A Categorical Programming Language

Tatsuya Hagino

Doctor of Philosophy
University of Edinburgh
1987

Author's address:

Tatsuya Hagino
Data Processing Center
Kyoto University
Kyoto 606
Japan

Abstract

A theory of data types and a programming language based on category theory are presented.

Data types play a crucial role in programming. They enable us to write programs easily and elegantly. Various programming languages have been developed, each of which may use different kinds of data types. Therefore, it becomes important to organize data types systematically so that we can understand the relationship between one data type and another and investigate future directions which lead us to discover exciting new data types.

There have been several approaches to systematically organize data types: algebraic specification methods using algebras, domain theory using complete partially ordered sets and type theory using the connection between logics and data types. Here, we use category theory. Category theory has proved to be remarkably good at revealing the nature of mathematical objects, and we use it to understand the true nature of data types in programming.

We organize data types under a new categorical notion of F, G-*dialgebras* which is an extension of the notion of adjunctions as well as that of T-algebras. T-algebras are also used in domain theory, but while domain theory needs some primitive data types, like products, to start with, we do not need any. Products, coproducts and exponentiations (i.e. function spaces) are defined exactly like in category theory using adjunctions. F, G-dialgebras also enable us to define the natural number object, the object for finite lists and other familiar data types in programming. Furthermore, their symmetry allows us to have the dual of the natural number object and the object for infinite lists (or lazy lists).

We also introduce a functional programming language in a categorical style. It has no primitive data types nor primitive control structures. Data types are declared using F, G-dialgebras and each data type is associated with its own control structure. For example, natural numbers are associated with primitive recursion. We define the meaning of the language operationally by giving a set of reduction rules. We also prove that any computation in this programming language terminates using Tait's computability method.

A specification language to describe categories is also included. It is used to give a formal semantics to F, G-dialgebras as well as to give a basis to the categorical programming language we introduce.

Acknowledgements

The greatest thanks go to Professor Rod Burstall who, first of all, accepted me as a Ph. D. student, then, supervised me during my Ph. D. study and, further, gave me an opportunity to continue working in Edinburgh. He gave me not only useful advices but also much-needed mental support. I would also thank to his wife, Sissi Burstall, who invited me to their house often and made my stay in Edinburgh very pleasant.

I am also grateful to Professor Reiji Nakajima for having encouraged me to come to Edinburgh and to Professor Heisuke Hironaka for having helped me to solve the financial problem for studying in Edinburgh.

Many lectures I attended in the first year enlightened me a lot: domain theory, operational semantics, denotational semantics, algebraic specification, category theory, and so on. I did not know what category theory really is before I came to Edinburgh. Thanks to Andrzej Tarlecki, Edmund Robinson and John Cartmell for having helped me to overcome the initial difficulty of category theory. John Cartmell also helped me a lot through discussions in the early stage of the thesis. I am also in debt to Furio Honsell who introduced me to Tait's method for proving normalization theorems of lambda calculi when I stuck in the normalization proof of CPL. Many other people in Edinburgh helped me with their comments and through discussions. I would especially like to thank Bob McKay and Paul Taylor for reading the early drafts and discovering some disastrous mistakes.

This thesis is written using LaTeX on a Sun workstation and printed by an Apple LaserWriter. I would like to thank George Cleland and Hugh Stabler for providing wonderful computing facilities and software to Laboratory of Foundation of Computer Science.

My Ph. D. study at the University of Edinburgh has been funded by the Educational Project for Japanese Mathematical Scientists, Harvard University, by the Overseas Research Students Award and by the Science and Engineering Research Council Research Fellowship.

Contents

Chapter 1

Introduction

This is an exploration of data types through category theory. It is an attempt to achieve better understanding of data types, their uniform classification, and discovery of a new world of data types. Data types have been with us since the very first programming languages. Even some machine languages now have some concept of data types, but early programming languages had only a fixed number of data types, like integers, reals and strings, and/or a fixed number of data type constructors, like array constructors and record constructors. When we gradually realized how important data types were, programming languages started having richer and richer data types. A number of programming languages now allow us to define our own data types. Some might even say that the richer they are, the better the programming languages are. Programming languages can be classified by the way how they handle data types.

There is no question about the importance of data types. Much research in this area has produced various kinds of data types, so varied that one cannot capture them all. We now need to systematically organize data types. We want to know the connection between one data type and another. We want to know the reason why those data types are with us and while some other data types are not. After getting a clear view of data types, we might find the future direction to discover other important data types.

There have been already some attempts to organize data types. We can name some of the important ones: *Domain theory* is one, *algebraic specification* is another and *type theory* is one where a lot of research is going on at the moment. In this thesis, we will present yet another attempt to organize data types. We do so by using category theory. We call our data types *Categorical Data Types* (or *CDT* for short).

One might ask "Why category theory?" Category theory is known as highly abstract mathematics. Some call it abstract nonsense. It chases abstract arrows and diagrams, proves nothing but about those arrows and diagrams, rarely talks about what arrows are for and often concepts go beyond one's imagination. However, when this 'abstract nonsense' works, it is like magic. One may discover a simple theorem actually means very deep things and some concepts beautifully unify and connect things which are unrelated before.

In ordinary mathematics, whether we are aware or not, we are in the world of set theory. Mathematics has been so well developed with set theory that we can hardly do anything without it. Therefore, it is very natural that semantics of programming languages is generally based on set theory. Note that it is often said that most of programming languages do not have set theoretic semantics and, therefore, domain theory has been developed, but this does not contradict with "semantics based on set theory", because domain theory itself is based on set theory. A domain is a set with certain properties.

Set theory is a powerful tool, but sometimes this power disfigures beautiful objects so that we cannot directly see their natural properties. For example, in set theory it is not easy to see either the duality between injective and surjective functions or the duality between cartesian products and disjoint sums. It is in category theory that these dualities come out clearly. Category theory concentrates on the outer behaviour of objects. It does not care what is in an object, whereas set theory is all about what is in an object. It is interesting to know that seeing from the outside reveals the nature of an object more naturally than seeing its inside. For example, one of the most important concepts discovered by category theory is *adjunction* (or *adjoint situation*), which is strikingly simple but very beautiful and unifies various concepts under the same name.

Our slogan is: "*category theory can provide a better and more natural understanding of mathematical objects than set theory*", so we use it to guide our tour around the world of data types. Note that we do not mean to abandon set theory by this. We will still heavily rely on it, but our intuition should not be obstructed by it.

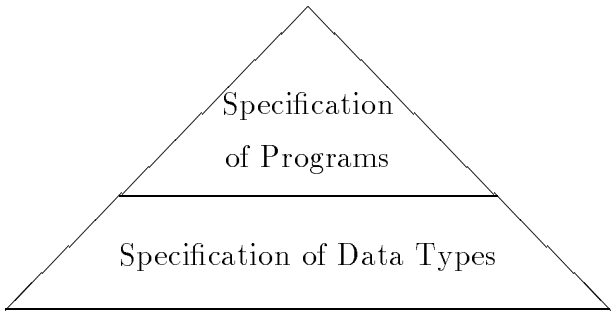
1.1 Backgrounds

1.1.1 Algebraic Specification Methods

Algebraic specification methods were first developed to describe what programs do. They are not like operational semantics or denotational semantics. These semantics also describe what programs do but in a different way. They describe it by giving meaning to each part of programs. They need to know how programs are written, that is, they need actual codes. Whereas, algebraic specification methods never talk about how programs are implemented. They describe their behaviour abstractly viewing from outside.

This abstract view point, seeing from outside, led to the discovery of *abstract data types* (see e.g. [Goguen, Thatcher and Wagner 78]). It is interesting to know that algebraic specification methods were started to describe programs but it also developed a theory of data types. Since algebraic specification methods try to describe things from an outside point of view, they cannot talk about the concrete nature of data types which programs handle. Therefore, the data types also needed to be abstracted and, thus, abstract data types have been developed. We may divide algebraic specification

methods into two: specification of data types and specification of programs.



Algebraic Specification

It is the former, specification of data types, that concerns us in this thesis.

Algebraic specification methods got their method of describing data types from abstract algebras in mathematics. Mathematicians have been using abstract algebras for about a century. Abstract algebras are only concerned with concrete real algebras insofar as they satisfy some laws. For example, a set with a binary operation is a group when the operation is associative, there is an identity and every element is invertible. A real set and a real operation can be anything, integers and +, general linear matrixes and their multiplication, and so on. Any theorem established for general groups can be applied to any real groups. There are various kinds of abstract algebras: groups, rings, fields, and so on. Those abstract algebras can be presented uniformly by *universal algebras*. Algebraic specification methods use a many-sorted version of universal algebras.

Naïvely speaking, an algebraic specification is a triple (S, Σ, E), where S is a set of sorts, Σ is a $S^* \times S$-indexed set of operations and E is a set of equations over Σ. For example, in an algebraic specification language CLEAR [Burstall and Goguen 80, Burstall and Goguen 82] a specification of lists may be as follows.

```
constant List =
    theory
        sorts element, list
        opns nil : list
             cons : element, list -> list
             head : list -> element
             tail : list -> list
        eqns all e : element, l : list, head(cons(e,l)) = e
             all e : element, l : list, tail(cons(e,l)) = l
     endth
```

S is { `element`, `list` }, $\Sigma_{\texttt{list}}$ is { `nil` }, $\Sigma_{\texttt{list element}}$ is { `head` }, and so on. E consists of the two equations above.

There are several problems about this specification as we will see immediately after we say what a specification means. An algebraic specification (S, Σ, E) defines a class of many sorted algebras each of which, say A, consists of an S-sorted set $|A|$ and functions $f_A : |A|_{s_1} \times \ldots \times |A|_{s_n} \longrightarrow |A|_s$ for each $f \in \Sigma_{s_1 \ldots s_n s}$ which satisfy the equations in E.

The first problem of the above specification is that not only lists satisfy it but also many of other data types as well. There is actually no way to make it describe only lists so long as we stick to first order methods. We need something of second order. The way algebraic specification methods usually obtain this is to put *data constraints*. We rely on the categorical fact that *the initial algebra is unique up to isomorphism*. In this case, we put a data constraint onto the sort '`list`', but not to '`element`' because if we put a data constraint onto '`element`' then the '`element`' sort would be empty.

The second problem is that '`head`' and '`tail`' are partial functions. The specification does not say what is '`head(nil)`' or what is '`tail(nil)`'. In order to fix this problem, we have to introduce, for example, error algebras or go into partial algebras.

The third problem is that although we put a data constraint on '`list`' it is not immediately obvious that '`nil`' and '`cons`' can construct all the lists. Some algebraic specification languages do distinguish these constructors from the others.

The fourth problem is about the sort '`element`'. We actually need it as a parameter. When we use this specification, '`element`' denotes a particular data type defined by another specification and we need a way to plug in any specification of '`element`' into this specification. Actually, CLEAR has this facility. '`List`' can be defined as '`procedure`' taking parametrized type '`element`'. However, this new specification no longer corresponds to a class of algebras but to something one level higher.

Many other problems there might be, but most of them have been solved in one way or another. The important point we would like to make is that the naïve idea of

$$\text{algebraic specification} = \text{universal algebra}$$

does not work well and we have to put a lot of other ideas into algebraic specification methods. One might wonder why so many complications are needed to define everyday objects like lists.

In CDT, we stick to the very simple relation

$$\text{categorical data type} = F,G\text{-dialgebra}$$

F,G-dialgebras can be seen as an extension of universal algebras (see section 3.1). We do not need to introduce meta arguments or any other complicated ideas into CDT in order to define lists or other basic data types.

1.1.2 Domain Theory

Domain theory was started with denotational semantics [Stoy 77, Scott 76]. In order to give denotational semantics to programs, we need several domains to which the denotations are mapped. Those domains are often interwoven and recursively defined. The most famous example of this is the following D.

$$D \cong D \to D$$

This domain D was necessary to give denotational semantics to the untyped lambda calculus. In general, we would like to solve the following domain equation:

$$D \cong F(D)$$

where $F(D)$ is a domain expression involving D.

Though domains are mathematical objects and not necessarily representable in computers, the idea of recursively defined data types has been adopted into several programming languages. For example, we can have a domain L for lists of A elements by solving[1]

$$L \cong 1 + A \times L,$$

and in the original version of ML [Gordon, Milner and Wordsworth 79], we could define the data type for lists just like the same.

```
abstype 'a list = unit + 'a # 'a list
    with ...
```

On the other hand, some domains cannot be represented in the same way. For example, we can have a domain I for infinite lists of A elements by solving

$$I \cong A \times I_\perp$$

where $I_\perp$ is the lifting of I by adding the new least element, but we cannot define infinite lists in ML in a similar way.

Comparing with algebraic specification methods, in domain theory we can define data types easily and there is no complication of parametrized data types, but we have some difficulty of defining operations over data types. In algebraic specification methods we define operations together with data types, but in domain theory we have to define them using the isomorphisms of domain equations.

If an algebraic specification (S, Σ, E) has no equational constraints (i.e. $E = \emptyset$), the initial algebra can be given by solving the following domain equations.

$$|A|_s \cong \sum_{f \in \Sigma_{s_1 \ldots s_n s}} |A|_{s_1} \times \ldots \times |A|_{s_n}$$

By this connection, we can see the possibility of combining algebraic specification methods and domain theory together. Actually, data types in the current Standard ML [Milner 84, Harper, MacQueen and Milner 86] are defined in this mixed fashion (see also section 5.3).

Categorically, we can go the other way round. If $F(D)$ is a covariant functor, the initial fixed point of $F(D)$ can be characterized as the initial F-algebra. A F-algebra is a categorical generalization of an ordinary algebra. The main idea we borrow from domain theory is this connection between initial fixed points and initial algebras.

[1] We have to say what kind of domains we are dealing with. Let us say in this thesis that a domain is a complete partially ordered set with the least element and a function between domains needs to be continuous and strict.

After becoming familiar with category theory, one can notice the dual connection between final fixed points and final co-algebras. People rarely talked about them until recently [Arbib and Manes 80]. One of the reasons is that co-algebras are not so popular and another reason is that final fixed points are often the same as initial fixed points in domain theory. However, in CDT we will use this dual connection as well. Final co-algebras give us some very intersting data types like infinite lists. We defined infinite lists by the initial fixed point of

$$I \cong A \times I_\perp$$

Actually, what we were doing using the lifting $I_\perp$ is to get the final fixed point of

$$I \cong A \times I.$$

1.2 Basic Category Theory

This section is to roughly introduce some categorical concepts we will use in the rest of this thesis. The author refers to category theory text books like [Mac Lane 71], [Arbib and Manes 75] and [Lambek and Scott 86] for a more detailed account of category theory.

A *category* $\mathcal{C}$ is given by

- a collection of *objects* $|\mathcal{C}|$,
- for any pair of objects A and B, a collection $\mathrm{Hom}_{\mathcal{C}}(A,B)$ of *morphisms* from domain A to codomain B, (we write $f\colon A \to B$ for $f \in \mathrm{Hom}_{\mathcal{C}}(A,B)$)
- for any objects A, B and C, an operation called *composition* denoted by '$\circ$' from $\mathrm{Hom}_{\mathcal{C}}(B,C) \times \mathrm{Hom}_{\mathcal{C}}(A,B)$ to $\mathrm{Hom}_{\mathcal{C}}(A,C)$ which is associative,

$$(f \circ g) \circ h = f \circ (g \circ h)$$

- for any object A, an *identity* morphism $\mathbf{I}_A\colon A \to A$ such that for any $f\colon B \to A$ and any $g\colon A \to C$

$$\mathbf{I}_A \circ f = f \qquad \text{and} \qquad g \circ \mathbf{I}_A = g$$

Two objects A and B are called *isomorphic* if there are two morphisms $f\colon A \to B$ and $g\colon B \to A$ such that

$$f \circ g = \mathbf{I}_B \qquad \text{and} \qquad g \circ f = \mathbf{I}_A.$$

f and g are called *isomorphisms*.

The *opposite category* $\mathcal{C}^{\mathrm{op}}$ of a category $\mathcal{C}$ is defined by reversing the direction of all the morphisms in $\mathcal{C}$.

$$\mathrm{Hom}_{\mathcal{C}^{\mathrm{op}}}(A,B) = \mathrm{Hom}_{\mathcal{C}}(B,A)$$

We may write $\mathcal{C}^{\mathrm{op}}$ morphism $f^{\mathrm{op}}\colon A \to B$ for $\mathcal{C}$ morphism $f\colon B \to A$.

The *product category* $\mathcal{C} \times \mathcal{D}$ of a category $\mathcal{C}$ and a category $\mathcal{D}$ is given as

- ◇ a $\mathcal{C} \times \mathcal{D}$ object is $\langle A, B \rangle$ for a $\mathcal{C}$ object A and a $\mathcal{D}$ object B
- ◇ a $\mathcal{C} \times \mathcal{D}$ morphism from $\langle A, B \rangle$ to $\langle A', B' \rangle$ is $\langle f, g \rangle$ for a $\mathcal{C}$ morphism $f: A \to B$ and a $\mathcal{D}$ morphism $g: A \to B$.

A *covariant functor* F from a category $\mathcal{C}$ to a category $\mathcal{D}$ (we write $F: \mathcal{C} \to \mathcal{D}$) is given by

- ◇ associating a $\mathcal{D}$ object $F(A)$ for every $\mathcal{C}$ object A
- ◇ associating a $\mathcal{D}$ morphism $F(f): F(A) \to F(B)$ for every $\mathcal{C}$ morphism $f: A \to B$ such that

$$F(\mathbf{I}_A) = \mathbf{I}_{F(A)} \qquad \text{and} \qquad F(f \circ g) = F(f) \circ F(g)$$

A *contravariant functor* is defined in a similar way except that $F(f): F(B) \to F(A)$.

A *natural transformation* α from a covariant functor $F: \mathcal{C} \to \mathcal{D}$ to a covariant functor $G: \mathcal{C} \to \mathcal{D}$ (we write $\alpha: F \dot{\to} G$) is given by

- ◇ associating a $\mathcal{D}$ morphism $\alpha_A: F(A) \to G(A)$ for every $\mathcal{C}$ object A such that for any $\mathcal{C}$ morphism $f: A \to B$ the following diagram commutes.

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

When every α_A is an isomorphism, we call α *natural isomorphism*.

Two functors $F: \mathcal{C} \to \mathcal{D}$ and $G: \mathcal{D} \to \mathcal{C}$ are called *adjoints* if there exists a natural isomorphism

$$\psi_{A,B}: \mathrm{Hom}_{\mathcal{D}}(F(A), B) \xrightarrow{\cong} \mathrm{Hom}_{\mathcal{C}}(A, G(B)).$$

F is called the left adjoint functor of G and G is called the right adjoint functor of F. We also call $\psi_{A,B}$ (or its inverse $\psi^{-1}_{A,B}$) *factorizer* or *mediating morphism*.

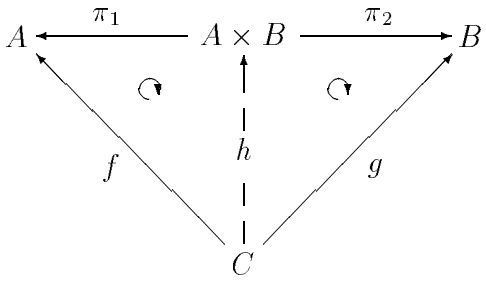
1.3 Development of Categorical Data Types

The motivation of CDT was to adopt the categorical way of defining data types into specification languages. Anybody educated using set theory has quite a shock when he first sees the way category theory works. It gives a totally different point of view to things which are familiar. Things which were vaguely connected suddenly are fitted into systematic places. It seems that the nature of things is finally revealed.

There are many beautiful concepts discovered through category theory, but here we concentrate only one of them, namely *adjunction* (or *adjoint situation*). In [Mac Lane 71], one will find many equivalent forms of the definition of adjunction (we gave one of them in section 1.2). One may be first at a loss for chosing the definition. Adjunction

is so versatile that it can be seen in a number of different forms and it is sometimes difficult to understand it if one sticks to a particular form of the definition. The form is not important if the spirit is understood. Adjunction can be regarded as a property of two functors or because of the unique correspondence between two functors it can be seen as defining one of them from the other. It is the latter which is important to us because it is a typical way of defining things in category theory. Let us see an example.

Using set theory, we can define what the product of two sets is, what the product of two groups is, what the product of two topological spaces is, and so on. Each definition is obviously deferent from the others but all of them are called by the same name, *product*. Why is that so? Is there any common property which all the different kinds of products should satisfy? Can we give the general definition of *product*? Category theory can give an affirmative answer to these questions. The categorical definition of products is

> For object A and B, the product $A \times B$ is an object such that there are two morphisms
>
> $$\pi_1 : A \times B \longrightarrow A \qquad \text{and} \qquad \pi_2 : A \times B \longrightarrow B$$
>
> and for any given two morphisms
>
> $$f : C \longrightarrow A \qquad \text{and} \qquad g : C \longrightarrow B$$
>
> there is a unique morphism $h : C \longrightarrow A \times B$ such that the following diagram commutes.
>
> 
>
>
> It is easily shown that any two objects satisfy this definition are isomorphic. We may write $\langle f, g \rangle$ for h.

This definition is general enough to cover the definition of products for sets, groups, topological spaces, and so on. We no longer need to define products for each individual case.

The generality should not be bound only in mathematics. Why should it not equally be appropriate to the definition of products in programming languages? The product data type of type A and type B is usually defined as a type of records whose first component is of type A and the second one is of type B, but this definition is like one in set theory. It assumes too much about how elements of data types are represented. It is not acceptable as an abstract description of the product data type. If the product data type is defined as an abstract data type, how can we present it?

We can directly adopt the categorical definition of products. There are five ingredients in the definition.

1. two given objects A and B,
2. the object $A \times B$ we are defining,
3. two morphisms $\pi_1 : A \times B \longrightarrow A$ and $\pi_2 : A \times B \longrightarrow B$,
4. $\langle f, g \rangle : C \longrightarrow A \times B$ for $f : C \longrightarrow A$ and $g : C \longrightarrow B$, and
5. the commutative diagram.

We may write these down as follows

$$
\begin{array}{l}
\text{object } A \times B \text{ is} \\
\quad \pi_1 : A \times B \longrightarrow A \\
\quad \pi_2 : A \times B \longrightarrow B \\
\quad \langle f, g \rangle : C \longrightarrow A \times B \text{ for } f : C \longrightarrow A \text{ and } g : C \longrightarrow B \\
\text{where} \\
\quad \pi_1 \circ \langle f, g \rangle = f \\
\quad \pi_2 \circ \langle f, g \rangle = g \\
\quad \pi_1 \circ h = f \wedge \pi_2 \circ h = g \Rightarrow h = \langle f, g \rangle \\
\text{end object.}
\end{array}
\qquad (*)
$$

Can we call this a categorical definition of the product data type constructor? Although this is an exact copy of the categorical definition, it has somehow lost the spirit of category theory; its beauty; its simplicity. The categorical definition of products we gave is in a disguised form of adjunction. The definition could have been sufficient to just say that the product functor is the right adjoint of the diagonal functor. The previous definition expands this into plain words so that there are a lot of duplications. One of them is that the type of $\langle\ ,\ \rangle$ can be deduced from the type of π_1 and π_2. If f and g has the same type as π_1 and π_2 except replacing $A \times B$ by C, $\langle f, g \rangle$ is a morphism from C to $A \times B$. Another duplication is that the commutative diagram can also be deduced from the rest. There are no other trivial ways to make diagrams involving π_1, π_2, f, g and $\langle f, g \rangle$. Therefore, the definition of product data types can be written simply as

$$
\begin{array}{l}
\text{object } A \times B \text{ is} \\
\quad \pi_1 : A \times B \longrightarrow A \\
\quad \pi_2 : A \times B \longrightarrow B \\
\text{end object.}
\end{array}
$$

This supplies the minimal information to get back to $(*)$. Now, we have to use the fact that $A \times B$ is defined by adjunction (it was not necessary in $(*)$). Let us indicate this by saying it is a *right object* as well as declaring $\langle\ ,\ \rangle$.

$$
\begin{array}{l}
\text{right object } A \times B \text{ with } \langle\ ,\ \rangle \text{ is} \\
\quad \pi_1 : A \times B \longrightarrow A \\
\quad \pi_2 : A \times B \longrightarrow B \\
\text{end object}
\end{array}
$$

This is the declaration of the product data type constructor in CDT (except for minor changes).

Let us examine the generality and simplicity of this declaration mechanism through examples. Let us try exponentials B^A. The functor $\bullet^A$ is defined as the right adjoint functor of $\bullet \times A$. The definition in CDT is

$$\begin{array}{l}\text{right object } B^A \text{ with curry( ) is}\\ \quad \text{eval}: B^A \times A \longrightarrow B\\ \text{end object}\end{array}$$

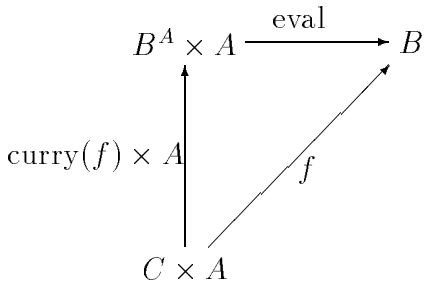
We can derive the usual definition of exponentials from this definition. First, the type of curry( ) should be

$$\frac{f: C \times A \longrightarrow B}{\text{curry}(f): C \longrightarrow B^A}.$$

The type of f is obtained from the type of 'eval' just replacing B^A by C. The commutative diagram which curry(f) gives can be obtained by connecting

$$B^A \times A \xrightarrow{\text{eval}} B \qquad \text{and} \qquad C \times A \xrightarrow{f} B$$

by curry$(f): C \longrightarrow B^A$. The only way to connect them together results



The morphism denoted by curry(f) is the unique one which makes this diagram commute. Thus, we recovered the ordinary definition of exponentials.

We said 'right object' for products and exponentials. It is natural to think that we also have '*left object*' as dual. The dual of products are coproducts. Let us define them in CDT.

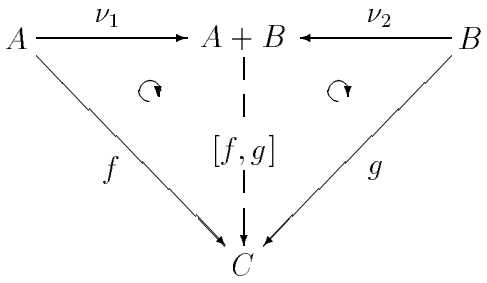
$$\begin{array}{l}\text{left object } A + B \text{ with [,] is}\\ \quad \nu_1: A \longrightarrow A + B\\ \quad \nu_2: B \longrightarrow A + B\\ \text{end object}\end{array}$$

The type of [,] can be obtained from the type of ν_1 and ν_2.

$$\frac{f: A \longrightarrow C \qquad g: B \longrightarrow C}{[f, g]: A + B \longrightarrow C}$$

Note that $[f, g]$ goes from $A + B$ to C not the other way round as it would be if it were a right object. The name 'left object' came from the fact that $A + B$ is in the left hand side of $\longrightarrow$. Remember that $A \times B$ was in the right hand side of $\longrightarrow$ for $\langle\ ,\ \rangle$.

A natural way of connecting f and g with ν_1 and ν_2 by $[f,g]$ gives us the ordinary commutative diagram which $[f,g]$ should satisfy.



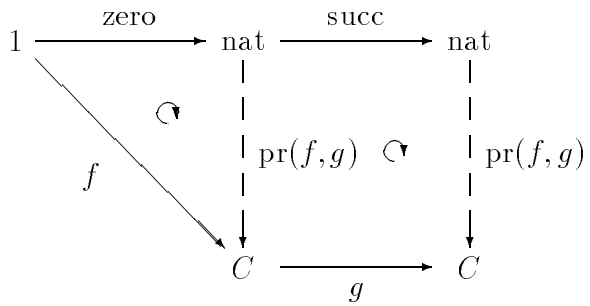
We demonstrated that we can express basic categorical constructs in CDT. Those constructs, or data types, are primitives in ordinary programming languages. Can we declare more familiar data types? In fact, the 'left object' declaration gives all those which can be defined by algebraic methods with no equations. 'Without equations' seems that we cannot define much, but actually it gives us all the important data types of ordinary programming languages. For example, natural numbers can be defined as

$$\begin{array}{l}\text{left object nat with pr(,) is}\\ \quad \text{zero}\colon 1 \longrightarrow \text{nat}\\ \quad \text{succ}\colon \text{nat} \longrightarrow \text{nat}\\ \text{end object}\end{array}$$

This is very much like a specification of natural numbers in algebraic specification methods except that we do not have the predecessor function or plus or times and that we have something called pr(,). From analogy of the types of [,] and $\langle$, $\rangle$, the type of pr(,) should be

$$\frac{f\colon 1 \longrightarrow C \qquad g\colon C \longrightarrow C}{\text{pr}(f,g)\colon \text{nat} \longrightarrow C}.$$

We also obtain the diagram characterizing 'nat' as we did for products and others.



This is exactly the definition of 'nat' being a natural number object in category theory and it is well-known that we can define all the primitive recursive functions using pr(,). For example, the addition function can be defined by

$$\text{add} \stackrel{\text{def}}{=} \text{eval} \circ \langle \text{pr}(\text{curry}(\pi_2), \text{curry}(\text{succ} \circ \text{eval})) \circ \pi_1, \pi_2 \rangle.$$

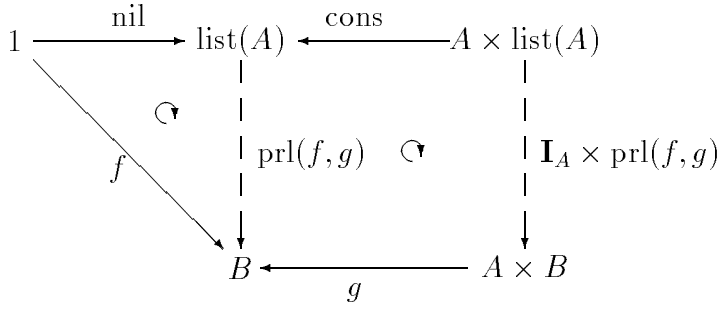
As another example, we give the definition of lists in CDT. It is

$$\begin{array}{l}\text{left object list}(A)\text{ with prl}(\ ,\ )\text{ is}\\ \quad \text{nil}\colon 1 \longrightarrow \text{list}(A)\\ \quad \text{cons}\colon A \times \text{list}(A) \longrightarrow \text{list}(A)\\ \text{end object}\end{array}$$

The type of prl(,) is

$$\frac{f\colon 1 \longrightarrow C \qquad g\colon A \times C \longrightarrow C}{\text{prl}(f,g)\colon \text{list}(A) \longrightarrow C}.$$

The diagram is



Remember that our definition of lists in CLEAR had 'head' and 'tail', but we do not declare them here. We can define them by prl(,).

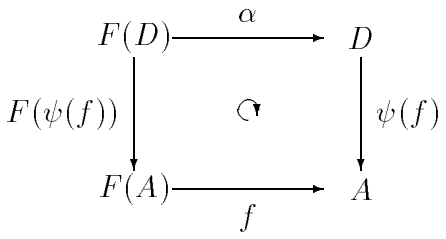
$$\begin{array}{rll}\text{head} & \stackrel{\text{def}}{=} \text{prl}(\nu_2, \nu_1 \circ \pi_1) & : \text{list}(A) \longrightarrow A + 1\\ \text{tail} & \stackrel{\text{def}}{=} [\nu_1 \circ \pi_2, \nu_2] \circ \text{prl}(\nu_2, \nu_1 \circ \langle \pi_1, [\text{cons}, \text{nil}] \rangle) & : \text{list}(A) \longrightarrow \text{list}(A) + 1\end{array}$$

'Without equations' is not a disadvantage to define everyday data types.

By the connection between initial fixed points and F-algebras, we can define the initial fixed point D of a covariant functor $F(X)$ as follows.

$$\begin{array}{l}\text{left object } D\text{ with } \psi(\ )\text{ is}\\ \quad \alpha\colon F(D) \longrightarrow D\\ \text{end object}\end{array}$$

α gives one direction of the isomorphism between D and $F(D)$, and $\psi(\ )$ gives unique arrows.



We have been using 'left object' more than 'right object', but they are dual and there are equally as many right objects as left objects. Just they are not familiar in ordinary

programming languages. For example, the following definition gives the data type for infinite lists.

$$\begin{array}{l}\text{right object inflist}(A)\text{ with fold}(\ ,\ )\text{ is}\\ \quad \text{hd}\colon \text{inflist}(A) \longrightarrow A\\ \quad \text{tl}\colon \text{inflist}(A) \longrightarrow \text{inflist}(A)\\ \text{end object}\end{array}$$

This gives the final fixed point of $I \cong A \times I$.

We have devised, based on category theory, a simple way of defining data types. The next question is whether we can adopt this method into ordinary programming languages. The answer is negative. Although what we can define in this way is far less than what we can define using algebraic specification languages with equations, we still have some strange things that can be defined in this way. Let us see an example. We defined $\text{list}(A)$ as a parametrized data type but in fact it is a functor.

$$\text{list}(f)\colon \text{list}(A) \longrightarrow \text{list}(B)$$

for a morphism $f\colon A \longrightarrow B$ is often called *map function*. In LISP it is 'MAPCAR' and in ML it is 'map'. The general declaration mechanism of CDT allows us to define the left and right adjoint functors of $\text{list}(A)$ which could not exist in the world of programming. We need to put a restriction to prevent these objects. The restriction will come out of a notion of computability in our setting. Interestingly, it turns out the category should be

$$\text{cartesian closed} + \text{initial fixed points} + \text{final fixed points}$$

We might see the similarity between this and the connection of lambda calculus and cartesian closed categories.

By putting this computability restriction, we can regard CDT as not only a device of defining data types but also a programming language. We program in a categorical fashion; there are no concrete data but morphisms; programs are also morphisms; there are no variables in programs. The computation in this language is reduction from morphisms to canonical ones. For example, we can reduce

$$\text{add} \circ \langle \text{succ} \circ \text{zero}, \text{succ} \circ \text{zero} \rangle \Rightarrow \text{succ} \circ \text{succ} \circ \text{zero}$$

which corresponds to the calculation of $1 + 1 = 2$.

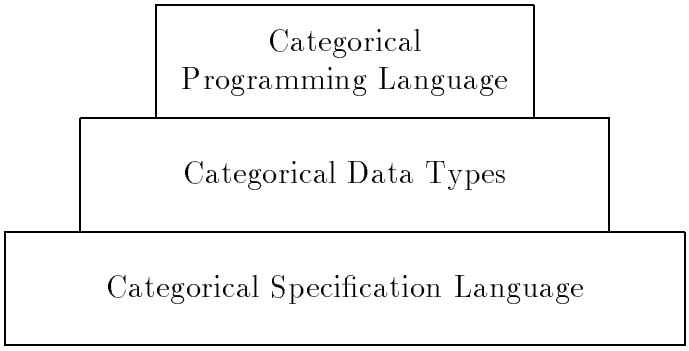
We summerize the characteristics of CDT as follows.

1. CDT uses categorical characterization of data types. We do not need to say things explicitly. All the equations are automatically generated for definitions.
2. CDT needs no primitive data types. Ordinary programming languages (e.g. PASCAL, LISP, ML) have primitive data types: natural numbers, lists, records, and so on, but CDT does not. They can be defined. Thus, CDT is analogous to algebraic specification methods where we can specify them as well. However, algebraic specification methods cannot specify higher order data types (i.e. function spaces or exponentials in a categorical term) nor can they specify products without using equations.

3. CDT can not only define products without explicitly mentioning equations but also can define exponentials.
4. CDT is symmetric in the sense that we can define initial algebras (or initial fixed points) as well as final co-algebras (or final fixed points).
5. Algebraic specification methods use initiality implicitly and do not use the unique homomorphisms between the initial algebras and the others, whereas CDT has explicit access to the unique morphisms. This gives the power of programming without going though equational characterization as it is necessary in algebraic specification methods.
6. Domain theory does not use the initiality explicitly either. The reason for this is that recursion in ordinary programming languages provide all the power of programming.
7. CDT defines functors. Functors are thought to be parametrized data types. Algebraic specification methods usually introduce parametrization later, but in CDT functorial behaviour of parametrized data types is treated at the base level.

1.4 In This Thesis

The theory of categorical data types is divided into three: *Categorical Specification Language*, *Categorical Data Types* and *Categorical Programming Language*.



Chapter 2 is about Categorical Specification Language (CSL for short). CSL is a specification language. It is an extension of ordinary algebraic specification languages. Whereas algebraic specifications specify algebras, it specifies categories. In order to specify categories, CSL has to handle functors, natural transformations and factorizers (or mediating morphisms). A CSL signature declares some functor names and their types (i.e. variances), some natural transformation names and their types and some factorizer names and their types. A CSL sentence is a conditional equation of functors, natural transformations and factorizers. A CSL model is a category equipped with functors, natural transformations and factorizers which have right types as are specified in the signature and which satisfy the sentences.

Chapter 3 gives the main idea of Categorical Data Types. The difference between CSL and CDT is that whereas CSL declares functors, natural transformations and

factorizers separately and connects them by sentences, CDT declare them together in a style of adjoint declarations. The semantics of CDT will be given informally in terms of F, G-algebras and formally in terms of CSL. There are some examples of data types we can define in CDT.

Chapter 4 is about Categorical Programming Language (CPL for short). CPL is a functional programming language which adopts the categorical declaration mechanism of data types from CDT. In order to define the notion of computation in CPL, we have to put some restrictions to CDT. We will introduce the notion of *elements* and *canonical elements* and present reduction rules to reduce elements to their equivalent canonical elements. We will also prove that any reduction in CPL terminates using Tait's computability method.

Each of those three languages, CSL, CDT and CPL, characterizes a category of data types in different ways.

	Syntax	**Semantics**
CSL	Signature and Sentences	Models
CDT	Adjoint Declarations	Freeness and Co-Freeness
CPL	Restricted Adjoint Declarations	Operational

In chapter 5, we will investigate the real consequences of our study in CDT. Section 5.1 is about an implementation of CPL as a real programming language. Section 5.2 is about the connection between CDT and typed lambda calculi, and finally in section 5.3 we will attempt to extend ML incorporating the CDT data type declaration mechanism.

1.5 Comparison with Other Works

Systematic studies of data types have already been carried out by various people in various contexts: ADJ in the context of initial algebras [Goguen, Thatcher and Wagner 78], Plotkin, Smyth and Lehmann in the context of domains [Lehmann and Smyth 81, Smyth and Plotkin 82] and Martin-Löf in the context of type theory [Martin-Löf 79]. This thesis is about a study of the same subject in the context of category theory. We do not try to extend the traditional approaches as Parasaya-Ghomi did for algebraic specification methods to include higher order types [Parasaya-Ghomi 82], nor do we try to unify two approaches together like [Dybjer 83], but we just directly use categorical methods of defining things.

Categorical Programming Language in chapter 4 might resemble Categorical Abstract Machine (CAM) by Curien [Curien 86], but he is only interested in cartesian closed categories whereas CPL deals with a class of different categories. Moreover, the reduction rules in CPL is systematically generated for products, coproducts, exponentials, natural numbers, and so on. We do not give any special reduction rules for any data types we define. CPL can be seen as

$$\text{CPL} = \text{CAM} + \text{initial data types} + \text{final data types}.$$

Actually, CAM can be absorbed into 'initial data types' and 'final data types', so in CPL we do not need to start with a particular set of reduction rules for cartesian closed categories. CPL has an ability to define cartesian closed categories and the introduction of data types also gives the control structure over those data types. Here is another slogan: "*control structures in programming languages come out of the structure of data types*".

Data Types		Control Structures
boolean	$\leftrightarrow$	`if` statement
disjoint union	$\leftrightarrow$	`case` statement
natural number	$\leftrightarrow$	primitive recursion, `for` statement
product	$\leftrightarrow$	pairing
function space	$\leftrightarrow$	function call

Barr and Wells uses *sketches* in [Barr and Wells 85] to describe algebras categorically. It is more powerful than ordinary algebraic specification methods because sketches can use any kind of limits whereas algebraic specification methods uses only products. It is interesting to investigate CSL by sketches.

Chapter 2

Categorical Specification Language

CDT can be seen from various points of view and can be presented in many ways. In this chapter, we present it as a specification language for categories (we call the specification language *Categorical Specification Language* or *CSL* for short). This is not the way originated, and it is difficult to recognize natural properties of data types in this way. We will give an alternative and more intuitive definition of CDT in chapter 3. However, the aim of CSL is to give mathematically rigorous background for the more intuitive presentation of CDT.

In applicative functional programming languages like ML, it is natural to see that their data types and functions form a category; each data type is an object; each function is a morphism; we have an identity function; and two functions can be composed in a usual way. We can also treat other programming languages including procedural ones semantically as defining domains and functions, and we can see that they form a category. These categories associated with programming languages reflect the characteristics of the programming languages. Thus, the study of data types can be carried out by examining these categories. CSL is a specification language for these categories.

Usually, a category is given by defining what an object consists of (e.g. a set for **Set**, the category of sets) and what a morphism between objects is (e.g. a set function for **Set**), but this is not the way CSL works. We are trying to understand a category in an abstract manner; we do not say what an object is; instead, we specify how it is constructed through its relationship among other objects. We saw in chapter 1 an ML data type for lists and it was a parameterized data type. We can now see it as a data type constructor; given a data type it constructs a new data type for the lists of the given data type. Categorically, constructors of objects are functors and they provide structures for categories. Remember that a cartesian closed category is a category with three functors, the terminal (constant) functor, the product functor and the exponential functor. Thus, CSL specifies a category equipped with some functors. Because the properties of functors are often described in terms of their interaction with natural transformations and factorizers (e.g. the binary product functor is explained with two natural transformations, π_1 and π_2, and factorizer $\langle\ ,\ \rangle$), CSL also specifies natural transformations and factorizers.

Let us make a comparison with algebraic specification languages like CLEAR [Burstall and Goguen 80, Burstall and Goguen 82]. An algebraic specification consists of declarations of some sorts and some operations on these sorts. Sorts are their data types. A model of an algebraic specification is a many-sorted algebra. On the other hand, a CSL specification consists of declarations of some functors, some natural transformations between them and some factorizers. Functors correspond to sorts, and natural transformations and factorizers correspond to operations. A model of a CSL specification is a category equipped with some functors, some natural transformations between them and some factorizers. It is abstract in the sense that the specification does not distinguish between equivalent categories (an algebraic specification does not distinguish isomorphic algebras). Note that, in general, the various models of a specification are not equivalent (e.g. not all the cartesian categories are equivalent).

