v':=t}(t_0), t)\rrbracket \,; \mathcal{C}\llbracket (v_C, v') \mapsto (t_j, v')\rrbracket \,; \ldots \\
&= \mathcal{C}\llbracket v \mapsto \sigma_{(v_C, v'):=(\sigma_{v':=t}(t_0), t)}(t_j, v')\rrbracket \,; \ldots \\
&= \mathcal{C}\llbracket v \mapsto \sigma_{v':=t}(\sigma_{v_C:=t_0}(t_j), v')\rrbracket \,; \ldots \\
&= \ldots \,; \mathrm{map}^{E_j}\{\mathcal{C}\llbracket (w_A, v') \mapsto w_A\rrbracket, \mathcal{C}\llbracket (w_L, v') \mapsto \left(\middle| v_C \mapsto \begin{array}{r} d_1 : t_1 \\ \ldots \\ d_n : t_n \end{array} \middle|\right)(w_L)\rrbracket\} \\
&= \ldots \,; \mathcal{C}\left[\!\!\left[(v_j, v') \mapsto E_j \begin{bmatrix} w_A \mapsto w_A \\ w_L \mapsto (\!|\ldots|\!)(w_L) \end{bmatrix} (v_j) \right]\!\!\right] \\
&= \mathcal{C}\llbracket v \mapsto \sigma_{(v_j, v'):=(\sigma_{v':=t}(\sigma_{v_C:=t_0}(t_j)), t)}(E_j[\ldots](v_j))\rrbracket \\
&= \mathcal{C}\llbracket v \mapsto \sigma_{v':=t}(\sigma_{v_j:=\sigma_{v_C:=t_0}(t_j)}(E_j[\ldots](v_j)))\rrbracket \\
&= \mathcal{C}\left[\!\!\left[v \mapsto \sigma_{v':=t}(E_j \begin{bmatrix} w_A \mapsto w_A \\ w_C \mapsto \left(\middle| v_C \mapsto \begin{array}{r} d_1 : t_1 \\ \ldots \\ d_n : t_n \end{array} \middle|\right)(w_C) \end{bmatrix} (\sigma_{v_C:=t_0}(t_j))) \right]\!\!\right] \\
&= \mathcal{C}\left[\!\!\left[v \mapsto \sigma_{v':=t}(d_j(\left(\middle| v_C \mapsto \begin{array}{r} d_1 : t_1 \\ \ldots \\ d_n : t_n \end{array} \middle|\right)(t_0))) \right]\!\!\right] \\
&= \mathcal{C}\left[\!\!\left[v \mapsto \sigma_{v':=t}(\left(\middle| v_C \mapsto \begin{array}{r} d_1 : t_1 \\ \ldots \\ d_n : t_n \end{array} \middle|\right)(t_0)) \right]\!\!\right] ; d_j
\end{aligned}
$$

□

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*$^{\mathit{Sum}}$, *case*$^{\mathit{Sum}}$, and *map*$^{\mathit{Sum}}$ 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 10.1.3 *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 [7] 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 $\mathit{fold}^{\mathit{Sum}}\{f_1, f_2\}$ and $\mathit{case}^{\mathit{Sum}}\{f_1, f_2\}$. 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 $\mathit{map}^{\mathit{Sum}}\{f_1, f_2\}$ and $\mathit{case}^{\mathit{Sum}}\{f_1\,;b_0, f_2\,;b_1\}$. These two morphisms can be quickly verified to be equal in a predistributive category. □

We now state the major theorem of this section:

Proposition 10.1.4 *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 10.1.2. 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:

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

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

(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:

$$
\mathcal{C} \left\llbracket v_X \mapsto d_i(\left(\left| v_C \mapsto \begin{array}{c} d_1 : t_1 \\ \ldots \\ d_n : t_n \end{array} \right|\right) (t)) \right\rrbracket
$$

$$
= \quad \mathcal{C} \left\llbracket v_X \mapsto \left(\left| v_C \mapsto \begin{array}{c} d_1 : t_1 \\ \ldots \\ d_n : t_n \end{array} \right|\right) (t) \right\rrbracket \,;\, d_i
$$

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

(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.