As CSL specifies functors, the treatment of parametrized data types is different from algebraic specification languages. It specifies one level higher objects. The concept of parameterized data types and how to combine them play very essential roles in algebraic specification languages, but parameterized data types are treated in their meta-level (one level higher than the level treating algebras), that is specifications themselves are parameterized rather than dealing with parameterized sorts in specifications. In CSL, on the other hand, parameterized data types are the basic objects in specifications. In one specification, several parameterized data types can be declared and their relationship is directly specified. Therefore, combining specifications does not play as important a role as it does in algebraic specification languages.

Our goal in this chapter is to define the specification language CSL. In section 2.1, we will introduce several notations for dealing with functors which will be necessary later. In section 2.2, we will define the CSL signatures and in section 2.3 the CSL structures. The definition of CSL sentences and the CSL satisfaction relation will be in section 2.5 which follows section 2.4 in which we will introduce expressions involving natural transformations and factorizers. Finally, in section 2.6 we will show that there is an initial CSL structure for each CSL theory.

2.1 A Functorial Calculus

Before giving the definition of CSL signatures, we will look at some aspects of functors. This will be a kind of a functorial calculus though not as abstract as [Kelly 72] is. [Kelly 72] develops a calculus of combining functors as we will do in this section, but for the purpose of solving the coherence problems, and he treats many variable functors quite extensively. However, he does not say much about mixed variant functors which we are interested in. We will also generalize variances to include free-variance and fixed-variance for uniform treatment.

Functors are very much like ordinary functions except that functors have variances. Let F be a unary functor $\mathcal{C} \to \mathcal{C}$ and G be a binary functor $\mathcal{C} \times \mathcal{C} \to \mathcal{C}$. Then, we can

combine them to get more complex functors:

$$G(F(X),Y) \qquad F(G(X,F(Y))) \qquad G(F(X),G(X,Y)) \qquad \ldots$$

We call them *functorial expressions*. Of course, not every such expression denotes a functor. For example, $G(X,X)$ is not a proper functor if G is covariant in one argument and contravariant in the other. It is a functor if G is covariant in both arguments or contravariant in both arguments, or if G does not depend on one of the arguments.

In order to cope with these situations uniformly, we introduce two new variances: fixed-variance and free-variance. We say that a functor $F(X)$ is *fixed-variant* in X if F is not functorial in X, that is, F maps objects to objects, but not morphisms. We also say that a functor $F(X)$ is *free-variant* in X if F does not depend on X. Therefore, when $G{:}\mathcal{C}\times\mathcal{C}\to\mathcal{C}$ is covariant in the first argument and contravariant in the second, $G(X,X)$ is a fixed-variant functor.

Let us introduce the symbols for variances.

Definition 2.1.1: Let **Var** be the set of variances $\{+,\ -,\ \bot,\ \top\}$: $+$ for covariance, $-$ for contravariance, $\bot$ for free-variance and $\top$ for fixed-variance. □

The next definition is extending the notion of opposite categories.

Definition 2.1.2: Let $\mathcal{C}$ be a category.

1. $\mathcal{C}^+$ is $\mathcal{C}$ itself.
2. $\mathcal{C}^-$ is the opposite category of $\mathcal{C}$.
3. $\mathcal{C}^\bot$ is the category which has only one object and only one morphism (i.e. the identity of the one object). We may call the category *one point category*.
4. $\mathcal{C}^\top$ is the category which has the same objects as $\mathcal{C}$ but no morphisms except identities. □

In this way, we can regard the variances as functions mapping categories to categories (i.e. **Cat** $\to$ **Cat**, where **Cat** is the category of (small) categories) or we can even regard them as a monoid acting on **Cat**.

Definition 2.1.3: $\langle$**Var**, $\bullet\rangle$ is a commutative monoid with unit $+$, where the monoid operation $\bullet{:}$ **Var** $\times$ **Var** $\to$ **Var** is defined by the following table.

$\bullet$	$\bot$	$+$	$-$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$+$	$\bot$	$+$	$-$	$\top$
$-$	$\bot$	$-$	$+$	$\top$
$\top$	$\bot$	$\top$	$\top$	$\top$

□

Proposition 2.1.4: $\langle \mathbf{Var}, \bullet \rangle$ is a monoid acting on **Cat**, that is, $\mathcal{C}^{u \bullet v} = (\mathcal{C}^u)^v$ for any $u, v \in \mathbf{Var}$.
Proof: We have to show $(\mathcal{C}^-)^- = \mathcal{C}$ (i.e. the opposite of opposite is itself) and so on, but they are trivial. []

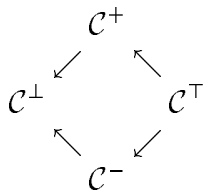
Since we will deal with many variable functors and they are functors from products of categories $\mathcal{C} \times \cdots \times \mathcal{C}$ to a category $\mathcal{C}$, **Var** action for product categories should be investigated.

Proposition 2.1.5: **Var** action on **Cat** distributes over products, that is, for any categories $\mathcal{C}$ and $\mathcal{D}$

$$(\mathcal{C} \times \mathcal{D})^u \cong \mathcal{C}^u \times \mathcal{D}^u.$$

Proof: In case u is $-$, it says that the opposite of the product category is isomorphic to the product of the opposite categories and it is the case because $\langle f, g \rangle^{\mathrm{op}} : \langle A, B \rangle \to \langle C, D \rangle \iff \langle f, g \rangle : \langle C, D \rangle \to \langle A, B \rangle \iff f : C \to A$ and $g : D \to B \iff f^{\mathrm{op}} : A \to C$ and $g^{\mathrm{op}} : B \to D \iff \langle f^{\mathrm{op}}, g^{\mathrm{op}} \rangle : \langle A, B \rangle \to \langle C, D \rangle$. The other cases are trivial. []

We need one more preparation before talking about mixed variant functors. The category $\mathcal{C}^\top$ can be embedded into $\mathcal{C}^+$ as well as into $\mathcal{C}^-$ and they themselves are embedded into $\mathcal{C}^\perp$, that is we have the following embedding functors.



We introduce a partial order on **Var** to respect these embeddings.

Definition 2.1.6: $\sqsubseteq$ is a partial order on **Var** such that $\perp \sqsubseteq + \sqsubseteq \top$ and $\perp \sqsubseteq - \sqsubseteq \top$. []

From the definition, it is clear that

Proposition 2.1.7: If $u \sqsubseteq v$, there is an embedding functor $e_{u,v} : \mathcal{C}^v \to \mathcal{C}^u$. []

Now we can start talking about mixed variant functors and their calculus. As a function is associated with an arity (simply a natural number), a mixed variant functor over a particular category $\mathcal{C}$ is associated with a *varity* which is a sequence of variances. For example, binary functor $G : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ which is contravariant in the first argument and covariant in the second is a functor with a varity $-+$.

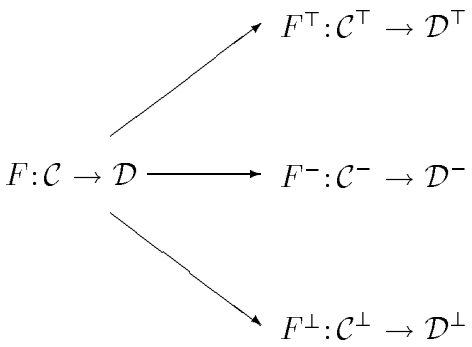
Definition 2.1.8: A mixed variant functor F of varity $v_1 \ldots v_n$ is a (covariant) functor from $\mathcal{C}^{v_1} \times \cdots \times \mathcal{C}^{v_n}$ to $\mathcal{C}$. []

When we are given a $\mathbf{Var}^*$-indexed set Γ of primitive mixed variant functors, where $F \in \Gamma_{v_1 \ldots v_n}$ is a functor of varity $v_1 \ldots v_n$, we would like to establish how to combine these primitive functors and get more complex functors like $H(G(X, Y), F(X))$.

Firstly, we extend the action of variances on categories to that on functors. For example, from a contravariant functor $F : \mathcal{C}^- \to \mathcal{C}$ we get a covariant functor $F^- : \mathcal{C} \to \mathcal{C}^-$ as $F^-(f : A \to B) \stackrel{\mathrm{def}}{=} F(f^{\mathrm{op}} : B \to A)$.

Definition 2.1.9: For a functor $F:\mathcal{C} \to \mathcal{D}$,

1. a functor $F^+:\mathcal{C}^+ \to \mathcal{D}^+$ is F itself,
2. a functor $F^-:\mathcal{C}^- \to \mathcal{D}^-$ is given by $F^-(A) = A$ for an object A in $\mathcal{C}$ and $F^-(f^{\mathrm{op}}) = F(f)^{\mathrm{op}}$ for a morphism f in $\mathcal{C}$,
3. a functor $F^\perp:\mathcal{C}^\perp \to \mathcal{D}^\perp$ is the identity functor since both $\mathcal{C}^\perp$ and $\mathcal{D}^\perp$ are the one point category, and
4. a functor $F^\top:\mathcal{C}^\top \to \mathcal{D}^\top$ has the same object mapping as F but no morphism mapping.



□

This definition is forced from the definition of $\mathcal{C}^u$, so **Var** action has more structure.

Proposition 2.1.10: For a functor $F:\mathcal{C} \to \mathcal{D}$, $(F^u)^v = F^{u \bullet v}$.
Proof: We have to check this for all the combinations of u and v. For example, $(F^-)^\top = F^\top$ is true because F^- only changes the mapping of morphisms but $(F^-)^\top$ forgets it. □

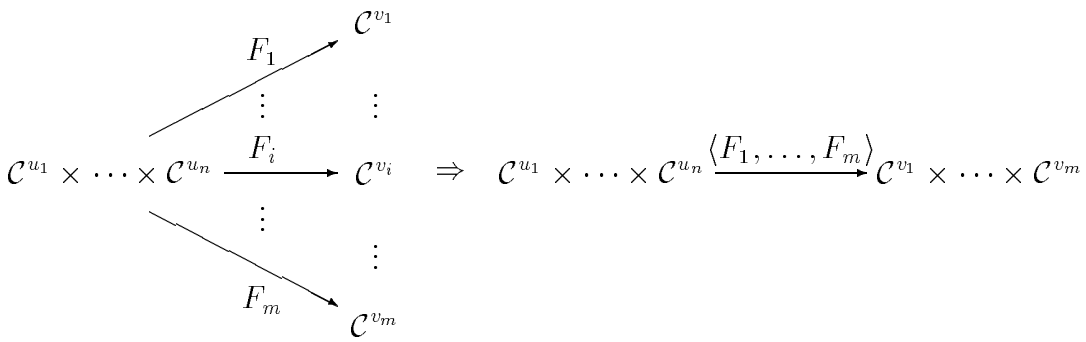
We have the two propositions which give us the basis for combining mixed variant functors.

Proposition 2.1.11: For functors $F:\mathcal{C}^{u_1} \times \cdots \times \mathcal{C}^{u_n} \to \mathcal{C}^{v_1} \times \cdots \times \mathcal{C}^{v_m}$ and $G:\mathcal{C}^{v_1} \times \cdots \times \mathcal{C}^{v_m} \to \mathcal{C}$ (i.e. varity $v_1 \ldots v_m$), $G \circ F$ is a functor of $\mathcal{C}^{u_1} \times \cdots \times \mathcal{C}^{u_n} \to \mathcal{C}$ (i.e. varity $u_1 \ldots u_n$).

$$\mathcal{C}^{u_1} \times \cdots \times \mathcal{C}^{u_n} \xrightarrow{F} \mathcal{C}^{v_1} \times \cdots \times \mathcal{C}^{v_m} \xrightarrow{G} \mathcal{C}$$

Proof: Trivial from the definition of composition of functors. □

Proposition 2.1.12: For functors $F_1{:}\mathcal{C}^{u_1} \times \cdots \times \mathcal{C}^{u_n} \to \mathcal{C}^{v_1}, \ldots, F_m{:}\mathcal{C}^{u_1} \times \cdots \times \mathcal{C}^{u_n} \to \mathcal{C}^{v_m}$, $\langle F_1, \ldots, F_m \rangle$ is a functor of $\mathcal{C}^{u_1} \times \cdots \times \mathcal{C}^{u_n} \to \mathcal{C}^{v_1} \times \cdots \times \mathcal{C}^{v_m}$.



Proof: It is trivial from the definition of products in **Cat**. []

The two propositions allow us to combine functors only if the source and target categories match exactly. For example, $F{:}\mathcal{C} \to \mathcal{C}$ and $G{:}\mathcal{C}^- \to \mathcal{C}$ cannot be composed into $G \circ F$. Therefore, we have to first convert functors of $\mathcal{C}^{v_1} \times \cdots \times \mathcal{C}^{v_n} \to \mathcal{C}$ into those of $\mathcal{C}^{v'_1} \times \cdots \times \mathcal{C}^{v'_n} \to \mathcal{C}^u$. There are two ways to do so.

Firstly, from definition 2.1.9, functor F of varity $v_1 \ldots v_n$, that is F is a functor of $\mathcal{C}^{v_1} \times \cdots \times \mathcal{C}^{v_n} \to \mathcal{C}$, into

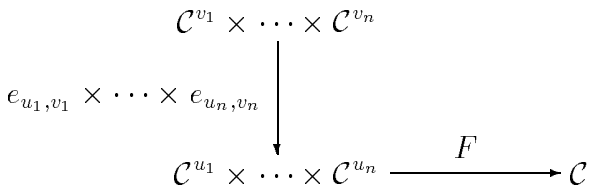
$$F^u{:}(\mathcal{C}^{v_1} \times \cdots \times \mathcal{C}^{v_n})^u \to \mathcal{C}^u \cong \mathcal{C}^{v_1 \bullet u} \times \cdots \times \mathcal{C}^{v_n \bullet u} \to \mathcal{C}^u$$

The isomorphism is from proposition 2.1.5 and proposition 2.1.4.

The other way of conversion is using embedding functors and coercing functors into greater variances (e.g. covariant functor can be a fixed variant functor).

Definition 2.1.13: If $u_1 \sqsubseteq v_1, \ldots, u_n \sqsubseteq v_n$ and F is a functor of varity $u_1 \ldots u_n$, we can coerce it to a functor of varity $v_1 \ldots v_n$ by

$$F\big|^{v_1 \ldots v_n}_{u_1 \ldots u_n} \stackrel{\text{def}}{=} F \circ (e_{u_1,v_1} \times \cdots \times e_{u_n,v_n}).$$



We may write $F\big|^{v_1 \ldots v_n}$ when $u_1 \ldots u_n$ is obvious. []

Let us now define the composition of mixed variant functors.

Definition 2.1.14: Let F be a functor of varity $u_1 \ldots u_n$ and $G_1, \ldots, G_n$ be functors of varity $v_{11} \ldots v_{1m}, \ldots, v_{n1} \ldots v_{nm}$, respectively. Then we have a functor $F[G_1, \ldots, G_n]$

of varity $w_1 \dots w_m$ where $w_i \stackrel{\text{def}}{=} u_1 \bullet v_{1i} \sqcup \cdots \sqcup u_n \bullet v_{ni}$[1]. The definition of the functor is

$$F[G_1, \dots, G_n] \stackrel{\text{def}}{=} F \circ \langle G_1^{u_1}|^{w_1 \dots w_m}, \dots, G_n^{u_n}|^{w_1 \dots w_m} \rangle$$

Proof of well-definedness: G_i is a functor of $\mathcal{C}^{v_{i1}} \times \cdots \times \mathcal{C}^{v_{im}} \to \mathcal{C}$. $G_i^{u_i}$ is a functor of $\mathcal{C}^{u_i \bullet v_{i1}} \times \cdots \times \mathcal{C}^{u_i \bullet v_{im}} \to \mathcal{C}^{u_i}$. Since $u_i \bullet v_{i1} \sqsubseteq w_1, \dots, u_i \bullet v_{im} \sqsubseteq w_m$, from definition 2.1.13 $G_i^{u_i}|^{w_1 \dots w_m}$ is a functor of $\mathcal{C}^{w_1} \times \cdots \times \mathcal{C}^{w_m} \to \mathcal{C}^{u_i}$. From proposition 2.1.12 $\langle G^{u_1}|^{w_1 \dots w_m}, \dots, G^{u_n}|^{w_1 \dots w_m} \rangle$ is a functor of $\mathcal{C}^{w_1} \times \cdots \times \mathcal{C}^{w_m} \to \mathcal{C}^{u_1} \times \cdots \times \mathcal{C}^{u_n}$. Therefore, from proposition 2.1.11,

$$F \circ \langle G^{u_1}|^{w_1 \dots w_m}, \dots, G^{u_n}|^{w_1 \dots w_m} \rangle : \mathcal{C}^{w_1} \times \cdots \times \mathcal{C}^{w_m} \to \mathcal{C}$$

is a functor of varity $w_1 \dots w_m$. []

The variances of $G_1, \dots, G_n$ are appropriately modified according to the varity of F and then the least upper bound is taken so that we can pair them together.

We need some lemmas to show the associativity of the composition.

Lemma 2.1.15: Let $u \sqsubseteq v$ and F be a functor of varity $w_1 \dots w_n$.

1. For any w, $u \bullet w \sqsubseteq v \bullet w$ and $(e_{u,v})^w = e_{u \bullet w, v \bullet w}$.
2. The following diagram commutes.

$$\begin{array}{ccc} \mathcal{C}^{v \bullet w_1} \times \cdots \times \mathcal{C}^{v \bullet w_n} & \xrightarrow{F^v} & \mathcal{C}^v \\ \Big\downarrow {\scriptstyle e_{u,v}^{w_1} \times \cdots \times e_{u,v}^{w_n}} & \circlearrowright & \Big\downarrow {\scriptstyle e_{u,v}} \\ \mathcal{C}^{u \bullet w_1} \times \cdots \times \mathcal{C}^{u \bullet w_n} & \xrightarrow{F^u} & \mathcal{C}^v \end{array}$$

   In other words, the action of **Var** on functors is natural with respect to the partial order $\sqsubseteq$.

3. $e_{u,v} \circ F^v = F^u|^{w_1 \bullet v \dots w_n \bullet v}$
4. $F|^{u_1 \dots u_n}|^{v_1 \dots v_n} = F|^{u_1 \sqcup v_1 \dots u_n \sqcup v_n}$
5. $(F \circ G)^u = F^u \circ G^u$
6. $\langle F_1, \dots, F_n \rangle^u = \langle F_1^u, \dots, F_n^u \rangle$
7. $(F|^{v_1 \dots v_n})^u = F^u|^{v_1 \bullet u \dots v_n \bullet u}$

[1] **Var** is a commutative semiring with unit: $\sqcup$ as its addition and $\bullet$ as its multiplication. If we express varities as vectors, then varity of $F[G_1, \dots, G_n]$ can be computed by the following matrix multiplication.

$$(w_1, \dots, w_m) = (u_1, \dots, u_n) \begin{pmatrix} v_{11} & \dots & v_{1m} \\ \vdots & \ddots & \vdots \\ v_{n1} & \dots & v_{nm} \end{pmatrix}$$

Proof: We have to check any pairs of u and v 1 and 2 hold from the definitions. 3 follows 2. 4, 5 and 6 are easy to show and 7 follows them.

$$\begin{aligned}
&(F|^{v_1 \dots v_n})^u = (F \circ (e_{w_1,v_1} \times \cdots \times e_{w_n,v_n}))^u \\
&= F^u \circ (e_{w_1,v_1} \times \cdots \times e_{w_n,v_n})^u = F^u \circ ((e_{w_1,v_1})^u \times \cdots \times (e_{w_n,v_n})^u) \\
&= F^u \circ (e_{w_1 \bullet u, v_1 \bullet u} \times \cdots \times e_{w_n \bullet u, v_n \bullet u}) = F^u|^{v_1 \bullet u \dots v_n \bullet u}
\end{aligned}$$ []

Proposition 2.1.16: Let $F, G_1, \dots, G_n$ and $H_1, \dots, H_m$ be functors of the following varities:

$$\begin{gathered}
F\colon u_1 \dots u_n \\
G_1\colon v_{11} \dots v_{1m}, \quad \dots \quad, G_n\colon v_{n1} \dots v_{nm} \\
H_1\colon w_{11} \dots w_{1l}, \quad \dots \quad, H_n\colon w_{m1} \dots w_{ml}
\end{gathered}$$

Then, the following equality between functors holds:

$$(F[G_1, \dots, G_n])[H_1, \dots, H_m] = F[G_1[H_1, \dots, H_m], \dots, G_n[H_1, \dots, H_m]].$$

Proof: Let varity $a_1 \dots a_m$, $b_{11} \dots b_{1l}, \dots, b_{n1} \dots b_{nl}$ and $c_1 \dots c_l$ be

$$\begin{gathered}
(a_1, \dots, a_m) \stackrel{\text{def}}{=} (u_1, \dots, u_n) \begin{pmatrix} v_{11} \dots v_{1m} \\ \vdots \ \ddots \ \vdots \\ v_{n1} \dots v_{nm} \end{pmatrix}, \\
\begin{pmatrix} b_{11} \dots b_{1l} \\ \vdots \ \ddots \ \vdots \\ b_{n1} \dots b_{nl} \end{pmatrix} \stackrel{\text{def}}{=} \begin{pmatrix} v_{11} \dots v_{1m} \\ \vdots \ \ddots \ \vdots \\ v_{n1} \dots v_{nm} \end{pmatrix} \begin{pmatrix} w_{11} \dots w_{1l} \\ \vdots \ \ddots \ \vdots \\ w_{m1} \dots w_{ml} \end{pmatrix}, \quad \text{and} \\
(c_1, \dots, c_l) \stackrel{\text{def}}{=} (u_1, \dots, u_n) \begin{pmatrix} v_{11} \dots v_{1m} \\ \vdots \ \ddots \ \vdots \\ v_{n1} \dots v_{nm} \end{pmatrix} \begin{pmatrix} w_{11} \dots w_{1l} \\ \vdots \ \ddots \ \vdots \\ w_{m1} \dots w_{ml} \end{pmatrix}.
\end{gathered}$$

Then,

$$\begin{aligned}
&(F[G_1, \dots, G_n])[H_1, \dots, H_m] \\
&= F \circ \langle G_1^{u_1}|^{a_1 \dots a_m}, \dots, G_n^{u_n}|^{a_1 \dots a_m} \rangle \circ \langle H_1^{a_1}|^{c_1 \dots c_l}, \dots, H_m^{a_m}|^{c_1 \dots c_l} \rangle \\
&= F \circ \langle G_1^{u_1}|^{a_1 \dots a_m} \circ \langle H_1^{a_1}|^{c_1 \dots c_l}, \dots \rangle, \dots \rangle \\
&= F \circ \langle G_1^{u_1} \circ (e_{v_{11} \bullet u_1, a_1} \times \cdots \times e_{v_{1m} \bullet u_1, a_m}) \circ \langle H_1^{a_1}|^{c_1 \dots c_l}, \dots \rangle, \dots \rangle \\
&= F \circ \langle G_1^{u_1} \circ \langle e_{v_{11} \bullet u_1, a_1} \circ H_1^{a_1}|^{c_1 \dots c_l}, \dots \rangle, \dots \rangle \\
&= F \circ \langle G_1^{u_1} \circ \langle H^{v_1 1 \bullet u_1}|^{w_{11} \bullet v_1 \bullet u_1 \dots w_{1l} \bullet v_1 \bullet u_1}|^{c_1 \dots c_l}, \dots \rangle, \dots \rangle \\
&= F \circ \langle G_1^{u_1} \circ \langle H^{v_1 1 \bullet u_1}|^{c_1 \dots c_l}, \dots \rangle, \dots \rangle
\end{aligned}$$

whereas

$$\begin{aligned}
&F[G_1[H_1, \dots, H_m], \dots, G_n[H_1, \dots, H_m]] \\
&= F \circ \langle (G_1 \circ \langle H_1^{v_{11}}|^{b_{11} \dots b_{1l}}, \dots, H_m^{v_{1m}}|^{b_{11} \dots b_{1l}} \rangle)^{u_1}|^{c_1 \dots c_l}, \dots \rangle \\
&= F \circ \langle G_1^{u_1} \circ \langle H_1^{v_{11} \bullet u_1}|^{b_{11} \bullet u_1 \dots b_{1l} \bullet u_1}|^{c_1 \dots c_l}, \dots \rangle, \dots \rangle \\
&= F \circ \langle G_1^{u_1} \circ \langle H_1^{v_{11} \bullet u_1}|^{c_1 \dots c_l}, \dots \rangle, \dots \rangle
\end{aligned}$$

Therefore, the proposition holds. []

Mixed variant functors form an algebraic theory but with an extra structure. Mixed variant functors are functions between four sorts, $+$, $-$, $\perp$ and $\top$.

Finally, we can show what functorial expressions like $H(G(X,Y),F(X))$ mean. Let Γ be a $\mathbf{Var}^*$-indexed set of primitive functor names[2]. Then, we define

$\mathbf{FE}(\Gamma) \stackrel{\text{def}}{=}$ the set of terms constructed from Γ and variables like term algebras.

$\mathbf{CFE}(\Gamma) \stackrel{\text{def}}{=} \{\ \lambda(X_1,\ldots,X_n).E \mid E \in \mathbf{FE}(\Gamma)$ and $X_1,\ldots,X_n$ includes all the variables in $E\ \}$

We call elements of $\mathbf{FE}(\Gamma)$ *functorial expressions* over Γ and call elements of $\mathbf{CFE}(\Gamma)$ *closed functorial expressions* over Γ. As we can convert any lambda closed term in a term algebra to a morphism in the corresponding algebraic theory, we can convert any closed functorial expression into functors. Let ξ be an assignment of functor symbols in $\Gamma_{v_1\ldots v_n}$ to real functors of varity $v_1\ldots v_n$ in $\mathcal{C}$, that is, ξ is a $\mathbf{Var}^*$-indexed function such that

$$\xi_{v_1\ldots v_n}\colon \Gamma_{v_1\ldots v_n} \to \mathbf{Funct}(\mathcal{C}^{v_1}\times\cdots\times\mathcal{C}^{v_n},\mathcal{C})$$

where $\mathbf{Funct}(\mathcal{D},\mathcal{E})$ is the category of functors from $\mathcal{D}$ to $\mathcal{E}$. We may write ξF for $\xi_{v_1\ldots v_n}(F)$. We can extend assignment ξ to that over $\mathbf{CFE}(\Gamma)$ as follows

$$\xi(\lambda(X_1,\ldots,X_n).E) \stackrel{\text{def}}{=} \xi_{X_1,\ldots,X_n}(E)$$

where $\xi_{X_1,\ldots,X_n}$ assigns expressions in $\mathbf{FE}(\Gamma)$ which have $X_1,\ldots,X_n$ as variables to functors of n variables and is defined by

$$\xi_{X_1,\ldots,X_n}(X_i) \stackrel{\text{def}}{=} \Pi^n_i\colon \mathcal{C}^{\perp}\times\cdots\mathcal{C}^{+}\times\cdots\mathcal{C}^{\perp}\to\mathcal{C}$$

$$\xi_{X_1,\ldots,X_n}(F(E_1,\ldots,E_n)) \stackrel{\text{def}}{=} \xi F[\xi_{X_1,\ldots,X_n}(E_1),\ldots,\xi_{X_1,\ldots,X_n}(E_n)]$$

where Π^n_i is the ith projection of the n fold product of $\mathcal{C}$.

Example 2.1.17: Let Γ consist of a unary functor symbol F which is covariant, a functor symbol G of varity $++$ and a functor symbol H of varity $-+$. Let ξ be an assignment of Γ. Then, $\lambda(X,Y).H(G(X,Y),F(X))$ denotes the following functor by ξ.

$$\xi(\lambda(X,Y).H(G(X,Y),F(X))) = \xi H[\xi G[\Pi^2_1,\Pi^2_2],\xi F[\Pi^2_1]]$$

The varity of Π^2_1 is $+\perp$, that of ξF is $+$ and, therefore, from definition 2.1.14, $\xi F[\Pi^2_1]$ has varity $+\perp$. The varity of Π^2_1 is $+\perp$, that of Π^2_2 is $\perp+$, that of ξG is $++$ and from definition 2.1.14 the varity of $\xi G[\Pi^2_1,\Pi^2_2]$ is

$$(+,+)\begin{pmatrix}+ & \perp\\ \perp & +\end{pmatrix} = (+\bullet+\sqcup+\bullet\perp,+\bullet\perp\sqcup+\bullet+) = (+\sqcup\perp,\perp\sqcup+) = (+,+)$$

Finally, the varity of the whole functor is

$$(-,+)\begin{pmatrix}+ & +\\ + & \perp\end{pmatrix} = (-\bullet+\sqcup+\bullet+,-\bullet+\sqcup+\bullet\perp) = (-\sqcup+,-\sqcup\perp) = (\top,-),$$

so it is fixed-variant in X and contravariant in Y. []

[2]From now on, we distinguish names (or symbols) from what they denote.

The last proposition in this section is to establish the relationship between the syntactic substitution of functorial expressions and the composition of functors defined in definition 2.1.14.

Proposition 2.1.18: Let

$$\lambda(X_1,\ldots,X_n).E \qquad \text{and} \qquad \lambda(Y_1,\ldots,Y_m).E_1,\ \ldots,\ \lambda(Y_1,\ldots,Y_m).E_n$$

be closed functorial expressions. Then,

$$\xi(\lambda(X_1,\ldots,X_n).E)[\xi(\lambda(Y_1,\ldots,Y_m).E_1),\ldots,\xi(\lambda(Y_1,\ldots,Y_m).E_n)] \\ = \xi(\lambda(Y_1,\ldots,Y_m).E[E_1/X_1,\ldots,E_n/X_n])$$

where $E[E_1/X_1,\ldots,E_n/X_n]$ is E in which $X_1,\ldots,X_n$ are replaced by $E_1,\ldots,E_n$, respectively.
Proof: We can easily prove it by structural induction on E using proposition 2.1.16 []

From this correspondence, for closed functorial expression K of n variables and $L_1,\ldots,L_n$ of m variables, we write

$$K[L_1,\ldots,L_n]$$

for a closed functorial expression of m variables which is obtained by replacing n variables in K by $L_1,\ldots,L_n$. Then, the proposition is

$$\xi(K[L_1,\ldots,L_n]) = \xi K[\xi L_1,\ldots,\xi L_n].$$

2.2 Signatures of Categorical Specification Language

A specification normally consists of a *signature*, which says what kind of sorts there are and what kinds of operations there are, and a set of *sentences* (or *equations*), which give properties of the operations. A specification defines a class of *models* which have what the signature says and satisfies the sentences. Therefore, in order to define a specification language, we have to define what its signatures are, what its sentences are and what its models are. However, it is often convenient to define, first, models without being constrained by sentences and, then, define a *satisfaction relation* between a statement and a model determining whether the statement is true in the model or not. A model which satisfies all the sentences is called a *theory model*. See *institutions* [Goguen and Burstall 83] for a categorical abstract definition of what specification languages are.

A CSL signature will be divided into three parts; in the first part, we will declare some names for functors, which will serve as parameterized data types (or data type constructors); in the second part, we will declare some names for natural transformations, which will serve as polymorphic functions over the parameterized data types; and in the third part, we will declare some names for factorizers (or mediating morphisms), which will be necessary to put initial or final constrains on the data types.

The first part can be presented as a $\mathbf{Var}^*$-indexed set Γ. F in $\Gamma_{v_1 \ldots v_n}$ is said to have varity $v_1 \ldots v_n$. We may write $F(v_1, \ldots, v_n)$ to indicate this.

Γ looks almost like an equational signature for algebraic specification languages. It is as if **Var** were the set of sorts and Γ were a set of operations over the sorts. The only difference is that Γ is not a $\mathbf{Var}^* \times \mathbf{Var}$-indexed set but simply a $\mathbf{Var}^*$-indexed set. This is because we can apply **Var** to functors to get other functors as we explained in definition 2.1.9, so it is sufficient to give Γ as a $\mathbf{Var}^*$-indexed set.

Note that Γ is one sorted. Each signature describes only one category. However, because we are dealing with one level higher objects, it has the power to describe more than one data type (or sort) inside one signature. We will illustrate this later in this section.

In the second part, a CSL signature introduces some symbols for natural transformations. Normally, a natural transformation is defined as follows: given two functors $F, G: \mathcal{C} \to \mathcal{D}$, a natural transformation $\alpha: F \stackrel{\cdot}{\to} G$ is a function which assigns to each $A \in |\mathcal{C}|$ a morphism $\alpha_A: F(A) \to G(A)$ such that for any morphism $f: A \to B$ in $\mathcal{C}$ the following diagram commutes.

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

As we have seen in section 2.1, closed functorial expressions provide more complicated functors constructed from primitive functors in Γ. Let Λ be $\mathbf{CFE}(\Gamma)$. We index the set of natural transformations by two closed functorial expressions, that is, the second part of a CSL signature is given by a $\Lambda \times \Lambda$-indexed set Δ; $\alpha \in \Delta_{K,L}$ will denote a natural transformation $\xi K \stackrel{\cdot}{\to} \xi L$, where ξK and ξL will be the denotations of K and L, respectively. We may write $\alpha: K \stackrel{\cdot}{\to} L$ to indicate this. Since a natural transformation should go between functors of the same number of variables, $\Delta_{K,L}$ should be empty when K and L have different number of variables. Even if they have the same number of variables, their variances may be different. In that case, we take the least upper bound of two variances.

The third and the final part of a CSL signature introduces symbols for factorizers (or mediating morphisms). In general, a factorizer is an isomorphism associated with an adjunction. If $F \vdash G$ where $F: \mathcal{C} \to \mathcal{D}$ and $G: \mathcal{D} \to \mathcal{C}$, the factorizer ψ gives the following natural isomorphism between hom sets.

$$\psi: \mathrm{Hom}_{\mathcal{D}}(F(A), B) \xrightarrow{\cong} \mathrm{Hom}_{\mathcal{C}}(A, G(B))$$

For example, the factorizer associated with the binary product functor 'prod' ($: \mathcal{C} \times \mathcal{C} \to \mathcal{C}$) is 'pair' which gives for any two morphisms $f: C \to A$ and $g: C \to B$ a morphism

$\mathrm{pair}(f,g)\colon C \to \mathrm{prod}(A,B)$. We can write this situation as the following rule.

$$\frac{C \xrightarrow{f} A \qquad C \xrightarrow{g} B}{C \xrightarrow{\mathrm{pair}(f,g)} \mathrm{prod}(A,B)}$$

In CSL, factorizers are given by a $(\Lambda\times\Lambda)^*\times(\Lambda\times\Lambda)$-indexed set Ψ ($\Lambda = \mathbf{CFE}(\Gamma)$). The index $(\Lambda\times\Lambda)^*$ specifies the type of morphisms to which a factorizer can be applied and the index $\Lambda\times\Lambda$ specifies the type of morphisms obtained by applying the factorizer. For

$$\psi \in \Psi_{\langle K_1,L_1\rangle\ldots\langle K_m,L_m\rangle,\langle K,L\rangle},$$

we may write it as the following rule.

$$\frac{f_1\colon K_1 \to L_1 \quad \ldots \quad f_m\colon K_m \to L_m}{\psi(f_1,\ldots,f_m)\colon K \to L}$$

where $f_1,\ldots,f_n$ are auxiliary names of morphisms introduced for this rule. As we have restricted the indexed set Δ, Ψ should be indexed by functors of the same number of variables, and also we take the least upper bound of the variances and regard it as the overall variance.

Hence, we come to the definition of CSL signatures.

Definition 2.2.1: A *CSL signature* is a triple $\langle\Gamma,\Delta,\Psi\rangle$, where Γ is a $\mathbf{Var}^*$-indexed set, Δ is a $\Lambda\times\Lambda$-indexed set (where $\Lambda = \mathbf{CFE}(\Gamma)$) for natural transformations and Ψ is a $(\Lambda\times\Lambda)^*\times(\Lambda\times\Lambda)$-indexed set for factorizers. As a restriction to the triple, $\Delta_{K,L}$ should be empty if K and L has the different number of variables and $\Psi_{\langle\langle K_1,L_1\rangle\ldots\langle K_m,L_m\rangle,\langle K,L\rangle\rangle}$ should also be empty if K_i, L_i, K and L do not have the same number of variables. □

As an example, we will give a CSL signature for cartesian closed categories.

Example 2.2.2: A cartesian closed category can be characterized as a category having three special functors: terminal object '1' (which is a constant functor), binary product 'prod' and exponential 'exp'. Therefore,

$$\Gamma_{()} = \{1\}, \qquad \Gamma_{++} = \{\mathrm{prod}\}, \qquad \Gamma_{-+} = \{\mathrm{exp}\},$$

where '()' denotes the empty string in $\mathbf{Var}^*$. The rest of Γ_s are empty. We sometimes write the index set Γ as

$$\{1,\ \mathrm{prod}(+,+),\ \mathrm{exp}(-,+)\}.$$

The product functor 'prod' is associated with two natural transformations which give projection morphisms.

$$\pi_1\colon \lambda(X,Y).\mathrm{prod}(X,Y) \dot{\to} \lambda(X,Y).X$$
$$\pi_2\colon \lambda(X,Y).\mathrm{prod}(X,Y) \dot{\to} \lambda(X,Y).Y$$

If there is no ambiguity, we may write them by listing their components as follows:

$$\pi_{1\,A,B}\colon \mathrm{prod}(A,B) \to A$$
$$\pi_{2\,A,B}\colon \mathrm{prod}(A,B) \to B$$

Functors (Γ)	$1 \quad \mathrm{prod}(+,+) \quad \exp(-,+)$
Natural Transformations (Δ)	$\pi_1 : \mathrm{prod}(A,B) \to A$ $\pi_2 : \mathrm{prod}(A,B) \to B$ $\mathrm{ev} : \mathrm{prod}(\exp(A,B),A) \to B$
Factorizers (Ψ)	$\dfrac{}{! : A \to 1}$ $\dfrac{f : C \to A \qquad g : C \to B}{\mathrm{pair}(f,g) : C \to \mathrm{prod}(A,B)}$ $\dfrac{h : \mathrm{prod}(C,A) \to B}{\mathrm{curry}(h) : C \to \exp(A,B)}$

Figure 2.1: CSL Signature for Cartesian Closed Categories

We might even omit subscripts from $\pi_{1\,A,B}$ and $\pi_{2\,A,B}$. The exponential functor 'exp' is associated with one natural transformation which gives evaluation morphisms.

$$\mathrm{ev} : \lambda(X,Y).\mathrm{prod}(\exp(X,Y),X) \dot{\to} \lambda(X,Y).Y$$

Note that the variance of these two functors is $\top+$. Therefore, Δ is

$$\begin{aligned}
\Delta_{\lambda(X,Y).\mathrm{prod}(X,Y),\lambda(X,Y).X} &= \{\pi_1\} \\
\Delta_{\lambda(X,Y).\mathrm{prod}(X,Y),\lambda(X,Y).Y} &= \{\pi_2\} \\
\Delta_{\lambda(X,Y).\mathrm{prod}(\exp(X,Y),X),\lambda(X,Y).Y} &= \{\mathrm{ev}\}
\end{aligned}$$

For any other combinations of closed functorial expression K and L, $\Delta_{K,L}$ is empty.

Finally, there are three factorizers for the three functors: '!' for '1', 'pair' for 'prod' and 'curry' for 'exp'.

$$\frac{}{! : \lambda(X).X \to \lambda(X).1}$$

$$\frac{f : \lambda(X,Y,Z).Z \to \lambda(X,Y,Z).X \qquad g : \lambda(X,Y,Z).Z \to \lambda(X,Y,Z).Y}{\mathrm{pair}(f,g) : \lambda(X,Y,Z).Z \to \lambda(X,Y,Z).\mathrm{prod}(X,Y)}$$

$$\frac{h : \lambda(X,Y,Z).\mathrm{prod}(Z,X) \to \lambda(X,Y,Z).Y}{\mathrm{curry}(h) : \lambda(X,Y,Z).Z \to \lambda(X,Y,Z).\exp(X,Y)}$$

If there is no ambiguity, we might write these rules down as in figure 2.1, where we summarize the definition of the CSL signature for cartesian closed categories. We will omit the tedious formal definition of Ψ as an indexed set. □

As we mentioned earlier, a CSL signature is one sorted, but because it handles higher objects, it is no less powerful than a many sorted equational signature. Let us demonstrate this. An equational signature is given by a pair $\langle S, \Sigma \rangle$ where S is a set (of sort names) and Σ is an $S^* \times S$-indexed set (of operator names). We will translate it to a corresponding CSL signature $\langle \Gamma, \Delta, \Psi \rangle$. Since functors play a role of sorts, we declare a constant functor for each sort in S.

$$\Gamma_{()} = S,$$

and we have a binary product functor to deal with sequences of sorts,[3] and the terminal object for constants.

$$\Gamma_{++} = \{ \text{ prod } \} \qquad \text{and} \qquad \Gamma_{()} = \{ 1 \}$$

Operations will be translated to natural transformations: for each operation $o \in \Sigma_{s_1 \ldots s_n, s}$, we have a natural transformation of the same name.

$$o\colon \text{prod}(s_1, \text{prod}(\ldots, \text{prod}(s_{n-1}, s_n))) \to s$$

that is,

$$\Delta_{\lambda().\text{prod}(s_1,\text{prod}(\ldots,\text{prod}(s_{n-1},s_n))),\lambda().s} = \Sigma_{s_1 \ldots s_n, s}$$

We also have two projections for 'prod' in Δ and Ψ has only one factorizer 'pair' for pairing. It is easy to see that the CSL signature $\langle \Gamma, \Delta, \Psi \rangle$ corresponds to the equational signature $\langle S, \Sigma \rangle$.

Proposition 2.2.3: A many-sorted equational signature can be represented by a CSL signature. □

It is interesting to know that we represented a many-sorted equational signature by a category with products because an *algebraic theory* can exactly be given as a category with products [Lawvere 63].

Let us make CSL signatures form a category by extending a pre-CSL signature morphism in a natural way. Intuitively, a signature morphism (not only in CSL but in general) does some renamings of symbols and/or some mergings of symbols.

Definition 2.2.4: A *CSL signature morphism* σ from a CSL signature $\langle \Gamma, \Delta, \Psi \rangle$ to a CSL signature $\langle \Gamma', \Delta', \Psi' \rangle$ is a triple $\langle \sigma', \sigma'', \sigma''' \rangle$ consisting of

1. a $\mathbf{Var}^*$-indexed function $\sigma'_{v_1 \ldots v_n}\colon \Gamma_{v_1 \ldots v_n} \to \Gamma'_{v_1 \ldots v_n}$ for mapping functor names,
2. a $\Lambda \times \Lambda$-indexed function (where $\Lambda = \mathbf{CFE}(\Gamma)$) $\sigma''_{K,L}\colon \Delta_{K,L} \to \Delta'_{\sigma' K, \sigma' L}$ for mapping natural transformation names, and
3. a $(\Lambda \times \Lambda)^* \times (\Lambda \times \Lambda)$-indexed function

   $$\sigma'''_{\langle\langle K_1,L_1\rangle \ldots \langle K_m,L_m\rangle,\langle K,L\rangle\rangle}\colon \Psi_{\langle\langle K_1,L_1\rangle \ldots \langle K_m,L_m\rangle,\langle K,L\rangle\rangle} \to \Psi'_{\langle\langle \sigma' K_1,\sigma' L_1\rangle \ldots \langle \sigma' K_m,\sigma' L_m\rangle,\langle \sigma' K,\sigma' L\rangle\rangle}$$

   for mapping factorizer names.

We often write σF for $\sigma'_{v_1 \ldots v_n}(F)$, $\sigma\alpha$ for $\sigma''_{K,L}(\alpha)$ and $\sigma\psi$ for $\sigma'''_{\langle\langle K_1,L_1\rangle \ldots,\langle K,L\rangle\rangle}(\psi)$. □

Definition 2.2.5: The *category of CSL signatures* **CSig** has CSL signatures as its objects and CSL signature morphisms as its morphisms. The identity morphism on a CSL signature $\langle \Gamma, \Delta, \Psi \rangle$ consists of the corresponding identity functions for the components and the composition of CSL morphisms is given by combining the component-wise compositions as indexed functions. This clearly forms a category. □

[3] We could have n-ary product functors for all natural numbers as well, but since we can represent them using a binary one, we only declare the binary one.

2.3 Structures of Categorical Specification Language

In this section, we will define *CSL structures*. A CSL signature specifies symbols for functors, natural transformations and factorizers, so intuitively, a CSL structure is a category associated with these functors, natural transformations and factorizers.

Definition 2.3.1: Given a CSL signature $\langle \Gamma, \Delta, \Psi \rangle$, a *CSL structure* $\langle \mathcal{C}, \xi \rangle$ is a small category $\mathcal{C}$ together with an assignment ξ

1. ξ assigns each functor name of varity $v_1 \dots v_n$ to a functor $\mathcal{C}^{v_1} \times \cdots \times \mathcal{C}^{v_n} \to \mathcal{C}$.[4] As we have seen in section 2.1, ξ can be extended to the assignment of closed functorial expressions to functors.

2. ξ assigns each natural transformation name $\alpha \in \Delta_{K,L}$, where K and L are closed functorial expressions of n variables, to a set of $\mathcal{C}$ morphisms

$$\xi\alpha_{A_1,\dots,A_n} : \xi K(A_1, \dots, A_n) \to \xi L(A_1, \dots, A_n)$$

   for arbitrary $\mathcal{C}$ objects $A_1, \dots, A_n$.[5]

3. Each factorizer symbol $\psi \in \Psi_{\langle\langle K_1,L_1\rangle,\dots,\langle K,L\rangle\rangle}$, where $K_1, L_1, \dots, K_m, L_m, K, L$ are closed functorial expressions of n variables, is assigned to a set of **Set** functions

$$\xi\psi_{A_1,\dots,A_n} : \prod_{i=1}^{m} \mathrm{Hom}_{\mathcal{C}}(\xi K_i(A_1, \dots, A_n), \xi L_i(A_1, \dots, A_n)) \\ \to \mathrm{Hom}_{\mathcal{C}}(\xi K(A_1, \dots, A_n), \xi L(A_1, \dots, A_n))$$

   for arbitrary $\mathcal{C}$ objects $A_1, \dots, A_n$. □

Note that we do not assign a natural transformation symbol to a natural transformation, but it is mapped to a set of morphisms and whether they form a natural transformation or not is left to be stated by equations.

Example 2.3.2: Let $\langle \Gamma, \Delta, \Psi \rangle$ be the CSL signature for cartesian closed categories defined in example 2.2.2. Then, any cartesian closed category is a CSL structure of this signature by obvious assignment of the symbols to the functors, natural transformations and factorizers. However, the converse is not true. We can have a category

[4] We could say ξ is a $\mathbf{Var}^*$-indexed function

$$\xi_{v_1\dots v_n} : \Gamma_{v_1\dots v_n} \to \mathbf{Funct}(\mathcal{C}^{v_1} \times \cdots \times \mathcal{C}^{v_n}, \mathcal{C}).$$

[5] We could express it as a $\Lambda\times\Lambda$-indexed function (where $\Lambda = \mathbf{CFE}(\Gamma)$)

$$\xi_{K,L} : \Delta_{K,L} \to \mathbf{Nat}(\xi K|^{\top\dots\top}, \xi L|^{\top\dots\top}).$$

which has three functors, three natural-transformation-look-alikes and three factorizer-look-alikes. CSL structures do not require mapping natural transformation symbols to natural transformations but only to a set of morphisms, nor do they require mapping factorizer symbols to factorizers. The factorizer-look-alikes may not give unique morphisms or may not commute some diagrams. []

CSL structure morphisms are defined simply as a kind of homomorphisms. They keep the structure nicely.

Definition 2.3.3: Given a CSL signature $\langle \Gamma, \Delta, \Psi \rangle$, a *CSL structure morphism* from a CSL structure $\langle \mathcal{C}, \xi \rangle$ to another $\langle \mathcal{D}, \zeta \rangle$ is a covariant functor $T: \mathcal{C} \to \mathcal{D}$ such that

1. for any F in $\Gamma_{v_1 \ldots v_n}$
$$T \circ \xi F = \zeta F \circ (T^{v_1} \times \cdots \times T^{v_n})$$
holds,
$$\begin{array}{ccc} \mathcal{C}^{v_1} \times \cdots \times \mathcal{C}^{v_n} & \xrightarrow{\xi F} & \mathcal{C} \\ {\scriptstyle T^{v_1} \times \cdots \times T^{v_n}} \downarrow & \circlearrowright & \downarrow \\ \mathcal{D}^{v_1} \times \cdots \times \mathcal{D}^{v_n} & \xrightarrow[\zeta F]{} & \mathcal{D} \end{array}$$

2. for any $\alpha \in \Delta_{K,L}$ and for any $\mathcal{C}$ objects $A_1, \ldots, A_n$
$$T(\xi \alpha_{A_1,\ldots,A_n}) = \zeta \alpha_{T(A_1),\ldots,T(A_n)},$$
and

3. for any $\psi \in \Psi_{\langle K_1,L_1 \rangle \ldots \langle K_m,L_m \rangle, \langle K,L \rangle}$, for any $\mathcal{C}$ objects $A_1, \ldots, A_n$ and for any $\mathcal{C}$ morphisms $f_i: \xi K_i(A_1, \ldots, A_n) \to \xi L_i(A_1, \ldots, A_n)$ $(i = 1, \ldots, m)$,
$$T(\xi \psi_{A_1,\ldots,A_n}(f_1, \ldots, f_m)) = \zeta \psi_{T(A_1),\ldots,T(A_n)}(T(f_1), \ldots, T(f_m)). \; []$$

Hence, the category of CSL structures is:

Definition 2.3.4: For a CSL signature $\langle \Gamma, \Delta, \Psi \rangle$, the category of CSL structures, **CMod**$(\langle \Gamma, \Delta, \Psi \rangle)$, has CSL structures as objects and CSL structure morphisms as morphisms; the identity morphism on $\langle \mathcal{C}, \xi \rangle$ is the identity functor $\mathbf{I}_{\mathcal{C}}$ and the composition of morphisms is the composition of their underlying functors. []

2.4 Functorial Calculus (revisit)

In section 2.1 we saw functorial expressions denote functors. In this section we will see an expression involving natural transformation symbols and factorizer symbols denote

a set of morphisms.[6] For example, under the signature of cartesian closed categories given in example 2.2.2, what should the following expression denote?

$$\pi_1 \circ \mathrm{prod}(\mathrm{curry}(\pi_2), \mathbf{I})$$

We even have a problem for expressions like $\pi_1 \circ \pi_2$ because π_1 and π_2 are not projections of the same product: π_2 is of $\mathrm{prod}(A, \mathrm{prod}(B, C))$ and π_1 is of $\mathrm{prod}(B, C)$. Actually, natural transformations are polymorphic like ML functions are.[7] As we defined in definition 2.3.1, π_1 denotes a set of morphisms.

$$\xi\pi_{1\,A,B} : \xi\,\mathrm{prod}(A, B) \to A$$

We have to figure out for each occurrence of π_1 what A and B are.

Definition 2.4.1: For a CSL signature $\langle \Gamma, \Delta, \Psi \rangle$, we have a set $\mathbf{Exp}(\Gamma, \Delta, \Psi)$ of *CSL expressions* defined by the following BNF.

$$e ::= \mathbf{I} \mid e_1 \circ e_2 \mid \alpha \mid \psi(e_1, \ldots, e_m) \mid F(e_1, \ldots, e_n) \mid f$$

where $\alpha \in \Delta$, $\psi \in \Psi$ and f is a variable for morphisms. We also have a set $\mathbf{AExp}(\Gamma, \Delta, \Psi)$ of *CSL annotated expressions* defined by the following BNF.

$$e ::= \mathbf{I}[K] \mid e_1 \circ e_2 \mid \alpha[K_1, \ldots, K_n] \mid \psi[K_1, \ldots, K_n](e_1, \ldots, e_m) \mid F(e_1, \ldots, e_n) \mid f$$

where $K_1, \ldots, K_n$ are closed functorial expressions over Γ. []

It is trivial to see that for each annotated expression there is a corresponding expression (i.e. forgetting all the annotations, [...]), which we call *skeleton* of the annotated expression. We are going to type-check an expression first and, then, we determine its denotation. Annotated expressions are used to remember the type-checking information inside expressions. We will give the typing rules for annotated expressions and show that every expression has the most general annotated expression and we take the type of this annotated expression as the type of the expression.

First, we define the notion of unification.

Definition 2.4.2: A closed functorial expression K of n variables is said to be *more general* than a closed functorial expression K' of m variables if there are closed functorial expressions $K_1, \ldots, K_n$ of m variables such that

$$K[K_1, \ldots, K_n] \equiv K'$$

[6] Some may want it to denote a natural transformation, but the treatment of factorizers seems very complicated to do so.

[7] Conversely, ML polymorphic functions are natural transformations. The author has not yet seen the definite statement of this, but it is a folklore among computer scientists dealing with category theory. This fact is rarely used in practice, but it sometimes helps to understand the behaviour of polymorphic functions. For example, any ML function f of type $\alpha\mathrm{list} \to \mathrm{int}$ should never depend on elements in the list but only to the length of list. As another example, if f is of type $\alpha\mathrm{list} \to \alpha\mathrm{list}$ and if we apply it to an integer list $[1, 4, 3, 5]$ and get $[5, 1, 3]$, we know the result of applying f to $[2, 8, 6, 10]$ (each element is doubled) without actually applying it. The result should be $[10, 2, 6]$, i.e. each element of the result is doubled as well.

where $\equiv$ is the equivalence relation ignoring variable renaming. (Trivially, $K \equiv K'$ implies $\xi K = \xi K'$.) []

Definition 2.4.3: Closed functorial expressions K and L are said to be *unifiable* when there is a closed functorial expression K' such that K' is less general than K as well as L. []

Proposition 2.4.4: If closed functorial expressions K and L are unifiable, then there is a most general unification, that is, there exist $K_1, \ldots, K_n$ and $L_1, \ldots, L_m$ such that $K[K_1, \ldots, K_n] \equiv L[L_1, \ldots, L_m]$ and for any K' which is less general than K and K' there are $K'_1, \ldots, K'_l$ such that $K' \equiv K[K_1, \ldots, K_n][K'_1, \ldots, K'_l]$.
Proof: Same as ordinary unification of terms. []

Let us now define the type of annotated expressions.

Definition 2.4.5: Let $\langle \Gamma, \Delta, \Psi \rangle$ be a CSL signature. An annotated expression e has a type $\rho \vdash e\colon K \to L$ when it can be derived from the following rules, where ρ is a given assignment of each morphism variable to its type and K and L are closed functorial expressions.

1. For the identity, $\rho \vdash \mathbf{I}[K]\colon K \to K$.

2. For the composition,

$$\frac{\rho \vdash e_1\colon K' \to L \qquad \rho \vdash e_2\colon K \to K'}{\rho \vdash e_1 \circ e_2\colon K \to L}.$$

3. For a natural transformation $\alpha \in \Delta_{K,L}$, where K and L are of n variables,

$$\rho \vdash \alpha[K_1, \ldots, K_n]\colon K[K_1, \ldots, K_n] \to L[K_1, \ldots, K_n]$$

4. For a factorizer $\psi \in \Psi_{\langle\langle K_1, L_1\rangle \ldots \langle K_m, L_m\rangle, \langle K, L\rangle\rangle}$, where K_i, L_i, K, L are of n variables,

$$\frac{\rho \vdash e_i\colon K_i[K'_1, \ldots, K'_n] \to L_i[K'_1, \ldots, K'_n]}{\rho \vdash \psi[K'_1, \ldots, K'_n](e_1, \ldots, e_m)\colon K[K'_1, \ldots, K'_n] \to L[K'_1, \ldots, K'_n]}$$

5. For a functor $F \in \Gamma_{v_1 \ldots v_n}$

$$\frac{\rho \vdash e_i\colon K_i \xrightarrow{v_i} L_i}{\rho \vdash F(e_1, \ldots, e_n)\colon F[K_1, \ldots, K_n] \to F[L_1, \ldots, L_n]}$$

   where $e_i\colon K_i \xrightarrow{v_i} L_i$ is $e_i\colon K_i \to L_i$ if v_i is $+$ or $\perp$, $e_i\colon L_i \to K_i$ if v_i is $-$ and $\mathbf{I}\colon K_i \to K_i$ if v_i is $\top$.

6. For a morphism variable f, $\rho \vdash f\colon \rho(f)$. []

Definition 2.4.6: We say an annotated expression e is *more general* than e' if there exist closed functorial expressions $K_1, \ldots, K_n$ such that $e' \equiv e[K_1, \ldots, K_n]$, where $e[K_1, \ldots, K_n]$ is e with all its annotations being composed with $K_1, \ldots, K_n$, e.g. $\alpha[L_1, \ldots, L_m][K_1, \ldots,$ is $\alpha[L_1[K_1, \ldots, K_n], \ldots, L_m[K_1, \ldots, K_n]]$. []

Proposition 2.4.7: If an annotated expression e has a type $\rho \vdash e: K \to L$, $e[K_1, \ldots, K_n]$ has the following type.

$$\rho[K_1, \ldots, K_n] \vdash e[K_1, \ldots, K_n]: K[K_1, \ldots, K_n] \to L[K_1, \ldots, K_n]$$

where $\rho[K_1, \ldots, K_n](f)$ is $K'[K_1, \ldots, K_n] \to L'[K_1, \ldots, K_n]$ when $\rho(f)$ is $K' \to L'$.
Proof: It can easily proved from definition 2.4.5 by structural induction on e. []

Proposition 2.4.8: Let $e \in \mathbf{Exp}(\Gamma, \Delta, \Psi)$ be an expression of a CSL signature $\langle \Gamma, \Delta, \Psi \rangle$. If there exists an annotated expression e' whose skeleton is e and if it has a type, then there exists a most general annotated expression which has a type and whose skeleton is e.
Proof: It can be proved by structural induction on e. Here, we present an algorithm of calculating a most general annotated expression.

1. If e is $\mathbf{I}$, the most general annotated expression is $\mathbf{I}[I]$, where I is the identity closed functorial expression $\lambda(X).X$.

2. If e is $e_1 \circ e_2$, from induction hypothesis, we have the most general annotated expressions for e_1 and e_2.

   $$\rho_1 \vdash e_1': K \to K' \qquad \rho_2 \vdash e_2': L \to L'$$

   We unify K with L' achieving $K[K_1, \ldots, K_n] \equiv L'[L_1, \ldots, L_m]$. The most general annotated expression for e and its type is

   $$\rho \vdash e_1'[K_1, \ldots, K_n] \circ e_2'[L_1, \ldots, L_m]: L[L_1, \ldots, L_m] \to K'[K_1, \ldots, K_n]$$

   where ρ is the result of combining $\rho_1[K_1, \ldots, K_n]$ and $\rho_2[L_1, \ldots, L_m]$.

3. If e is a natural transformation $\alpha \in \Delta_{K,L}$, the most general annotated expression and its type is

   $$\rho \vdash \alpha[\Pi_1^n, \ldots, \Pi_n^n]: K \to L,$$

   where Π_i^n is the closed functorial expression $\lambda(X_1, \ldots, X_n).X_i$.

4. If e is $\psi(e_1, \ldots, e_m)$ for a factorizer $\psi \in \Psi_{\langle\langle K_1, L_1\rangle, \ldots, \langle K, L\rangle\rangle}$, from induction hypothesis, we have annotated expressions e_i' such that

   $$\rho_i \vdash e_i': K_i' \to L_i'.$$

   We unify K_i' with K_i and L_i' with L_i. If the unification is successful, we have the most general annotated expression for e.

   $$\rho \vdash \psi[J_1, \ldots, J_n](e_1'[J_{11}', \ldots], \ldots, e_m'[J_{1m}', \ldots]): K[J_1, \ldots, J_n] \to L[J_1, \ldots, J_n]$$

where $J_1, \ldots, J_n, J'_{11}, \ldots$ are the substitution for K_i and L_i obtained from the unification and ρ is the result of combining $\rho_i[J'_{1i}, \ldots]$.

5. If e is $F(e_1, \ldots, e_n)$, let the most general annotated expressions for e_i be

$$\rho_i \vdash e'_i : K_i \xrightarrow{v_i} L_i.$$

Then, the most general annotated expression for e and its type is

$$\rho \vdash F(e'_1, \ldots, e'_n) : F[K_1, \ldots, K_n] \to F[L_1, \ldots, L_n].$$

6. If e is a morphism variable f, its most general annotated expression is itself and has the following type.

$$\rho \vdash f : \lambda(X, Y).X \to \lambda(X, Y).Y \qquad \square$$

From this proposition,

Definition 2.4.9: We define the type of an expression $e \in \mathbf{Exp}(\Gamma, \Delta, \Psi)$ to be the type of the most general annotated expression e' whose skeleton is e. $\square$

Let us finally define the denotation of expressions in $\mathbf{Exp}(\Gamma, \Delta, \Psi)$ when a CSL structure $\langle \mathcal{C}, \xi \rangle$ is given. Since each expression e is associated uniquely to the most general annotated expression e' by proposition 2.4.8, we can define the denotation of e to be that of e' and define the denotation of annotated expressions.

Definition 2.4.10: Let $\langle \Gamma, \Delta, \Psi \rangle$ be a CSL signature and $\langle \mathcal{C}, \xi \rangle$ be a CSL structure. For an annotated expression $e \in \mathbf{AExp}(\Gamma, \Delta, \Psi)$ of type $\rho \vdash e : K \to L$ (where K and L are of l variables), we define its denotation, ξe, to be a set of morphisms

$$(\xi e)_{A_1, \ldots, A_l} : \xi K(A_1, \ldots, A_l) \to \xi L(A_1, \ldots, A_n)$$

for any $\mathcal{C}$ objects $A_1, \ldots, A_l$ and for any morphism variable assignment ω (where $\omega(f, A_1, \ldots, A_l)$ gives a morphism of type $\xi K'(A_1, \ldots, A_l) \to \xi L'(A_1, \ldots, A_l)$ when $\rho(f)$ is $K' \to L'$).

1. For the identity,

$$(\xi \mathbf{I}[K])_{A_1, \ldots, A_l} \stackrel{\text{def}}{=} \mathbf{I}_{\xi K[A_1, \ldots, A_l]}$$

2. For compositions,

$$(\xi e_1 \circ e_2)_{A_1, \ldots, A_l} \stackrel{\text{def}}{=} (\xi e_1)_{A_1, \ldots, A_l} \circ (\xi e_2)_{A_1, \ldots, A_l}$$

3. For natural transformations,

$$(\xi \alpha[K_1, \ldots, K_n])_{A_1, \ldots, A_l} \stackrel{\text{def}}{=} \xi \alpha_{\xi K_1(A_1, \ldots, A_l), \ldots, \xi K_n(A_1, \ldots, A_l)}$$

4. For factorizers,

$$(\xi\psi[K_1,\ldots,K_n](e_1,\ldots,e_m))_{A_1,\ldots,A_l} \stackrel{\text{def}}{=} \psi_{\xi K_1(A_1,\ldots,A_l),\ldots,\xi K_n(A_1,\ldots,A_l)}((\xi e_1)_{A_1,\ldots,A_l},\ldots,(\xi e_m)_{A_1,\ldots,A_l})$$

5. For functors,

$$(\xi F(e_1,\ldots,e_n))_{A_1,\ldots,A_l} \stackrel{\text{def}}{=} \xi F((\xi e_1)_{A_1,\ldots,A_l},\ldots,(\xi e_n)_{A_1,\ldots,A_l})$$

6. For morphism variables,

$$(\xi f)_{A_1,\ldots,A_n} \stackrel{\text{def}}{=} \omega(f,A_1,\ldots,A_l)$$

Proof of well-definedness: We have to show, for example, for a natural transformation $\alpha \in \Delta_{K,L}$, $(\xi\alpha[K_1,\ldots,K_n])_{A_1,\ldots,A_l}$ is a morphism from

$$\xi K[K_1,\ldots,K_n](A_1,\ldots,A_l) \qquad \text{to} \qquad \xi L[K_1,\ldots,K_n](A_1,\ldots,A_l).$$

This holds because from definition 2.3.1 $\xi\alpha_{\xi K_1(A_1,\ldots,A_l),\ldots,\xi K_n(A_1,\ldots,A_l)}$ is a morphism from $\xi K(\xi K_1(A_1,\ldots,A_l),\ldots))$ to $\xi L(\xi K_1(A_1,\ldots,A_l),\ldots)$ and from proposition 2.1.18 it is from $\xi K[K_1,\ldots,K_n](A_1,\ldots,A_l)$ to $\xi L[K_1,\ldots,K_n](A_1,\ldots,A_l)$. □

Example 2.4.11: Let $\langle\Gamma,\Delta,\Psi\rangle$ be a CSL signature for cartesian closed categories presented in example 2.2.2 and $\langle\mathcal{C},\xi\rangle$ be a CSL structure of this signature where $\mathcal{C}$ is a cartesian closed category and ξ is the standard assignment (i.e. the product symbol to the product functor and so on). Let us find out the denotation of $\mathrm{ev}\circ\mathrm{pair}(f,\pi_2)$. First, we have to find out the corresponding most general annotated expression and its type by the algorithm used to prove proposition 2.4.8.

1. $\mathrm{ev}\circ\mathrm{pair}(f,\pi_2)$ is given by composing ev and $\mathrm{pair}(f,\pi_2)$, so we need to calculate the most general annotated expressions for these two sub-expressions first.

   (a) ev is a natural transformation, and its most general annotated expression is

   $$\rho_1 \vdash \mathrm{ev}[\Pi_1^2,\Pi_2^2]\colon \mathrm{prod}[\mathrm{exp},\Pi_1^2] \to \Pi_2^2.$$

   (b) $\mathrm{pair}(f,\pi_2)$ is given by applying the factorizer pair to f and π_2.

      i. 'f' is a morphism variable, its most general annotated expression is

      $$\rho_2 \vdash f\colon \Pi_1^2 \to \Pi_2^2.$$

      ii. π_2 is a natural transformation, and its most general annotated expression is

      $$\rho_3 \vdash \pi_2[\Pi_1^2,\Pi_1^2]\colon \mathrm{prod} \to \Pi_2^2.$$

pair has the type

$$\langle\langle\Pi_3^3,\Pi_1^3\rangle\langle\Pi_3^3,\Pi_2^3\rangle,\langle\Pi_3^3,\mathrm{prod}[\Pi_1^3,\Pi_2^3]\rangle\rangle$$

By unification, we get the most general annotated expression for $\mathrm{pair}(f,\pi_2)$.

$$\rho_4 \vdash \mathrm{pair}[\Pi_3^3,\Pi_2^3,\mathrm{prod}[\Pi_1^3,\Pi_2^3]](f,\pi_2[\Pi_1^3,\Pi_2^3]): \\ \mathrm{prod}[\Pi_1^3,\Pi_2^3]\to\mathrm{prod}[\Pi_3^3,\Pi_2^3]$$

where ρ_4 maps f to $\mathrm{prod}[\Pi_1^3,\Pi_2^3]\to\Pi_3^3$.

Unifying $\mathrm{prod}[\Pi_3^3,\Pi_2^3]$ and $\mathrm{prod}[\mathrm{exp},\Pi_1^2]$, we get the most general annotated expression for $\mathrm{ev}\circ\mathrm{pair}(f,\pi_2)$.

$$\rho_5 \vdash \mathrm{ev}[\Pi_2^3,\Pi_3^3]\circ\mathrm{pair}[\mathrm{exp}[\Pi_2^3,\Pi_3^3],\Pi_2^3,\mathrm{prod}[\Pi_1^3,\Pi_2^3]](f,\pi_2[\Pi_1^3,\Pi_2^3]) \\ :\mathrm{prod}[\Pi_1^3,\Pi_2^3]\to\Pi_3^3$$

where ρ_5 maps f to $\mathrm{prod}[\Pi_1^3,\Pi_2^3]\to\mathrm{exp}[\Pi_2^3,\Pi_3^3]$.

From definition 2.4.10, the denotation of this annotated expression is a set of morphisms for objects A, B and C and a morphism variable assignment

$$\omega(f,A,B,C):\xi\mathrm{prod}(A,B)\to\xi\mathrm{exp}(B,C).$$

1. $(\xi\mathrm{ev}[\Pi_2^3,\Pi_3^3])_{A,B,C}=\xi\mathrm{ev}_{B,C}$
2. $(\xi f)_{A,B,C}=\omega(f,A,B,C)$
3. $(\xi\pi_2[\Pi_1^3,\Pi_2^3])_{A,B,C}=\xi{\pi_2}_{A,B}$
4. $(\xi\mathrm{pair}[\ldots](f,\pi_2[\ldots]))_{A,B,C}=$
   $\xi\mathrm{pair}_{\xi\mathrm{exp}(B,C),B,\xi\mathrm{prod}(A,B)}(\omega(f,A,B,C),\xi{\pi_2}_{A,B})$
5. $(\xi\mathrm{ev}[\ldots]\circ\mathrm{pair}[\ldots](f,\pi_2[\ldots]))_{A,B,C}=$
   $\xi\mathrm{ev}_{B,C}\circ\xi\mathrm{pair}_{\xi\mathrm{exp}(B,C),B,\xi\mathrm{prod}(A,B)}(\omega(f,A,B,C),\xi{\pi_2}_{A,B})$

Therefore, the denotation of $\mathrm{ev}\circ\mathrm{pair}(f,\pi_2)$ is

$$\xi\mathrm{ev}_{B,C}\circ\xi\mathrm{pair}_{\xi\mathrm{exp}(B,C),B,\xi\mathrm{prod}(A,B)}(\omega(f,A,B,C),\xi{\pi_2}_{A,B}) \qquad \square$$

2.5 Sentences and Satisfaction Relation of Categorical Specification Language

In this section, we will finish defining the specification language CSL at last. First, we define what a CSL sentence is.

Definition 2.5.1: A *CSL conditional equation* is a sequence of CSL expression pairs and a CSL expression pair. We usually write it as

$$e_1 = e_1' \wedge \ldots \wedge e_n = e_n' \Rightarrow e = e',$$

or simply $e = e'$ if the preceding sequence is empty. To be typed, it needs to share the same morphism variable environment, e_i and e_i' have to have the same type and e and e' have to have the same type. We may write the types as follows:

$$\rho \vdash e_1 = e_1' : K_1 \to L_1 \wedge \ldots \wedge e_n = e_n' : K_n \to L_n \Rightarrow e = e' : K \to L$$

We write $\mathbf{CEq}(\Gamma, \Delta, \Psi)$ for the set of all the CSL conditional equations which can be typed. []

The CSL conditional equations are the CSL sentences. We now have to define the *satisfaction relation* for CSL. We have separately defined what CSL structures are and what CSL conditional equations are. The satisfaction relation connects these two together so that we can say a CSL conditional equation holds or not in a particular CSL structure.

Definition 2.5.2: Let $\langle \Gamma, \Delta, \Psi \rangle$ be a CSL signature. A CSL structure $\langle \mathcal{C}, \xi \rangle$ satisfies a CSL conditional equation

$$e_1 = e_1' \wedge \ldots \wedge e_n = e_n' \Rightarrow e = e'$$

having a type

$$\rho \vdash e_1 = e_1' : K_1 \to L_1 \wedge \ldots \wedge e_n = e_n' : K_n \to L_n \Rightarrow e = e' : K \to L,$$

if and only if for any objects $A_1, \ldots, A_l$ and any morphism variable assignment ω we have either

1. a CSL equation $e_i = e_i'$ such that $(\xi e_i)_{A_1,\ldots,A_l} \neq (\xi e_i')_{A_1,\ldots,A_l}$, or
2. $(\xi e)_{A_1,\ldots,A_l} = (\xi e')_{A_1,\ldots,A_l}$.

We will write

$$\langle \mathcal{C}, \xi \rangle \models e_1 = e_1' \wedge \ldots \wedge e_n = e_n' \Rightarrow e = e'$$

when $\langle \mathcal{C}, \xi \rangle$ satisfies this CSL conditional equation. []

We have defined the specification language, CSL: CSL signatures, CSL structures, CSL conditional equations and CSL satisfaction relation. We could have defined it as an institution (see [Goguen and Burstall 83]) by defining **CMod** as a contravariant functor and showing CSL satisfaction condition.

Finally, let us finish presenting the CSL theory (i.e. a pair of CSL signature and CSL conditional equations) of cartesian closed categories.

Example 2.5.3: We have presented the signature for cartesian closed categories in example 2.2.2 (or figure 2.1), so all we have to do is to list the CSL conditional equations. (Note that they are not conditional for this example.)

1. $f = !$
2. $1 = \mathbf{I}$
3. $\text{pair}(\pi_1, \pi_2) = \mathbf{I}$
4. $\pi_1 \circ \text{pair}(f, g) = f$
5. $\pi_2 \circ \text{pair}(f, g) = g$
6. $\text{pair}(f, g) \circ h = \text{pair}(f \circ h, g \circ h)$
7. $\text{prod}(f, g) = \text{pair}(f \circ \pi_1, g \circ \pi_2)$
8. $\text{curry}(\text{ev}) = \mathbf{I}$
9. $\text{ev} \circ \text{curry}(\text{prod}(f, \mathbf{I})) = f$
10. $\text{curry}(f) \circ g = \text{curry}(f \circ \text{prod}(g, \mathbf{I}))$
11. $\exp(f, g) = \text{curry}(g \circ \text{eval} \circ \text{prod}(f, \mathbf{I}))$

The naturality of π_1, π_2 and ev can be derived from these equations. For example, the naturality of π_1 is shown by

$$\pi_1 \circ \text{prod}(f, g) = \pi_1 \circ \text{pair}(f \circ \pi_1, g \circ \pi_2) = f \circ \pi_1. \qquad \square$$

2.6 Free Categories

One of the major advantages of algebraic specification methods using equations or conditional equations over other specification methods is that any theory has an initial model (i.e. the initial object in the category of models which satisfy the theory). This also holds for CSL, and in this section, we will construct an initial structure for a CSL theory. Remember that a CSL structure is a pair of a category and an interpretation. The category of an initial CSL structure corresponds to a so-called *free category*.

Given a CSL signature $\langle \Gamma, \Delta, \Psi \rangle$ and a set Θ of CSL conditional equations, we are going to define a special category $\mathcal{C}$ and an interpretation ξ. For simplicity, we assume that Γ does not contain any free-variant functors. (We can always get such a signature by discarding free-variant arguments. This does not affect its semantics at all.)

Definition 2.6.1: We say that a closed functorial expression is *ground* if it has no variables, that is, its basic form is $\lambda().E$. We take ground closed functorial expressions as the objects of $\mathcal{C}$. $\square$

The definition of morphisms of $\mathcal{C}$ is a little bit more complicated, so we define them step by step.

Definition 2.6.2: A ground annotated expression is an annotated expression such that

1. all the annotation, $[K_1, \ldots, K_n]$ consists of ground closed functorial expressions, and
2. it does not contain any morphism variables. []

Proposition 2.6.3: If a ground annotated expression e has a type $\rho \vdash e: K \to L$, and both K and L are ground functorial expressions.
Proof: We can easily prove it from definition 2.4.5 by structural induction. []

The following will give us the basis of the morphisms in $\mathcal{C}$.

Definition 2.6.4: For ground closed functorial expressions K and L, we define

$$\mathbf{GExp}(K, L) \stackrel{\text{def}}{=} \{\ e \mid e \text{ is ground and } \emptyset \vdash e: K \to L\ \}\ []$$

To make **GExp** proper morphisms, we introduce a family of equivalence relations $\equiv$ indexed by a pair of ground functorial expressions. Each $\equiv_{K,L}$ is an equivalence relation on $\mathbf{GExp}(K, L)$.

Definition 2.6.5: We define $\equiv$ to be the smallest relation satisfying the following conditions. (In the following, to simplify the presentation, we omit indexes of $\equiv$ if there is no ambiguity.)

1. $\equiv$ is an equivalence relation, that is, reflexive, symmetric and transitive.
2. If $e_1 \equiv_{K,L} e_1'$ and $e_2 \equiv_{K',K} e_2'$, then $e_1 \circ e_2 \equiv_{K',L} e_1' \circ e_2'$.
3. If $e \in \mathbf{GExp}(K, L)$, then $\mathbf{I}[L] \circ e \equiv e$ and $e \circ \mathbf{I}[K] \equiv e$.
4. For a functor symbol $F \in \Gamma_{v_1 \ldots v_n}$, if $e_1 \equiv e_1'$, ... and $e_n \equiv e_n'$, then $F(e_1, \ldots, e_n) \equiv F(e_1', \ldots, e_n')$.
5. For a factorizer symbol $\psi \in \Psi_{\langle\langle K_1', L_1'\rangle \ldots \langle K_l', L_l'\rangle, \langle K', L'\rangle\rangle}$ and ground functorial expressions $K_1, \ldots, K_n$, if $e_1 \equiv e_1'$, ... and $e_l \equiv e_l'$, then
$$\psi[K_1, \ldots, K_n](e_1, \ldots, e_l) \equiv \psi[K_1, \ldots, K_n](e_1', \ldots, e_l').$$
6. Finally, for a conditional CSL equation $e_1 = e_1' \wedge \ldots \wedge e_n = e_n' \Rightarrow e = e' \in \Theta$ whose type is
$$\rho \vdash e_1 = e_1': K_1' \to L_1' \wedge \ldots \wedge e_n = e_n': K_n' \to L_n' \Rightarrow e = e': K' \to L'$$
with $\rho(f_i)$ is $K_i'' \to L_i''$, ground functorial expressions $K_1, \ldots, K_m$, and ground annotated expressions
$$e_i'' \in \mathbf{GExp}(K_i''[K_1, \ldots, K_m], L_i''[K_1, \ldots, K_m]),$$
if for all $j = 1, \ldots, n$
$$(e_j[K_1, \ldots, K_m])[e_1'', \ldots, e_l''/f_1, \ldots, f_l] \equiv (e_j'[K_1, \ldots, K_m])[e_1'', \ldots, e_l''/f_1, \ldots, f_l],$$

then

$$(e[K_1,\ldots,K_m])[e''_1,\ldots,e''_l/f_1,\ldots,f_l] \equiv (e'[K_1,\ldots,K_m])[e''_1,\ldots,e''_l/f_1,\ldots,f_l]. \;[]$$

We can now define the morphisms of $\mathcal{C}$.

Definition 2.6.1 (continued)**:** The $\mathcal{C}$ morphisms from K to L are the equivalence classes of $\mathbf{GExp}(K,L)$ by $\equiv_{K,L}$, or simply,

$$\mathrm{Hom}_{\mathcal{C}}(K,L) \stackrel{\mathrm{def}}{=} \mathbf{GExp}(K,L)/\equiv_{K,L}.$$

We write $\langle e\rangle$ for the equivalence class to which e belongs. []

Proposition 2.6.6: $\mathcal{C}$ is a category.
Proof: The identity morphism of a $\mathcal{C}$ object K is given by $\langle \mathbf{I}[K]\rangle$. The composition of $\langle e\rangle\colon K \to L$ and $\langle e'\rangle\colon K' \to K$ is defined by

$$\langle e\rangle \circ \langle e'\rangle \stackrel{\mathrm{def}}{=} \langle e \circ e'\rangle.$$

The composition is trivially associative, and the satisfiability of the absorption rules of the identities is guaranteed by the third condition of the equivalence relation $\equiv$ defined in definition 2.6.5. []

Let us define an interpretation ξ so that $\langle \mathcal{C}, \xi\rangle$ is a CSL structure of the CSL signature $\langle \Gamma, \Delta, \Psi\rangle$.

Definition 2.6.7: The definition of ξ is divided into three.

1. For a functor symbol $F \in \Gamma_{v_1\ldots v_n}$, ξF is a functor $\mathcal{C}^{v_1\ldots v_n} \to \mathcal{C}$ defined by

$$\xi F(K_1,\ldots,K_n) \stackrel{\mathrm{def}}{=} F[K_1,\ldots,K_n)], \quad \text{and}$$
$$\xi F(\langle e_1\rangle,\ldots,\langle e_n\rangle) \stackrel{\mathrm{def}}{=} \langle F(e_1,\ldots,e_n)\rangle.$$

2. For a natural transformation symbol $\alpha \in \Delta_{K,L}$, $\xi\alpha$ is

$$\xi\alpha_{K_1,\ldots,K_n} \stackrel{\mathrm{def}}{=} \langle \alpha[K_1,\ldots,K_n]\rangle.$$

3. Finally, for a factorizer symbol $\psi \in \Psi_{\langle\langle K_1,L_1\rangle\ldots\langle K_n,L_n\rangle,\langle K,L\rangle\rangle}$, $\xi\psi$ is

$$\xi\psi_{K'_1,\ldots,K'_n}(\langle e_1\rangle,\ldots,\langle e_n\rangle) \stackrel{\mathrm{def}}{=} \langle \psi[K'_1,\ldots,K'_n](e_1,\ldots,e_n)\rangle. \;[]$$

Proposition 2.6.8: $\langle \mathcal{C}, \xi\rangle$ is a CSL structure of $\langle \Gamma, \Delta, \Psi\rangle$. Moreover, it is a theory structure of the CSL theory given by the set Θ of CSL conditional equations.
Proof: The condition 6 in definition 2.6.5 makes it satisfy the conditional equations. []

We have constructed a special CSL structure $\langle \mathcal{C}, \xi \rangle$, and we will next show that it is the initial object in the full-subcategory of $\mathbf{CMod}(\langle \Gamma, \Delta, \Psi \rangle)$ of all the structures satisfying Θ.

Theorem 2.6.9: For any CSL structure satisfying Θ, there is a unique CSL structure morphism from $\langle \mathcal{C}, \xi \rangle$.
Proof: Let $\langle \mathcal{D}, \zeta \rangle$ be an arbitrary CSL structure satisfying Θ. Using the denotation of annotated expressions on this structure (see definition 2.4.10), we define a functor T from $\mathcal{C}$ to $\mathcal{D}$ as follows:

$$T(K) \stackrel{\text{def}}{=} \zeta K \quad \text{and} \quad T(\langle e \rangle) \stackrel{\text{def}}{=} \zeta e.$$

Note that, since K is a ground functorial expression ζK is an $\mathcal{D}$ object, and that, since e is a ground annotated expression, ζe is a $\mathcal{D}$ morphism (it does not need objects $A_1, \ldots, A_m$ or a morphism variable assignment ω). It is easy to see that T is a (covariant) functor (note that we have to show the well-definedness first). It is also not so difficult to show that T is a CSL structure morphism and that it is the unique one from $\langle \mathcal{C}, \xi \rangle$ to $\langle \mathcal{D}, \zeta \rangle$ (simply extending the result on algebraic specifications). []

The advantage of working in an initial CSL structure is that, if we show that a conditional equation holds in the initial one, then we automatically know that it holds in any CSL structure. In chapter 4, we will define the symbolic computation in categories and it can be regarded as the computation in free categories.

Chapter 3

Categorical Data Types

In chapter 2, we introduced categorical data types from a point of view of a specification language defining categories. Although the specification language CSL is a rigorous language, it is rather tedious and categoritians may never define categories in that way. In this chapter, we will give another presentation of categorical data types, which will be simpler and more intuitive. This is also the way Categorical Data Types originated. Note that we are not discarding CSL completely and that the semantics of categorical data types will be given in terms of CSL.

Section 3.1 is an extended introduction to categorical data types. We will investigate some conventional data types and introduce a new uniform categorical way of defining data types. In section 3.2, we will make this new way into the CDT declaration mechanism. Section 3.3 will give various examples of CDT declarations. In section 3.4, CDT declarations will be connected to CSL theories, and finally in section 3.5 we will give a construction of CDT data types.

3.1 What are Categorical Data Types?

The need for pairs of two (or more) items of data often arises when we write programs. It is often the case that a function or procedure takes more than one argument and this means that a pair (or a record) of data needs to be passed to the function or procedure. In another situation, we may want to declare a new data type whose element is a pair of elements of other data types. In PASCAL, we can define such data types using its `record ... end` construct. For example, `intchar` whose element is a pair of an integer and a character can be declared as:

```
type intchar = record
                   first: integer;
                   second: char
               end;
```

In ML, this can be done by

```
type intchar = int * string;
```

(where since ML does not have a type for representing characters, we have to use '`string`' type whose element is a sequence of characters). These two languages have the means of constructing data types of pairs from already-existing data types. Let us call the constructors *product type constructors*. Most of the current programming languages have product type constructors in one way or another as their built-in primitives because they are so essential that we can even say that no programming language is complete without them.