□

10.2 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) $\mathcal{P}[\![f]\!] \equiv \{v \mapsto \mathcal{P}_v[\![f]\!]\}$ 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) $\mathcal{P}[\![id_T]\!] = \{v_T \mapsto v_T\}$

(iv) $\mathcal{P}[\![!_T]\!] = \{v_T \mapsto ()\}$

(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) $\mathcal{P}\llbracket c_i \rrbracket = \{v \mapsto c_i(v)\}$ where v is a variable base for the domain of c_i

$$\text{(ii)}\quad \mathcal{P}\llbracket \mathit{fold}^L\{h_1, \ldots, h_n\} \rrbracket = \{(v_L, v_X) \mapsto \left\{\left|\begin{array}{c} c_1 : v_1 \mapsto \mathcal{P}\llbracket h_1 \rrbracket(v_1, v_X) \\ \ldots \\ c_n : v_n \mapsto \mathcal{P}\llbracket h_n \rrbracket(v_n, v_X) \end{array}\right|\right\}(v_L)\}$$

$$\text{(iii)}\quad \mathcal{P}\llbracket \mathit{case}^L\{h_1, \ldots, h_n\} \rrbracket = \{(v_L, v_X) \mapsto \left\{\begin{array}{c} c_1(v_1) \mapsto \mathcal{P}\llbracket h_1 \rrbracket(v_1, v_X) \\ \ldots \\ c_n(v_n) \mapsto \mathcal{P}\llbracket h_n \rrbracket(v_n, v_X) \end{array}\right\}(v_L)\}$$

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

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

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

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

$$\text{(iii)}\quad \mathcal{P}\llbracket \mathit{record}^R\{g_1, \ldots, g_n\} \rrbracket = \{(v'_C, v_X) \mapsto \left(\begin{array}{c} d_1 : \mathcal{P}\llbracket g_1 \rrbracket(v'_C, v_X) \\ \ldots \\ d_n : \mathcal{P}\llbracket g_n \rrbracket(v'_C, v_X) \end{array}\right)\}$$

$$\text{(iv)}\quad \mathcal{P}\llbracket \mathit{map}^R\{f_1, \ldots, f_m\} \rrbracket = \{(v_R, v_X) \mapsto R\left[\begin{array}{c} w_1 \mapsto \mathcal{P}\llbracket f_1 \rrbracket(w_1, v_X) \\ \ldots \\ w_m \mapsto \mathcal{P}\llbracket f_m \rrbracket(w_m, v_X) \end{array}\right](v_R)\}$$

Theorem 10.2.1 *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 [7] 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:

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

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

L fold factorizer uniqueness:

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

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

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 $\mathit{fold}^L\{h_1, \ldots, h_n\}$ are equal.

R unfold factorizer reduction:

$$\mathcal{P}\llbracket \mathit{unfold}^R\{g_1, \ldots, g_n\} \,;\, d_i \rrbracket$$

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

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 10.2.2 *$\mathcal{C}$ 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. This induction is not to be confused with categorical structural induction as described early in Chapter 9. Rather, the induction is a conventional syntactical one based on sub-terms that form complete combinator expressions themselves. The cartesian cases were proved in [7]. The remaining non-trivial cases — the fold and unfold factorizers — both follow exactly the pattern of proof for the fold factorizer:

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

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:

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

$$\begin{aligned}
&= \langle \mathcal{P}[\![\mathcal{C}[\![v_X \mapsto t]\!]]\!], \mathcal{P}[\![id]\!]\rangle\,;\{(v_L, v_X) \mapsto \left\{\left|\begin{array}{c} \cdots \\ c_i : v_i \mapsto \mathcal{P}[\![\mathcal{C}[\![(v_i, v_X) \mapsto t_i]\!]]\!](v_i, v_X) \\ \cdots \end{array}\right|\right\}(v_L)\} \\
&= \langle \{v_X \mapsto t\}, \{v_X \mapsto v_X\}\rangle\,;\{(v_L, v_X) \mapsto \left\{\left|\begin{array}{c} \cdots \\ c_i : v_i \mapsto \{(v_i, v_X) \mapsto t_i\}(v_i, v_X) \\ \cdots \end{array}\right|\right\}(v_L)\} \\
&= \{v_X \mapsto \left\{\left|\begin{array}{c} \cdots \\ c_i : v_i \mapsto \{(v_i, v_X) \mapsto t_i\}(v_i, v_X) \\ \cdots \end{array}\right|\right\}(t)\} \\
&= \{v_X \mapsto \left\{\left|\begin{array}{c} \cdots \\ c_i : v_i \mapsto t_i \\ \cdots \end{array}\right|\right\}(t)\}
\end{aligned}$$

□

10.3 An Alternative View: Precartesian Sketches

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 8.1 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 this chapter, 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 [55, 54]. The complete development of this particular method is presented in [10].