In order to understand the nature of product type constructors, let us suppose a programming language which does not have them as primitives and that we have to define them in terms of others. This might mean that we need in the language some kind of one level higher operations which can define not types but type constructors. Let us refer to an algebraic specification language CLEAR,[1] and see its ability to define a product type constructor.

```
constant Triv =
    theory
        sort element
    endth

procedure Prod(A:Triv,B:Triv) =
    theory
        data sort prod
        opns pair: element of A,element of B -> prod
             pi1 : prod -> element of A
             pi2 : prod -> element of B
        eqns all a:element of A,b:element of B,
                                  pi1(pair(a,b)) = a
             all a:element of A,b:element of B,
                                  pi2(pair(a,b)) = b
    endth
```

This defines an algebra P of three sorts: two '`element`' sorts (let us call them '`A-element`' and '`B-element`' to distinguish them) and a '`prod`' sort. The underlying set $|P|_{\texttt{prod}}$ is the (set) product of two sets $|P|_{\texttt{A-element}}$ and $|P|_{\texttt{B-element}}$. The declaration also defines three operations (or functions), '`pair`', '`pi1`' and '`pi2`', satisfying the two equations listed. Note that the following equation can be proved using induction on '`prod`' sort (we have the induction principle on this sort because of the initial data constraint).

```
all x: prod, pair(pi1(x),pi2(x)) = x
```

An algebraic specification language like CLEAR is powerful enough to allow us to define all kinds of type constructors (including ordinary data types as constant type constructors) in a uniform way without having any particular primitives. However,

[1] CLEAR can be institution independent, but here we refer the one that uses the equational algebraic institution.

because of its use of equations, we cannot adopt its declaration mechanism in ordinary programming languages.[2] We might be puzzled that we need equations even to define such very basic data types as products.

Let us find whether there is any other ways of defining product type constructors by examining the foundation, namely mathematics. Modern mathematics uses set theory extensively because of its power. Set theory does not have a product type constructor as its primitive construct either, so it is defined by means of other constructs. For sets A and B, their (set) product is defined as:

$$A \times B \stackrel{\text{def}}{=} \{\ (x, y) \mid x \in A,\ y \in B\ \}$$

where (x, y) is really an abbreviation of $\{\{x\}\ \{x\ y\}\}$. Although this looks very simple, it actually needs some work to show from the axioms of set theory that this is actually a set. Set theory uses the power of first order logic so heavily that it is much harder to put the set theory formalism into a programming language than to put the algebraic specification formalism. As an example, let $\phi(x)$ be a first order formula. Then, from the comprehension axiom (or the replacement axiom), we have the set

$$\{\ x \in A \mid \phi(x)\ \}$$

where A is a set. $\phi(x)$ can be anything expressible by first order logic (using quantifiers and negations), and this is too much powerful to investigate the basic property of data types. It disfigures the beauty behind this powerful definition mechanism and we cannot see through it easily.

Set theory has achieved a firm position as the foundation of mathematics, but there are some alternatives. Category theory is one of them. It has been proved that category theory has a remarkable ability to disclose true nature of mathematical objects. For example, a product constructor (or categorically a functor $\mathcal{C} \times \mathcal{C} \to \mathcal{C}$) is beautifully characterized as the right adjoint of the diagonal functor $\mathcal{C} \to \mathcal{C} \times \mathcal{C}$.[3] Expanding the definition of adjunctions, this means that we have the following natural isomorphism:

$$\mathrm{Hom}_{\mathcal{C}}(C, A) \times \mathrm{Hom}_{\mathcal{C}}(C, B) \simeq \mathrm{Hom}_{\mathcal{C}}(C, \mathrm{prod}(A, B)) \qquad (*)$$

(natural in A, B and C). We use 'prod' for the product functor to follow the notation we use later. The isomorphic function from the left-hand side to the right-hand side is the factorizer of this adjunction and we write 'pair' for it. We can rewrite this adjoint situation as the following rule:

$$\frac{C \stackrel{f}{\longrightarrow} A \qquad C \stackrel{g}{\longrightarrow} B}{C \stackrel{\mathrm{pair}(f,g)}{\longrightarrow} \mathrm{prod}(A, B)} \qquad (**)$$

Given two morphisms $f{:}C \to A$ and $g{:}C \to B$, $\mathrm{pair}(f, g)$ give a morphism of $C \to \mathrm{prod}(A, B)$. This correspondence is one-to-one and is natural in A, B and C.

[2] There is a programming language OBJ [Goguen and Tardo 79] which treats equations as a kind of programs (as rewrite rules).

[3] There may be more than one right adjoint of the diagonal functor, but they are isomorphic. Therefore, we say '*the*' right adjoint rather than '*a*' right adjoint.

Comparing with the product type constructor '`Prod`' defined by CLEAR, we have the same 'pair' though the previous one takes two elements as the arguments and this one takes two morphisms instead, and now we can find the things corresponding to two projections '`pi1`' and '`pi2`' as well. Replacing C by $\mathrm{prod}(A,B)$ in $(*)$, we get:

$$\mathrm{Hom}_{\mathcal{C}}(\mathrm{prod}(A,B),A) \times \mathrm{Hom}_{\mathcal{C}}(\mathrm{prod}(A,B),B) \simeq \mathrm{Hom}_{\mathcal{C}}(\mathrm{prod}(A,B),\mathrm{prod}(A,B)).$$

We have a very special morphism in $\mathrm{Hom}_{\mathcal{C}}(\mathrm{prod}(A,B),\mathrm{prod}(A,B))$, namely the identity. Because of the isomorphism, there exist unique morphisms of $\mathrm{prod}(A,B) \to A$ and $\mathrm{prod}(A,B) \to B$ which are mapped to the identity by 'pair', and these are the projections. We name them 'pi1' and 'pi2' as well. Because of the very way they are defined, it is trivial that

$$\mathrm{pair}(\mathrm{pi1},\mathrm{pi2}) = \mathbf{I}. \qquad (+)$$

Furthermore, from the naturality in C of $(*)$, we have the rule

$$\frac{C \xrightarrow{\mathrm{pair}(f,g)} \mathrm{prod}(A,B) \xrightarrow{\mathrm{pi1}} A \qquad C \xrightarrow{\mathrm{pair}(f,g)} \mathrm{prod}(A,B) \xrightarrow{\mathrm{pi2}} B}{C \xrightarrow{\mathrm{pair}(f,g)} \mathrm{prod}(A,B) \xrightarrow{\mathrm{pair}(\mathrm{pi1},\mathrm{pi2})} \mathrm{prod}(A,B)},$$

and, if we express it by equations and use $(+)$,

$$\mathrm{pair}(\mathrm{pi1} \circ \mathrm{pair}(f,g), \mathrm{pi2} \circ \mathrm{pair}(f,g)) = \mathrm{pair}(\mathrm{pi1},\mathrm{pi2}) \circ \mathrm{pair}(f,g) = \mathrm{pair}(f,g).$$

Since 'pair' is isomorphic, we can conclude that the following equations hold:

$$\mathrm{pi1} \circ \mathrm{pair}(f,g) = f \quad \text{and} \quad \mathrm{pi2} \circ \mathrm{pair}(f,g) = g.$$

These are exactly the ones which we listed when defining '`Prod`' in CLEAR. Note that this time they are derived equations. By saying that 'prod' is the right adjoint to the diagonal functor, we get these equations automatically. This shows how neat the categorical definition is.

Another advantage of categorical definition is that we can form the dual definition easily. We defined the product functor as the right adjoint of the diagonal functor. Then, it is natural to ask what is the left adjoint of the diagonal functor. It is the coproduct functor $\mathcal{C} \times \mathcal{C} \to \mathcal{C}$. In the category of sets, the copoduct of two sets A and B is their disjoint sum

$$A + B \stackrel{\mathrm{def}}{=} \{\ 0\ \} \times A \bigcup \{\ 1\ \} \times B.$$

It is not easy to see in set theory that this is the dual of $A \times B$. In PASCAL, we can define coproducts by means of variant record. In ML, we used to have a built-in coproduct type constructor '`+`', but the new Standard ML does not. Instead, '`+`' can be defined by the following '`datatype`' declaration.

```
datatype 'a + 'b = in1 of 'a | in2 of 'b;
```

We cannot define the product type constructor by a '`datatype`' declaration in ML, but we can define its dual. ML looks non-symmetric from this. In CLEAR, we can define a coproduct type constructor as follows:

```
Procedure CoProd(A:Triv,B:Triv) =
    theory
        data sort coprod
        opns in1: element of A -> coprod
             in2: element of B -> coprod
    endth
```

Again, this cannot be seen as the dual of '`Prod`' we defined earlier; here we do not use equations; we have only two operations whereas we had three. This shows that CLEAR is not symmetric either.

Now, in category theory, the coproduct functor $\mathcal{C} \times \mathcal{C} \to \mathcal{C}$ which we call 'coprod' is defined by the dual of $(*)$, by just changing the direction of arrows.

$$\mathrm{Hom}_{\mathcal{C}}(A, C) \times \mathrm{Hom}_{\mathcal{C}}(B, C) \simeq \mathrm{Hom}_{\mathcal{C}}(\mathrm{coprod}(A, B), C)$$

We name the isomorphic function going from the left-hand side to the right-hand side 'case' (we could call it 'copair' to emphasize the duality to 'pair', but, since it plays a role of '`case`' statements of ML or C (or PASCAL), we call it 'case'). Writing the adjunction as a rule,

$$\frac{A \xrightarrow{f} C \qquad B \xrightarrow{g} C}{\mathrm{coprod}(A, B) \xrightarrow{\mathrm{case}(f,g)} C}.$$

Two injections $\mathrm{in1}\colon A \to \mathrm{coprod}(A, B)$ and $\mathrm{in2}\colon B \to \mathrm{coprod}(A, B)$ are defined as the morphisms which 'case' maps to the identity of $\mathrm{coprod}(A, B)$. As before, we can derive some equations easily.

From what we have looked at, it seems a good idea to design a category theory based (programming or specification) language which has the ability to define functors by means of adjunctions. Since it is convenient to introduce names for unit natural transformations and factorizers at the same time, we regard an adjunction as a triple of a functor, a unit natural transformation and a factorizer (see, for example, [Mac Lane 71] for many equivalent ways of defining adjunctions). Therefore, a category theory based language may have the following two forms of declaring new functors:

$$\begin{array}{l}\text{let } \langle F, \alpha, \psi \rangle \text{ be right adjoint of } G \\ \text{let } \langle F, \alpha, \psi \rangle \text{ be left adjoint of } G\end{array}$$

where G is a functor we already have, F is the new functor we are defining, α is the associated unit natural transformation and ψ is the associated factorizer. One problem is that we need to have some primitive functors with which we start. We definitely need diagonal functors for we want to define product and coproduct functors. In order to define the natural number object (which is a constant functor), we need a pretty complicated functor G. The problem is how to represent such G.

Let us investigate how other languages and theories define the data type of natural numbers. In set theory, it has the axiom of infinity which says the existence of natural numbers. This may look rather artificial. In PASCAL, there is no intuitively easy way to define it. In ML, though it is a built-in data type for efficiency, we could define it as:

```
datatype nat = zero | succ of nat;
```

Note the recursiveness in this definition. Essentially, we need some kind of recursiveness to define a data type of natural numbers. In CLEAR, one can define it as

```
constant Nat =
  theory
    data sort nat
    opns zero: nat
         succ: nat -> nat
  endth
```

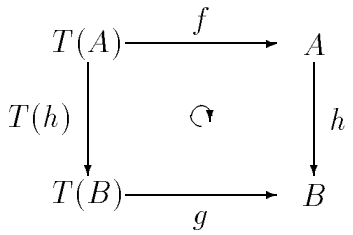
This is very much similar to the one in ML, though we often think that the CLEAR definition is based on the initial algebra semantics whereas the ML definition is based on domain theory. In domain theory, a data type of natural numbers can be defined as the solution of the following domain equation

$$N \cong 1 + N. \qquad (*)$$

The initial solution of $(*)$ can be calculated as a colimit of a sequence of domains, but we do not go into its detail here. As a connection to the initial algebra semantics, the initial solution can be characterized as the initial T-algebra, where T is a functor $T(X) \stackrel{\text{def}}{=} 1 + X$ in this case. In general, given a category and an endo-functor T, we can form a category of T-algebras.

Definition 3.1.1: For a category $\mathcal{C}$ and an endo-functor $T{:}\mathcal{C} \to \mathcal{C}$, the category of T-algebras is defined

1. its objects are pairs $\langle A, f\rangle$ where A is a $\mathcal{C}$ object and f is a $\mathcal{C}$ morphism $T(A) \to A$, and
2. its morphisms $h{:}\langle A, f\rangle \to \langle B, g\rangle$ are $\mathcal{C}$ morphisms $h{:}A \to B$ which make the following diagram commute.

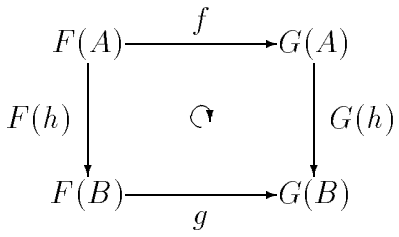


Note that this is a weaker version of the category of T-algebras defined in many category theory books (e.g. [Mac Lane 71]) where T needs to be a monad. []

We can dualize definition 3.1.1 to define T-coalgebras. However, in the theory of categorical data types ('CDT theory' for short), we combine the two definitions together.

Definition 3.1.2: Let $\mathcal{C}$ and $\mathcal{D}$ be categories and both F and G be functors from $\mathcal{C}$ to $\mathcal{D}$. We define an F, G-dialgebra[4] as

1. its objects are pairs $\langle A, f\rangle$ where A is a $\mathcal{C}$ object and f is a $\mathcal{D}$ morphism of $F(A) \to G(A)$, and
2. its morphisms $h{:}\langle A, f\rangle \to \langle B, g\rangle$ are $\mathcal{C}$ morphisms $h{:}A \to B$ such that the following diagram commutes.



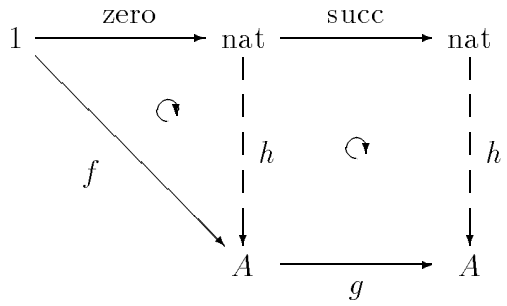
In the case where F or G is contravariant, we have to modify the direction of some arrows.

It is easy to show that it is a category; let us write $\mathbf{DAlg}(F, G)$ for it. Note that $\mathbf{DAlg}(T, \mathbf{I})$ is the category of T-algebras and $\mathbf{DAlg}(\mathbf{I}, T)$ is the category of T-coalgebras. □

This is a very simple extension of definition 3.1.1, yet its symmetry and dividing the source category from the target one give us greater freedom. With T-algebras, we need to use the coproduct functor to define the domain of natural numbers, but by F, G-dialgebra we do not. Let $\mathcal{C}$ be any category and $\mathcal{D}$ be its product $\mathcal{C} \times \mathcal{C}$. We define the functors F and G as

$$F(A) \stackrel{\text{def}}{=} \langle 1, A\rangle \quad \text{and} \quad G(A) \stackrel{\text{def}}{=} \langle A, A\rangle.$$

Let $\langle \text{nat}, \langle \text{zero}, \text{succ}\rangle\rangle$ be the initial F, G-dialgebra. From the definition, 'nat' is a $\mathcal{C}$ object, 'zero' is a $\mathcal{C}$ morphism of $1 \to \text{nat}$ and 'succ' is a $\mathcal{C}$ morphism of $\text{nat} \to \text{nat}$. The initiality means that for any $\mathbf{DAlg}(F, G)$ object $\langle A, \langle f, g\rangle\rangle$ there exists a unique $\mathbf{DAlg}(F, G)$ morphism $h{:}\langle \text{nat}, \langle \text{zero}, \text{succ}\rangle\rangle \to \langle A, \langle f, g\rangle\rangle$. If we spell out the definition, this means that for any $\mathcal{C}$ object A and any $\mathcal{C}$ morphisms $f{:}1 \to A$ and $g{:}A \to A$ there exists a unique $\mathcal{C}$ morphism $h{:}\,\text{nat} \to A$ which makes the following diagram commute.



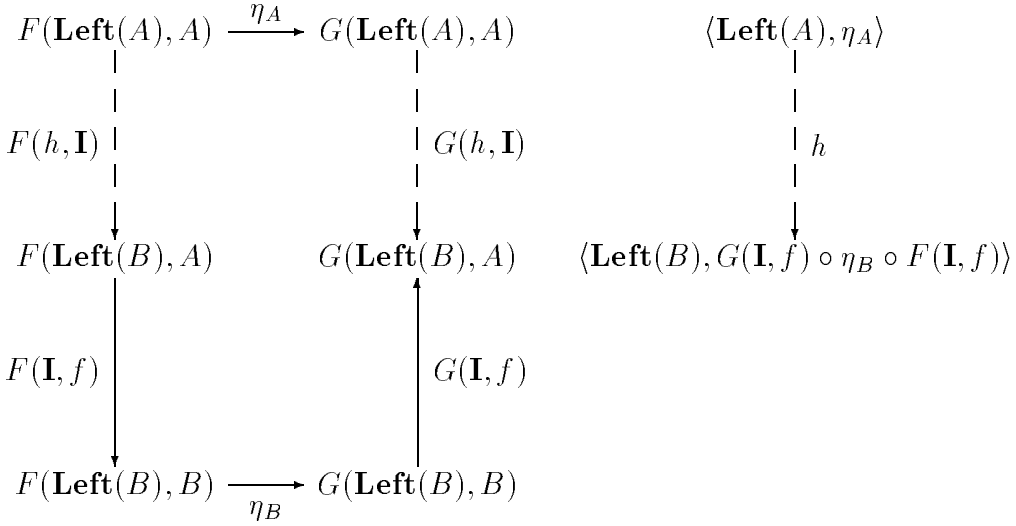
[4]The name *dialgebra* was suggested by Bob McKay.

This is exactly the definition of 'nat' being a natural number object in category theory.

To get further generality of F, G-dialgebras, we parametrize F and G.

Definition 3.1.3: Let $\mathcal{C}$, $\mathcal{D}$ and $\mathcal{E}$ be categories, and let $F: \mathcal{C} \times \mathcal{D} \to \mathcal{E}$ and $G: \mathcal{C} \times \mathcal{D}^- \to \mathcal{E}$ be functors. We define $\langle \mathbf{Left}[F,G](A), \eta_A \rangle$ for a $\mathcal{D}$ object A to be the initial object in the category $\mathbf{DAlg}(F(\ \cdot\ , A), G(\ \cdot\ , A))$. Dually, we define $\langle \mathbf{Right}[F,G](A), \epsilon_A \rangle$ to be the final object. We may write $\mathbf{Right}(A)$ for $\mathbf{Right}[F,G](A)$ and $\mathbf{Left}(A)$ for $\mathbf{Left}[F,G](A)$ if the context makes F and G clear. []

Proposition 3.1.4: If $\mathbf{Left}[F,G](A)$ exists for every object $A \in |\mathcal{D}|$, $\mathbf{Left}[F,G]$ denotes a functor $\mathcal{D} \to \mathcal{C}$ (i.e. we can extend it to $\mathcal{D}$ morphisms). Dually, if $\mathbf{Right}[F,G](A)$ exists for every $A \in |\mathcal{D}|$, $\mathbf{Right}[F,G]$ denotes a functor $\mathcal{D}^- \to \mathcal{C}$.
Proof: We first need to define what $\mathbf{Left}[F,G](f)$ is for a $\mathcal{D}$ morphism $f: A \to B$. We define it as the morphism $h: \mathbf{Left}(A) \to \mathbf{Left}(B)$ which fills in the following diagram.



The unique existence of the morphism is provided because $\langle \mathbf{Left}(A), \eta_A \rangle$ is the initial object of $\mathbf{DAlg}(F(\ \cdot\ , A), G(\ \cdot\ , A))$. In other words, $\mathbf{Left}(f)$ is the unique morphism which satisfies

$$G(\mathbf{Left}(f), f) \circ \eta_B \circ F(\mathbf{Left}(f), f) = \eta_A.$$

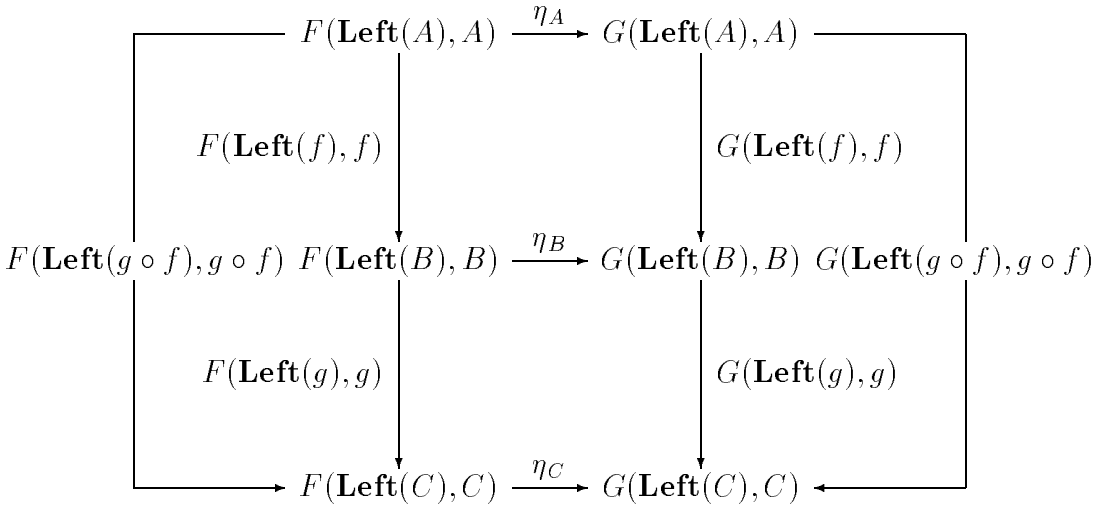
Let us check that **Left** is in fact a functor. Trivially,

$$G(\mathbf{I}, \mathbf{I}) \circ \eta_A \circ F(\mathbf{I}, \mathbf{I}) = \eta_A.$$

Therefore, $\mathbf{Left}(\mathbf{I}_A) = \mathbf{I}_{\mathbf{Left}(A)}$. For morphisms $f: A \to B$ and $g: B \to C$,

$$\begin{aligned} & G(\mathbf{Left}(g) \circ \mathbf{Left}(f), g \circ f) \circ \eta_C \circ F(\mathbf{Left}(g) \circ \mathbf{Left}(f), g \circ f) \\ = \ & G(\mathbf{Left}(f), f) \circ G(\mathbf{Left}(g), g) \circ \eta_C \circ F(\mathbf{Left}(g), g) \circ F(\mathbf{Left}(f), f) \\ = \ & G(\mathbf{Left}(f), f) \circ \eta_B \circ F(\mathbf{Left}(f), f) \\ = \ & \eta_A \end{aligned}$$

Therefore, $\mathbf{Left}(g) \circ \mathbf{Left}(f) = \mathbf{Left}(g \circ f)$.



Dually, we can prove that **Right** is a functor. □

'**Left**' and '**Right**' may suggest a connection with left and right adjoint functors. In fact,

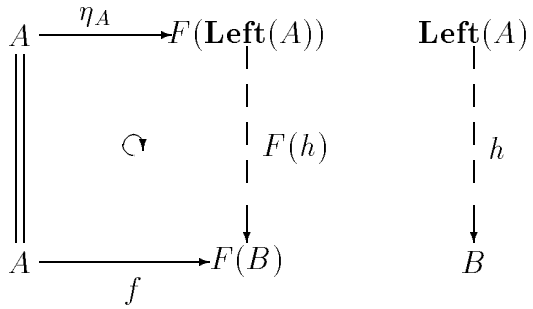
Proposition 3.1.5: For a functor $F{:}\mathcal{C} \to \mathcal{D}$, its left adjoint functor can be denoted by

$$\mathbf{Left}[\lambda(X,Y).Y, \lambda(X,Y).F(X)]$$

and, dually, its right adjoint functor can be denoted by

$$\mathbf{Right}[\lambda(X,Y).F(X), \lambda(X,Y).Y].$$

Proof: Let us only check the left adjoint case. We see $\lambda(X,Y).Y$ as a functor $\mathcal{C} \times \mathcal{D} \to \mathcal{D}$ and $\lambda(X,Y).F(X)$ as a functor $\mathcal{C} \times \mathcal{D}^- \to \mathcal{D}$. From definition 3.1.3, **Left** is a functor $\mathcal{D} \to \mathcal{C}$. If we spell out the condition of $\langle \mathbf{Left}(A), \eta_A \rangle$ being the initial algebra, it means that for any $\mathcal{C}$ object B and a $\mathcal{C}$ morphism $f{:}\, A \to F(B)$ there exists a unique $\mathcal{C}$ morphism $h{:}\, \mathbf{Left}(A) \to B$ such that the following diagram commutes.



This is exactly the condition of **Left** being the left adjoint functor of F. Dually, we can prove that $\mathbf{Right}[\lambda(X,Y).F(X), \lambda(X,Y).Y]$ is the right adjoint. □

Hence, definition 3.1.2 of F,G-dialgebras covers both T-algebras and adjoints so that it enables us to define products, coproducts, natural number object, and so on in a uniform way.

3.2 Data Type Declarations in Categorical Data Types

In the previous section, we have looked at some ways of defining data types in some languages. In this section, we will introduce how to define data types in CDT.

If we were only interested in functors, only $\mathbf{Left}[F,G]$ and $\mathbf{Right}[F,G]$ defined in the previous section would be needed, but we do want morphisms (or natural transformations) and factorizers which will make up some kind of programs, the meaning of which we will examine in chapter 4 (e.g. how to execute them).

$\mathbf{Left}[F,G]$ and $\mathbf{Right}[F,G]$ have been defined for functors $F{:}\mathcal{C}\times\mathcal{D}\to\mathcal{E}$ and $G{:}\mathcal{C}\times\mathcal{D}^{-}\to\mathcal{E}$, where $\mathcal{C}$, $\mathcal{D}$ and $\mathcal{E}$ are some categories. Remember the aim of CDT; we would like to define (or specify, or study) a category of data types. Therefore, $\mathcal{C}$, $\mathcal{D}$ and $\mathcal{E}$ should somehow be related to this category. The simplest we can think of is that they are the product categories of this category and all the functors are in the form of $\mathcal{C}^s\to\mathcal{C}$, where s is a sequence of variances (i.e. $s\in\mathbf{Var}^*$) and $\mathcal{C}^{v_1\ldots v_n}\stackrel{\text{def}}{=}\mathcal{C}^{v_1}\times\cdots\times\mathcal{C}^{v_n}$.

In order to simplify the presentation, let us use the vector notation and write, for example, $\vec{F}$ for a sequence of functors $\langle F_1,\ldots,F_n\rangle$ where all of them have the same type $\mathcal{C}^s\to\mathcal{C}$, that is, $\vec{F}$ is a functor of $\mathcal{C}^s\to\mathcal{C}^n$.

From definition 3.1.3, for $\vec{F}{:}\mathcal{C}\times\mathcal{C}^s\to\mathcal{C}^n$ and $\vec{G}{:}\mathcal{C}\times\mathcal{C}^{-\bullet s}\to\mathcal{C}^n$, $\mathbf{Left}[F,G]$ is a functor $\mathcal{C}^s\to\mathcal{C}$ and $\mathbf{Right}[F,G]$ is a functor $\mathcal{C}^{-\bullet s}\to\mathcal{C}$, where $u\bullet v_1\ldots v_n\stackrel{\text{def}}{=}u\bullet v_1\ldots u\bullet v_n$.

Hence, we come to the definition of CDT declarations.

Definition 3.2.1: In CDT theory, there are two forms of declaring new functors. One is to define a functor $L{:}\mathcal{C}^s\to\mathcal{C}$ by

$$\begin{array}{l}\text{left object } L(\vec{X}) \text{ with } \psi \text{ is}\\ \quad \vec{\alpha}{:}\,\vec{F}(L,\vec{X})\to\vec{G}(L,\vec{X})\\ \text{end object}\end{array}\qquad(*)$$

and the other is to define a functor $R{:}\mathcal{C}^{-\bullet s}\to\mathcal{C}$ by

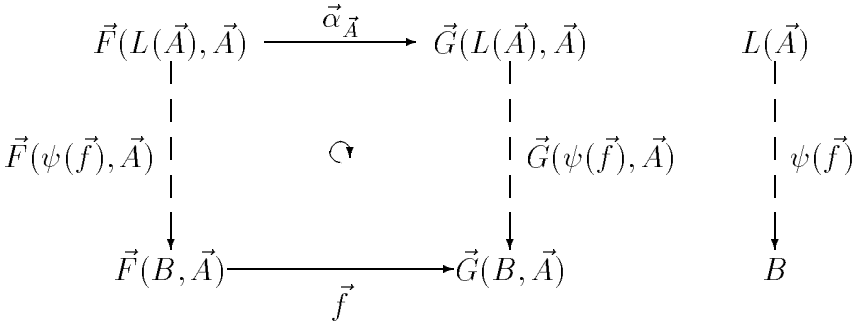
$$\begin{array}{l}\text{right object } R(\vec{X}) \text{ with } \psi \text{ is}\\ \quad \vec{\alpha}{:}\,\vec{F}(R,\vec{X})\to\vec{G}(R,\vec{X})\\ \text{end object}\end{array}$$

where $\vec{X}$ is a sequence $\langle X_1,\ldots,X_n\rangle$ of variables, ψ is the associated factorizer, $\vec{\alpha}$ is a sequence $\langle\alpha_1,\ldots,\alpha_m\rangle$ of the associated natural transformations, and $\vec{F}$ and $\vec{G}$ are sequences $\langle F_1,\ldots,F_m\rangle$ and $\langle G_1,\ldots,G_m\rangle$, respectively, of functors which we have as primitives or we have already defined and whose type is $F_i{:}\mathcal{C}\times\mathcal{C}^s\to\mathcal{C}$ and $G_i{:}\mathcal{C}\times\mathcal{C}^{-\bullet s}\to\mathcal{C}$, respectively. Semantically, L is $\mathbf{Left}[\vec{F},\vec{G}]$ and R is $\mathbf{Right}[\vec{F},\vec{G}]$. We may call L left functor or left object and R right functor or right object. []

If we expand the definition of **Left**, $(*)$ defines for any $\mathcal{C}$ objects $\vec{A} = \langle A_1, \ldots, A_n \rangle$ an object $L(\vec{A})$ and a morphism

$$\vec{F}(L(\vec{A}), \vec{A}) \xrightarrow{\vec{\alpha}_{\vec{A}}} \vec{G}(L(\vec{A}), \vec{A}),$$

and for any object B and a sequence of morphisms $\vec{f}: \vec{F}(B, \vec{A}) \to \vec{G}(B, \vec{A})$, $\psi(\vec{f})$ denotes the unique morphism which makes the following diagram commute.



In definition 3.2.1, it is not immediately clear what kind of $\vec{F}$ and $\vec{G}$ is allowed. We vaguely stated that they are primitive or have been defined already. In order to clarify this point, we go back to CSL and regard a CDT declaration as an extension of a given CSL signature.

Definition 3.2.2: Let $\langle \Gamma, \Delta, \Psi \rangle$ be a CSL signature. A CDT declaration $D \in$ **Decl** is given by the following BNF expression.

$$\begin{aligned} D ::= \ & \{ \text{ left } | \text{ right } \} \text{ object } F(X_1, \ldots, X_n) \text{ with } \psi \text{ is} \\ & \quad \alpha_1 : E_1 \to E_1' \\ & \quad \ldots \\ & \quad \alpha_m : E_m \to E_m' \\ & \text{end object} \end{aligned}$$

where F is a new functor symbol, ψ is a new factorizer symbol, $\alpha_1, \ldots, a_m$ are new natural transformation symbols, and E_i and E_i' $(i = 1, \ldots, m)$ are well-formed functorial expressions (under this signature $\langle \Gamma, \Delta, \Psi \rangle$) whose variables are $X_1, \ldots, X_n$ and F (here we use F as a formal parameter like we use its function name inside a function body in PASCAL). []

We need to put restriction on the variance of F in the functorial expressions such that for each $i = 1, \ldots, m$

1. the variance of F in E_i should be either covariant or free,
2. the variance of F in E_i' should also be either covariant or free, and
3. either the variance of F in E_i should be covariant or that in E_i' should be covariant.

We could have allowed F to be contravariant (as indeed the original definition of CDT did), but it turned out that the generality by contravariance was of very little use,

so, because it simplifies the following argument, we restrict ourselves only to covariant functors. The third condition above is to make the extension consistent (each α_i should somehow relate to the functor we are declaring).

Let us calculate the variance of F. If the variance of $\lambda(F, X_1, \ldots, X_n).E_i$ is $v_i s_i$ ($v_i \in$ **Var** for the variance of F and $s_i \in$ **Var*** for the variance of $X_1, \ldots, X_n$) and that of $\lambda(F, X_1, \ldots, X_n).E'_i$ is $v'_i s'_i$, $\lambda(F, X_1, \ldots, X_n).E_i$ denotes a functor $\mathcal{C}^{v_i} \times \mathcal{C}^{s_i} \to \mathcal{C}$ and $\lambda(F, X_1, \ldots, X_n).E'_i$ denotes a functor $\mathcal{C}^{v'_i} \times \mathcal{C}^{s'_i} \to \mathcal{C}$. The restriction above states that $v_i \sqcup v'_i = +$. From proposition 3.1.4, the variance of F in case that it is declared by a left CDT declaration should be

$$\bigsqcup_{i=1}^{m} s_i \sqcup - \bullet s'_i, \qquad (*)$$

and in case that it is declared by a right one, the variance should be

$$\bigsqcup_{i=1}^{m} - \bullet s_i \sqcup s'_i, \qquad (**)$$

where **Var*** is a partially ordered set with the ordering given by $u_1 \ldots u_n \sqsubseteq v_1 \ldots v_n$ if and only if $u_1 \sqsubseteq v_1$, ...and $u_n \sqsubseteq v_n$.

Definition 3.2.2 (continued)**:** A CDT declaration gives an extension of CSL signature.

$$\langle \Gamma, \Delta, \Psi \rangle \hookrightarrow \langle \Gamma \cup \{ F \}, \Delta \cup \{ \alpha_1, \ldots, \alpha_m \}, \Psi \cup \{ \psi \} \rangle$$

where the variance of F is given by $(*)$ or $(**)$, the type of α_i is

$$\lambda(X_1, \ldots, X_n).E_i[F(X_1, \ldots, X_n)/F] \dot{\to} \lambda(X_1, \ldots, X_n).E'_i[F(X_1, \ldots, X_n)/F],$$

and the type of ψ is

$$\frac{f_i \colon \lambda(X, X_1, \ldots, X_n).E_i[X/F] \to \lambda(X, X_1, \ldots, X_n).E'_i[X/F] \quad (i = 1, \ldots, m)}{\psi(f_1, \ldots, f_m) \colon \lambda(X, X_1, \ldots, X_n).F(X_1, \ldots, X_n) \to \lambda(X, X_1, \ldots, X_n).X}$$

by a left CDT declaration and

$$\frac{f_i \colon \lambda(X, X_1, \ldots, X_n).E_i[X/F] \to \lambda(X, X_1, \ldots, X_n).E'_i[X/F] \quad (i = 1, \ldots, m)}{\psi(f_1, \ldots, f_m) \colon \lambda(X, X_1, \ldots, X_n).X \to \lambda(X, X_1, \ldots, X_n).F(X_1, \ldots, X_n)}$$

by a right one. []

We will see in section 3.4 a CDT declaration as a CSL theory extension so that the semantics of a CDT declaration can be given by a CSL structure.

3.3 Examples of Categorical Data Types

In this section, we will present several examples of categorical data types declared by

$$\begin{array}{l}\text{left object } F(X_1,\ldots,X_n) \text{ with } \psi \text{ is}\\ \quad \alpha_1 : E_1 \to E_1'\\ \qquad \ldots\\ \quad \alpha_m : E_m \to E_m'\\ \text{end object}\end{array} \qquad (*)$$

for left objects and by

$$\begin{array}{l}\text{right object } F(X_1,\ldots,X_n) \text{ with } \psi \text{ is}\\ \quad \alpha_1 : E_1 \to E_1'\\ \qquad \ldots\\ \quad \alpha_m : E_m \to E_m'\\ \text{end object}\end{array} \qquad (**)$$

for right objects.

3.3.1 Terminal and Initial Objects

Let us start with an empty CSL signature $\langle \emptyset, \emptyset, \emptyset \rangle$. The simplest case of $(*)$ and $(**)$ is when $n = m = 0$. If we consider the case when $n = m = 0$ in $(**)$, we get the declaration of the terminal object.

right object 1 with !
end object

(We omitted the keyword 'is' to make the declaration look nicer.) From the definition, this defines an object '1' and for any object A there is a unique morphism '!' from A to '1'.

$$A \xrightarrow{\quad ! \quad} 1$$

Therefore, it really is the terminal object.

Dually, if we change the keyword 'right' to 'left' in the definition of the terminal object, we get the definition of the initial object.

left object 0 with !!
end object

The factorizer '!!' gives a unique morphism from '0' to any object A.

$$0 \xrightarrow{\quad !! \quad} A$$

3.3.2 Products and CoProducts

Next, we define products and coproducts. The binary product functor can be declared as the following right object.

$$\begin{array}{l}\text{right object prod}(X,Y)\text{ with pair is}\\ \quad \text{pi1}\colon \text{prod} \to X\\ \quad \text{pi2}\colon \text{prod} \to Y\\ \text{end object}\end{array}$$

From definition 3.2.2, this defines a functor symbol 'prod' whose variance is '++' (i.e. covariant in both arguments), two natural transformation symbols 'pi1' and 'pi2' whose types are

$$\text{pi1}\colon \lambda(X,Y).\text{prod}(X,Y) \dot{\to} \lambda(X,Y).X$$

$$\text{pi2}\colon \lambda(X,Y).\text{prod}(X,Y) \dot{\to} \lambda(X,Y).Y$$

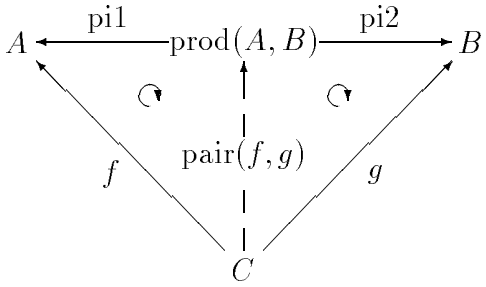
and a factorizer symbol 'pair' whose type is

$$\frac{f\colon \lambda(Z,X,Y).Z \to \lambda(Z,X,Y).X \qquad g\colon \lambda(Z,X,Y).Z \to \lambda(Z,X,Y).Y}{\text{pair}(f,g)\colon \lambda(Z,X,Y).Z \to \lambda(Z,X,Y).\text{prod}(X,Y)}.$$

If we write this down in a more understandable way, it becomes the familiar definition of the binary product, that is 'prod' has two unit morphisms

$$A \xleftarrow{\text{pi1}} \text{prod}(A,B) \xrightarrow{\text{pi2}} B$$

and $\text{pair}(f,g)$ gives the unique morphism for any morphisms $f\colon C \to A$ and $g\colon C \to B$ such that the following diagram commutes.



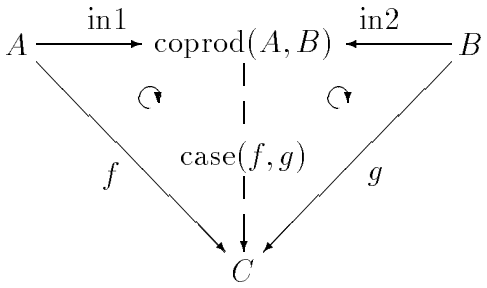
Note that the CDT declaration of the binary product functor is very similar to the 'Prod' theory in CLEAR defined in 3.1. One of the differences is that in CLEAR 'pair' is treated as a function in the same class as 'pi1' and 'pi2' but in CDT 'pair' is quite different from 'pi1' and 'pi2'. Another one is that in CLEAR 'Prod' is declared as the initial algebra so it is close to CDT's left object but in CDT 'prod' is naturally a right object because the product functor is the *right* adjoint of the diagonal functor.

Dually, we can define the binary coproduct functor as a left object.

left object coprod(X, Y) with case is
in1:$X \to$ coprod
in2:$Y \to$ coprod
end object

Again, this declaration looks very close to the one in CLEAR (defined in section 3.1), but note that we have 'case' in CDT so that we can use it to write programs or to specify some properties.

Just writing the situation as a diagram,

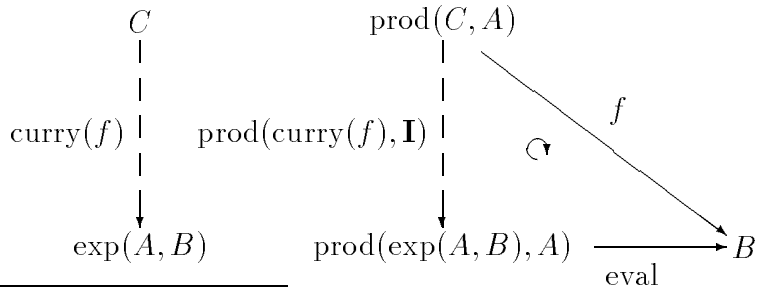


3.3.3 Exponentials

One of the objections against algebraic specification methods is that it cannot handle function spaces. CDT's declaration mechanism looks very close to that of algebraic specification methods, but CDT is based on category theory not on many-sorted algebras, and in category theory function spaces can be defined as exponentials. For objects A and B, the exponential of B by A is written as $\exp(A, B)$[5] satisfies the following natural isomorphism.

$$\mathrm{Hom}_{\mathcal{C}}(\mathrm{prod}(C, A), B) \simeq \mathrm{Hom}_{\mathcal{C}}(C, \exp(A, B))$$

In other words, The functor $\exp(A, \cdot\,)$ is the right adjoint of $\mathrm{prod}(\cdot\,, A)$. We write 'curry' for the factorizer and 'eval' for the counit natural transformation. Then, for any object C and any morphism $f\colon \mathrm{prod}(C, A) \to B$, $\mathrm{curry}(f)$ is the unique morphism from $\exp(A, B)$ to C such that the following diagram commutes.