Chapter 11

Strong Datatype Programming

This section will provide concreteness for the term logic formalization by introducing a variety of oft-used fundamental strong datatypes, and some computation rule examples. We then consolidate our development with an application code example: database accessing.

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.

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

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

have been employed as abstract syntax trees for language compilation.

Booleans

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

Natural Numbers

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

Finite Lists

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

Finite Binary Trees

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

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

Infinite Binary Trees

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

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

Infinite Lists

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

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

Finite Products

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

Finite Sums

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

Bushes

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

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

11.1 The Programming Paradigm

The programming design paradigm that fits well here is an inductive one that transforms initial states into a succession of states. We will illustrate it by presenting the set of computation rules for two particular strong datatypes: the initial type DBTree and the final type Colist. These rules, as well as the rules for the other common datatypes, are easily derivable from the axioms defined in Chapter 8.

The rules for DBTree are listed below. All context variables have been left implicit in order to focus on the essential computation.

- Fold-from-DBTree:

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

$$\left\{\!\left|\begin{array}{l} \text{dbleaf} : v_A \mapsto t_1(v_A) \\ \text{dbnode} : (v_B,(v_C,v'_C)) \mapsto t_2(v_B,v_C,v'_C) \end{array}\right|\!\right\}(\text{dbnode}(b,(t,t'))) \implies t_2(b,\{\!|\ldots|\!\}(t),\{\!|\ldots|\!\}(t'))$$

- Case-from-DBTree:

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

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

- Map-on-DBTree:

$$\text{DBTree}\begin{bmatrix} w_A \mapsto t_1(w_A) \\ w_B \mapsto t_2(w_B) \end{bmatrix}(\text{dbleaf}(a)) \implies \text{dbleaf}(t_1(a))$$

$$\text{DBTree}\begin{bmatrix} w_A \mapsto t_1(w_A) \\ w_B \mapsto t_2(w_B) \end{bmatrix}(\text{dbnode}(b,(t,t'))) \implies \text{dbnode}(t_2(b),(\text{DBTree}[\ldots](t),\text{DBTree}[\ldots](t')))$$

The Fold-from-DBTree operation computes "initial" states with t_1 from basic data at the leaves and feeds these state values to the second state transform that uses t_2 to create succeeding states. The recursion is clearly seen by noting the result of this operator being applied to a node — the result is expressed by re-applying this operation to both the left and the right sub-trees (t and t') supplied by the operand. It is the programmer's option to design an appropriate state datatype C as well as t_1 and t_2 to achieve the desired result of type C. Thus the computation should be viewed as inductively driven by the structure of the DBTree element being applied to by this operation.

The Case-from-DBTree operation performs exactly one transformation on the operand DBTree element according to whether its outermost structure is that of a leaf or a node.

If the element is a leaf, t_1 uses its leaf datum to build a result. For a node, t_2 is selected to compute from the node datum and its subtrees. Since only one pass is dictated, t_1 and t_2 must belong to the component types that comprise DBTree.

The Map-on-DBTree operation performs t_1 upon all leaf data and t_2 upon all node data to create a new DBTree structure whose leaves and nodes are possibly of new types. Again the recursion of the map operator is seen by its re-application to the sub-trees of an operand node.

Moving from the initial side to the final side of the programming paradigm, we now display the rules for the Colist datatype:

- Unfold-to-Colist:

$$\mathrm{cohead}(\left(\!\left|\; v_C \mapsto \begin{array}{r} \mathrm{cohead} : t_1(v_C) \\ \\ \mathrm{cotail} : t_2(v_C) \end{array} \right|\!\right)(t)) \quad \Longrightarrow \quad t_1(t)$$

$$\mathrm{cotail}(\left(\!\left|\; v_C \mapsto \begin{array}{r} \mathrm{cohead} : t_1(v_C) \\ \\ \mathrm{cotail} : b_0(t'_2(v_C)) \end{array} \right|\!\right)(t)) \quad \Longrightarrow \quad b_0(t'_2(t))$$

$$\mathrm{cotail}(\left(\!\left|\; v_C \mapsto \begin{array}{r} \mathrm{cohead} : t_1(v_C) \\ \\ \mathrm{cotail} : b_1(t'_2(v_C)) \end{array} \right|\!\right)(t)) \quad \Longrightarrow \quad b_1((\!|\ldots|\!)(t'_2(t)))$$

- Record-to-Colist:

$$\mathrm{cohead}(\left(v_C \mapsto \begin{array}{r} \mathrm{cohead} : t_1(v_C) \\ \\ \mathrm{cotail} : t_2(v_C) \end{array} \right)(t)) \quad \Longrightarrow \quad t_1(t)$$

$$\mathrm{cotail}(\left(v_C \mapsto \begin{array}{r} \mathrm{cohead} : t_1(v_C) \\ \\ \mathrm{cotail} : b_0(t'_2(v_C)) \end{array} \right)(t)) \quad \Longrightarrow \quad b_0(t'_2(t))$$

$$\mathrm{cotail}(\left(v_C \mapsto \begin{array}{r} \mathrm{cohead} : t_1(v_C) \\ \\ \mathrm{cotail} : b_1(t'_2(v_C)) \end{array} \right)(t)) \quad \Longrightarrow \quad b_1(t'_2(t))$$

- Map-on-Colist:

$$\text{cohead}(\text{Colist}\left[w_A \mapsto t(w_A)\right](\text{cohead} : t_1, \text{cotail} : t_2)) \implies t(t_1)$$

$$\text{cotail}(\text{Colist}\left[w_A \mapsto t(w_A)\right](\text{cohead} : t_1, \text{cotail} : b_0(t_2'))) \implies b_0(t(t_2'))$$

$$\text{cotail}(\text{Colist}\left[w_A \mapsto t(w_A)\right](\text{cohead} : t_1, \text{cotail} : b_1(t_2'))) \implies b_1(\text{Colist}.[].(t_2'))$$

The Unfold-to-Colist operation generates a record-like structure have two "field" labels : cohead and cotail. The operand t serves on the first pass as a seed state value for the variable v_C to compute *simultaneously* the first set of field values. In the particular case that a computed field value contains a component of the seed's type, i.e. C, then the operation is re-applied to that component value to create a record-within-a-field having the cohead and cotail field labels. The rules show this situation occurs when the field term has the form $b_1(t')$. If a computed field value has no seed-type component — as in the remaining Unfold-to-Colist rules — the field is fully evaluated, thereby ending recursion in that field. So this operation can be visualized as a set of state transformations — one per field — operating in unison to create successively more complete approximations of the record structure on each pass.

Since the recursion within fields may continue without end, a Colist can be infinitary. Any implementation must consequently evaluate this operation lazily as needed by the program. This infinitary aspect is prevalent among final datatypes.

The Record-to-Colist produces a record-like structure from a seed value in a single pass. This necessitates the supplied field terms to already be of the field type of the Colist datatype, i.e. the first approximation has the correct field types.

And in the same manner as for the Map-on-DBTree operation, the Map-on-Colist operation creates a new Colist structure, possibly of a new parameter type, where the base data of type A has been replaced wherever it occurs by the image of that data under t. The notation

$$(\text{cohead} : t_1, \text{cotail} : t_2)$$

means a Colist structure whose cohead projection will yield t_1 and whose cotail projection will give t_2. The reader may note that the second map rule can be simplified further since $b_0(t'_2) = b_0(t(t'_2)) = b_0(())$, the only term in the summand 1 of the datatype $1 + C$.

The reader is encouraged to derive the rules for other datatypes, particularly the simple ones of Nat and List, to see that they all conceptually follow the same general procedures of our examples.

11.2 A Programming Example

A strong datatype application case study is outlined in this section. It is intended to show the general capabilities of the term logic, especially in broad-based applications. Our example is the familiar one of database access.

First, the datatype chosen to store the keyed data in a format reminscent of an ordered B^+-tree is declared below. Storage takes place in "levels", a level being that part of the database at constant depth from the database root. A piece of data in a level may be null, a key-datum pair, or a list of ordered-staggered keys paired with sub-stores. Each sub-store contains all the data having keys related by the key ordering in a pre-defined way with its associated staggered key.

$$\begin{array}{ll}\textbf{data} & C \longrightarrow \mathrm{Datastore}(\mathrm{Key}, \mathrm{Data}) \quad = \\ & \mathrm{level} : C \longrightarrow 1 + (\mathrm{Key} \times \mathrm{Data}) + \mathrm{List}(\mathrm{Key} \times C).\end{array}$$

For access efficiency, we wish to assume an insertion/deletion strategy (implementation not shown) that reasonably balances the database store and maintains the database's maximum depth. Thus we declare a database to be simply a record that allows access to both the datastore and the current depth. This depth will be the principal data structure (a natural number) that drives the access program.