[5]Many category theory books use the notation B^A for the exponential of B by A.

The reason why the exponentials are function spaces is that their global elements are just morphisms.[6]

$$\mathrm{Hom}_{\mathcal{C}}(1, \exp(A,B)) \simeq \mathrm{Hom}_{\mathcal{C}}(\mathrm{prod}(1,A), B) \simeq \mathrm{Hom}_{\mathcal{C}}(A,B)$$

Let us write down the definition as a CDT declaration. Assume a CSL signature $\langle \Gamma, \Delta, \Psi \rangle$ which contains the definition of the binary product functor as we defined in subsection 3.3.2. Then, the exponential functor can be declared as follows.

right object $\exp(X,Y)$ with curry is
    eval: $\mathrm{prod}(\exp, X) \to Y$
end object

This is so far the most complicated CDT declaration. In the previous examples, functorial expressions E_i and E_i' in $(*)$ and $(**)$ are all simply variables. From definition 3.2.2, the CDT declaration above defines a functor symbol 'exp' of type $-+$, a natural transformation 'eval' of type

$$\lambda(X,Y).\mathrm{prod}(\exp(X,Y), X) \stackrel{\cdot}{\to} \lambda(X,Y).Y$$

and a factorizer 'curry' of type

$$\frac{f: \lambda(Z,X,Y).\mathrm{prod}(Z,X) \to \lambda(Z,X,Y).Y}{\mathrm{curry}(f): \lambda(Z,X,Y).Z \to \lambda(Z,X,Y).\exp(X,Y)}.$$

These types are what we expect them to be from the exponential adjunction. Let us once more convince ourselves that the semantics by F,G-dialgebras really defines the exponentials. $\langle \exp(A,B), \mathrm{eval}_{A,B} \rangle$ is the final F,G-dialgebra where $F(C) \stackrel{\mathrm{def}}{=} \mathrm{prod}(C,A)$ and $G(C) \stackrel{\mathrm{def}}{=} B$. This means that, for any $\langle C, f \rangle$ where C is an object and f is a morphism of $F(C) \to G(C)$, $\mathrm{curry}(f)$ is the unique morphism of

$$\langle \exp(A,B), \mathrm{eval}_{A,B} \rangle \to \langle C, f \rangle.$$

From definition 3.1.2 of F,G-dialgebras, $\mathrm{curry}(f)$ is the unique morphism $C \to \exp(A,B)$ which makes the following diagram commute.

$$\begin{array}{ccc}
F(C) = \mathrm{prod}(C,A) & \xrightarrow{\quad f \quad} & B = G(C) \\
\Big\downarrow F(\mathrm{curry}(f)) = \mathrm{prod}(\mathrm{curry}(f), \mathbf{I}) & \circlearrowright & \Big\| \mathbf{I} = G(\mathrm{curry}(f)) \\
F(\exp(A,B)) = \mathrm{prod}(\exp(A,B), A) & \xrightarrow[\mathrm{eval}_{A,B}]{} & B = G(\exp(A,B))
\end{array}$$

This is exactly the condition of 'exp' being the exponential functor.

[6] A *global element* of an object A is a morphism from the terminal object to A.

The declaration of the exponential functor in CDT very much looks like a declaration in a algebraic specification language (e.g. in CLEAR), but, as is well-known, we cannot define function spaces as algebras. The essential difference lies in that 'exp' is a right object, in other words, defined by the terminal data constraint rather than the initial one which CLEAR uses and in the availability of the factorizer 'curry'. If we define 'curry' as an ordinary function (or an ML function), its type is

$$(C \times A \to B) \to (C \to (A \to B))$$

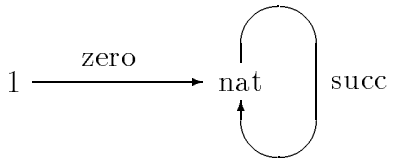
and this could never be a type of algebraic functions (i.e. functions defined by algebraic specification methods).

3.3.4 Natural Number Object

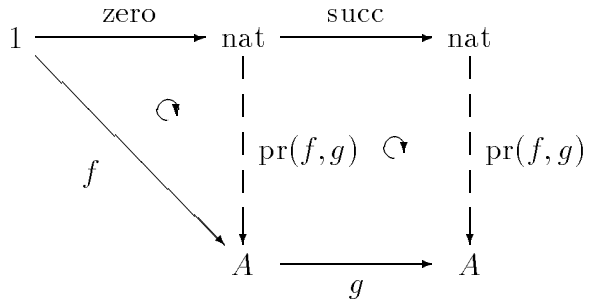
As we have already shown that the natural number object can be given by '**Left**', let us write it down as a CDT declaration. Although we can define the natural number object if we have only the terminal object, it is often very convenient to assume that a CSL signature $\langle \Gamma, \Delta, \Psi \rangle$ contains not only the terminal object but also the product functor and the exponential functor. The declaration of the natural number object as a CDT is

left object nat with pr is
    zero: $1 \to$ nat
    succ: nat $\to$ nat
end object

which defines a constant functor (i.e. an object) 'nat' with two morphisms 'zero' and 'succ'.



In addition, the factorizer 'pr' (standing for *p*rimitive *r*ecursion) gives for any morphisms $f: 1 \to A$ and $g: A \to A$ a unique morphism $\mathrm{pr}(f,g)$: nat $\to A$ such that the following diagram commutes.



As is well-known (e.g. [Goldblatt 79] chapter 13), 'pr' provides us to define any primitive recursive function.

Definition 3.3.1: For a category $\mathcal{C}$ with the natural number object 'nat', the terminal object '1' and the binary product functor 'prod', a morphism f is *primitive recursive* (on natural numbers) if it can be generated after finitely many steps by means of the following rules:

1. $f = \mathbf{I}_{\mathrm{nat}}\colon \mathrm{nat} \to \mathrm{nat}$,
2. $f = \mathrm{zero}\colon 1 \to \mathrm{nat}$,
3. $f = \mathrm{succ}\colon \mathrm{nat} \to \mathrm{nat}$,
4. $f = \mathrm{pi1}\colon \mathrm{prod}(\mathrm{nat}, \mathrm{nat}) \to \mathrm{nat}$,
5. $f = \mathrm{pi2}\colon \mathrm{prod}(\mathrm{nat}, \mathrm{nat}) \to \mathrm{nat}$,
6. $f = g \circ \mathrm{pair}(h, k)\colon A \to \mathrm{nat}$ for primitive recursive morphisms $g\colon \mathrm{prod}(\mathrm{nat}, \mathrm{nat}) \to \mathrm{nat}$, $h\colon A \to \mathrm{nat}$ and $k\colon A \to \mathrm{nat}$,
7. $f = g \circ \mathrm{prod}(h, k)\colon \mathrm{prod}(A, B) \to \mathrm{nat}$ for primitive recursive morphisms $g\colon \mathrm{prod}(\mathrm{nat}, \mathrm{nat}) \to \mathrm{nat}$, $h\colon A \to \mathrm{nat}$ and $k\colon B \to \mathrm{nat}$, and
8. $f\colon \mathrm{prod}(\mathrm{nat}, A) \to \mathrm{nat}$ satisfying
   (a) $f \circ \mathrm{pair}(\mathrm{zero} \circ !, \mathbf{I}) = g$, and
   (b) $f \circ \mathrm{pair}(\mathrm{succ} \circ \mathrm{pi1}, \mathrm{pi2}) = h \circ \mathrm{pair}(f, \mathbf{I})$

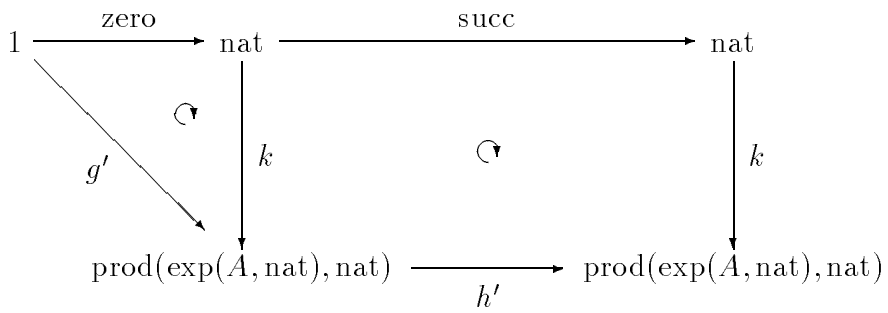
   for primitive recursive $g\colon A \to \mathrm{nat}$ and $h\colon \mathrm{prod}(\mathrm{nat}, \mathrm{prod}(\mathrm{nat}, A)) \to \mathrm{nat}$. □

This is a straight copy of the standard definition of primitive recursive functions on natural numbers.

Proposition 3.3.2: If a cartesian closed category $\mathcal{C}$ has the natural number object, it has all the primitive recursive morphisms.
Proof: It is suffice to show that there exists a morphism $f\colon \mathrm{prod}(\mathrm{nat}, A) \to \mathrm{nat}$ for any morphisms $g\colon A \to \mathrm{nat}$ and $h\colon \mathrm{prod}(\mathrm{nat}, \mathrm{prod}(\mathrm{nat}, A)) \to \mathrm{nat}$ such that

1. $f \circ \mathrm{pair}(\mathrm{zero} \circ !, \mathbf{I}) = g$, and
2. $f \circ \mathrm{pair}(\mathrm{succ} \circ \mathrm{pi1}, \mathrm{pi2}) = h \circ \mathrm{pair}(f, \mathbf{I})$.

There is a morphism $k\colon \mathrm{nat} \to \mathrm{prod}(\mathrm{exp}(A, \mathrm{nat}), \mathrm{nat})$ such that the following diagram commutes.



where g' and h' are

$$\begin{aligned} g' &\stackrel{\text{def}}{=} \text{pair}(\text{curry}(g \circ \text{pi2}), \text{zero}) \\ h' &\stackrel{\text{def}}{=} \text{pair}(\, \text{curry}(h \circ \text{pair}(\, \text{eval} \circ \text{pair}(\, \text{pi2}, \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{pi1} \circ \text{pi1}), \\ &\qquad\qquad\qquad\qquad\qquad\quad \text{pi2} \circ \text{pi1})), \\ &\qquad\qquad\quad \text{succ} \circ \text{pi2}). \end{aligned}$$

Therefore, k is $\text{pr}(g', h')$. Then, $f \stackrel{\text{def}}{=} \text{eval} \circ \text{prod}(\text{pi1} \circ k, \mathbf{I})$. We can easily show that this is what we wanted. []

For example, the morphism 'add' corresponding to the addition function of natural numbers can be given as

$$\text{add} \stackrel{\text{def}}{=} \text{eval} \circ \text{prod}(\text{pr}(\text{curry}(\text{pi2}), \text{curry}(\text{succ} \circ \text{eval})), \mathbf{I}).$$

It corresponds to the following usual definition of 'add'.

$$\begin{aligned} &\text{add}(0, y) = y \\ &\text{add}(x + 1, y) = \text{add}(x, y) + 1 \end{aligned}$$

Furthermore, we can easily prove categorically that 'add' satisfies familiar laws like commutativity (categorically $\text{add} \circ \text{pair}(\text{pi2}, \text{pi1}) = \text{add}$) and so on.

3.3.5 Lists

We have defined the data type of natural numbers in the previous subsection. Another algebraic data type which is often used in programming is the data type of lists. In CDT, the data type of lists is defined as follows:

$$\begin{aligned} &\text{left object list}(X) \text{ with prl is} \\ &\qquad \text{nil}\colon 1 \to \text{list} \\ &\qquad \text{cons}\colon \text{prod}(X, \text{list}) \to \text{list} \\ &\text{end object} \end{aligned}$$

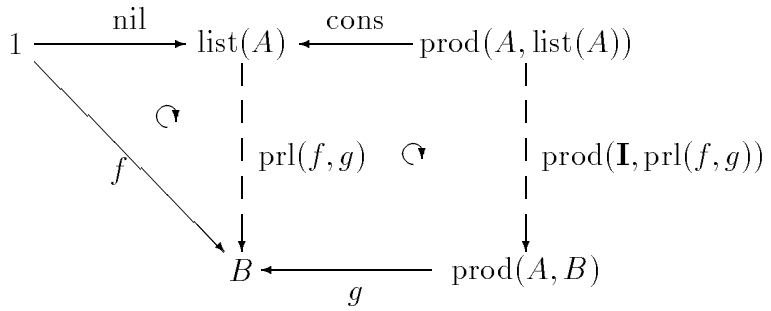
We needed to assume a CSL signature having the terminal object and the product functor. The declaration above defines a one argument covariant functor 'list', two natural transformations

$$\text{nil}\colon \lambda(X).1 \stackrel{\cdot}{\to} \lambda(X).\text{list}(X) \quad \text{and} \quad \text{cons}\colon \lambda(X).\text{prod}(X, \text{list}(X)) \stackrel{\cdot}{\to} \lambda(X).\text{list}(X)$$

and a factorizer 'prl' (standing for *p*rimitive *r*ecursion on *l*ist) whose type is

$$\frac{f\colon \lambda(Y, X).1 \to \lambda(Y, X).Y \qquad g\colon \lambda(Y, X).\text{prod}(X, Y) \to \lambda(Y, X).Y}{\text{prl}(f, g)\colon \lambda(Y, X).\text{list}(X) \to \lambda(Y, X).Y}.$$

As usual, we can express the situation as a diagram.



A global element of list(A) is normally constructed from 'nil' and 'cons'. For example, list(nat) has

$$\text{cons} \circ \text{pair}(\text{succ} \circ \text{zero}, \text{cons} \circ \text{pair}(\text{succ} \circ \text{succ} \circ \text{zero}, \text{nil}))$$

as a global element (in plain words, this element is the list of 1 and 2). 'nil' and 'cons' are usually called constructors of 'list'. We can see in general to define an algebraic CDT by listing its constructors. Destructors are defined using factorizers. In the case of 'list', 'hd' ($head$) and 'tl' ($tail$) can be defined as follows.

$$\text{hd} \stackrel{\text{def}}{=} \text{prl}(\text{in2}, \text{in1} \circ \text{pi1})$$
$$\text{tl} \stackrel{\text{def}}{=} \text{coprod}(\text{pi2}, \mathbf{I}) \circ \text{prl}(\text{in2}, \text{in1} \circ \text{prod}(\mathbf{I}, \text{case}(\text{cons}, \text{nil})))$$

Note that we have to define 'hd' and 'tl' as total functions (in a sense). The type of 'hd' is list(A) $\rightarrow$ coprod(A, 1) and is not list(A) $\rightarrow$ A. The type of 'tl' is also list(A) $\rightarrow$ coprod(list(A), 1). The type '1' is for error (like $\perp$ in a domain) and, for example, hd $\circ$ nil = in2.

As 'list' is a covariant functor, for a morphism $f\colon A \rightarrow B$ list(f)$\colon$list(A) $\rightarrow$ list(B) transforms a list of A elements to a list of B elements by applying f to each element. For example, list(succ)$\colon$list(nat) $\rightarrow$ list(nat) increments every element in a given list by one. In general, we have the following equations:

$$\text{list}(f) = \text{prl}(\text{nil}, \text{cons} \circ \text{prod}(f, \mathbf{I}))$$
$$\text{list}(f) \circ \text{nil} = \text{nil}$$
$$\text{list}(f) \circ \text{cons} \circ \text{pair}(x, l) = \text{cons} \circ \text{pair}(f \circ x, \text{list}(f) \circ l)$$

'list' corresponds to '`map`' function in ML and '`MAPCAR`' in LISP.

3.3.6 Final Co-Algebras (Infinite Lists and Co-Natural Number Object)

The objects we defined in the preceding subsections are all familiar either in category theory or in programming languages. Particularly, we have seen in subsection 3.3.4 and 3.3.5 the natural number object and the data type of lists, which are typical initial algebras. Recently, several works have been done about final coalgebras, which are

the dual of initial algebras (see [Arbib and Manes 80]). From their symmetry of CDT declarations, we can easily define final coalgebras in CDT as well as initial algebras.

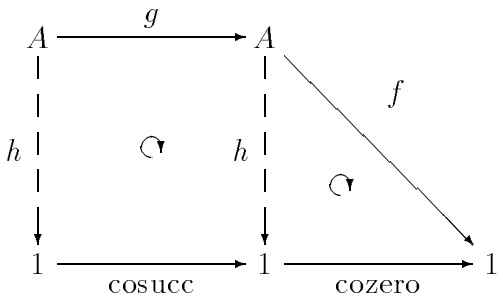
Let us dualize the declaration of the natural number object defined in subsection 3.3.4 by

left object nat with pr is
zero: $1 \to$ nat
succ: nat $\to$ nat
end object

If we simply replace 'left' by 'right' and change the direction of arrows, we get

right object conat with copr is
cozero: nat $\to 1$
cosucc: nat $\to$ nat
end object

Unfortunately, this is not an exciting object. We can prove that 'conat' is isomorphic to the terminal object as follows: from the uniqueness of terminal objects up to isomorphism in any category, we simply need to show that '1' is the terminal F, G-dialgebra for these particular F and G, that is, for any object A and morphisms $f: A \to 1$ and $g: A \to A$, there exists a unique morphism $h: A \to 1$ such that the following diagram commutes.

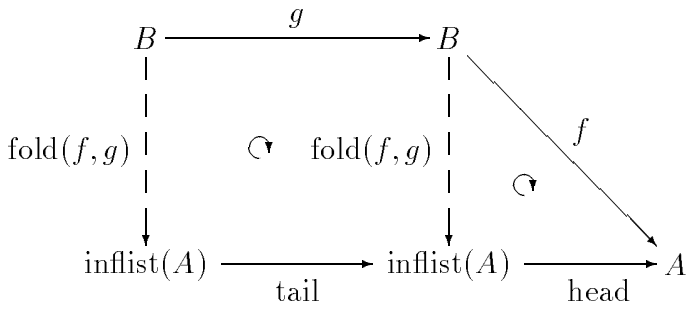


Indeed, we have this unique morphism $A \to 1$ because '1' is the terminal object and the above diagram trivially commutes.

Although the exact dual of the natural number object is not an interesting thing, we can modify it to get a CDT data type of infinite lists.

right object inflist(X) with fold is
head: inflist $\to X$
tail: inflist $\to$ inflist
end object

The diagram of explaining 'inflist' is



Since the functor 'inflist' is not so familiar in category theory or in conventional programming languages, let us find out what it is in the category of sets. We expect it to be a set of *infinite lists* in some sense.

Proposition 3.3.3: In the category of sets, for a set A, inflist(A) is the following set of ω-infinite sequences of elements in A.

$$\{\ (x_0, x_1, \ldots, x_n, \ldots) \mid x_i \in A\ \}$$

Proof: We define 'head' and 'tail' as follows:

$$\mathrm{head}((x_0, x_1, \ldots, x_n, \ldots)) \stackrel{\mathrm{def}}{=} x_0$$
$$\mathrm{tail}((x_0, x_1, \ldots, x_n, \ldots)) \stackrel{\mathrm{def}}{=} (x_1, x_2, \ldots, x_{n+1}, \ldots)$$

Let $\mathrm{fold}(f, g)(x)$ be a sequence $(h_0(x), h_1(x), \ldots, h_n(x), \ldots)$ for functions $f\colon B \to A$ and $g\colon B \to B$. The commutativity of the diagram above forces the following equations.

$$h_0(x) = f(x)$$
$$(h_1(x), h_2(x), \ldots, h_{n+1}(x), \ldots) = (h_0(x), h_1(x), \ldots, h_n(x), \ldots)$$

Therefore, $\mathrm{fold}(f, g)(x)$ is uniquely determined as

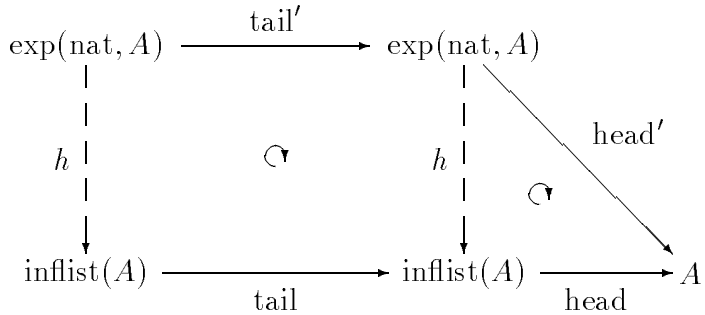
$$\mathrm{fold}(f, g)(x) \stackrel{\mathrm{def}}{=} (f(x), f(g(x)), \ldots, f(g^n(x)), \ldots) \qquad \square$$

Hence, at least in the category of sets, inflist(A) is really the data type of infinite lists of A elements.

More generally,

Proposition 3.3.4: In a cartesian closed category $\mathcal{C}$ with the natural number object, inflist(A) is isomorphic to $\exp(\mathrm{nat}, A)$.

Proof: Let us define $h\colon \exp(\mathrm{nat}, A) \to \mathrm{inflist}(A)$ to be the fill-in morphism of the following diagram.



where 'head'' and 'tail'' are

$$\text{head}' \stackrel{\text{def}}{=} \text{eval} \circ \text{pair}(\mathbf{I}, \text{zero} \circ !),$$
$$\text{tail}' \stackrel{\text{def}}{=} \text{curry}(\text{eval} \circ \text{prod}(\mathbf{I}, \text{succ})).$$

We define $h'\!:\text{inflist}(A) \to \exp(\text{nat}, A)$ to be

$$h' \stackrel{\text{def}}{=} \text{curry}(\text{eval} \circ \text{pair}(\text{pr}(\text{curry}(\text{head} \circ \text{pi2}), \exp(\text{tail}, \mathbf{I})) \circ \text{pi2}, \text{pi1})).$$

After some calculation, we can show that $h \circ h' = \mathbf{I}$ and $h' \circ h = \mathbf{I}$, so inflist(A) is isomorphic to $\exp(\text{nat}, A)$. []

This proposition tells us that

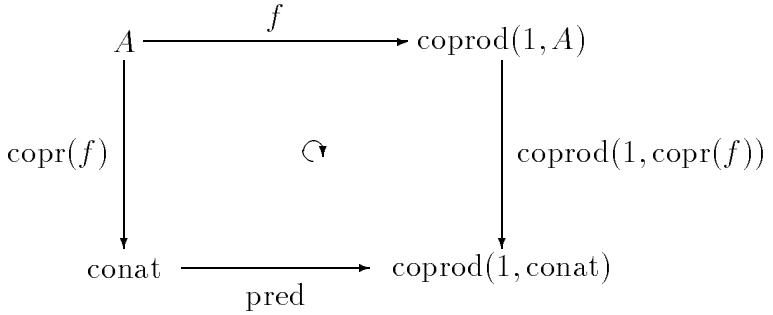
$$\text{inflist}(A) \cong A \times A \times \cdots \times A \times \cdots \cong \prod_{i=1}^{\infty} A.$$

Indeed, this is the dual of $\sum_{i=1}^{\infty} A$, the special case of which is the natural number object, nat $\cong \sum_{i=1}^{\infty} 1$.

We started this subsection by considering the dual of the natural number object. It led us to the CDT data type of infinite lists. We still have a different question whether there is a final coalgebra which resembles a data type of natural numbers. The answer is yes. The following right object defines a CDT data type of natural numbers plus alpha.

right object conat with copr is
    pred:conat $\to$ coprod(1, conat)
end object

The situation can be written as a diagram



The natural transformation 'pred' is the predecessor function and there is a morphism from 'nat' to 'conat' given by

$$\text{copr}(\text{pr}(\text{in1}, \text{in2} \circ \text{case}(\text{zero}, \text{succ})))$$

which we expect to be injective, but so far the author has been able neither to prove it nor to give a counter example. Note that there is always a morphism from the left object to its corresponding right object. 'Conat' has an interesting extra element, namely the infinity (∞). The ground element to denote it is

$$\text{infinity} \stackrel{\text{def}}{=} \text{copr}(\text{in2})$$

It is easy to prove that the predecessor of the infinity is itself (i.e. pred $\circ$ infinity = in2 $\circ$ infinity).

In the category of sets, 'conat' is really the set of natural numbers and the infinity.

Proposition 3.3.5: In **Set**, 'conat' is 'nat $\cup \{ \infty \}$'.
Proof: The predecessor function is defined as usual. Roughly speaking, for any function $f: A \to \mathrm{coprod}(1, A)$, $\mathrm{copr}(f)(x)$ is the number of applications of f to x to get the element of 1

$$\mathrm{copr}(f)(x) = n \qquad \text{where} \quad f^n(x) \in 1,$$

and, if it never results in the element of 1, $\mathrm{copr}(f)(x) = \infty$. We can easily show that this is the unique function which makes the 'conat' diagram commute. []

Therefore, in the category of sets, 'conat' is isomorphic to 'nat', but this is not the case for all the categories. There are some categories where 'conat' and 'nat' are not isomorphic.

Proposition 3.3.6: In category **TRF** of subsets of natural numbers as objects and total recursive functions as morphisms, there exists the natural number object but does not exist the co-natural number object.
Proof: **TRF**'s terminal object, initial object, product functor and coproduct functor are all the same as those of **Set**. For example, injections 'in1' and 'in2' for $\mathrm{coprod}(A, B)$ are trivially total and recursive, and for any two total recursive functions $f: A \to C$ and $g: B \to C$ $\mathrm{case}(f, g)$ is also total recursive function. We can write it down as a kind of program.

$$\mathrm{case}(f, g)(x) \stackrel{\mathrm{def}}{=} \text{if } x \in A \text{ then } f(x) \text{ else } g(x)$$

The natural number object 'nat' in **TRF** is also the ordinary set of natural numbers. 'Zero' and 'succ' are total recursive functions from the very definition of recursive functions and for $f: 1 \to A$ and $g: A \to A$ of total recursive functions, $\mathrm{pr}(f, g)$ can be written as the following program.

$$\mathrm{pr}(f, g)(n) \stackrel{\mathrm{def}}{=} \text{if } n = 0 \text{ then } f() \text{ else } g(\mathrm{pr}(f, g)(n-1))$$

which defines a total recursive function.

However, we cannot have the co-natural number object in **TRF**. The program of $\mathrm{copr}(f)$ for a total recursive function $f: A \to \mathrm{coprod}(1, A)$ can only be

$$\mathrm{copr}(f)(x) \stackrel{\mathrm{def}}{=} \text{if } f(x) \in 1 \text{ then } 0 \text{ else } \mathrm{copr}(f)(f(x)) + 1$$

which is recursive but not total. []

There is also a category which has both 'nat' and 'conat' and in which they are non-isomorphic. We will show this in chapter 4.

From the point of view of finding fixed points of functors, 'nat' is the initial fixed point of $F(X) \stackrel{\mathrm{def}}{=} 1 + X$ and 'conat' is the final fixed point of the same functor.

3.3.7 Automata

The declarations of initial algebras and final coalgebras do not use the full power of the CDT declaration mechanism. Their unit and counit natural transformations always have the form

$$\alpha: E \to F$$

for initial algebras and have the form

$$\alpha: F \to E$$

for final coalgebras, where E is any functorial expression but F is a variable (more specially, the variable which denotes the object we declare). Therefore, all we are doing is just listing constructors for initial algebras and listing destructors for final coalgebras. We will see what kinds of form define sensible functors in section 3.5. So far, the only exception was the exponentials. We will see another example in this subsection.

One of the interesting applications of category theory to computer science is to automata theory. [Arbib and Manes 75] presents the category $\mathbf{Dyn}(I)$ of I-dynamics whose object is

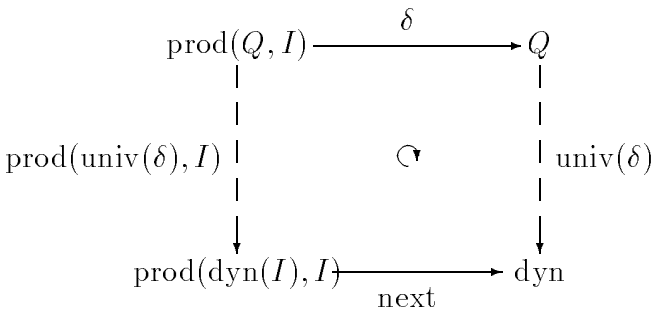
$$Q \times I \xrightarrow{\quad\delta\quad} Q,$$

where Q is the set of states, I is the set of inputs and δ is a *dynamics* which is a function determining the next state of the automaton according to the current state and input.

From this, we can construct a categorical data type of automata.

right object dyn(I) with univ is
    next: prod(dyn, I) → dyn
end object

Note that our dyn(I) for an object I is just an object; it is not a category like $\mathbf{Dyn}(I)$ is. The diagram which explains this right object is



For any dynamics $\delta: \mathrm{prod}(Q, I) \to Q$ and an initial state $q_0: 1 \to Q$, we get an automaton

$$1 \xrightarrow{\quad\mathrm{univ}(\delta) \circ q_0\quad} \mathrm{dyn}(I)$$

as a global element of $\mathrm{dyn}(I)$. Though we can put this automaton into the next state by applying 'next', we are never ever able to see its behaviour from the outside. Moreover, because of this non-observability, we can easily prove that $\mathrm{dyn}(I)$ is in fact isomorphic to the terminal object. In order to make 'dyn' a more sensible object, we need to add an output function. The new categorical data type of automata is[7]

$$\begin{array}{l}\text{right object } \mathrm{dyn}'(I,O) \text{ with } \mathrm{univ}' \text{ is}\\ \quad \mathrm{next}'\colon \mathrm{prod}(\mathrm{dyn}', I) \to \mathrm{dyn}'\\ \quad \mathrm{output}'\colon \mathrm{dyn}' \to O\\ \text{end object}\end{array}$$

For any dynamics $\delta\colon \mathrm{prod}(Q, I) \to Q$, any output function $\beta\colon Q \to O$ and an initial state $q_0\colon 1 \to Q$, we have a global element in $\mathrm{dyn}'(I, O)$

$$1 \xrightarrow{\mathrm{univ}'(\delta,\beta)\circ q_0} \mathrm{dyn}'(I, O).$$

We can obtain its next state by applying 'next'' and its output by 'output''. In addition, the following proposition holds.

Proposition 3.3.7: In a cartesian closed category, the categorical data type of Moore automata, $\mathrm{dyn}'(I, O)$, is isomorphic to $\exp(\mathrm{list}(I), O)$.
Proof: By defining two morphisms between them and proving that they form an isomorphism. □

3.3.8 Obscure Categorical Data Types

We have defined more or less familiar data types as categorical data types in the preceding subsections. One might ask whether CDT can define any data types which are unable to be defined in other languages or methods. The data type of automata is such an example and we can invent similar examples more, but still they are familiar (or we are just trying to express our familiar data types in CDT). In fact, CDT allows us very obscure data types, some of which may not be conceptualized in the human brain (at least not in the author's brain).

From the prime requirement of CDT, it can define right and left adjoint functors of any existing functors, and in subsection 3.3.5, we defined 'list' as a covariant functor, so that we can declare its left and right adjoint functors in CDT as follows.

$$\begin{array}{l}\text{left object } \mathrm{ladjlist}(X) \text{ with } \psi \text{ is}\\ \quad \alpha\colon X \to \mathrm{list}(\mathrm{ladjlist})\\ \text{end object}\end{array}$$

[7]The original definition we used was

$$\begin{array}{l}\text{right object } \mathrm{dyn}'(I,O) \text{ with } \mathrm{univ}' \text{ is}\\ \quad \mathrm{next}'\colon \mathrm{prod}(\mathrm{dyn}', I) \to \mathrm{prod}(\mathrm{dyn}', O)\\ \text{end object}\end{array}$$

which gave us the categorical data type of Mealy automata. The current definition gives us the data type of Moore automata.

right object radjlist(X) with ψ' is
α': list(radjlist) $\rightarrow X$
end object

Some questions arise immediately after defining these data types: are they familiar data types, and are they in any way useful? The answers to the both questions are unfortunately negative. For the left adjoint,

Proposition 3.3.8: In a cartesian closed category, 'ladjlist(A)' for any object A is isomorphic to the initial object.
Proof: It is easy to show that the initial object makes the characteristic diagram of 'ladjlist' commute. Note that in a cartesian closed category 'list(0)' is isomorphic to the terminal object so that the unit morphism of 'ladjlist' is the unique morphism to the terminal object. []

The right adjoint functor is more harmful than the left one.

Proposition 3.3.9: A cartesian closed category with 'radjlist' degenerates (i.e. all the objects are isomorphic).
Proof: We have the following morphism from the initial object.

$$1 \xrightarrow{\text{nil}} \text{list}(\text{radjlist}(0)) \xrightarrow{\alpha'} 0$$

Then, it is easy to show that the terminal object is isomorphic to the initial one. This further implies that any object in the category is isomorphic to the initial object.

$$A \cong \text{prod}(A, 1) \cong \text{prod}(A, 0) \cong 0 \qquad []$$

Most of the left and right adjoint functors of conventional data types follow the same pattern as 'list', that is, they are either trivial or destructive, so they are useless.

Hence, a natural question to ask ourselves is that what kind of categorical data types are useful. But what is the formal criteria of *useful* data types? We have not yet defined this. We will come back to this in chapter 4 and see it from a point of view of computability of categorical data types.

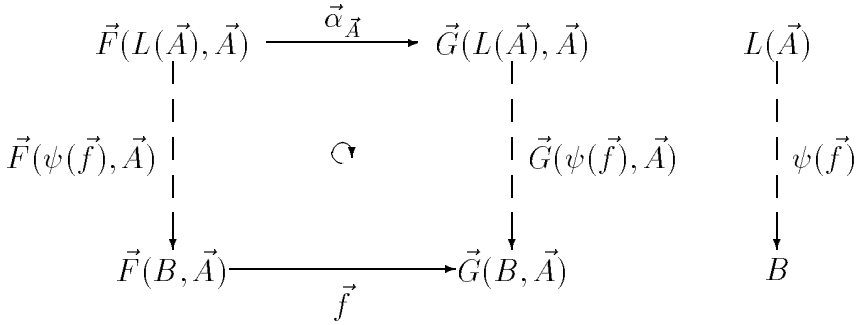
3.4 Semantics of Categorical Data Types

In definition 3.2.2, we associated a CDT declaration to a CSL signature extension (an injective morphism in **CSig**). In this section, we will see it as a CSL theory extension and give the precise semantics of CDT declarations.

First, from our informal intention of CDT declarations we have to figure out the CSL statements which characterize them. A CDT declaration

left object $L(X_1, \ldots, X_n)$ with ψ is
α_1: $E_1 \rightarrow E_1'$
$\cdots$
α_m: $E_m \rightarrow E_m'$
end object

is the syntactic form of defining the functor $L = \mathbf{Left}[\vec{F}, \vec{G}]$, where $\vec{F}$ and $\vec{G}$ are corresponding functors for $E_1, \ldots, E_m$ and $E'_1, \ldots, E'_m$, respectively. $\langle L(\vec{A}), \vec{\alpha} \rangle$ is the initial object of $\mathbf{DAlg}(\vec{F}, \vec{G})$ and ψ is its mediating morphism, that is, for any morphisms $\vec{f}: \vec{F}(B, \vec{A}) \to \vec{G}(B, \vec{A})$, $\psi(\vec{f})$ gives a unique morphism from $L(\vec{A})$ to B such that the following diagram commutes.



The commutativity of this diagram can be expressed as the following equation:

$$\vec{G}(\psi(\vec{f}), \vec{A}) \circ \vec{\alpha} = \vec{f} \circ \vec{F}(\psi(\vec{f}), \vec{A}), \qquad (*)$$

and the uniqueness can be expressed as the following conditional equation:

$$\vec{G}(h, \vec{A}) \circ \vec{\alpha} = \vec{f} \circ \vec{F}(h, \vec{A}) \Rightarrow h = \psi(\vec{f}) \qquad (**)$$

The two equations say everything about L, $\vec{\alpha}$ and ψ. Let us now translate them to CSL statements in $\langle \Gamma \cup \{L\}, \Delta \cup \{\alpha_1, \ldots, \alpha_m\}, \Psi \cup \{\psi\} \rangle$ so as to give complete description of the CDT declaration above in CSL. $(*)$ corresponds to the following m CSL equations:

$$E'_i[\psi(f_1, \ldots, f_m)/L] \circ \alpha_i = f_i \circ E_i[\psi(f_1, \ldots, f_m)/L]^8 \qquad (\mathrm{LEQ}_i)$$

$(i = 1, \ldots, m)$ and $(**)$ corresponds to the following conditional CSL equation.

$$E'_1[g/L] \circ \alpha_1 = f_1 \circ E_1[g/L] \wedge \ldots \wedge E'_m[g/L] \circ \alpha_m = f_m \circ E_m[g/L] \Rightarrow g = \psi(f_1, \ldots, f_m) \qquad (\mathrm{LCEQ})$$

In addition, we should have a CSL equation expressing functors by factorizers and natural transformations. We can extract such an equation from proposition 3.1.4.

$$L(h_1, \ldots, h_n) = \psi(E'_1[h_i/X_i] \circ \alpha_1 \circ E_1[h_i/X_i], \ldots, E'_m[h_i/X_i] \circ \alpha_m \circ E_m[h_i/X_i])^9 \qquad (\mathrm{LFEQ})$$

Therefore, the semantics of CDT declaration can be given as a CSL theory extension as follows.

[8] $E_i[\psi(f_1, \ldots, f_m)/L]$ means replacing the variable L by $\psi(f_1, \ldots, f_m)$ and replacing the other variables $X_1, \ldots, X_n$ by the identities, that is it is a shorthand for

$$E_i[\psi(f_1, \ldots, f_m)/L, \mathbf{I}/X_1, \ldots, \mathbf{I}/X_n].$$

[9] $E_1[h_i/X_i]$ is a shorthand for $E_1[\mathbf{I}/L, h_1/X_1, \ldots, h_n/X_n]$.

Definition 3.4.1: Given a CSL theory $\langle \Gamma, \Delta, \Psi, \Theta \rangle$, a CDT declaration

$$\begin{array}{l} \text{left object } L(X_1, \ldots, X_n) \text{ with } \psi \text{ is} \\ \quad \alpha_1 : E_1 \to E_1' \\ \quad \cdots \\ \quad \alpha_m : E_m \to E_m' \\ \text{end object,} \end{array}$$

where E_i and E_i' $(i = 1, \ldots, m)$ are CSL functorial expressions over $\langle \Gamma, \Delta, \Psi \rangle$ whose variables are $X_1, \ldots, X_n$ and L, is associated with a CSL theory morphism

$$\sigma_L : \langle \Gamma, \Delta, \Psi, \Theta \rangle \to \langle \Gamma \cup \{L\}, \Delta \cup \{\alpha_1, \ldots, \alpha_m\}, \Psi \cup \{\psi\}, \\ \Theta \cup \{\mathrm{LEQ}_1, \ldots, \mathrm{LEQ}_m, \mathrm{LCEQ}, \mathrm{LFEQ}\} \rangle$$

where the types of $L, \alpha_1, \ldots, \alpha_m$ and ψ are as given in definition 3.2.2. Dually, we can associate to a right object R

$$\begin{array}{l} \text{right object } R(X_1, \ldots, X_n) \text{ with } \psi \text{ is} \\ \quad \alpha_1 : E_1 \to E_1' \\ \quad \cdots \\ \quad \alpha_m : E_m \to E_m' \\ \text{end object,} \end{array}$$

a CSL theory morphism σ_R.

$$\sigma_R : \langle \Gamma, \Delta, \Psi, \Theta \rangle \to \langle \Gamma \cup \{R\}, \Delta \cup \{\alpha_1, \ldots, \alpha_m\}, \Psi \cup \{\psi\}, \\ \Theta \cup \{\mathrm{REQ}_1, \ldots, \mathrm{REQ}_m, \mathrm{RCEQ}, \mathrm{RFEQ}\} \rangle$$

where REQ_i, RCEQ and RFEQ are

$$\alpha_i \circ E_i[\psi(f_1, \ldots, f_m)/R] = E_i'[\psi(f_1, \ldots, f_m)/R] \circ f_i \qquad (\mathrm{REQ}_i)$$

$$\alpha_1 \circ E_1[g/R] = E_1'[g/R] \circ f_1 \wedge \ldots \\ \wedge \alpha_m \circ E_m[g/R] = E_m'[g/R] \circ f_m \Rightarrow g = \psi(f_1, \ldots, f_m) \qquad (\mathrm{RCEQ})$$

$$R(h_1, \ldots, h_n) = \\ \psi(E_1'[h_1/X_i] \circ \alpha_1 \circ E_1[h_i/X_i], \ldots, E_m'[h_i/X_i] \circ \alpha_m \circ E_m[h_i/X_i]) \qquad (\mathrm{RFEQ})$$

□

Example 3.4.2: Let $\langle \Gamma, \Delta, \Psi, \Theta \rangle$ be the CSL theory of cartesian closed categories (see examples 2.2.2 and 2.5.3). On top of this, as we have seen in section 3.3.4, we can define a natural number object by

$$\begin{array}{l} \text{left object nat with pr is} \\ \quad \text{zero}: 1 \to \text{nat} \\ \quad \text{succ}: \text{nat} \to \text{nat} \\ \text{end object.} \end{array}$$

The CSL statements characterizing this object are

$$\mathrm{pr}(f,g) \circ \mathrm{zero} = f \qquad (\mathrm{LEQ}_{\mathrm{nat},1})$$
$$\mathrm{pr}(f,g) \circ \mathrm{succ} = g \circ \mathrm{pr}(f,g) \qquad (\mathrm{LEQ}_{\mathrm{nat},2})$$
$$h \circ \mathrm{zero} = f \wedge h \circ \mathrm{succ} = g \circ h \Rightarrow h = \mathrm{pr}(f,g) \qquad (\mathrm{LCEQ}_{\mathrm{nat}})$$

The CSL extension σ_{nat} associated with the declaration above is:

$$\langle \Gamma, \Delta, \Psi \rangle \xrightarrow{\sigma_{\mathrm{nat}}} \langle \Gamma \cup \{\mathrm{nat}\}, \Delta \cup \{\mathrm{zero}, \mathrm{succ}\}, \Psi \cup \{\mathrm{pr}\}, \Theta \cup \{\mathrm{LEQ}_{\mathrm{nat},1}, \mathrm{LEQ}_{\mathrm{nat},2}, \mathrm{LCEQ}_{\mathrm{nat}}\} \rangle \ \square$$

Thus, each CDT declaration can be associated with a CSL theory extension. This can be thought as a semantics of CDT declarations. However, it is sometimes convenient to regard their semantics to be real categories.