$$\begin{array}{ll}\textbf{data} & C \longrightarrow \mathrm{Database}(\mathrm{Key}, \mathrm{Data}) \quad = \\ & \mathrm{store} : C \longrightarrow \mathrm{Datastore}(\mathrm{Key}, \mathrm{Data}) \\ & \mathrm{depth} : C \longrightarrow \mathrm{Nat}.\end{array}$$

A few preliminary definitions will make the coming procedural code clearer. We will let null_data be the "empty" datum nil where Data is assumed to be some List(A) datatype. Similarly we define an empty datastore and an empty database:

$$\textbf{def} \quad \text{null\_ds} = (\text{level} : b_0(())).$$

$$\textbf{def} \quad \text{null\_db} = (\text{depth} : \text{zero}, \text{store} : \text{null\_ds}).$$

We intend to search a database to retrieve a datum that has been paired with the argument key. If the key cannot be matched, the empty datum is returned. The search also presumes some ordering of the keys to narrow the search universe to a sub-store as we descend from one level to the next. For simplicity we will assume an "compare" predicate and an "equality" predicate are available from context to do the narrowing and matching. The search can now be implemented as follows:

$$\textbf{def}\ \text{search}(\text{key}, \text{db}) =$$

$$p_0\left(\left\{\left|\begin{array}{l} \text{zero} : () \mapsto (\text{null\_data}, \text{store}(\text{db})) \\ \text{succ} : (v_{\text{data}}, v_{\text{ds}}) \mapsto \left\{\begin{array}{l} \text{nil} \mapsto \text{scan\_store\_level}(\text{level}(v_{\text{ds}}), \text{key}) \\ \text{cons}(\_, \_) \mapsto (v_{\text{data}}, v_{\text{ds}}) \end{array}\right\}(v_{\text{data}}) \end{array}\right|\right\}(\text{depth}(\text{db}))\right).$$

Starting from zero, the depth drives the Fold-from-Nat to produce an initial state of the null datum paired with the datestore of the passed database (db) found by the store projection. The state represents the pairing of datum-found with the current sub-store to be searched. The second phrase of the fold operation is invoked by the succ constructors in the depth value. This phrase applies a Case-from-List operation to the found-datum state to create the next state. If the datum state is still null, i.e. not yet found, then the routine scan_store_level is applied to the top level of the current1 sub-store state. Scan_store_level creates a new state for the next fold pass by finding either the required datum or the next sub-store in the passed current sub-store level. If the datum has already been found, i.e. it has a cons as its outermost constructor, the cons phrase will simply pass along the found-datum and next sub-store states without change to form the next state. (The cons' arguments that would hold the found datum itself have been indicated as "don't cares".) The search ends when the last succ constructor of depth has been processed and the found datum, if any, has been projected out of the final state with p_0.

The scan_store_level routine, shown below, expects a level projection of a datastore. The outer Case-from-Sum operation distinguishes a null level from a non-null level. In the null-level (b_0) case, a null datum - null datastore pair is returned. The non-null-level case forces an inner Case-from-Sum operation to be applied to the level component that could be either a key-datum pair or a listed of keyed sub-stores. If the component is a key-datum pair, a key-match is attempted by the equality predicate. The Case-from-Bool acts on the predicate's outcome as an if-then-else clause to either return the key-matched datum or a null-datum. If the level component is a list, the routine scan_key_datastore_list is invoked to find the sub-store in the list to serve as the next one to be searched.

def scan_store_level(ds_level, key) =

$$\left\{\begin{array}{l} b_0(v) \mapsto (\mathrm{null\_data}, \mathrm{null\_ds}) \\ b_1(v) \mapsto \left\{\begin{array}{l} b_0(v_{\mathrm{kd}}) \mapsto \left\{\begin{array}{l} \mathrm{true} \mapsto (p_1(v_{\mathrm{kd}}), \mathrm{null\_ds}) \\ \mathrm{false} \mapsto (\mathrm{null\_data}, \mathrm{null\_ds}) \end{array}\right\} (\mathrm{equality}(p_0(v_{\mathrm{kd}}), \mathrm{key})) \\ b_1(v_{\mathrm{list}}) \mapsto \mathrm{scan\_key\_datastore\_list}(v_{\mathrm{list}}, \mathrm{key}) \end{array}\right\} (v) \end{array}\right\} (\mathrm{ds\_level})$$