Definition 3.4.3: A sequence of CDT declarations $D_1, \ldots, D_l$ defines a sequence of CSL theory extensions starting from the empty CSL theory.

$$\langle \emptyset, \emptyset, \emptyset, \emptyset \rangle \xrightarrow{\sigma_{D_1}} \langle \Gamma_1, \Delta_1, \Psi_1, \Theta_1 \rangle \xrightarrow{\sigma_{D_2}} \cdots \xrightarrow{\sigma_{D_l}} \langle \Gamma_l, \Delta_l, \Psi_l, \Theta_l \rangle$$

We define a CSL structure of the CDT declaration sequence $D_1, \ldots, D_l$ to be a category which is a CSL theory structure of $\langle \Gamma_l, \Delta_l, \Psi_l, \Theta_l \rangle$, and the free structure of $D_1, \ldots, D_l$ to be the free category of this CSL theory (see section 2.6). $\square$

If we do not rely on any ways of defining functors other than CDT declarations and if we do not accept any pre-defined functors, it is inevitable to start with the empty CSL theory. We have defined cartesian closed categories as a CSL theory in examples 2.2.2 and 2.5.3, but we can do so in CDT starting from the empty theory by just declaring the terminal object (see subsection 3.3.1), products (see subsection 3.3.2) and exponentials (see subsection 3.3.3). The advantage of the latter is that we neither need to think about equations nor need to do tedious typing of functors, natural transformations or factorizers. These things come out automatically, so it is easy to define categories and there is less chance to make mistakes.

We have introduced CDT declarations from **Left** and **Right**, but we could not connect them formally. Now, after having models of CDT as categories, we can do so.

Proposition 3.4.4: Let $\langle \mathcal{C}, \xi \rangle$ be a CSL structure of a CDT declaration sequence D_1, $\ldots$, D_l. If D_i is

left object $L(X_1, \ldots, X_n)$ with ψ is
$\alpha_1 : E_1 \to E_1'$
$\cdots$
$\alpha_m : E_m \to E_m'$
end object

then $\xi L = \mathbf{Left}(\vec{F}, \vec{G})$ where

$$\vec{F} \stackrel{\mathrm{def}}{=} \langle \xi\lambda(L, X_1, \ldots, X_n).E_1, \ldots, \xi\lambda(L, X_1, \ldots, X_n).E_m \rangle$$
$$\vec{G} \stackrel{\mathrm{def}}{=} \langle \lambda(L, X_1, \ldots, X_n).E_1', \ldots, \lambda(L, X_1, \ldots, X_n).E_m' \rangle.$$

Name	CDT Declaration	CSL Statements
Terminal	right object 1 with ! end object	$f = !$
Initial	left object 0 with !! end object	$f = !!$
Products	right object prod(X, Y) with pair is pi1: prod $\to X$ pi2: prod $\to Y$ end object	pi1 $\circ$ pair$(f, g) = f$ pi2 $\circ$ pair$(f, g) = g$ $h =$ pair(pi1 $\circ h$, pi2 $\circ h$) prod$(f, g) =$ pair$(f \circ$ pi1$, g \circ$ pi2)
Coproducts	left object coprod(X, Y) with case is in1: $X \to$ coprod in2: $Y \to$ coprod end object	case$(f, g) \circ$ in1 $= f$ case$(f, g) \circ$ in2 $= g$ $h =$ case$(h \circ$ in1$, h \circ$ in2) coprod$(f, g) =$ case(in1 $\circ f$, in2 $\circ g$)
Exponentials	right object exp(X, Y) with curry is eval: prod(exp, X) $\to Y$ end object	eval $\circ$ prod(curry(f), $\mathbf{I}$) $= f$ $h =$ curry(eval $\circ$ prod$(h, \mathbf{I})$) exp$(f, g) =$ curry$(g \circ$ eval $\circ$ prod$(\mathbf{I}, f))$
NNO	left object nat with pr is zero: $1 \to$ nat succ: nat $\to$ nat end object	pr$(f, g) \circ$ zero $= f$ pr$(f, g) \circ$ succ $= g \circ$ pr(f, g) $h \circ$ succ $= g \circ h \Rightarrow h =$ pr$(h \circ$ zero$, g)$
Lists	left object list(X) with prl is nil: $1 \to$ list cons: prod$(X,$list$) \to$ list end object	prl$(f, g) \circ$ nil $= f$ prl$(f, g) \circ$ cons $=$ prod$(\mathbf{I},$ prl$(f, g)) \circ g$ $h \circ$ cons $=$ prod$(\mathbf{I}, h) \circ g \Rightarrow$ $h =$ prl$(h \circ$ nil$, g)$ list$(f) =$ prl(nil, cons $\circ$ prod$(f, \mathbf{I}))$
Infinite Lists	right object inflist(X) with fold is head: inflist $\to X$ tail: inflist $\to$ inflist end object	head $\circ$ fold$(f, g) = f$ tail $\circ$ fold$(f, g) =$ fold$(f, g) \circ g$ tail $\circ h = g \circ h \Rightarrow h =$ fold(head $\circ h, g)$ inflist$(f) =$ fold$(f \circ$ head, tail)
Co-NNO	right object conat with copr is pred: conat $\to$ coprod$(1,$ conat$)$ end object	pred $\circ$ copr$(f) =$ coprod$(\mathbf{I},$ copr$(f)) \circ f$ $h \circ$ pred $= f \circ$ coprod$(\mathbf{I}, h) \Rightarrow$ $h =$ copr(f)
Automata	right object dyn$'(I, O)$ with univ$'$ is next$'$: prod(dyn$', I) \to$ dyn$'$ output$'$: dyn$' \to O$ end object	next$' \circ$ prod(univ$'(f, g), \mathbf{I}) =$ univ$'(f, g) \circ f$ output$' \circ$ univ$'(f, g) = g$ next$' \circ$ prod$(h, \mathbf{I}) = h \circ f \Rightarrow$ $h =$ univ$'(f,$ output$' \circ h)$

Figure 3.1: CDT Objects

The similar thing holds for right CDT declarations.
Proof: Trivial, because we set the CSL statements so that this proposition holds. □

Finally, in this section, let us summarize the objects we have defined in this chapter and their characteristic CSL statements in figure 3.1.

3.5 Existence of Left and Right

In section 3.1, we have introduced functors **Left**$[F, G]$ and **Right**$[F, G]$ with the condition which characterizes them, but we did not consider whether such functors exist or not. In this section, we study them mathematically and present a condition of the existence.

Let us recall the standard construction of initial T-algebras (see, for example, [Scott 76, Lehmann and Smyth 81]).

Proposition 3.5.1: For a ω-cocomplete category $\mathcal{C}$ (i.e. it has colimit of any ω-chain

diagram) and an endo-functor $T{:}\mathcal{C} \to \mathcal{C}$ which is ω-cocontinuous (i.e. it preserves colimit of any ω-chain diagram), its initial T-algebra is given by the colimit of the following ω-chain diagram.[10]

$$0 \xrightarrow{!!} T(0) \xrightarrow{T(!!)} T^2(0) \xrightarrow{T^2(!!)} \cdots \xrightarrow{T^{n-1}(!!)} T^n(0) \xrightarrow{T^n(!!)} \cdots \quad []$$

As we presented in section 3.1, $\mathbf{Left}[F,G]$ is a generalization of initial T-algebras, where F is a functor of $\mathcal{C} \times \mathcal{D} \to \mathcal{E}$ and G is of $\mathcal{C} \times \mathcal{D}^- \to \mathcal{E}$. We will reduce the existence problem of **Left** to that of corresponding T-algebras. For a $\mathcal{D}$ object A, from its definition $\langle \mathbf{Left}[F,G](A), \eta_A \rangle$ is the initial object in the category $\mathbf{DAlg}(F(\,\cdot\,,A), G(\,\cdot\,,A))$, so

$$F(\mathbf{Left}[F,G](A), A) \xrightarrow{\eta_A} G(\mathbf{Left}[F,G](A), A)$$

Now, if $G(\,\cdot\,,A)$ has a left adjoint functor, say $H(\,\cdot\,,A){:}\mathcal{E} \to \mathcal{C}$, this morphism η_A has its one-to-one corresponding morphism

$$H(F(\mathbf{Left}[F,G](A), A), A) \longrightarrow \mathbf{Left}[F,G](A).$$

This means that $\mathbf{Left}[F,G](A)$ is a T-algebra, where $T(B) \stackrel{\text{def}}{=} H(F(B,A),A)$, and we naturally expect this T-algebra to be special. It really is the initial T-algebra, so we can formulate the following theorem.

Theorem 3.5.2: Let $F{:}\mathcal{C} \times \mathcal{D} \to \mathcal{E}$ and $G{:}\mathcal{C} \times \mathcal{D}^- \to \mathcal{E}$ be functors. If

1. $\mathcal{C}$ is ω-cocomplete,
2. for each $\mathcal{D}$ object A, $G_A \stackrel{\text{def}}{=} G(\,\cdot\,,A){:}\mathcal{C} \to \mathcal{E}$ has a left adjoint $H_A{:}\mathcal{E} \to \mathcal{C}$, and
3. for each $\mathcal{D}$ object A, $F_A \stackrel{\text{def}}{=} F(\,\cdot\,,A){:}\mathcal{C} \to \mathcal{E}$ is ω-cocontinuous,

then $\mathbf{Left}[F,G](A)$ exists in $\mathcal{C}$ and

$$\mathbf{Left}[F,G](A) = \operatorname*{colimit}_n (H_A \circ F_A)^n(0)$$

Proof: Since a left adjoint is cocontinuous, $T_A \stackrel{\text{def}}{=} H_A \circ F_A$ is ω-cocontinuous if F_A is so. All we have to show is that the initial T_A-algebra gives the initial object of $\mathbf{DAlg}(F_A, G_A)$, and the rest follows from proposition 3.5.1.

Let the initial T_A-algebra be I paired with a morphism

$$H_A(F_A(I)) \xrightarrow{\iota} I.$$

[10] In general, this sequence might not converge at ω. In such a case, we may extend the sequence up to any ordinal such that

1. $T^{\alpha+1}(0) = T(T^\alpha(0))$, and
2. $T^\beta(0) = \operatorname*{colimit}_{\alpha<\beta} T^\alpha(0)$ for a limit ordinal β (treating 0 as a limit ordinal).

For an object $\langle B, f\rangle$ in $\mathbf{DAlg}(F_A, G_A)$

$$F_A(B) \xrightarrow{\quad f \quad} G_A(B)$$

we have to construct the unique morphism from I to B which makes a certain diagram commute. To do so, let us name the factorizer of the adjunction $H_A \vdash G_A$ ψ, that is, ψ is the natural isomorphism $\mathrm{Hom}_{\mathcal{C}}(H_A(C), D) \xrightarrow{\cong} \mathrm{Hom}_{\mathcal{E}}(C, G_A(D))$. Then, $\psi^{-1}(f)$ gives a T_A-algebra

$$H_A(F_A(B)) \xrightarrow{\quad \psi^{-1}(f) \quad} B$$

and since $\langle I, \iota\rangle$ is the initial T_A-algebra, there exists a unique morphism $g\colon I \to B$ such that the following diagram commutes.

$$\begin{array}{ccc} H_A(F_A(I)) & \xrightarrow{\quad \iota \quad} & I \\ {\scriptstyle H_A(F_A(g))}\downarrow & \circlearrowright & \downarrow{\scriptstyle g} \\ H_A(F_A(B)) & \xrightarrow[\quad \psi^{-1}(f) \quad]{} & B \end{array} \qquad (*)$$

The naturality of ψ converts this diagram to the following commutative diagram.

$$\begin{array}{ccc} F_A(I) & \xrightarrow{\quad \psi(\iota) \quad} & G_A(I) \\ {\scriptstyle F_A(g)}\downarrow & \circlearrowright & \downarrow{\scriptstyle G_A(g)} \\ F_A(B) & \xrightarrow[\quad f \quad]{} & G_A(B) \end{array} \qquad (**)$$

(i.e. $\psi(g \circ \iota) = G_A(g) \circ \psi(\iota)$ and $\psi(\psi^{-1}(f) \circ H_A(F_A(g))) = f \circ F_A(g)$). That showed the existence of g. The uniqueness of g is no more difficult. If $g\colon I \to B$ satisfies $(**)$, then by applying ψ^{-1} we get $(*)$ back again and so there g should be unique. □

Example 3.5.3: Trivially, if $\mathcal{E}$ is $\mathcal{C}$ and $G\colon \mathcal{C} \times \mathcal{D}^{-} \to \mathcal{C}$ is the projection functor, G_A is the identity which has the left adjoint H_A which is also the identity functor. In this case, the above theorem is essentially stating the same thing as proposition 3.5.1. □

Example 3.5.4: Another simple case is that $\mathcal{E}$ is $\mathcal{C} \times \mathcal{C}$ and G_A is the diagonal functor. Its left adjoint is the binary coproduct functor. $F_A\colon \mathcal{C} \to \mathcal{C} \times \mathcal{C}$ can be

decomposed into F'_A and F''_A both of which are functors of $\mathcal{C} \times \mathcal{D} \to \mathcal{C}$ such that $F_A(B) = \langle F'_A(B), F''_A(B) \rangle$. The theorem states that $\mathbf{Left}[F,G](A)$ is the initial T_A-algebra where $T_A(B) \stackrel{\text{def}}{=} F'_A(B) + F''_A(B)$. This explains that our natural number object in subsection 3.3.4 is the initial T-algebra where $T(B) \stackrel{\text{def}}{=} 1 + B$. □

Theorem 3.5.2 has its dual form for $\mathbf{Right}[F,G]$.

Theorem 3.5.5: Let $F{:}\mathcal{C} \times \mathcal{D} \to \mathcal{E}$ and $G{:}\mathcal{C} \times \mathcal{D}^- \to \mathcal{E}$ be functors. If

1. $\mathcal{C}$ is ω-complete,
2. for each $\mathcal{D}$ object A, $F_A \stackrel{\text{def}}{=} F(\,\cdot\,, A){:}\mathcal{C} \to \mathcal{E}$ has a right adjoint $K_A{:}\mathcal{E} \to \mathcal{C}$, and
3. for each $\mathcal{D}$ object A, $G_A \stackrel{\text{def}}{=} G(\,\cdot\,, A){:}\mathcal{C} \to \mathcal{E}$ is ω-continuous,

then $\mathbf{Right}[F,G](A)$ exists in $\mathcal{C}$ and

$$\mathbf{Right}[F,G](A) = \lim_n (K_A \circ F_A)^n(1) \qquad \square$$

Example 3.5.6: As an application of this theorem, let us calculate $\mathrm{dyn}'(I,O)$. For this, we should take $\mathcal{D}$ to be $\mathcal{C} \times \mathcal{C}^-$, $\mathcal{E}$ to be $\mathcal{C} \times \mathcal{C}$, $F{:}\mathcal{C} \times \mathcal{C} \times \mathcal{C}^- \to \mathcal{C} \times \mathcal{C}$ to be $F(A,I,O) \stackrel{\text{def}}{=} \langle A \times I, A \rangle$ and $G{:}\mathcal{C} \times \mathcal{C}^- \times \mathcal{C} \to \mathcal{C} \times \mathcal{C}$ to be $G(A,I,O) \stackrel{\text{def}}{=} \langle A, O \rangle$. $F_{I,O}$ has a right adjoint $H_{I,O}(D,E) \stackrel{\text{def}}{=} \exp(I,D) \times E$. Therefore, $\mathrm{dyn}'(I,O)$ is the final $T_{I,O}$-coalgebra, where $T_{I,O}(A) \stackrel{\text{def}}{=} H_{I,O}(G_{I,O}(A)) = \exp(I,A) \times O$. Now, let us calculate $\lim_n T^n_{I,O}(1)$.

$$\begin{aligned}
T^0_{I,O}(1) &\cong 1 \\
T^1_{I,O}(1) &\cong \exp(I,1) \times O \cong O \cong \exp(1,O) \\
T^2_{I,O}(1) &\cong T_{I,O}(O) \cong \exp(I,O) \times O \cong \exp(1+I,O) \\
T^3_{I,O}(1) &\cong \exp(I, \exp(1+I,O)) \times O \\
&\cong \exp(I \times (1+I), O) \times O \\
&\cong \exp(1+I+I^2, O) \\
&\cdots \\
T^n_{I,O}(1) &\cong \exp(1+I+I^2+\cdots+I^{n-1}, O) \\
&\cdots
\end{aligned}$$

Therefore, $\mathrm{dyn}'(I,O) \cong \lim_n T^n_{I,O}(1) \cong \exp(1+I+I^2+\cdots, O) \cong \exp(\mathrm{list}(I), O)$. □

Chapter 4

Computation and Categorical Data Types

In chapter 2 section 2.4, we introduced CSL expressions $\mathbf{Exp}(\Gamma, \Delta, \Psi)$ which denote polymorphic functions in CSL structures. In this chapter, we will see them as programs and see how they can be executed. One of the ways to treat specifications as programs is to regard equations as rewrite rules, but in our case, CSL statements are in general conditional equations and, therefore, it is quite difficult to treat them as rewrite rules. Furthermore general rewriting cannot be regarded as real computation unless rules are Church-Rosser, otherwise, rewriting is more close to theorem proving.

There is no other way so long as we are dealing with CSL specifications. Remember that CSL was introduced in order to give semantics to categorical data types we have investigated in chapter 3. In CDT theory, we are not dealing with arbitrary CSL specifications, but some special ones, ones which are associated with CDT declarations. Therefore, we have much more hope for executing these special CSL specifications than arbitrary ones. For example, cartesian closed categories are, as is well-known, connected to lambda calculus which is a model of computation, so we can put some evaluation mechanism into them. [Curien 86], for example, has developed such a system. A difference of our approach from his is that we do not restrict ourselves only to cartesian closed categories. One of the main aims of CDT is to study categories formed by programming languages, and we should not presume any structure in the categories without proper reasons. We can only accept the cartesian closed structure in CDT if this is necessary for putting the concept of computation to it. As we will see later in this chapter, our categories are cartesian closed (with some extra structure), and by then we should know why the cartesian closed structure is necessary.

CSL expressions in section 2.4 and the CDT declaration mechanism in chapter 3 give us the basis of Categorical Programming Language (CPL) to which we will devote this chapter. CPL tries to extract the computable part of CDT. As we have seen in subsection 3.3.8, CDT in general allows us to define very strange objects which have no concept of being computable. We are going to put restrictions to the form of CDT declarations in CPL.

Therefore, in CPL, we can declare data types by CDT (with restrictions). As we have seen in section 3.4, this will determine a CSL in which we can have CSL expressions introduced in section 2.4. CSL expressions are the programs in CPL. There is no difference between programs and data. From the categorical point of view, there is nothing inside objects, that is, there are no data inside data types. CSL expressions whose domain is the terminal object are called *CPL elements*. The execution in CPL is essentially a reduction of a CPL element to a canonical irreducible CPL element.

Following the result of this chapter, CPL will not only exist on paper, but also can be implemented. This will be presented in chapter 5.

In section 4.1 we introduce a restriction to objects we can declare in CPL and a set of reduction rules for CPL computation. In section 4.2 we see an example of computation in CPL. In section 4.3 we prove that any computation in CPL terminates (i.e. the reductions are normalizing) by Tait's computability method. In section 4.4 we show some properties about objects in CPL and, finally, section 4.5 gives another set of reduction rules for CPL computation which reduces CPL elements into intuitively more canonical elements.

4.1 Reduction Rules for Categorical Programming Language

In this section, we will present some basic definitions together with the reduction rules for CPL. Let us assume in the following discussion that we have defined categorical data types by a sequence of CDT declarations, $D_1, \ldots, D_l$, and that we have obtained the corresponding CSL, $\langle \Gamma, \Delta, \Psi, \Theta \rangle$ as in definition 3.4.3.

In ordinary programming languages, we distinguish programs and data. We feed data into a program; it processes the data and then outputs the result data. We cannot feed a program into another program and data cannot process other data or programs. However, it is true that data are a special kind of programs, very simple ones. We can write data directly into programs as initialization statements or as assignments. For example, natural numbers like 1, 132, 59, etc. are data as well as constants in programs. In some languages like LISP, there is no difference between data and programs at all. CPL is not as liberated as LISP, but both data and programs are morphisms and data are just special morphisms having a special domain object.

As we know, category theory deals with the external structure of objects rather than their inner structure so it is not proper to think about data inside objects. However, we do sometimes need something similar to elements in set theory. We say *elements* in category theory are morphisms whose domain is the terminal object.

$$1 \xrightarrow{e} A$$

We say e is an element of A. If we think in the category of sets (**Set**), '1' is the one-point set and a morphism from the one-point set to a set corresponds to an element in

the set.

$$\mathrm{Hom}_{\mathbf{Set}}(1, A) \cong A$$

Hence, the definition of elements in CPL is:

Definition 4.1.1: Given a CSL signature $\langle \Gamma, \Delta, \Psi \rangle$, a *CPL element* e is a CSL expression with no morphism variables whose domain type is the terminal object.

$$\rho \vdash e : \lambda(X_1, \ldots, X_n).1 \to K$$

(see section 2.4 for typing). In case $n = 0$ which is very often the case, we may write $\vdash e : 1 \to E$ where E is a functorial expression without variables. We also say that e is an element of E. []

In order for this definition to be sensible, we need to have the terminal object in our category. For simplicity, we assume that D_1 is the CDT declaration of the terminal object as is presented in subsection 3.3.1.

Though this definition is a natural way of defining elements in category theory, it introduces non-symmetry in CPL. Remember that CDT is meant to be symmetric: e.g. if we have an object of natural numbers, we should have its dual, an object of co-natural numbers, and so on. Since we treat '1' as a special object and the elements in CPL are defined in this way, we destroy the beauty of symmetry. We will see the consequence of this shortly.

Example 4.1.2: Assume that we have defined all the objects in section 3.3. Then

$$\begin{gathered}\mathrm{succ} \circ \mathrm{zero}, \\ \mathrm{pi2} \circ \mathrm{pair}(\mathrm{succ} \circ \mathrm{zero}, \mathrm{nil}), \\ \mathrm{eval} \circ \mathrm{prod}(\mathrm{pr}(\mathrm{curry}(\mathrm{pi2}), \mathrm{curry}(\mathrm{succ} \circ \mathrm{eval})), \mathbf{I}) \circ \mathrm{pair}(\mathrm{succ} \circ \mathrm{succ} \circ \mathrm{zero}, \mathrm{succ} \circ \mathrm{zero})\end{gathered}$$

are all CPL elements. []

As we can see from this example, we cannot regard all the elements as data. 'succ∘zero' and 'pair(succ ∘ zero, nil)' can be data, but 'pi1 ∘ pair(succ ∘ zero, nil)' cannot be. We call special data-like CPL elements *canonical CPL elements*. The definition is:

Definition 4.1.3: A *canonical CPL element* is a CPL element which wholly consists of natural transformations introduced by left objects and factorizers by right objects. Formally, a canonical CPL element, $c \in \mathbf{CE}$, is defined by

$$c ::= \mathbf{I} \mid \alpha_L \circ c \mid \psi_R(e_1, \ldots, e_n) \circ c$$

where α_L is a natural transformation of a left object and ψ_R is a factorizer of a right object. Note we often do not write $\mathbf{I}$ at the tail of canonical elements. []

From this definition, 'pair' and 'curry' can form canonical elements, but 'pi1', 'pi2' or 'eval' cannot. So for the right objects, factorizers are the means of creating canonical elements. On the other hand, 'zero', 'succ' and 'cons' can form canonical elements,

but not 'pr' or 'case'. So for the left objects, natural transformations are the means of creating canonical elements.

	Canonical	**Non-Canonical**
left object	natural transformations e.g. zero, succ, nil, cons	factorizers e.g. case, pr, prl
right object	factorzers e.g. pair, curry	natural transformations e.g. pi1, pi2, eval

Definition 4.1.3 is beautifully symmetrical as we hoped from the symmetry of CDT. In addition, if we look at the canonical elements in other programming languages, we note that 'pair' corresponds to making pairs of data, 'curry' corresponds to lambda abstraction and both create canonical things (i.e. we cannot process them any more as themselves), and also we note that 'pr' and 'case' are programming constructs corresponding to primitive recursive definitions and case statements and they can never be data. Therefore, definition 4.1.3 fits well with the usual notion of data.

Of course, this is not the only definition of canonical elements. It defines quite lazy canonical elements. It does not care what are inside factorizers. For example,

$$\text{pair}(\text{pi1} \circ \text{pair}(\text{succ} \circ \text{zero}, \text{nil}), \text{zero})$$

is a canonical element of 'prod(nat, nat)' from this definition. Some might not want to call it canonical because it is equal to another canonical element

$$\text{pair}(\text{succ} \circ \text{zero}, \text{zero})$$

which looks more *canonical*. However, this is because 'prod' is a rather special object. In general, expressions inside factorizers may not be elements, so we cannot demand them to be canonical (or we have to treat product-like objects special). One might still accept this easily since canonical elements in Martin-Löf's type theory are similar to ours, but another queer aspect of our definition is that we accept

$$\text{pair}(\text{pi2}, \text{pi1}) \circ \text{pair}(\text{succ}, \mathbf{I}) \circ \text{zero}$$

as a canonical element. It is equal to the following element

$$\text{pair}(\text{pi2} \circ \text{pair}(\text{succ}, \mathbf{I}) \circ \text{zero}, \text{pi1} \circ \text{pair}(\text{succ}, \mathbf{I}) \circ \text{zero}),$$

but this is again because of the special property of 'prod'. In general

$$\psi_R(e_1, \ldots, e_n) \circ c$$

is not equal to something in the form of $\psi_R(e'_1, \ldots, e'_n)$. For example, we cannot always find other simpler canonical elements which is equal to '$\text{fold}(f, g) \circ c$' (where 'fold' is the factorizer of 'inflist').

There are many views of computation, but in CPL computation is the reduction of an element to a canonical element equal to the given element.

$$e \Rightarrow c$$

Since the reduction is not straightforward, we want to do step-by-step reduction. Therefore, we modify the above form of reduction to the following one.

$$\langle e, c\rangle \Rightarrow c'$$

This means that a CSL expression e applied to a CPL canonical element c is reduced to a CPL canonical element c'. It is like calculating the result of applying an element c to a function e. Obviously, $e \circ c$ should be semantically equal to c'. Since a CPL element is a morphism from the terminal object and '$\mathbf{I}$' is its canonical element, if we want to reduce an arbitrary CPL element e we can ask for the following reduction.

$$\langle e, \mathbf{I}\rangle \Rightarrow c$$

In the following, we assume that the associativity of '$\circ$' (the composition of morphisms) has been taken care by some means so that in our rules we do not consider it. We also assume that there are no functors in CPL elements because we can always replace them with factorizers and natural transformations. For example,

$$\mathrm{eval} \circ \mathrm{prod}(\mathrm{curry}(\mathrm{succ}), \mathrm{nil})$$

is equivalent to

$$\mathrm{eval} \circ \mathrm{pair}(\mathrm{curry}(\mathrm{succ}) \circ \mathrm{pi1}, \mathrm{nil} \circ \mathrm{pi2}).$$

Let us now define rules for the reductions. The simplest one is for identities. Since $\mathbf{I} \circ c$ is equal to c, we should have the following rule.

$$\langle \mathbf{I}, c\rangle \Rightarrow c \qquad \text{(IDENT)}$$

Next, for the composition, we naturally have

$$\frac{\langle e_2, c\rangle \Rightarrow c'' \qquad \langle e_1, c''\rangle \Rightarrow c'}{\langle e_1 \circ e_2, c\rangle \Rightarrow c'} \qquad \text{(COMP)}$$

In case that e is a natural transformation introduced by a left object or a factorizer introduced by a right object, the rule is easy because the composition of e with a canonical element is canonical by definition 4.1.3.

$$\langle \alpha_L, c\rangle \Rightarrow \alpha_L \circ c \qquad \text{(L-NAT)}$$

$$\langle \psi_R(e_1, \ldots, e_n), c\rangle \Rightarrow \psi_R(e_1, \ldots, e_n) \circ c \qquad \text{(R-FACT)}$$

For example, $\langle \mathrm{succ}, \mathrm{zero}\rangle \Rightarrow \mathrm{succ} \circ \mathrm{zero}$ and

$$\langle \mathrm{curry}(\mathrm{pi2}), \mathrm{pair}(\mathrm{zero}, \mathrm{succ} \circ \mathrm{zero})\rangle \Rightarrow \mathrm{curry}(\mathrm{pi2}) \circ \mathrm{pair}(\mathrm{zero}, \mathrm{succ} \circ \mathrm{zero}).$$

Difficulty comes in for the other cases, i.e. when e is a natural transformation of a right object or a factorizer of a left object. Let us first consider the case for a factorizer ψ_L introduced by the following left object.

left object $L(X)$ with ψ_L is
    $\alpha_L : E(L, X) \to E'(L, X)$
end object

From the property of this object, we have a CSL equation

$$E'(\psi_L(e), \mathbf{I}) \circ \alpha_L = e \circ E(\psi_L(e), \mathbf{I}).$$

An instance of this equation is '$\mathrm{pr}(f,g) \circ \mathrm{succ} = g \circ \mathrm{pr}(f,g)$', and in this case we should have a rewrite rule from '$\mathrm{pr}(f,g) \circ \mathrm{succ}$' to '$g \circ \mathrm{pr}(f,g)$'. Therefore, in general we might have the following rule.

$$\frac{\langle e \circ E(\psi_L(e), \mathbf{I}), c\rangle \Rightarrow c'}{\langle E'(\psi_L(e), \mathbf{I}), \alpha_L \circ c\rangle \Rightarrow c'}$$

However, this rule is not what we wanted. We wanted a rule for $\langle \psi_L(e), c''\rangle \Rightarrow c'''$. In order that the above rule to be a one we want, $E'(L,X)$ should be simply L, and we get

$$\frac{\langle e \circ E(\psi_L(e), \mathbf{I}), c\rangle \Rightarrow c'}{\langle \psi_L(e), \alpha_L \circ c\rangle \Rightarrow c'}. \qquad \text{(L-FACT)}$$

The restriction that demands $E'(L,X)$ should be L is the first restriction we put onto objects in order to obtain the CPL computation rules. The left objects introduced in chapter 3, initial object, coproducts, natural number object and lists, all satisfy this restriction except the left adjoint functor of 'list' which we do not expect to be in the world of computation. If $E'(L,X)$ is something other than L, we are allowing to have the left adjoint L' of $E'(\,\cdot\,, X)$ by

left object $L'(X,Y)$ with $\psi_{L'}$ is
$\quad \alpha_{L'}\colon Y \to E'(LL', X)$
end object.

In familiar categories (e.g. the category of sets), a functor $F\colon \mathcal{C} \to \mathcal{C}$ hardly has a left adjoint: e.g. the product functor $\bullet \times A$ does not have one, nor does the coproduct functor $\bullet + A$.

As an example of L-FACT, let us write the rules for the factorizer associated with the natural number object. There are two rules for two natural transformations, 'zero' and 'succ'.

$$\frac{\langle e, c\rangle \Rightarrow c'}{\langle \mathrm{pr}(e,e'), \mathrm{zero} \circ c\rangle \Rightarrow c'} \qquad \frac{\langle e' \circ \mathrm{pr}(e,e'), c\rangle \Rightarrow c'}{\langle \mathrm{pr}(e,e'), \mathrm{succ} \circ c\rangle \Rightarrow c'}$$

Let us now consider the last case, the case for the reduction rule of a natural transformation α_R introduced by the following right object declaration.

right object $R(X)$ with ψ_R is
$\quad \alpha_R\colon E(R,X) \to E'(R,X)$
end object

From the property of this object, we have a CSL equation

$$\alpha_R \circ E(\psi_R(e), \mathbf{I}) = E'(\psi_R(e), \mathbf{I}) \circ e.$$

An instance of this equation is 'pi1 $\circ$ pair$(f,g) = f$', and in this case we should have a rule to rewrite 'pi1 $\circ$ pair(f,g)' to 'f'. Therefore, in general we have a rule rewriting from the left-hand side to the right-hand side as the following rule.

$$\frac{\langle E'(\psi_R(e), \mathbf{I}) \circ e, c\rangle \Rightarrow c'}{\langle \alpha_R, E(\psi_R(e), \mathbf{I}) \circ c\rangle \Rightarrow c'}$$

However, this rule is not quite right because $E(\psi_R(e), \mathbf{I}) \circ c$ is not a canonical element. We cannot have functors in canonical elements. Therefore, what the rule should really look like is

$$\frac{c = E(\psi_R(e), \mathbf{I}) \circ c'' \qquad \langle E'(\psi_R(e), \mathbf{I}) \circ e, c''\rangle \Rightarrow c'}{\langle \alpha_R, c\rangle \Rightarrow c'}. \tag{+}$$

We now have a different problem of finding out an expression e and a canonical element c'' from a given canonical element c such that

$$c = E(\psi_R(e), \mathbf{I}) \circ c''.$$

Since we are dealing with computation, we need a mechanical way of solving this problem. Let us introduce another form of reduction rules.

$$\langle c, E, R\rangle \rightsquigarrow \langle \psi_R(e), c''\rangle \tag{$*$}$$

such that $c = E(\psi_R(e), \mathbf{I}) \circ c''$ where E is a functorial expression in which R is a variable. By these rules, we can rewrite the rule (+) to

$$\frac{\langle c, E(R, X), R\rangle \rightsquigarrow \langle \psi_R(e), c''\rangle \qquad \langle E'(\psi_R(e), \mathbf{I}) \circ e, c''\rangle \Rightarrow c'}{\langle \alpha_R, c\rangle \Rightarrow c'}. \tag{R-NAT}$$

We now have to list the rules for ($*$). If E is simply R itself, c should be $E(\psi_R(e), \mathbf{I}) \circ c'' = \psi_R(e) \circ c''$. Therefore, the rule should be

$$\langle \psi_R(e) \circ c'', R, R\rangle \rightsquigarrow \langle \psi_R(e), c''\rangle. \tag{R-NAT-V}$$

Next, if $E(R, X)$ does not depend on R, c is $E(\psi_R(e), \mathbf{I}) \circ c'' = c''$, so the rule may be

$$\langle c, E, R\rangle \rightsquigarrow \langle \psi_R(e), c\rangle,$$

but where does $\psi_R(e)$ come from? We cannot determine e. Therefore, $E(R, X)$ must not be independent from R (i.e. $E(R, X)$ should not be free of R).

The case left is when $E(R, X)$ is not a variable but a real functorial expression. Let it be

$$F(\hat{E}_1(R, X), \ldots, \hat{E}_n(R, X)), \tag{1}$$

where F is a functor name. In this case, the equation we are solving is

$$c = F(\hat{E}_1(\psi_R(e), \mathbf{I}), \ldots, \hat{E}_n(\psi_R(e), \mathbf{I})) \circ c''. \tag{2}$$

Since functors are always represented by their associated factorizers, the equation looks like

$$c = \psi_F(\cdots \circ \hat{E}_1(\psi_R(e), \mathbf{I}) \circ \cdots \circ \hat{E}_n(\psi_R(e), \mathbf{I}) \circ \cdots) \circ c''. \tag{3}$$

As we can see, we might have to pick up the same $\psi_R(e)$ from more than one place and this would be a trouble. Because $\psi_R(e)$ is a general CSL expression and we cannot do theorem proving to show two CSL expressions are equal when doing the CPL computation. Therefore, we need to restrict that $\psi_R(e)$ should appear only once in (3). In order for this, we have to first restrict that only one $\hat{E}_i$ in (2) contains R. Without loss of generality, we can assume it is $\hat{E}_1(R, X)$, and since we are only interested in the first argument of F, we assume that F is a unary functor and $E(R, X)$ is $F(\hat{E}(R, X))$. Now (2) looks like

$$c = F(\hat{E}(\psi_R(e), \mathbf{I})) \circ c'', \tag{2'}$$

but still (3) might have more than one $\psi_R(e)$ because when we expand functors by factorizers using (LFEQ) or (RFEQ) in definition 3.4.1, we might duplicate $\psi_R(e)$. Before stating the restriction to guarantee the no-duplication of $\psi_R(e)$, let us note that F should be a functor introduced by a right object declaration. This is because, if we look at the equation (3), c is a canonical element consisting of left natural transformations and right factorizers, so ψ_F should be a right factorizer. Let the following be the declaration of F.

$$\begin{array}{l} \text{right object } F(Y) \text{ with } \psi_F \text{ is} \\ \quad \beta_1 \colon \tilde{E}_1(F, Y) \to \tilde{E}'_1(F, Y) \\ \quad \cdots \\ \quad \beta_m \colon \tilde{E}_m(F, Y) \to \tilde{E}'_m(F, Y) \\ \text{end object} \end{array} \tag{4}$$

From the equation (RFEQ) in definition 3.4.1, (3) really is

$$\begin{array}{l} c = \psi_F(\ \tilde{E}'_1(\mathbf{I}, \hat{E}(\psi_R(e), \mathbf{I})) \circ \beta_1 \circ \tilde{E}_1(\mathbf{I}, \hat{E}(\psi_R(e), \mathbf{I})), \\ \qquad \cdots \\ \quad \tilde{E}'_m(\mathbf{I}, \hat{E}(\psi_R(e), \mathbf{I})) \circ \beta_m \circ \tilde{E}_m(\mathbf{I}, \hat{E}(\psi_R(e), \mathbf{I}))\ ) \circ c'' \end{array} \tag{3'}$$

In order that $\psi_R(e)$ should appear only once in this equation, Y should appear only once in one of $\tilde{E}_i$ and $\tilde{E}'_i$. In order to show that Y should not be in one of $\tilde{E}'_i$, let F be simply

$$\begin{array}{l} \text{right object } F(Y) \text{ with } \psi_F \text{ is} \\ \quad \beta \colon Y \to F \\ \text{end object.} \end{array} \tag{4'}$$

(3) becomes

$$c = \psi_F(\beta \circ \hat{E}(\psi_R(e), \mathbf{I})) \circ c''.$$

We may demand that c should be $\psi_F(\hat{e}) \circ \hat{c}$ and find $\psi_R(e)$ such that

$$\hat{e} = \beta \circ \hat{E}(\psi_R(e), \mathbf{I}),$$

but how can we solve this equation? In general, we need theorem proving for this. We should have reduced the problem recursively into

$$\langle \check{c}, \check{E}, R \rangle \rightsquigarrow \langle \psi_R(e), \check{c} \rangle,$$

but there is no way to do this if Y appears in one of $\tilde{E}_i$, because $\hat{e}$ should never be an element. The typing rule in definition 3.2.2 gives us

$$\frac{\hat{e}: Y \to Z}{\psi_F(\hat{e}): Z \to F(Y)}$$

and we cannot choose Y to be the terminal object. If Y and F were the other way round in $(4')$, we could choose Z to be the terminal object to make $\hat{e}$ an element. Therefore, if Y is in one of $\tilde{E}'_i$ in (4), we have a possibility of reducing the problem of solving $(3')$ into a smaller problem of the same kind. Without losing generality, we can assume that Y only appears in $\tilde{E}'_1$ in (4). Now, $(3')$ is

$$c = \psi_F(\tilde{E}'_1(\mathbf{I}, \hat{E}(\psi_R(e), \mathbf{I})) \circ \beta_1, \beta_2, \ldots, \beta_m) \circ c''. \qquad (3'')$$

We demand c to be $\psi_F(\hat{e}_1, \hat{e}_2, \ldots, \hat{e}_m) \circ \hat{c}$, so $(3'')$ is further rewritten to

$$\psi_F(\hat{e}_1, \hat{e}_2, \ldots, \hat{e}_m) \circ \hat{c} = \psi_F(\tilde{E}'_1(\mathbf{I}, \hat{E}(\psi_R(e), \mathbf{I})) \circ \beta_1, \beta_2, \ldots, \beta_m) \circ c''. \qquad (3''')$$

Here, we cannot jump to the conclusion that $\hat{e}_1$ is $\tilde{E}'_1(\mathbf{I}, \hat{E}(\psi_R(e), \mathbf{I})) \circ \beta_1$, $\hat{e}_2$ is β_2, $\ldots$, and $\hat{c}$ is c'', because a part of $\hat{c}$ may well contribute to form $\tilde{E}'_1(\mathbf{I}, \hat{E}(\psi_R(e), \mathbf{I}))$. What is desirable is that we could rewrite $\psi_F(\hat{e}_1, \hat{e}_2, \ldots, \hat{e}_m) \circ \hat{c}$ to $\psi_F(\hat{e}'_1, \hat{e}'_2, \ldots, \hat{e}'_m)$ like $\mathrm{pair}(f, g) \circ h = \mathrm{pair}(f \circ h, g \circ h)$. This is possible when each $\tilde{E}'_i$ in (4) does not depend on F.

Proposition 4.1.4: Let R be a right object defined by

$$\begin{array}{l}\text{right object } R(X_1, \ldots, X_n) \text{ with } \psi_R \text{ is} \\ \quad \alpha_1: E_1(R, X_1, \ldots, X_n) \to E'_1(X_1, \ldots, X_n) \\ \quad \cdots \\ \quad \alpha_m: E_m(R, X_1, \ldots, X_n) \to E'_m(X_1, \ldots, X_n) \\ \text{end object}\end{array}$$

(note that E'_i does not depend on R), then

$$\psi_R(e_1, \ldots, e_m) \circ e' = \psi_R(e_1 \circ E_1(e', \mathbf{I}, \ldots, \mathbf{I}), \ldots, e_m \circ E_m(e', \mathbf{I}, \ldots, \mathbf{I})) \qquad \text{(REQC)}$$

Proof: From (RCEQ) in definition 3.4.1,

$$\begin{array}{c}\alpha_1 \circ E_1(\psi_R(e_1, \ldots, e_m) \circ e', \mathbf{I}, \ldots, \mathbf{I}) = f_1 \wedge \\ \cdots \\ \alpha_m \circ E_m(\psi_R(e_1, \ldots, e_m) \circ e', \mathbf{I}, \ldots, \mathbf{I}) = f_m \wedge \\ \Rightarrow \psi_R(e_1, \ldots, e_m) \circ e' = \psi_R(f_1, \ldots, f_m).\end{array}$$

Using (REQ_i) and the fact that E_i is covariant in R, we get

$$\begin{array}{r}e_1 \circ E_1(e', \mathbf{I}, \ldots, \mathbf{I}) = f_1 \wedge \ldots \wedge e_m \circ E_m(e', \mathbf{I}, \ldots, \mathbf{I}) = f_m \\ \Rightarrow \psi_R(e_1, \ldots, e_m) \circ e' = \psi_R(f_1, \ldots, f_m).\end{array}$$

Therefore,

$$\psi_R(e_1, \ldots, e_m) \circ c' = \psi_R(e_1 \circ E_1(e', \mathbf{I}, \ldots, \mathbf{I}), \ldots, e_m \circ E_m(e', \mathbf{I}, \ldots, \mathbf{I})). \; \square$$

We call this kind of right objects *unconditioned*. The name indicates that the objects are characterized without using conditional CSL equations. In fact, (REQ$_i$), (REQC), (RFEQ) and the following (REQI) characterize the unconditioned right objects.[1]

$$\psi_R(\alpha_1, \ldots, \alpha_m) = \mathbf{I} \tag{REQI}$$

Therefore, we assume that in (4) F does not appear in any of $\tilde{E}'_i$. As we have already assumed that Y appears only in $\tilde{E}'_1$, (4) now looks like

$$\begin{array}{l}\text{right object } F(Y) \text{ with } \psi_F \text{ is}\\ \quad \beta_1 \colon \tilde{E}_1(F) \to \tilde{E}'_1(Y)\\ \quad \beta_2 \colon \tilde{E}_2(F) \to \tilde{E}'_2\\ \qquad \cdots\\ \quad \beta_m \colon \tilde{E}_m(F) \to \tilde{E}'_m\\ \text{end object}\end{array}$$

($\tilde{E}'_2, \ldots, \tilde{E}'_m$ do not depend on F or Y in this case, but in general F might have parameters other than Y and they can appear in $\tilde{E}'_2, \ldots, \tilde{E}'_m$), and (3) is

$$\begin{aligned}&\psi_F(\hat{e}_1 \circ \tilde{E}_1(\hat{c}), \hat{e}_2 \circ \tilde{E}_e(\hat{c}), \ldots, \hat{e}_m \circ \tilde{E}_m(\hat{c}))\\ &\quad = \psi_F(\tilde{E}'_1(\hat{E}(\psi_R(e), \mathbf{I})) \circ \beta_1 \circ \tilde{E}_1(c''), \beta_2 \circ \tilde{E}_2(c''), \ldots, \beta_m \circ \tilde{E}_m(c'')).\end{aligned} \tag{$\hat{3}$}$$

If, furthermore, the first argument of ψ_F is an element, we can reduce the problem of solving (3) into

$$\hat{e}_1 \circ \tilde{E}_1(\hat{c}) = \tilde{E}'_1(\hat{E}(\psi_R(e), \mathbf{I})) \circ \beta_1 \circ \tilde{E}_1(c''). \tag{5}$$

Since $\hat{e}_1 \circ \tilde{E}_1(\hat{c})$ is an element, we can ask its canonical element and it becomes almost like the original object (3) except E is replaced. Let us see the typing rule for ψ_F defined in definition 3.2.2.

$$\frac{f_1 \colon \tilde{E}_1(Z) \to \tilde{E}'_1(Y) \qquad \ldots}{\psi_F(f_1, \ldots) \colon Z \to F(Y)}$$

Since ψ_F in ($\hat{3}$) is making an element, Z above should be the terminal object 1. In order that f_1 above is also an element, $\tilde{E}_1(Z)$ (or $\tilde{E}_1(1)$ because Z is 1) should also the terminal object. Because $\tilde{E}_1(Z)$ cannot be independent from Z, $\tilde{E}_1(Z)$ should be simply Z. Therefore, (5) is

$$\hat{e}_1 \circ \hat{c} = \tilde{E}'_1(\hat{E}(\psi_R(e), \mathbf{I})) \circ \beta_1 \circ c''. \tag{5$'$}$$

As $\hat{e}_1 \circ \hat{c}$ is an element, we can ask its canonical element by

$$\langle \hat{e}_1, \hat{c} \rangle \Rightarrow \hat{c}', \tag{6}$$

and we can also ask to solve

$$\hat{c}' = \tilde{E}'_1(\hat{E}(\psi_R(e), \mathbf{I})) \circ \hat{c}'' \tag{7}$$

[1] We can define unconditioned left objects as dual.

by

$$\langle \hat{c}', \tilde{E}_1'(\hat{E}(R,X)), R\rangle \rightsquigarrow \langle \psi_R(e), \hat{c}''\rangle.$$

If we set c'' to be

$$\psi_F(\hat{c}'', \hat{e}_2 \circ \hat{E}_2(\hat{c}), \ldots, \hat{e}_m \circ \hat{E}_m(\hat{c})),$$

$(\hat{3})$ is satisfied from (REQ_i) in definition 3.4.1.

$$\begin{aligned}
&\tilde{E}_1'(\hat{E}(\psi_R(e), \mathbf{I})) \circ \beta_1 \circ \tilde{E}_1(c'') \\
&= \tilde{E}_1'(\hat{E}(\psi_R(e), \mathbf{I})) \circ \beta_1 \circ \tilde{E}_1(\psi_F(\hat{c}'', \hat{e}_2 \circ \hat{E}_2(\hat{c}), \ldots, \hat{e}_m \circ \hat{E}_m(\hat{c}))) \\
&= \tilde{E}_1'(\hat{E}(\psi_R(e), \mathbf{I})) \circ \beta_1 \circ \hat{c}'' \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (\text{from } (\mathrm{REQ}_i)) \\
&= \hat{c}' \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (\text{from } (7)) \\
&= \hat{e}_1 \circ \hat{c} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (\text{from } (6))
\end{aligned}$$

$$\begin{aligned}
&\beta_i \circ \tilde{E}_i(c'') \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (i = 2, \ldots, m) \\
&= \beta_i \circ \tilde{E}_i(\psi_F(\hat{c}'', \hat{e}_2 \circ \hat{E}_2(\hat{c}), \ldots, \hat{e}_m \circ \hat{E}_m(\hat{c}))) \\
&= \hat{e}_i \circ \tilde{E}_i(\hat{c}) \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (\text{from } (\mathrm{REQ}_i))
\end{aligned}$$

Therefore, we got the following rule.

$$\frac{\langle \hat{e}_1, \hat{c}\rangle \Rightarrow \hat{c}' \qquad \langle \hat{c}', \tilde{E}_1'(\hat{E}(R,X)), R\rangle \rightsquigarrow \langle \psi_R(e), \hat{c}''\rangle}{\begin{array}{l}\langle \psi_F(\hat{e}_1, \hat{e}_2, \ldots, \hat{e}_m) \circ \hat{c}, F(\hat{E}(R,X)), R\rangle \\ \qquad \rightsquigarrow \langle \psi_R(e), \psi_F(\hat{c}'', \hat{e}_2 \circ \tilde{E}_2(\hat{c}), \ldots, \hat{e}_m \circ \hat{E}_m(\hat{c}))\rangle\end{array}} \qquad (\text{R-NAT-F})$$

As an example of the rules (R-FACT), (R-NAT), (R-NAT-V) and (R-NAT-F), let us write the rules for the exponentials. The CDT declaration of the exponentials is

$$\begin{array}{l}\text{right object } \exp(X,Y) \text{ with curry is} \\ \qquad \text{eval: } \mathrm{prod}(\exp, X) \to Y \\ \text{end object}\end{array}$$

as we have seen in subsection 3.3.3. The rule (R-FACT) is simply

$$\langle \mathrm{curry}(e), c\rangle \Rightarrow \mathrm{curry}(e) \circ c,$$

and the rule (R-NAT) is

$$\frac{\langle c, \mathrm{prod}(\exp, X), \exp\rangle \rightsquigarrow \langle \mathrm{curry}(e), c''\rangle \qquad \langle e, c''\rangle \Rightarrow c'}{\langle \mathrm{eval}, c\rangle \Rightarrow c'}.$$

The rule (R-NAT-V) is simply

$$\langle \mathrm{curry}(e) \circ c'', \exp, \exp\rangle \rightsquigarrow \langle \mathrm{curry}(e), c''\rangle,$$

and, finally, the rule (R-NAT-F) is

$$\frac{\langle \hat{e}_1, \hat{c}\rangle \Rightarrow \hat{c}' \qquad \langle \hat{c}', \exp, \exp\rangle \rightsquigarrow \langle \mathrm{curry}(e), \hat{c}''\rangle}{\langle \mathrm{pair}(\hat{e}_1, \hat{e}_2) \circ \hat{c}, \mathrm{prod}(\exp, X), \exp\rangle \rightsquigarrow \langle \mathrm{curry}(e), \mathrm{pair}(\hat{c}'', \hat{e}_2 \circ \hat{c})\rangle}.$$

We may simplify the last three rules and have the next one instead.

$$\frac{\langle e_1, c\rangle \Rightarrow \mathrm{curry}(e) \circ c'' \qquad \langle e, \mathrm{pair}(c'', e_2 \circ c)\rangle \Rightarrow c'}{\langle \mathrm{eval}, \mathrm{pair}(e_1, e_2) \circ c\rangle \Rightarrow c'}$$

We will see an example using these rules in the next section.

In order to obtain (R-NAT-F), we have put several restrictions to the right object declaration. To state the restrictions formally, let us introduce the notion '*productive*'.

Definition 4.1.5: Functorial expressions which are *productive* in X are generated by the following two rules.

1. X itself is a functorial expression productive in X.
2. If $P(Y_1, \ldots, Y_n)$ is a functor which is productive in Y_i and E is a functorial expression productive in X, $P(E_1, \ldots, E_{i-1}, E_i, E_{i+1}, \ldots, E_n)$ is productive in X, where $E_1, \ldots, E_{i-1}, E_{i+1}, \ldots, E_n$ are functorial expressions which do not contain X.

A functor $P(Y_1, \ldots, Y_n)$ is called productive in its i-th argument Y_i when P is declared as a right object and its declaration

$$\begin{array}{l}\text{right object } P(Y_1, \ldots, Y_n) \text{ with } \psi_P \text{ is}\\ \quad \alpha_{P,1}\colon E_{P,1}(P, Y_1, \ldots, Y_n) \to E'_{P,1}(P, Y_1, \ldots, Y_n)\\ \qquad\qquad \cdots\\ \quad \alpha_{P,m}\colon E_{P,m}(P, Y_1, \ldots, Y_n) \to E'_{P,m}(P, Y_1, \ldots, Y_n)\\ \text{end object}\end{array}$$

satisfies

1. P is unconditioned (i.e. P does not appear in $E'_{P,1}, \ldots, E'_{P,m}$),
2. Y_i does not appear in $E_{P,1}, \ldots, E_{P,m}$,
3. Y_i appears only one of $E'_{P,1}, \ldots, E'_{P,m}$, so let us assume that it appears in $E'_{P,j}$ only,
4. $E_{P,j}$ is simply P, and
5. $E'_{P,j}$ is a functorial expression productive in Y_i.

Therefore, the declaration above may look more like

$$\begin{array}{l}\text{right object } P(Y_1, \ldots, Y_i, \ldots, Y_n) \text{ with } \psi_P \text{ is}\\ \quad \alpha_{P,1}\colon E_{P,1}(P, Y_1, \ldots, Y_{i-1}, Y_{i+1}, \ldots, Y_n) \to\\ \qquad\qquad E'_{P,1}(Y_1, \ldots, Y_{i-1}, Y_{i+1}, \ldots, Y_n)\\ \qquad\qquad \cdots\\ \quad \alpha_{P,j}\colon P \to E'_{P,j}(Y_1, \ldots, Y_{i-1}, Y_i, Y_{i+1}, \ldots, Y_n) \qquad \text{(P)}\\ \qquad\qquad \cdots\\ \quad \alpha_{P,m}\colon E_{P,m}(P, Y_1, \ldots, Y_{i-1}, Y_{i+1}, \ldots, Y_n) \to\\ \qquad\qquad E'_{P,m}(Y_1, \ldots, Y_{i-1}, Y_{i+1}, \ldots, Y_n)\\ \text{end object}\end{array}$$

$\alpha_{P,j}$ may be called *projection* to Y_i. □

The functor 'prod' is a typical productive functor. It is productive in its both arguments. The functor 'exp' is not productive in any of its arguments. A functorial expression 'prod(prod(X,exp(Y,Z)),prod(X,U))' is productive in U but not in X or Y or Z.

Definition 4.1.6: A right object R is called *computable* if its declaration

$$\begin{array}{l}\text{right object } R(X_1,\ldots,X_n) \text{ with } \psi_R \text{ is}\\ \quad \alpha_{R,1}\colon E_{R,1}(R,X_1,\ldots,X_n) \to E'_{R,1}(R,X_1,\ldots,X_n)\\ \qquad\qquad \cdots\\ \quad \alpha_{R,m}\colon E_{R,m}(R,X_1,\ldots,X_n) \to E'_{R,m}(R,X_1,\ldots,X_n)\\ \text{end object}\end{array} \qquad \text{(RC)}$$

satisfies that $E_{R,1},\ldots,E_{R,m}$ are functorial expressions productive in R. We also call a left object L *computable* when its declaration is

$$\begin{array}{l}\text{left object } L(X_1,\ldots,X_n) \text{ with } \psi_L \text{ is}\\ \quad \alpha_{L,1}\colon E_{L,1}(L,X_1,\ldots,X_n) \to L\\ \qquad\qquad \cdots\\ \quad \alpha_{L,m}\colon E_{L,m}(L,X_1,\ldots,X_n) \to L\\ \text{end object}\end{array} \qquad \text{(LC)}$$

□

The reduction rules we have defined in this section work when all the objects we define are computable, and all the objects declared in section 3.3 are computable except the obscure ones in subsection 3.3.8. Obviously, we did not want to have those obscure objects in our datatype system and the computability constraint gets rid of them. In other words, *the categories which are defined by declaring computable objects cannot be richer than cartesian closed category with recursive objects.* Note that we did not make the restriction in the beginning. We had the ability to declare a lot of other objects, but it turned out that in order to be able to discuss the computability in CDT, the categories should be cartesian closed with recursive objects. This signifies the importance of cartesian closed categories in computer science yet again (e.g. the models of typed lambda calculus correspond to cartesian closed categories).

Note that not all the productive objects are computable by the definition 4.1.5, but from now on we only treat computable objects, so productive objects mean computable productive objects.

Definition 4.1.7: Let $D_1,\ldots,D_l$ be a sequence of CDT declarations defining only computable objects and let $\langle\Gamma,\Delta,\Psi,\Theta\rangle$ be the corresponding CSL theory defined by definition 3.4.3. Then, we can have computation rules for CPL elements over the CSL signature $\langle\Gamma,\Delta,\Psi\rangle$. The computation rules are divided into two: those in the form of

$$\langle e,c\rangle \Rightarrow c'$$

where c and c' are canonical elements and e and c can be composed (i.e. $e \circ c$ is an element), and those in the form of

$$\langle c, E, R\rangle \rightsquigarrow \langle \psi_R(e_1, \ldots, e_m), c'\rangle$$

where c and c' are canonical elements, R is a right functor name, E is a functorial expression productive in R and, if c is an element of E', E should be more general than E'.

1. IDENT
$$\langle \mathbf{I}, c\rangle \Rightarrow c$$

2. COMP
$$\frac{\langle e_2, c\rangle \Rightarrow c'' \qquad \langle e_1, c''\rangle \Rightarrow c'}{\langle e_1 \circ e_2, c\rangle \Rightarrow c'}$$

3. L-NAT
$$\langle \alpha_{L,j}, c\rangle \Rightarrow \alpha_{L,j} \circ c$$

4. R-FACT
$$\langle \psi_R(e_1, \ldots, e_m), c\rangle \Rightarrow \psi_R(e_1, \ldots, e_m) \circ c$$

5. L-FACT
$$\frac{\langle e_j \circ E_{L,j}[\psi_L(e_1, \ldots, e_m)/L], c\rangle \Rightarrow c'}{\langle \psi_L(e_1, \ldots, e_m), \alpha_{L,j} \circ c\rangle \Rightarrow c'}$$

6. R-NAT
$$\frac{\begin{array}{c}\langle c, E_{R,j}, R\rangle \rightsquigarrow \langle \psi_R(e_1, \ldots, e_m), c''\rangle \\ \langle E'_{R,j}[\psi_R(e_1, \ldots, e_m)/R] \circ e_j, c''\rangle \Rightarrow c'\end{array}}{\langle \alpha_{R,j}, c\rangle \Rightarrow c'}$$

7. R-NAT-V
$$\langle \psi_R(e_1, \ldots, e_m) \circ c'', R, R\rangle \rightsquigarrow \langle \psi_R(e_1, \ldots, e_m), c''\rangle$$

8. R-NAT-F
$$\frac{\begin{array}{c}R \in E_i \qquad Y_i \in E'_{P,j} \qquad \langle \hat{e}_j, c\rangle \Rightarrow c' \\ \langle c', E'_{P,j}[E_i/Y_i], R\rangle \rightsquigarrow \langle \psi_R(e_1, \ldots, e_{m'}), c''\rangle\end{array}}{\begin{array}{l}\langle \psi_P(\hat{e}_1, \ldots, \hat{e}_m) \circ c, P(E_1, \ldots, E_n), R\rangle \rightsquigarrow \\ \quad \langle \psi_R(e_1, \ldots, e_{m'}), \psi_P(\hat{e}_1 \circ E_{P,1}[c/P], \ldots, \\ \quad\quad \hat{e}_{j-1} \circ E_{P,j-1}[c/P], c'', \hat{e}_{j+1} \circ E_{P,j+1}[c/P], \ldots, \hat{e}_m \circ E_{P,m}[c/P])\rangle\end{array}}$$

where the objects L, R and P are defined as (LC), (RC) and (P), respectively, and $R \in E_i$ means that R appears in a functorial expression E_i. []

We have to show that the rules are well-formed, but we have to show their soundness at the same time. We will do this in section 4.3 as well as showing that every reduction terminates (in other words, every element is normalizable by these rules).

4.2 An Example of using Reduction Rules

In this section, we will see an example computation in CPL using the reduction rules defined in the previous section. Since computation by hand is very tiresome, we do only one example, but we will see some more examples of CPL computation done by a computer in chapter 5.

Let us assume that we have declared the terminal object as in subsection 3.3.1, products as in subsection 3.3.2, exponentials as in subsection 3.3.3 and natural number object as in subsection 3.3.4. We can write down instances of the reduction rules in definition 4.1.7 as follows.

$$\langle \mathbf{I}, c\rangle \Rightarrow c \qquad \text{(IDENT)}$$

$$\frac{\langle e_2, c\rangle \Rightarrow c'' \qquad \langle e_1, c''\rangle \Rightarrow c'}{\langle e_1 \circ e_2, c\rangle \Rightarrow c'} \qquad \text{(COMP)}$$

For the terminal object, we have

$$\langle !, c\rangle \Rightarrow ! \circ c. \qquad \text{(R-FACT}_!\text{)}$$

For products, we have

$$\langle \mathrm{pair}(e_1, e_2), c\rangle \Rightarrow \mathrm{pair}(e_1, e_2) \circ c, \qquad \text{(R-FACT}_{\mathrm{pair}}\text{)}$$

$$\frac{\langle c, R, R\rangle \rightsquigarrow \langle \mathrm{pair}(e_1, e_2), c''\rangle \qquad \langle e_1, c''\rangle \Rightarrow c'}{\langle \mathrm{pi1}, c\rangle \Rightarrow c'}, \qquad \text{(R-NAT}_{\mathrm{pi1}}\text{)}$$

and

$$\langle \mathrm{pair}(e_1, e_2) \circ c, \mathrm{prod}, \mathrm{prod}\rangle \rightsquigarrow \langle \mathrm{pair}(e_1, e_2), c\rangle. \qquad \text{(R-NAT-V}_{\mathrm{prod}}\text{)}$$

If we combine (R-NAT$_{\mathrm{pi1}}$) and (R-NAT-V$_{\mathrm{prod}}$) together, we get a familiar rule

$$\frac{\langle e_1, c\rangle \Rightarrow c'}{\langle \mathrm{pi1}, \mathrm{pair}(e_1, e_2) \circ c\rangle \Rightarrow c'}. \qquad \text{(R-NAT}'_{\mathrm{pi1}}\text{)}$$

Similarly, for 'pi2', we have

$$\frac{\langle e_2, c\rangle \Rightarrow c'}{\langle \mathrm{pi2}, \mathrm{pair}(e_1, e_2) \circ c\rangle \Rightarrow c'}. \qquad \text{(R-NAT}'_{\mathrm{pi2}}\text{)}$$

For exponentials, as we have seen in the previous section, we have the following two rules.

$$\langle \mathrm{curry}(e), c\rangle \Rightarrow \mathrm{curry}(e) \circ c \qquad \text{(R-FACT}_{\mathrm{curry}}\text{)}$$

$$\frac{\langle e_1, c\rangle \Rightarrow \mathrm{curry}(e) \circ c'' \qquad \langle e, \mathrm{pair}(c'', e_2 \circ c)\rangle \Rightarrow c'}{\langle \mathrm{eval}, \mathrm{pair}(e_1, e_2) \circ c\rangle \Rightarrow c'} \qquad \text{(R-NAT}'_{\mathrm{eval}}\text{)}$$

For natural number object, we have

$$\langle \mathrm{zero}, c\rangle \Rightarrow \mathrm{zero} \circ c, \qquad \text{(L-NAT}_{\mathrm{zero}}\text{)}$$

$$\langle \mathrm{succ}, c\rangle \Rightarrow \mathrm{succ} \circ c, \qquad \text{(L-NAT}_{\mathrm{succ}}\text{)}$$

$$\frac{\langle e_1, c\rangle \Rightarrow c'}{\langle \mathrm{pr}(e_1, e_2), \mathrm{zero} \circ c\rangle \Rightarrow c'}, \qquad (\text{L-FACT}_{\mathrm{zero}})$$

and

$$\frac{\langle e_2 \circ \mathrm{pr}(e_1, e_2), c\rangle \Rightarrow c'}{\langle \mathrm{pr}(e_1, e_2), \mathrm{succ} \circ c\rangle \Rightarrow c'}. \qquad (\text{L-FACT}_{\mathrm{succ}})$$

Now let us try to calculate '$1 + 1$' in CPL. The addition function is defined in subsection 3.3.4 as

$$\mathrm{add} = \mathrm{eval} \circ \mathrm{prod}(\mathrm{pr}(\mathrm{curry}(\mathrm{pi2}), \mathrm{curry}(\mathrm{succ} \circ \mathrm{eval})), \mathbf{I}).$$

If we expand 'prod' by 'pair', we get

$$\mathrm{add} = \mathrm{eval} \circ \mathrm{pair}(\mathrm{pr}(\mathrm{curry}(\mathrm{pi2}), \mathrm{curry}(\mathrm{succ} \circ \mathrm{eval})) \circ \mathrm{pi1}, \mathrm{pi2}).$$

Therefore, the calculation '$1 + 1$' corresponds to the following reduction.

$$\langle \mathrm{eval} \circ \mathrm{pair}(\mathrm{pr}(\mathrm{curry}(\mathrm{pi2}), \mathrm{curry}(\mathrm{succ} \circ \mathrm{eval})) \circ \mathrm{pi1}, \mathrm{pi2}), \mathrm{pair}(\mathrm{succ} \circ \mathrm{zero}, \mathrm{succ} \circ \mathrm{zero})\rangle \Rightarrow c \qquad (1)$$

From (COMP),

$$\frac{\begin{array}{c}\langle \mathrm{pair}(\mathrm{pr}(\mathrm{curry}(\mathrm{pi2}), \mathrm{curry}(\mathrm{succ} \circ \mathrm{eval})) \circ \mathrm{pi1}, \mathrm{pi2}), \mathrm{pair}(\mathrm{succ} \circ \mathrm{zero}, \mathrm{succ} \circ \mathrm{zero})\rangle \Rightarrow c_1 \\ \langle \mathrm{eval}, c_1\rangle \Rightarrow c\end{array}}{\langle \mathrm{eval} \circ \mathrm{pair}(\mathrm{pr}(\mathrm{curry}(\mathrm{pi2}), \mathrm{curry}(\mathrm{succ} \circ \mathrm{eval})) \circ \mathrm{pi1}, \mathrm{pi2}), \mathrm{pair}(\mathrm{succ} \circ \mathrm{zero}, \mathrm{succ} \circ \mathrm{zero})\rangle \Rightarrow c}$$

From (R-FACT$_{\mathrm{pair}}$), c_1 is

$$\mathrm{pair}(\mathrm{pr}(\mathrm{curry}(\mathrm{pi2}, \mathrm{curry}(\mathrm{succ} \circ \mathrm{eval}) \circ \mathrm{pi1}, \mathrm{pi2}) \circ \mathrm{pair}(\mathrm{succ} \circ \mathrm{zero}, \mathrm{succ} \circ \mathrm{zero}),$$

so we need to calculate

$$\langle \mathrm{eval}, \mathrm{pair}(\mathrm{pr}(\mathrm{curry}(\mathrm{pi2}, \mathrm{curry}(\mathrm{succ} \circ \mathrm{eval}) \circ \mathrm{pi1}, \mathrm{pi2}) \circ \mathrm{pair}(\mathrm{succ} \circ \mathrm{zero}, \mathrm{succ} \circ \mathrm{zero})\rangle \Rightarrow c.$$

From (R-NAT$'_{\mathrm{eval}}$),

$$\frac{\begin{array}{c}\langle \mathrm{pr}(\mathrm{curry}(\mathrm{pi2}), \mathrm{curry}(\mathrm{succ} \circ \mathrm{eval})) \circ \mathrm{pi1}, \mathrm{pair}(\mathrm{succ} \circ \mathrm{zero}, \mathrm{succ} \circ \mathrm{zero})\rangle \Rightarrow \mathrm{curry}(e_1) \circ c_2 \\ \langle e_1, \mathrm{pair}(c_2, \mathrm{pi2} \circ \mathrm{pair}(\mathrm{succ} \circ \mathrm{zero}, \mathrm{succ} \circ \mathrm{zero}))\rangle \Rightarrow c\end{array}}{\langle \mathrm{eval}, \mathrm{pair}(\mathrm{pr}(\mathrm{curry}(\mathrm{pi2}, \mathrm{curry}(\mathrm{succ} \circ \mathrm{eval}) \circ \mathrm{pi1}, \mathrm{pi2}) \circ \mathrm{pair}(\mathrm{succ} \circ \mathrm{zero}, \mathrm{succ} \circ \mathrm{zero})\rangle \Rightarrow c.} \qquad (2)$$

From (COMP),

$$\frac{\begin{array}{c}\langle \mathrm{pi1}, \mathrm{pair}(\mathrm{succ} \circ \mathrm{zero}, \mathrm{succ} \circ \mathrm{zero})\rangle \Rightarrow c_3 \\ \langle \mathrm{pr}(\mathrm{curry}(\mathrm{pi2}), \mathrm{curry}(\mathrm{succ} \circ \mathrm{eval})), c_3\rangle \Rightarrow \mathrm{curry}(e_1) \circ c_2\end{array}}{\langle \mathrm{pr}(\mathrm{curry}(\mathrm{pi2}), \mathrm{curry}(\mathrm{succ} \circ \mathrm{eval})) \circ \mathrm{pi1}, \mathrm{pair}(\mathrm{succ} \circ \mathrm{zero}, \mathrm{succ} \circ \mathrm{zero})\rangle \Rightarrow \mathrm{curry}(e_1) \circ c_2} \qquad (3)$$

From (R-NAT$'_{\text{pi1}}$),

$$\frac{\langle \text{succ} \circ \text{zero}, ! \rangle \Rightarrow c_3}{\langle \text{pi1}, \text{pair}(\text{succ} \circ \text{zero}, \text{succ} \circ \text{zero}) \rangle \Rightarrow c_3},$$

and from (L-NAT$_{\text{zero}}$) and (L-NAT$_{\text{succ}}$), c_3 is

$$\text{succ} \circ \text{zero} \circ !.$$

Going back to (3), we need to calculate

$$\langle \text{pr}(\text{curry}(\text{pi2}), \text{curry}(\text{succ} \circ \text{eval})), \text{succ} \circ \text{zero} \circ ! \rangle \Rightarrow \text{curry}(e_1) \circ c_2.$$

From (L-FACT$_{\text{succ}}$),

$$\frac{\langle \text{curry}(\text{succ} \circ \text{eval}) \circ \text{pr}(\text{curry}(\text{pi2}), \text{curry}(\text{succ} \circ \text{eval})), \text{zero} \circ ! \rangle \Rightarrow \text{curry}(e_1) \circ c_2}{\langle \text{pr}(\text{curry}(\text{pi2}), \text{curry}(\text{succ} \circ \text{eval})), \text{succ} \circ \text{zero} \circ ! \rangle \Rightarrow \text{curry}(e_1) \circ c_2}$$

and from (COMP)

$$\frac{\langle \text{pr}(\text{curry}(\text{pi2}), \text{curry}(\text{succ} \circ \text{eval})), \text{zero} \circ ! \rangle \Rightarrow c_4 \quad \langle \text{curry}(\text{succ} \circ \text{eval}), c_4 \rangle \Rightarrow \text{curry}(e_2) \circ c_2}{\langle \text{curry}(\text{succ} \circ \text{eval}) \circ \text{pr}(\text{curry}(\text{pi2}), \text{curry}(\text{succ} \circ \text{eval})), \text{zero} \circ ! \rangle \Rightarrow \text{curry}(e_1) \circ c_2} \tag{4}$$

From (L-FACT$_{\text{zero}}$),

$$\frac{\langle \text{curry}(\text{pi2}), ! \rangle \Rightarrow c_4}{\langle \text{pr}(\text{curry}(\text{pi2}), \text{curry}(\text{succ} \circ \text{eval})), \text{zero} \circ ! \rangle \Rightarrow c_4}$$

and from (R-FACT$_{\text{curry}}$), c_4 is 'curry(pi2) $\circ$!'. Going back to (4), e_1 is 'succ $\circ$ eval' and c_2 is curry(pi2) $\circ$!. Therefore, going back to (2), we now have to calculate

$$\langle \text{succ} \circ \text{eval}, \text{pair}(\text{curry}(\text{pi2}) \circ !, \text{pi2} \circ \text{pair}(\text{succ} \circ \text{zero}, \text{succ} \circ \text{zero})) \rangle \Rightarrow c.$$

From (COMP),

$$\frac{\langle \text{eval}, \text{pair}(\text{curry}(\text{pi2}) \circ !, \text{pi2} \circ \text{pair}(\text{succ} \circ \text{zero}, \text{succ} \circ \text{zero})) \rangle \Rightarrow c_5 \quad \langle \text{succ}, c_5 \rangle \Rightarrow c}{\langle \text{succ} \circ \text{eval}, \text{pair}(\text{curry}(\text{pi2}) \circ !, \text{pi2} \circ \text{pair}(\text{succ} \circ \text{zero}, \text{succ} \circ \text{zero})) \rangle \Rightarrow c} \tag{5}$$

From (R-NAT$'_{\text{eval}}$),

$$\frac{\langle \text{curry}(\text{pi2}) \circ !, ! \rangle \Rightarrow \text{curry}(e_2) \circ c_6 \quad \langle e_2, \text{pair}(c_6, \text{pi2} \circ \text{pair}(\text{succ} \circ \text{zero}, \text{succ} \circ \text{zero})) \rangle \Rightarrow c_5}{\langle \text{eval}, \text{pair}(\text{curry}(\text{pi2}) \circ !, \text{pi2} \circ \text{pair}(\text{succ} \circ \text{zero}, \text{succ} \circ \text{zero})) \rangle \Rightarrow c_5}$$

From (R-FACT$_!$) and (R-FACT$_{\text{curry}}$), e_2 is 'pi2' and c_6 is '$! \circ !$', so we have

$$\langle \text{pi2}, \text{pair}(! \circ !, \text{pi2} \circ \text{pair}(\text{succ} \circ \text{zero}, \text{succ} \circ \text{zero}))\rangle \Rightarrow c_5$$

From (R-NAT$'_{\text{pi2}}$),

$$\frac{\langle \text{pi2} \circ \text{pair}(\text{succ} \circ \text{zero}, \text{succ} \circ \text{zero}), !\rangle \Rightarrow c_5}{\langle \text{pi2}, \text{pair}(! \circ !, \text{pi2} \circ \text{pair}(\text{succ} \circ \text{zero}, \text{succ} \circ \text{zero}))\rangle \Rightarrow c_5}$$

From (COMP),

$$\frac{\begin{array}{c}\langle \text{pair}(\text{succ} \circ \text{zero}, \text{succ} \circ \text{zero}), !\rangle \Rightarrow c_6 \\ \langle \text{pi2}, c_6\rangle \Rightarrow c_5\end{array}}{\langle \text{pi2} \circ \text{pair}(\text{succ} \circ \text{zero}, \text{succ} \circ \text{zero}), !\rangle \Rightarrow c_5}$$

From (R-FACT$_{\text{pair}}$), c_6 is

$$\text{pair}(\text{succ} \circ \text{zero}, \text{succ} \circ \text{zero}) \circ !,$$

so we have

$$\langle \text{pi2}, \text{pair}(\text{succ} \circ \text{zero}, \text{succ} \circ \text{zero}) \circ !\rangle \Rightarrow c_5$$

and (R-NAT$'_{\text{pi2}}$)

$$\frac{\langle \text{succ} \circ \text{zero}, !\rangle \Rightarrow c_5}{\langle \text{pi2}, \text{pair}(\text{succ} \circ \text{zero}, \text{succ} \circ \text{zero}) \circ !\rangle \Rightarrow c_5}$$

By using (L-NAT$_{\text{zero}}$) and (L-NAT$_{\text{succ}}$), c_5 is

$$\text{succ} \circ \text{zero} \circ !.$$

Now, we go back to (5) and we need to calculate

$$\langle \text{succ}, \text{succ} \circ \text{zero} \circ !\rangle \Rightarrow c,$$

but this is straightforward from (L-NAT$_{\text{succ}}$) and c is

$$\text{succ} \circ \text{succ} \circ \text{zero} \circ !.$$

Therefore, the reduction (1) is

$$\begin{array}{l}\langle \text{eval} \circ \text{pair}(\text{pr}(\text{curry}(\text{pi2}), \text{curry}(\text{succ} \circ \text{eval})) \circ \text{pi1}, \text{pi2}), \\ \qquad \text{pair}(\text{succ} \circ \text{zero}, \text{succ} \circ \text{zero})\rangle \Rightarrow \text{succ} \circ \text{succ} \circ \text{zero} \circ !\end{array}$$

that is, we have shown '$1 + 1$' is '2' in CPL.

4.3 Well-Definedness and Normalization Theorem for Reduction Rules

In section 4.1, we discussed what is computation in CPL and obtained a set of reduction rules (definition 4.1.7). Usual questions to be asked when we get reduction rules are, firstly, whether they are well-defined or not and, secondly, whether they are normalizing or not. In this section, we will answer both questions affirmatively.

Let us assume in this section that we are working in a CSL theory $\langle \Gamma, \Delta, \Psi, \Theta \rangle$ which is obtained by a sequence of CDT declarations, $D_1, \ldots, D_l$, each of which is a computable object declaration.

First, we prove the well-definedness of the reduction rules.

Theorem 4.3.1: Well-Definedness Theorem: Let $e \circ c$ be an element in $\langle \Gamma, \Delta, \Psi \rangle$ and c be a canonical element. If from the rules listed in definition 4.1.7 we deduce

$$\langle e, c \rangle \Rightarrow c',$$

then c' is a canonical element and $e \circ c = c'$ holds in $\langle \Gamma, \Delta, \Psi, \Theta \rangle$ (or in any CSL theory structure of this).
Proof: We prove at the same time that for a canonical element c, a right functor R, a functorial expression E productive in R which is more general than the type of c, if the rules in definition 4.1.7 deduce

$$\langle c, E, R \rangle \rightsquigarrow \langle e, c' \rangle,$$

then e is $\psi_R(e_1, \ldots, e_m)$ for some $e_1, \ldots, e_n$, c' is a canonical element and $c = E[\psi(e_1, \ldots, e_m)/R] \circ c'$ holds in $\langle \Gamma, \Delta, \Psi, \Theta \rangle$.

The proof is done by mathematical induction on the length of reduction, so all we have to do is to check each reduction rule.

1. IDENT

   $$\langle \mathbf{I}, c \rangle \Rightarrow c$$

   It is trivial from $\mathbf{I} \circ c = c$.

2. COMP

   $$\frac{\langle e_2, c \rangle \Rightarrow c'' \qquad \langle e_1, c'' \rangle \Rightarrow c'}{\langle e_1 \circ e_2, c \rangle \Rightarrow c'}$$

   From the induction hypothesis c' is canonical and $c' = e_1 \circ c'' = e_1 \circ e_2 \circ c$.

3. L-NAT

   $$\langle \alpha_{L,j}, c \rangle \Rightarrow \alpha_{L,j} \circ c$$

   It is trivial since $\alpha_{L,j} \circ c$ is a canonical element.

4. R-FACT

   $$\langle \psi_R(e_1, \ldots, e_m), c \rangle \Rightarrow \psi_R(e_1, \ldots, e_m) \circ c$$

   It is again trivial since $\psi_R(e_1, \ldots, e_m) \circ c$ is a canonical element.

5. L-FACT

$$\frac{\langle e_j \circ E_{L,j}[\psi_L(e_1,\ldots,e_m)/L], c\rangle \Rightarrow c'}{\langle \psi_L(e_1,\ldots,e_m), \alpha_{L,j} \circ c\rangle \Rightarrow c'}$$

If $\alpha_{L,j} \circ c$ is canonical, c is canonical, and from (LEQ_j)

$$e_j \circ E_{L,j}[\psi_L(e_1,\ldots,e_m)/L] \circ c = \psi_L(e_1,\ldots,e_m) \circ \alpha_{L,j} \circ c.$$

Therefore, the premise of the rule is well-formed, so from the induction hypothesis c' is canonical, and

$$c' = e_j \circ E_{L,j}[\psi_L(e_1,\ldots,e_m)/L] \circ c = \psi_L(e_1,\ldots,e_m) \circ \alpha_{L,j} \circ c.$$

6. R-NAT

$$\frac{\begin{array}{c}\langle c, E_{R,j}, R\rangle \rightsquigarrow \langle \psi_R(e_1,\ldots,e_m), c''\rangle \\ \langle E'_{R,j}[\psi_R(e_1,\ldots,e_m)/R] \circ e_j, c''\rangle \Rightarrow c'\end{array}}{\langle \alpha_{R,j}, c\rangle \Rightarrow c'}$$

Since $\alpha_{R,j}$ can be composed with c and $\alpha_{R,j}$ has the type $E_{R,j} \to E'_{R,j}$, c is an element of a functorial expression not more general than $E_{R,j}$. From the induction hypothesis, c'' is a canonical element and

$$E_{R,j}[\psi_R(e_1,\ldots,e_m)/R] \circ c'' = c.$$

Therefore,

$$\begin{aligned}
&\alpha_{R,j} \circ c \\
&= \alpha_{R,j} \circ E_{R,j}[\psi_R(e_1,\ldots,e_m)/R] \circ c'' \quad \ldots\ldots\ldots\ldots\ldots\ldots \text{(from } (\mathrm{REQ}_j)) \\
&= E'_{R,j}[\psi_R(e_1,\ldots,e_m)/R] \circ e_j \circ c'' \quad \ldots\ldots\ldots\ldots\ldots\ldots \text{(from hypothesis)} \\
&= c. \quad \ldots\ldots\ldots\ldots\ldots\ldots \text{(from hypothesis)}
\end{aligned}$$

7. R-NAT-V

$$\langle \psi_R(e_1,\ldots,e_m) \circ c'', R, R\rangle \rightsquigarrow \langle \psi_R(e_1,\ldots,e_m), c''\rangle$$

R is a variable, so it is more general than anything, so the rule is well-formed. Since $\psi_R(e_1,\ldots,e_m) \circ c''$ is a canonical element, so is c''.

8. R-NAT-F

$$\frac{\begin{array}{c}R \in E_i \qquad Y_i \in E'_{P,j} \qquad \langle \hat{e}_j, c\rangle \Rightarrow c' \\ \langle c', E'_{P,j}[E_i/Y_i], R\rangle \rightsquigarrow \langle \psi_R(e_1,\ldots,e_{m'}), c''\rangle\end{array}}{\begin{array}{l}\langle \psi_P(\hat{e}_1,\ldots,\hat{e}_m) \circ c, P(E_1,\ldots,E_n), R\rangle \rightsquigarrow \\ \quad \langle \psi_R(e_1,\ldots,e_{m'}), \psi_P(\hat{e}_1 \circ E_{P,1}[c/P],\ldots, \\ \quad\quad \hat{e}_{j-1} \circ E_{P,j-1}[c/P], c'', \hat{e}_{j+1} \circ E_{P,j+1}[c/P],\ldots,\hat{e}_m \circ E_{P,m}[c/P])\rangle\end{array}}$$

$\psi_P(\hat{e}_1,\ldots) \circ c = \psi_P(\hat{e}_1 \circ E_{P,1}[c/P],\ldots)$ and from the typing rule of ψ_P, $\hat{e}_j \circ E_{P,j}[c/P] = \hat{e}_j \circ c$ is an element of a functorial expression which is not more general

than $E'_{P,j}[E_1/Y_1,\ldots]$ which is not more general than $E'_{P,j}[E_i/Y_i]$. Therefore, the premises of the rule are well-formed and

$$\begin{aligned}
&\hat{e}_j \circ E_{P,j}[c/P] \\
&= \hat{e}_j \circ c \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (E_{P,j} \text{ is simply } P) \\
&= c' \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (\text{from } \langle \hat{e}_j, c\rangle \Rightarrow c') \\
&= E'_{P,j}[E_i/Y_i][\psi_R(e_1,\ldots)/R] \circ c''. \quad \ldots\ldots\ldots\ldots (\text{from } \langle c', E'_{P,j}[E_i/Y_i], R\rangle \rightsquigarrow \ldots)
\end{aligned}$$

Therefore,

$$\begin{aligned}
&P(E_1,\ldots)[\psi_R(e_1,\ldots)/R] \circ \psi_P(\hat{e}_1 \circ E_{P,1}[c/P],\ldots,c'',\ldots) \\
&= \psi_P(\alpha_{P,1},\ldots,E'_{P,j}[E_i/Y_i][\psi_R(e_1,\ldots)/R],\ldots) \\
&\qquad\qquad \circ \psi_P(\hat{e}_1 \circ E_{P,1}[c/P],\ldots,c'',\ldots) \quad \ldots\ldots\ldots\ldots (\text{expand } P) \\
&= \psi_P(\hat{e}_1 \circ E_{P,1}[c/P],\ldots,E'_{P,j}[E_i/Y_i][\psi_R(e_1,\ldots)/R] \circ c'',\ldots) \\
&\qquad\qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (\text{from 4.1.4 and } (\mathrm{REQ}_k)) \\
&= \psi_P(\hat{e}_1 \circ E_{P,1}[c/P],\ldots,\hat{e}_j \circ E_{P,j}[c/P],\ldots) \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (\text{from above}) \\
&= \psi_P(\hat{e}_1,\ldots,\hat{e}_m) \circ c. \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (\text{from 4.1.4})
\end{aligned}$$

Of course, $\psi_P(\hat{e}_1 \circ E_{P,1}[c/P],\ldots,c'',\ldots)$ is a canonical element.