Locating the next sub-store presumes that the keys of the list of key - sub-store pairs are ordered in some fashion to narrow the search. The concerned routine below is driven by this list to successively compare the passed key with the staggered keys to find the desired key range. The initial state is set to a null-datastore and range-not-found pair. If the proper range is found, the routine bypasses further comparisons by setting the found state and returns the corresponding sub-store to the search for scanning. Otherwise, the null datastore is returned.

$$\mathbf{def}\ \mathrm{scan\_key\_datastore\_list}(\mathrm{kdsl}, \mathrm{key}) =$$

$$p_0(\left(\!\left|\begin{array}{l} \mathrm{nil} \mapsto (\mathrm{null\_ds}, \mathrm{false}) \\ \mathrm{cons} : (v_{\mathrm{kds}}, (v_{\mathrm{ds}}, v_{\mathrm{fnd}})) \mapsto \left\{\begin{array}{l} \mathrm{true} \mapsto (p_1(v_{\mathrm{kds}}), \mathrm{true}) \\ \mathrm{false} \mapsto (v_{\mathrm{ds}}, v_{\mathrm{fnd}}) \end{array}\right\} (\mathrm{and}(\mathrm{compare}(\mathrm{key}, p_0(v_{\mathrm{kds}})), \mathrm{not}(v_{\mathrm{fnd}})) \end{array}\right|\!\right) (\mathrm{kdsl}))$$

This example shows the typical combination of 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.

Chapter 12

Related Work and Conclusions

We have described a consistent and functionally complete term logic with support for strong datatypes which serves as a categorical programming language. Such a categorical programming language has been implemented: **Charity** is currently an operational testbed for categorical programming and uses a mixture of eager initial and lazy final datatypes (Cockett and Fukushima [11]). As a programming language it is remarkably useable (considering its formal nature and the fact that all programs strongly normalize). The arithmetic strength of the setting is weaker than most comparable systems (as there is no exponentiation) but does allow, for example, the definition of Ackermann's function.

Categorical computation remains relatively inefficient compared even to conventional functional languages, so techniques such as optimizing the translation of the term logic to combinators in order to speed up execution will be important. Recent intriguing results by Barry Jay [25] on the efficiency of iteration within categorical computation may help illuminate such optimizations. The discovery of additional specializations of the fold and unfold factorizers to outfit a categorical programmer's "toolkit" in the sense hinted by David Turner [50] would further this goal. Also, optimizing via program transformations similar to those developed by Malcolm [35, 34] for general Hagino datatypes may be feasible. Additionally, the current research surge in functional programming analysis from the perspectives of control, continuations, and partial evaluation might well be worthwhile in this setting.

Cockett's recent investigations [9] indicate that attractive abstract data structure specifications, such as those espoused by Walters [52], are highly applicable to this programming setting. Also, preliminary investigations have begun towards incorporating dependent types into the system which would yield further improvements in abstract data structure expressiveness.

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.

Others, of course, have investigated calculi based in initial (inductive) and final (co-inductive) datatypes. The interested reader should compare this work with Malcolm's syntactic calculus [34] and Greiner's variant of the polymorphic lambda calculus [16]. Categorical initial and final algebras underlie both well-developed treatments, but neither grapples with the environmental scoping problem nor reaches a user-programming language level that supports datatype declarations.

Future explorations may include finding the precise role of parametricity within our settings and its connection to generalizing naturality. This interest is motivated by

Hasegawa's work [18, 19] that asserted the equivalence of parametricity to the presence of categorical products and coproducts in models of polymorphic calculi. Because our declaration style had been mapped by Wraith into a model of the polymorphic lambda calculus and since our setting is closed under true products and coproducts, we expect formalized parametricity to be demonstrable for our category of strong datatypes.

Another future interest is the highly concurrent semantics of the unfold operations. These operators may allow the term logic's adaptation to parallel functional programming language design and particularly for operating systems design.

On another front, language expressibility might be enhancible by moving some aspects of categorical logic in categorical programming, e.g. dependent types and type classes. Preliminary work on a indexed category-based combinator reduction system featuring the latter has been reported by Hilken and Rydeheard [21, 22].

Finally, in the tradition of many earlier categorically based works, we, too, take our leave with slogans that summarize the motivation and justification of this effort:

A category can be not only a model of computation but also a medium.
Categorical datatypes always satisfy standard models.

Bibliography

[1] A. Asperti and G. Longo. *Categories, Types, and Structures.* MIT Press, 1991.

[2] M. Barr and C. Wells. *Category Theory for Computing Science.* Prentice-Hall, 1990.

[3] J. Bénabou. Fibrations petites et localement petites. *C. R. Acad. Sc. Paris*, 281:A897–A900, 1975.

[4] J. Bènabou. Fibered categories and the foundations of naive category theory. *Journal of Symbolic Logic*, 50(1):10–37, 1985.

[5] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, 1977.

[6] J. Cartmell. Generalized algebraic theories and contextual categories. *Annals of Pure and Applied Logic*, 32:209–243, 1986.

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

[8] J. R. B. Cockett. List-arithmetic distributive categories: Locoi. *Journal of Pure and Applied Algebra*, 66:1–29, 1990.

[9] J. R. B. Cockett. Conditional control is not quite categorical control. In Graham Birtwistle, editor, *IV Higher Order Workshop, Banff*, Workshops in Computing, pages 190–217. Springer-Verlag, 1991.

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

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

[12] J. R. B. Cockett and D. Spencer. Strong categorical datatypes I. In R. A. G. Seely, editor, *International Meeting on Category Theory 1991*, volume 13 of *Canadian Mathematical Society Proceedings*, pages 141–169. AMS, Montreal, 1992.

[13] J. A. Goguen and R. D. Burstall. Institutions: Abstract model theory for specification and programming. *J. A. C. M.*, 39(1):95–146, January 1992.

[14] J. W. Gray. Fibered and cofibred categories. In *Conference on Categorical Algebra*, La Jolla, California, 1965.

[15] J. W. Gray. *Formal Category Theory I: Adjointness for 2-Categories*, volume 391 of *Lecture Notes in Mathematics*. Springer-Verlag, 1974.

[16] J. Greiner. Programming with inductive and co-inductive types. Technical Report CMU-CS-92-109, Carnegie Mellon University, Pittsburgh, January 1992.

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

[18] R. Hasegawa. Categorical data types in parametric polymorphism. In *Fourth Asian Logic Conference*, RIMS, 1990.

[19] R. Hasegawa. Parametricity of extensionally collapsed term models of polymorphism and their categorical properties. In *Theoretical Aspects of Computer Science*, 1991.

[20] B. Hilken and D. E. Rydeheard. Indexed categories for program development. Technical report UMCS-90-10-1, University of Manchester, 1990.

[21] B. P. Hilken and D. E. Rydeheard. Towards a categorical semantics of type classes. Technical report, University of Manchester, December 1991. Technical report.

[22] B. P. Hilken and D. E. Rydeheard. Towards a categorical semantics of type classes. In *Mathematical Foundations of Computer Science*, pages 191–201, 1991.

[23] J. M. E. Hyland and A. M. Pitts. The theory of constructions: Categorical semantics and topos-theoretic models. *Contemporary Mathematics*, 92:137–199, 1989.

[24] B. Jacobs. *Categorical type theory*. PhD thesis, University of Nijmegen, 1991.

[25] C. B. Jay. Tail recursion from universal invariants. In D. H. Pitts et al., editors, *Category Theory and Computer Science*, volume 530 of *Lecture Notes in Computer Science*, pages 151–163, Paris, France, September 1991. Springer-Verlag.

[26] P. T. Johnstone and eds. R. Parè. *Indexed Categories and their Applications*, volume 661 of *Lecture Notes in Mathematics*. Springer-Verlag, 1978.

[27] G. M. Kelly. Elementary observations on 2-categorical limits. *Bulletin of the Australian Mathematical Society*, 39:301–317, 1989.

[28] G. M. Kelly and S. MacLane. *Closed coherence for a natural transformation*, pages 1–28. Lecture Notes in Mathematics, Vol. 281. Springer-Verlag, 1972.

[29] G. M. Kelly and R. Street. Review of the elements of 2-categories. In *Proceedings Sydney Category Theory Seminar 1972/73*, number 420 in Springer Lecture Notes in Mathematics, 1974.

[30] A. Kock. Strong functors and monoidal monads. *Archiv der Mathematik*, 23:113–120, 1972.

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

[32] F. W. Lawvere and S. Schanuel. *Conceptual Mathematics.* University of Buffalo Press, 1991.

[33] 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.

[34] G. Malcolm. *Algebraic Data Types and Program Transformation.* PhD thesis, University of Groningen, 1990.

[35] G. Malcolm. Data structures and program transformations. *Science of Computer Programming*, 14:255–279, 1990.

[36] E. G. Manes and M. A. Arbib. Parametrized datatypes do not need highly constrained parameters. *Information and Control*, 52:139–158, 1982.

[37] C. McLarty. *Elementary Categories, Elementary Toposes.* Oxford University Press, 1992.

[38] E. Moggi. An abstract view of programming languages. Lecture notes, June 1989.

[39] E. Moggi. Computational lambda-calculus and monads. In *4th Annual Symposium on Logic in Computer Science*, pages 14–23, 1989.

[40] E. Moggi. Notions of computation and monads. *Information and Control*, 93:55–92, 1991.

[41] P. Panangaden, P. Mendler, and M. Schwartzbach. Recursively defined types in constructive type theory. In *Resolution of Equations in Algebraic Structures*, volume 1. Academic Press, 1989.

[42] D. Pavlović. *Predicates and Fibrations.* PhD thesis, Utrecht, November 1990.

[43] W. Phoa. Fibrations (outline). Lecture notes, April 1992.

[44] B. Pierce. *Basic Category Theory for Computer Scientists.* MIT Press, 1991.

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

[46] L. Romàn. On recursive principles in cartesian categories. Preprint, 19.

[47] M. Smyth and G. Plotkin. The category theoretic solution of recursive domain equations. *SIAM Journal of Computing*, 11:761–783, 1982.

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

[49] P. Taylor. *Recursive domains, indexed category theory and polymorphism.* PhD thesis, University of Cambridge, 1986.

[50] D. Turner. Duality and de morgan principles for lists. *Bulletin of the European Association of Theoretical Computer Science*, 45:229–237, October 1991.

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

[52] R. F. C. Walters. Data types in distributive categories. *Bulletin of the Australian Mathematical Society*, 40:79–82, 1989.

[53] R. F. C. Walters. *Categories, Logic, and Computer Science.* Cambridge University Press, 1992.

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

[55] 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.

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

Biographical Note

My terrestrial appointment began two days after Eisenhower's birthday in the year of D-Day - October 16, 1944 - in my parents' bedroom near Elmore, Ohio. Because of a valedictorianship at Harris-Elmore High School and the top score on the National High School Mathematics Exam for the entire northwest quadrant of Ohio in 1962, I became a General Motors Merit Scholar at Bowling Green State University in Bowling Green, Ohio. Six semesters later, in 1965, I had pocketed a baccalureate in mathematics and chemistry. As a Teaching Fellow at the University of Michigan I earned a perfunctory Master's Degree in Mathematics in 1966, from which I happily escaped to the warmer mathematics research clime at The Ohio State University for the next three years. Although I found algebra and topology to mix like alcohol and water, the meager prospects for being a well-fed and sober algebraic topologist in 1969 induced my entry into industrial software development. I re-entered university in 1974 for two years of graduate study at the University of Toronto to gain an electronics engineering education, specializing in integrated circuits. I then believed a nuts-and-bolts engineering background to be necessary for future successful involvement in new computing systems, and my experiences have proved me right. I scurried on to major jobs in designing computing machines - hardware and software mixtures - that aimed at a variety of high-tech markets. Some were: operating systems for ultra-high-speed data acquisition via telemetry, fault-tolerant peripheral controllers, I/O channels for capability-object based CISC integrated circuits, and file-transfer protocols within telecommunication work stations. The arrival of my daughter Morgan in 1985 was the nudge for taking stock, veering away from industry and committing life's continuation to the higher track of learning for *my* sake. I entered the Oregon Graduate Institute in September, 1987, as that year's Tektronix Computer Science Fellow. I gratefully received this award again the following year. General department research assistantships supported me the rest of the way.

My publications are listed below. But I cannot ignore the real achievement I performed at OGI during these tremendous years of growth and flux for the CSE department: steadfastly sitting at the same beat-up oak desk in the same dusty office in the same cobwebbed position facing westward from September 15, 1987 to August 5, 1992. Like a rock.

- J.R.B. Cockett and D. Spencer, *Strong Categorical Datatypes II : A term logic for categorical programming.* Presented at 2nd Montreal Workshop on Programming Language Theory (December, 1991). Submitted to Theoretical Computer Science.
- J.R.B. Cockett and D. Spencer, *Strong Categorical Datatypes I.* International Meeting on Category Theory (Montreal, June, 1991). Canadian Mathematical Society Proceedings 13 (1992) 141-169.
- D. Spencer, *A Survey of Categorical Computation: Fixed Points, Partiality, Combinators, ... Control?* Bulletin of the European Association of Theoretical Computer Science 43 (February, 1991) 285-312.
- D. Spencer, *Computable Type Theories for Specification.* Oregon Graduate Institute Computer Science Research Symposium (May, 1989). Prize award.