We have proved the theorem as well as shown the well-formedness of the reduction rules. []

Next, we prove the normalization, that is to prove the following theorem.

Theorem 4.3.2: Normalization Theorem: For a canonical element c and a CSL expression e whose domain is compatible with the codomain of c (i.e. we can have $e \circ c$ as an element), there is a canonical element c' such that

$$\langle e, c\rangle \Rightarrow c'$$

by the rules listed in definition 4.1.7. []

Before proving this theorem, we need some preparation. Note that the theorem proves the two things together: the existence of c' and the reducibility of $\langle e, c\rangle \Rightarrow c'$. Therefore, from the existence part, we will have the following corollary.

Corollary 4.3.3: For any element e in $\langle \Gamma, \Delta, \Psi\rangle$, there is a canonical element c such that $e = c$ holds in $\langle \Gamma, \Delta, \Psi, \Theta\rangle$. []

Showing the reducibility can be regarded as showing the termination of computation in CPL. So, every program terminates in CPL. This is what we expected because we only use primitive recursions.

First, we show a property of canonical elements.

Proposition 4.3.4: A canonical element of $R(E_1,\ldots,E_n)$ for a right object R defined by (RC) has the following form.

$$\psi_R(e_1,\ldots,e_m) \circ c$$

where c is another canonical element.
On the other hand, a canonical element of $L(E_1, \ldots, E_n)$ for a left object L defined by (LC) has the following form.

$$\alpha_{L,j} \circ c$$

where c is another canonical element.
Proof: A canonical element only consists of natural transformations of left objects and factorizers of right objects. Therefore, a canonical element should look like either

$$\alpha_{L,j} \circ c \qquad \text{or} \qquad \psi_R(e_1, \ldots, e_m) \circ c.$$

From the typing rules in section 2.4, the first one always gives an element of $L(E_1, \ldots, E_n)$ and the second one gives an element of $R(E_1, \ldots, E_n)$. []

From this proposition, we can see that whenever we have

$$\langle e, c \rangle \qquad \text{or} \qquad \langle c, E, R \rangle$$

we can always apply exactly one of the rules in definition 4.1.7. In other words, the computation in CPL is never stuck and deterministic. We can always proceed to *the* next computation step.

We are going to prove the normalization theorem by the *computability* method due to Tait [Tait 67]. This method is often used to show normalization of various systems especially that of lambda calculi (see, for example, [Stenlund 72] and [Lambek and Scott 86]) where two kinds of induction are used: induction on types and induction on structures. The method usually goes for lambda calculi as follows:

1. Define the notion of computable terms inductively on types.
2. Show that any computable term is normalizable.
3. Show that all the terms are computable by induction on terms.
4. Therefore, any term is normalizable.

The notion of computable terms is stronger than that of normalizable terms, but we need this stronger notion (which is a part of the essence of the Tait computability method) to carry out the normalization proof. The computability predicate also divide the two inductions involved in the proof clearly. In our case, the normalization proof goes as follows.

1. We define the notion of *computable canonical elements* inductively on types. Intuitively, a canonical element c is computable if
   (a) $\alpha_L \circ c$ is computable if c is computable.
   (b) $\psi_R(e) \circ c$ is computable if there is a reduction $\langle \alpha_R, \psi_R(e) \circ c \rangle \Rightarrow c'$ such that c' is computable, that is all the components of $\psi_R(e) \circ c$ are computable.
2. We define the notion of *calculability* for expressions (we could have called it computability as well as is conventional in proofs about lambda calculi, but we distinguish them for clarity). An expression e is *calculable* if for any computable canonical element c there is a reduction $\langle e, c \rangle \Rightarrow c'$ such that c' is computable. Note that an expression e is normalizable if for any canonical element e there is a reduction $\langle e, c \rangle \Rightarrow c$, whereas e is calculable if there is a reduction for any *computable* canonical element. Therefore, it should be easier to prove that an expression is calculable than to prove it is normalizable.

3. We will prove that all the expressions are calculable by structural induction.
4. As an easy corollary, we can show that any canonical element is computable.
5. Finally, we prove that all the reductions are normalizing.

First, let us assign for each n-ary functor F a function which given sets $C_1 \ldots, C_n$ of canonical elements of type $E_1, \ldots, E_n$ gives a set of canonical elements C of type $F(E_1, \ldots, E_n)$. We write $\tilde{F}$ for the function (i.e. $C = \tilde{F}(C_1, \ldots, C_n)$). $\tilde{F}$ is monotonic in the ith argument if F is covariant in the ith argument, and $\tilde{F}$ is anti-monotonic in the ith argument if F is contravariant in the ith argument.

Definition 4.3.5: 1. For a left object[2]

$$\begin{array}{l}\text{left object } L(X) \text{ with } \psi_L \text{ is}\\ \quad \alpha_L{:}E_L(L,X) \to L\\ \text{end object,}\end{array}$$

$\tilde{L}(C)$ is the minimal fixed point of the following monotonic function:

$$S \longmapsto \{\ \alpha_L \circ c \ \mid\ c \in \tilde{E}_L(S,C)\ \}$$

The minimal fixed point can be calculated as the least upper bound of the following ascending chain:

$$\tilde{L}_0(C) \subseteq \tilde{L}_1(C) \subseteq \cdots \subseteq \tilde{L}_n(C) \subseteq \tilde{L}_{n+1}(C) \subseteq \cdots \subseteq \tilde{L}_\omega(C) \subseteq \cdots$$

where $\tilde{L}_0(C)$ is $\emptyset$ and

$$\tilde{L}_{\beta+1}(C) \stackrel{\text{def}}{=} \{\ \alpha_L \circ c \ \mid\ c \in \tilde{E}_L(\tilde{L}_\beta(C),C)\ \}.$$

2. For a right object

$$\begin{array}{l}\text{right object } R(X) \text{ with } \psi_R \text{ is}\\ \quad \alpha_R{:}R \to E'_R(R,X)\\ \text{end object,}\end{array}$$

$\tilde{R}(C)$ is the maximal fixed point of the following monotonic function:

$$S \longmapsto \{\ \psi_R(e) \circ c \ \mid\ \langle \alpha_R, \psi_R(e) \circ c\rangle \Rightarrow c' \text{ such that } c' \in \tilde{E}'_R(S,C)\ \}$$

The maximal fixed point can be calculated as the greatest lower bound of the following descending chain:

$$\tilde{R}_0(C) \supseteq \tilde{R}_1(C) \supseteq \cdots \supseteq \tilde{R}_n(C) \supseteq \tilde{R}_{n+1}(C) \supseteq \cdots \supseteq \tilde{R}_\omega(C) \supseteq \cdots$$

where $\tilde{R}_0(C)$ is the set of all the canonical elements of $R(E)$ (where C is a set of canonical elements of type E) and

$$\tilde{R}_{\beta+1}(C) \stackrel{\text{def}}{=} \{\ \psi_R(e) \circ c \ \mid\ \langle \alpha_R, \psi_R(e) \circ c\rangle \Rightarrow c' \text{ such that } c' \in \tilde{E}'_R(\tilde{R}_\beta(C),C)\ \}.$$

[2]For simplicity, we define this for a simple left object, but the general case should be obvious.

3. For a right object[3]

$$\begin{array}{l}\text{right object } R'(X) \text{ with } \psi_{R'} \text{ is}\\ \quad \alpha_{R'}\colon \mathrm{prod}(R, E_{R'}(X)) \to E'_{R'}(R', X)\\ \text{end object,}\end{array}$$

$\tilde{R}'(C)$ is the maximal fixed point of the following monotonic function:

$$S \longmapsto \{\ \psi_{R'}(e) \circ c \ \mid\ \begin{array}{l}\text{For any } c' \in \tilde{E}_{R'}(C)\ \langle \alpha_{R'}, \mathrm{pair}(\psi_{R'}(e) \circ c, c')\rangle \Rightarrow c''\\ \quad \text{such that } c'' \in \tilde{E}'_{R'}(S, C)\ \ \}\end{array}$$

We can similarly define $\tilde{R}'_\beta(C)$.

Well-definedness: We have to show that $\tilde{F}$ is monotonic or anti-monotonic according to the variance of F. We prove this by induction on the order of declaration of objects.

1. If F is the left object L above and covariant, we show that $\tilde{L}_\beta$ is monotonic by induction on β. $\tilde{L}_0$ is trivially monotonic. $\tilde{L}_{\beta+1}(C)$ is

$$\{\ \alpha_L \circ c \ \mid\ c \in \tilde{E}_L(\tilde{L}_\beta(C), C)\ \}.$$

   From the induction hypothesis $\tilde{L}_\beta$ is monotonic. Since $E_L(L, X)$ is covariant in both L and X, from the other induction hypothesis $\tilde{E}_L$ is monotonic. Therefore, $\tilde{E}_L(\tilde{L}_\beta(C), C)$ is monotonic in C, and $\tilde{L}_{\beta+1}$ is monotonic. Hence, $\tilde{L}$ is monotonic. Similarly, we can show that $\tilde{L}$ is anti-monotonic if L is contravariant.

2. If F is the right object R above and covariant, we show that $\tilde{R}_\beta$ is monotonic by induction on β. $\tilde{R}_0$ is trivially monotonic. $\tilde{R}_{\beta+1}(C)$ is

$$\{\ \psi_R(e) \circ c \ \mid\ \langle \alpha_R, \psi_R(e) \circ c\rangle \Rightarrow c' \wedge c' \in \tilde{E}'_R(\tilde{R}_\beta(C), C)\ \}.$$

   From the induction hypothesis, $\tilde{R}_\beta$ is monotonic, and from the other induction hypothesis, $\tilde{E}'_R$ is monotonic in both arguments. Therefore, $\tilde{R}_{\beta+1}$ is monotonic, and by induction $\tilde{R}$ is monotonic. We can similarly show that $\tilde{R}$ is anti-monotonic if R is contravariant.

3. If F is the right object R' above and covariant, we show that $\tilde{R}'_\beta$ is monotonic by induction on β. $\tilde{R}'_0$ is trivially monotonic. $\tilde{R}'_{\beta+1}(C)$ is

$$\begin{array}{l}\{\ \psi_{R'}(e) \circ c \ \mid\ \forall c' \in \tilde{E}_{R'}(C)\ \langle \alpha_{R'}, \mathrm{pair}(\psi_{R'}(e) \circ c, c')\rangle \Rightarrow c'' \wedge\\ \qquad\qquad\qquad\qquad\qquad\qquad c'' \in \tilde{E}'_{R'}(\tilde{R}'_\beta(C), C)\ \}.\end{array}$$

   From the induction hypothesis, $\tilde{R}'_\beta$ is monotonic, and from the other induction hypothesis, $\tilde{E}_{R'}$ is anti-monotonic and $\tilde{E}'_{R'}$ is monotonic in both arguments. Therefore, $\tilde{R}'_{\beta+1}$ is monotonic, and by induction $\tilde{R}'$ is monotonic. We can similarly show that $\tilde{R}'$ is anti-monotonic if R' is contravariant. □

[3]For a right object R, since the domain of $\alpha_R\colon E_R(R, X) \to E'_R(R, X)$ needs to be productive in R, there are basically these two cases: when $E_R(R, X)$ is R and when it is $\mathrm{prod}(R, E''(X))$.

We now define the notion of *computability* and *calculability*.

Definition 4.3.6: The set $\Omega_{F(E_1,\ldots,E_n)}$ of *computable* canonical elements of type $F(E_1,\ldots,E_n)$ is defined inductively by $\tilde{F}(\Omega_{E_1},\ldots,\Omega_{E_n})$. []

Definition 4.3.7: An CSL expression e of type $E \to E'$ is called *calculable with respect to* $C \to C'$ for $C \subseteq \Omega_E$ and $C' \subseteq \Omega_{E'}$ if for any c in C there is a reduction $\langle e, c\rangle \Rightarrow c'$ such that c' is in C'. When e is calculable with respect to $\Omega_E \to \Omega_{E'}$, we simply say that e is *calculable*. []

Example 4.3.8: 1. For the terminal object '1'

right object 1 with !
end object,

since there is no natural transformation, any canonical element is computable. Let us use $\star$ to denote an arbitrary element of $\tilde{1}$.

2. For the left object 'nat' of natural numbers

left object nat with pr is
zero: $1 \to$ nat
succ: nat $\to$ nat
end object,

$\widetilde{\text{nat}}_0$ is $\emptyset$. $\widetilde{\text{nat}}_1$ is

$$\{\ \text{zero} \circ c,\ \text{succ} \circ c' \ \mid\ c \in \tilde{1} \wedge c' \in \widetilde{\text{nat}}_0\ \} = \{\ \text{zero} \circ \star\ \}.$$

Similarly $\widetilde{\text{nat}}_2$ consists of zero $\circ \star$ and succ $\circ$ zero $\circ \star$. In general, $\widetilde{\text{nat}}_n$ is the set of n elements corresponding to a set of 0, 1, 2, ..., and $n-1$. Therefore, $\widetilde{\text{nat}}$ is the set of all the canonical elements of nat.

3. For the right object 'prod' of products

right object prod(X,Y) with pair is
pi1: prod $\to X$
pi2: prod $\to Y$
end object,

a canonical element $\text{pair}(e_1,e_2) \circ c$ is computable if there are reductions

$$\langle \text{pi1}, \text{pair}(e_1,e_2) \circ c\rangle \Rightarrow c_1 \qquad \text{and} \qquad \langle \text{pi2}, \text{pair}(e_1,e_2) \circ c\rangle \Rightarrow c_2$$

such that c_1 and c_2 are computable, that is a canonical element of prod is computable if its components are computable. Since the reductions above are equivalent to $\langle e_1, c\rangle \Rightarrow c_1$ and $\langle e_2, c\rangle \Rightarrow c_2$, if c is computable and e_1 and e_2 are calculable, $\text{pair}(e_1,e_2) \circ c$ is computable.

4. For the right object 'exp' of exponentials

$$\begin{array}{l}\text{right object } \exp(X,Y) \text{ with curry is}\\ \quad \text{eval: } \mathrm{prod}(\exp, X) \to Y\\ \text{end object,}\end{array}$$

a canonical element $\mathrm{curry}(e) \circ c$ of type $\exp(E, E')$ is computable, if for any computable canonical element c' of type E' there is a reduction

$$\langle \mathrm{eval}, \mathrm{pair}(\mathrm{curry}(e) \circ c, c') \rangle \Rightarrow c''$$

and c'' is computable. The reduction is equivalent to $\langle e, \mathrm{pair}(c, c') \rangle \Rightarrow c''$. Since $\mathrm{pair}(c, c')$ is computable, there is such a reduction if e is calculable. Remember that $\mathrm{curry}(e)$ corresponds to the closure of e (or lambda closed term of e) so that we can say that a closure is calculable if the application with canonical elements always results canonical elements. This exactly corresponds to the definition of computability for lambda expressions (see [Stenlund 72]).

5. For the right object 'inflist' of infinite lists

$$\begin{array}{l}\text{right object } \mathrm{inflist}(X) \text{ with fold is}\\ \quad \text{head: inflist} \to X\\ \quad \text{tail: inflist} \to \mathrm{inflist}\\ \text{end object,}\end{array}$$

let us figure out the computable canonical elements of type inflist(nat). $\widetilde{\mathrm{inflist}}_0(\Omega_{\mathrm{nat}})$ is the set of all the canonical elements of type inflist(nat). $\widetilde{\mathrm{inflist}}_1(\Omega_{\mathrm{nat}})$ is

$$\begin{array}{ll} & \{\ \mathrm{fold}(e_1, e_2) \circ c \ \mid \ \langle \mathrm{head}, \mathrm{fold}(e_1, e_2) \circ c \rangle \Rightarrow c_1 \wedge c_1 \in \Omega_{\mathrm{nat}} \wedge \\ & \qquad\qquad \langle \mathrm{tail}, \mathrm{fold}(e_1, e_2) \circ c \rangle \Rightarrow c_2 \wedge c_2 \in \widetilde{\mathrm{inflist}}_0(\Omega_{\mathrm{nat}}) \ \} \\ = & \{\ \mathrm{fold}(e_1, e_2) \circ c \ \mid \ \langle e_1, c \rangle \Rightarrow c_1 \wedge \langle \mathrm{fold}(e_1, e_2) \circ e_2, c \rangle \Rightarrow c_2 \ \}. \end{array}$$

Therefore, if e_1 and e_2 are calculable and c is computable, $\mathrm{fold}(e_1, e_2) \circ c$ is in $\widetilde{\mathrm{inflist}}_1(\Omega_{\mathrm{nat}})$. We can inductively show that it is in any $\widetilde{\mathrm{inflist}}_\beta(\Omega_{\mathrm{nat}})$, and, therefore, it is in $\widetilde{\mathrm{inflist}}(\Omega_{\mathrm{nat}})$.

6. For the left object of ordinals

$$\begin{array}{l}\text{left object ord with pro is}\\ \quad \text{ozero: } 1 \to \mathrm{ord}\\ \quad \text{sup: } \exp(\mathrm{nat}, \mathrm{ord}) \to \mathrm{ord}\\ \text{end object,}\end{array}$$

$\widetilde{\mathrm{ord}}_0$ is empty, $\widetilde{\mathrm{ord}}_1$ is $\{\ \mathrm{ozero} \circ \star\ \}$, and

$$\begin{array}{ll} & \widetilde{\mathrm{ord}}_2 \\ = & \{\ \mathrm{sup} \circ c,\ \mathrm{ozero} \circ \star \ \mid \ c \in \widetilde{\exp}(\widetilde{\mathrm{nat}}, \widetilde{\mathrm{ord}}_1)\ \} \\ = & \{\ \mathrm{sup} \circ \mathrm{curry}(e) \circ c,\ \mathrm{ozero} \circ \star \ \mid \ \forall c' \in \widetilde{\mathrm{nat}}\ \langle e, \mathrm{pair}(c, c') \rangle \Rightarrow c'' \wedge \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad c'' \in \widetilde{\mathrm{ord}}_1\ \} \\ = & \{\ \mathrm{sup} \circ \mathrm{curry}(e) \circ c,\ \mathrm{ozero} \circ \star \ \mid \ \forall c' \in \Omega_{\mathrm{nat}}\ \langle e, \mathrm{pair}(c, c') \rangle \Rightarrow \mathrm{ozero} \circ \star\ \}. \end{array}$$

In general,

$$\widetilde{\mathrm{ord}}_{\beta+1} = \{\ \mathrm{sup} \circ \mathrm{curry}(e) \circ c,\ \mathrm{ozero} \circ \star \ \mid \\ \forall c' \in \Omega_{\mathrm{nat}}\ \langle e, \mathrm{pair}(c, c')\rangle \Rightarrow c'' \wedge c'' \in \widetilde{\mathrm{ord}}_{\beta}\ \}$$

Therefore, a canonical element $\mathrm{sup} \circ \mathrm{curry}(e) \circ c$ is computable if the following reductions always exist:

$$\begin{array}{l} \langle e, \mathrm{pair}(c, c_1)\rangle \Rightarrow \mathrm{sup} \circ \mathrm{curry}(e_1) \circ c'_1 \\ \quad \langle e_1, \mathrm{pair}(c'_1, c_2)\rangle \Rightarrow \mathrm{sup} \circ \mathrm{curry}(e_2) \circ c'_2 \\ \quad\quad \langle e_2, \mathrm{pair}(c'_2, c_3)\rangle \Rightarrow \mathrm{sup} \circ \mathrm{curry}(e_3) \circ c'_3 \\ \quad\quad\quad \cdots \\ \quad\quad\quad \langle e_\beta, \mathrm{pair}(c'_\beta, c_{\beta+1})\rangle \Rightarrow \mathrm{ozero} \circ \star \end{array} \qquad \square$$

The next proposition intuitively means that functors preserve the structure of data. For example, when the functor 'list' (or `map` in ML and `MAPCAR` in LISP) is applied to a list, it only changes the components of the list and preserves the length.

Proposition 4.3.9: Let F be n-ary functor, and $e_1, \ldots, e_n$ be CSL expressions calculable with respect to $C_i \to C'_i$. Then $F(e_1, \ldots, e_n)$ is calculable with respect to $\tilde{F}(C''_1, \ldots, C''_n) \to \tilde{F}(C'''_1, \ldots, C'''_n)$, where C''_i is C_i and C'''_i is C'_i if F is covariant in the ith argument and C''_i is C'_i and C'''_i is C_i if F is contravariant in the ith argument.
Proof: We prove this by induction on the order of declarations of objects.

1. Let F be a left object declared by

$$\begin{array}{l} \text{left object } L(X) \text{ with } \psi_L \text{ is} \\ \quad \alpha_L : E_L(L, X) \to L \\ \text{end object} \end{array}$$

which is covariant in X, e be a CSL expression which is calculable with respect to $C \to C'$. We prove that $L(e)$ is calculable with respect to $\tilde{L}_\beta(C) \to \tilde{L}_\beta(C')$ by induction on β. Trivially, $L(e)$ is calculable with respect to $\tilde{L}_0(C) \to \tilde{L}_0(C)$ because $\tilde{L}_0(C)$ is empty. Assume we have proved that $L(e)$ is calculable with respect to $\tilde{L}_\beta(C) \to \tilde{L}_\beta(C')$. An element of $\tilde{L}_{\beta+1}(C)$ is $\alpha_L \circ c$ such that $c \in \tilde{E}_L(\tilde{L}_\beta(C), C)$. From L-FACT we get

$$\dfrac{\dfrac{\dfrac{\langle \alpha_L \circ E_L(\mathbf{I}, e) \circ E_L(L(e), \mathbf{I}), c\rangle \Rightarrow c'}{\langle \alpha_L \circ E_L(\mathbf{I}, e) \circ E_L(\psi_L(\alpha_L \circ E_L(\mathbf{I}, c)), \mathbf{I}), c\rangle \Rightarrow c'}}{\langle \psi_L(\alpha_L \circ E_L(\mathbf{I}, e)), \alpha_L \circ c\rangle \Rightarrow c'}}{\langle L(e), \alpha_L \circ c\rangle \Rightarrow \alpha_L \circ c'}$$

Since $E_L(L, X)$ consists of functors declared before L, from the induction hypothesis, $E_L(L(e), \mathbf{I})$ is calculable with respect to $\tilde{E}_L(\tilde{L}_\beta(C), C) \to \tilde{E}_L(\tilde{L}_\beta(C'), C)$ and $E_L(\mathbf{I}, e)$ is calculable with respect to $\tilde{E}_L(\tilde{L}_\beta(C'), C) \to \tilde{E}_L(\tilde{L}_\beta(C'), C')$. Therefore, there is a reduction

$$\langle E_L(\mathbf{I}, e) \circ E_L(L(e), \mathbf{I}), c\rangle \Rightarrow c'$$

such that c' is in $\tilde{E}_L(\tilde{L}_\beta(C'), C')$. From definition 4.3.5, $\alpha_L \circ c'$ is in $\tilde{L}_{\beta+1}(C')$. Hence, $L(e)$ is calculable with respect to $\tilde{L}_{\beta+1}(C) \to \tilde{L}_{\beta+1}(C')$. By induction, $L(e)$ is calculable with respect to $\tilde{L}_\beta(C) \to \tilde{L}_\beta(C')$ for any β, and, therefore, $L(e)$ is calculable with respect to $\tilde{L}(C) \to \tilde{L}(C')$. When $L(X)$ is contravariant, we can similarly prove that $L(e)$ is calculable with respect to $\tilde{L}(C') \to \tilde{L}(C)$.

2. Let F be a right object declared by

$$\begin{array}{l}\text{right object } R(X) \text{ with } \psi_R \text{ is}\\ \quad \alpha_R\colon R \to E'_R(R, X)\\ \text{end object}\end{array}$$

which is covariant, and e be a CSL expression which is calculable with respect to $C \to C'$. We prove that $R(e)$ is calculable with respect to $\tilde{R}_\beta(C) \to \tilde{R}_\beta(C')$ by induction on β. From R-FACT, we have

$$\langle R(e), c\rangle \Rightarrow \psi_R(E'_R(\mathbf{I}, e) \circ \alpha_R) \circ c.$$

Trivially, $R(e)$ is calculable with respect to $\tilde{R}_0(C) \to \tilde{R}_0(C')$ because $\tilde{R}_0(C')$ is the set of all the canonical elements of R. Assume we have proved that $R(e)$ is calculable with respect to $\tilde{R}_\beta(C) \to \tilde{R}_\beta(C')$. An element of $\tilde{R}_{\beta+1}(C)$ is $\psi_R(e') \circ c'$ such that there is a reduction $\langle \alpha_R, \psi_R(e') \circ c'\rangle \Rightarrow c''$ and c'' is in $\tilde{E}'_R(\tilde{R}_\beta(C), C)$. We will show that the following canonical element is in $\tilde{R}_{\beta+1}(C')$:

$$\psi_R(E'_R(\mathbf{I}, e) \circ \alpha_R) \circ \psi_R(e') \circ c' \qquad (*)$$

From R-NAT, we have

$$\dfrac{\dfrac{\dfrac{\langle E'_R(R(e), \mathbf{I}) \circ E'_R(\mathbf{I}, e), c''\rangle \Rightarrow c'''}{\langle E'_R(R(e), \mathbf{I}) \circ E'_R(\mathbf{I}, e) \circ \alpha_R, \psi_R(e') \circ c'\rangle \Rightarrow c'''}}{\langle E'_R(\psi_R(E'_R(\mathbf{I}, c) \circ \alpha_R), \mathbf{I}), E'_R(\mathbf{I}, e) \circ \alpha_R, \psi_R(e') \circ c'\rangle \Rightarrow c'''}}{\langle \alpha_R, \psi_R(E'_R(\mathbf{I}, e) \circ \alpha_R) \circ \psi_R(e') \circ c'\rangle \Rightarrow c'''}$$

Since $E'_R(R, X)$ consists of functors declared before R, $E'_R(R(e), \mathbf{I}) \circ E'_R(\mathbf{I}, e)$ is calculable with respect to $\tilde{E}'_R(\tilde{R}_\beta(C), C) \to \tilde{E}'_R(\tilde{R}_\beta(C'), C')$ from the induction hypothesis. Therefore, c''' is in $\tilde{E}'_R(\tilde{R}_\beta(C'), C')$, and from definition 4.3.5, $(*)$ is in $\tilde{R}_{\beta+1}(C)$. Therefore, by induction, $R(e)$ is calculable with respect to $\tilde{R}_\beta(C) \to \tilde{R}_\beta(C')$ for any β, so it is calculable with respect to $\tilde{R}(C) \to \tilde{R}(C')$. When $R(X)$ is contravariant, we can similarly prove that $R(e)$ is calculable with respect to $\tilde{R}(C') \to \tilde{R}(C)$.

3. If F be a right object declared by

$$\begin{array}{l}\text{right object } R'(X) \text{ with } \psi_{R'} \text{ is}\\ \quad \alpha_{R'}\colon \mathrm{prod}(R', E_{R'}(X)) \to E'_R(R, X)\\ \text{end object,}\end{array}$$

we can similarly prove that $R(e)$ is calculable with respect to $\tilde{R}(C) \to \tilde{R}(C')$ (or with respect to $\tilde{R}(C') \to \tilde{R}(C)$ when $R(X)$ is contravariant). □

In the following few lemmas, we are to prove all the expressions are calculable.

Lemma 4.3.10: **I** is calculable.
Proof: We have to show that for any computable canonical element c there is a reduction of $\langle \mathbf{I}, c\rangle \Rightarrow c'$ and that c' is computable. This is immediate from the reduction rule IDENT and that c' is c []

Lemma 4.3.11: If both e_1 and e_2 are calculable, so is $e_1 \circ e_2$.
Proof: For any computable canonical element c' we have the following reduction from COMP:

$$\frac{\langle e_2, c\rangle \Rightarrow c'' \qquad \langle e_1, c''\rangle \Rightarrow c'}{\langle e_1 \circ e_2, c\rangle \Rightarrow c'}$$

Since e_2 is calculable, there is a reduction for $\langle e_2, c\rangle \Rightarrow c''$ so that c'' is computable. Since e_1 is calculable, there is a reduction for $\langle e_1, c''\rangle \Rightarrow c'$ so that c' is computable. Therefore, there is a reduction for $\langle e_1 \circ e_2, c\rangle \Rightarrow c'$ so that c' is computable. []

Lemma 4.3.12: For any natural transformation α_L of a left object L, α_L is calculable.
Proof: For any computable canonical element c, we have the following reduction by L-NAT:

$$\langle \alpha_L, c\rangle \Rightarrow \alpha_L \circ c$$

From definition 4.3.6, $\alpha_L \circ c$ is computable. Therefore, α_L is calculable. []

Lemma 4.3.13: For any natural transformation α_R of a right object R, α_R is calculable.
Proof: Let R be

right object $R(X)$ with ψ_R is
$\alpha_R : R \to E'_R(R, X)$
end object,

and c be a computable canonical element $R(E)$. For any β, c is in $\tilde{R}_{\beta+1}(\Omega_E)$. From definition 4.3.5, there exists a reduction $\langle \alpha_R, c\rangle \Rightarrow c'$ such that c' is in $\tilde{E}'_R(\tilde{R}_\beta(\Omega_E), \Omega_E)$. Because the result of reductions does not depend on β, c' is in

$$\tilde{E}'_R(\bigcap_\beta \tilde{R}_\beta(\Omega_E), \Omega_E) = \tilde{E}'_R(\tilde{R}(\Omega_E), \Omega_E) = \Omega_{E'_R(R(E),E)}.$$

Therefore, α_R is calculable. We can similarly prove that for a right object

right object $R'(X)$ with $\psi_{R'}$ is
$\alpha_{R'} : \mathrm{prod}(R', E_{R'}(X)) \to E'_{R'}(R', X)$
end object,

$\alpha_{R'}$ is calculable. []

Lemma 4.3.14: If e is calculable, so is $\psi_L(e)$ where L is a left object and ψ_L is its factorizer.

Proof: Let L be

left object $L(X)$ with ψ_L is
$\alpha_L: E_L(L, X) \to L$
end object,

and $e: E_L(E, E') \to E$ be a calculable CSL expression. We will prove that $\psi_L(e)$ is calculable with respect to $\tilde{L}_\beta(\Omega_{E'}) \to \Omega_E$ by induction on β. Trivially, it is calculable with respect to $\tilde{L}_0(\Omega_{E'}) \to \Omega_E$ because $\tilde{L}_0(\Omega_{E'})$ is empty. Assume we have proved that $\psi_L(e)$ is calculable with respect to $\tilde{L}_\beta(\Omega_{E'}) \to \Omega_E$. An element in $\tilde{L}_{\beta+1}(\Omega_{E'})$ is $\alpha_L \circ c$ for c which is in $\tilde{E}_L(\tilde{L}_\beta(\Omega_{E'}), \Omega_{E'})$. From L-FACT, we get

$$\frac{\dfrac{\langle E_L(\psi_L(e), \mathbf{I}), c\rangle \Rightarrow c'' \qquad \langle e, c''\rangle \Rightarrow c'}{\langle e \circ E_L(\psi_L(e), \mathbf{I}), c\rangle \Rightarrow c'}}{\langle \psi_L(e), \alpha_L \circ c\rangle \Rightarrow c'}$$

From proposition 4.3.9 and the induction hypothesis, $E_L(\psi_L(e), \mathbf{I})$ is calculable with respect to $\tilde{E}_L(\tilde{L}_\beta(\Omega_{E'}), \Omega_{E'}) \to \tilde{E}_L(\Omega_E, \Omega_{E'})$, and there is a reduction $\langle E_L(\psi_L(e), \mathbf{I}), c\rangle \Rightarrow c''$. Since e is calculable, there is a reduction $\langle e, c''\rangle \Rightarrow c'$ such that c' is in Ω_E. Therefore, $\psi_R(e)$ is calculable with respect to $\tilde{L}_{\beta+1}(\Omega_{E'}) \to \Omega_E$, and by induction it is calculable with respect to $\tilde{L}_\beta(\Omega_{E'}) \to \Omega_E$ for any β. Because $\Omega_{L(E')} = \tilde{L}(\Omega_{E'})$ is $\bigcup_\beta \tilde{L}_\beta(\Omega_{E'})$, we have proved that $\psi_L(e)$ is calculable. []

Lemma 4.3.15: If e is calculable, so is $\psi_R(e)$ where R is a right object and ψ_R is its factorizer.

Proof: Let R be

right object $R(X)$ with ψ_R is
$\alpha_R: R \to E'_R(R, X)$
end object,

and e be a CSL expression of type $E \to E'_R(E, E')$. We are to prove that $\psi_R(e): E \to R(E')$ is calculable. Since $\Omega_{R(E')} = \tilde{R}(\Omega_{E'})$ is $\bigcap_\beta \tilde{R}_\beta(\Omega_{E'})$, we prove that $\psi_R(e)$ is calculable with respect to $\Omega_E \to \tilde{R}_\beta(\Omega_{E'})$ by induction on β. Trivially, it is calculable with respect to $\Omega_E \to \tilde{R}_0(\Omega_{E'})$ because for any $c \in \Omega_E$ we have $\langle \psi_R(e), c\rangle \Rightarrow \psi_R(e) \circ c$ and $\tilde{R}_0(\Omega_{E'})$ is the set of all the canonical elements of type $R(E')$. Assume we have proved that $\psi_R(e)$ is calculable with respect to $\Omega_E \to \tilde{R}_\beta(\Omega_{E'})$. For any $c \in \Omega_E$ we have $\langle \psi_R(e), c\rangle \Rightarrow \psi_R(e) \circ c$. From R-NAT, we get

$$\frac{\langle e, c\rangle \Rightarrow c'' \qquad \langle E'_R(\psi_R(e), \mathbf{I}), c''\rangle \Rightarrow c'}{\langle \alpha_R, \psi_R(e) \circ c\rangle \Rightarrow c'}.$$

Since e is calculable, there is a reduction $\langle e, c\rangle \Rightarrow c''$ and c'' is in $\tilde{E}'_R(\Omega_E, \Omega_{E'})$. As we assumed that $\psi_R(e)$ is calculable with respect to $\Omega_E \to \tilde{R}_\beta(\Omega_{E'})$, $E'_R(\psi_R(e), \mathbf{I})$ is calculable with respect to $\tilde{E}'_R(\Omega_E, \Omega_{E'}) \to \tilde{E}'_R(\tilde{R}_\beta(\Omega_{E'}), \Omega_{E'})$ from proposition 4.3.9. Therefore, there is a reduction $\langle E'_R(\psi_R(e), \mathbf{I})c''\rangle \Rightarrow c'$ and c' is in $\tilde{E}'_R(\tilde{R}_\beta(\Omega_{E'}), \Omega_{E'})$. From definition 4.3.5, $\psi_R(e) \circ c$ is in $\tilde{R}_{\beta+1}(\Omega_{E'})$, so $\psi_R(e)$ is calculable with respect to $\Omega_E \to \tilde{R}_{\beta+1}(\Omega_{E'})$, and by induction it is calculable with respect to $\Omega_E \to \tilde{R}_\beta(\Omega_{E'})$ for any β. Therefore, it is calculable with respect to $\Omega_E \to \tilde{R}(\Omega_{E'})$. We can similarly

prove that for a right object

$$\begin{array}{l}\text{right object } R'(X) \text{ with } \psi_{R'} \text{ is}\\ \quad \alpha_{R'}\colon \text{prod}(R', E_{R'}(X)) \to E'_{R'}(R', X)\\ \text{end object,}\end{array}$$

$\psi_{R'}(e)$ is calculable. □

Theorem 4.3.16: Any CSL expression e is calculable.
Proof: This is proved by structural induction and each case follows from the lemmas, 4.3.10, 4.3.11, 4.3.12, 4.3.13, 4.3.14 and 4.3.15. □

Corollary 4.3.17: Any canonical element is computable.
Proof: As a canonical element c is a CSL expression, and therefore, from theorem 4.3.16 it is calculable. Because any canonical element of the terminal object 1 is computable, specially $\mathbf{I}$ is computable. Therefore, there is a reduction $\langle c, \mathbf{I}\rangle \Rightarrow c'$ such that c' is computable. Trivially, c' is c (using L-NAT, R-FACT and COMP), so c is computable. □

We now finish this section by proving the normalization theorem.

Proof of Normalization 4.3.2: From theorem 4.3.16, any expression e is calculable, and from corollary 4.3.17, any canonical element is computable. Therefore, from the definition 4.3.7 of calculable expressions, there is a reduction $\langle e, c\rangle \Rightarrow c'$. □

4.4 Properties of Computable Objects

In section 4.1, we saw that we have to restrict ourselves to computable objects (definition 4.1.6) in order to introduce our notion of computability into CDT. Let us see in this section some of the properties which these particular objects enjoy.

First, we show that computable left objects are fixed points of some domain equations.

Theorem 4.4.1: Let L be a computable left object declared as follows.

$$\begin{array}{l}\text{left object } L(X_1, \ldots, X_n) \text{ with } \psi_L \text{ is}\\ \quad \alpha_{L,1}\colon E_{L,1}(L, X_1, \ldots, X_n) \to L\\ \quad \cdots\\ \quad \alpha_{L,m}\colon E_{L,m}(L, X_1, \ldots, X_n) \to L\\ \text{end object}\end{array}$$

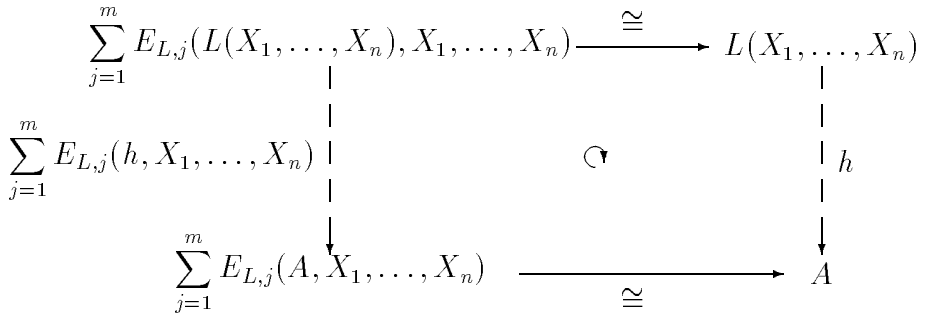
Then, the following isomorphism holds in any CSL structure which has L and coproducts.

$$L(X_1, \ldots, X_n) \cong \sum_{j=1}^{m} E_{L,j}(L(X_1, \ldots, X_n), X_1, \ldots, X_n)$$

where $\sum_{j=1}^{m}$ is the m-ary coproduct. Furthermore, if A is an object which satisfies

$$A \cong \sum_{j=1}^{m} E_{L,j}(A, X_1, \ldots, X_n),$$

there is a unique morphism h from $L(X_1, \ldots, X_n)$ to A such that the following diagram commutes.



Proof: For simplicity, we prove the isomorphism in case L does not have any parameters (i.e. $n = 0$). Therefore, the isomorphism we prove is

$$L \cong \sum_{j=1}^{m} E_{L,j}(L).$$

Let f be a morphism

$$[\alpha_{L,1}, \ldots, \alpha_{L,m}]$$

where $[\ , \ldots, \ ]$ is the factorizer of $\sum_{j=1}^{m}$. f is a morphism from $\sum_{j=1}^{m} E_{L,j}(L)$ to L. Let g be a morphism

$$\psi_L(\nu_1 \circ E_{L,1}(f), \ldots, \nu_m \circ E_{L,m}(f))$$

where ν_j is the j-th injection of $\sum_{j=1}^{m}$. g is a morphism from L to $\sum_{j=1}^{m} E_{L,j}(L)$. We show that f is the inverse of g. Let us first show that $f \circ g = \mathbf{I}$.

$$\begin{aligned}
& f \circ g \circ \alpha_{L,j} \\
&= f \circ \nu_j \circ E_{L,j}(f) \circ E_{L,j}(g) \quad \ldots\ldots \text{(from (LEQ}_j\text{))} \\
&= \alpha_{L,j} \circ E_{L,j}(f) \circ E_{L,j}(g) \\
&= \alpha_{L,j} \circ E_{L,j}(f \circ g) \quad \ldots\ldots (E_{L,j}(L) \text{ is covariant})
\end{aligned}$$

From (LCEQ),

$$f \circ g = \psi_R(\alpha_{L,1}, \ldots, \alpha_{L,m}) = \mathbf{I}.$$

The second equality holds again from (LCEQ). Next we show that $g \circ f = \mathbf{I}$.

$$\begin{aligned}
& g \circ f \\
&= [g \circ \alpha_{L,1}, \ldots, g \circ \alpha_{L,m}] \\
&= [\nu_1 \circ E_{L,1}(f \circ g), \ldots, \nu_m \circ E_{L,m}(f \circ g)] \\
&= [\nu_1 \circ E_{L,1}(\mathbf{I}), \ldots, \nu_m \circ E_{L,m}(\mathbf{I})] \\
&= [\nu_1, \ldots, \nu_m] \\
&= \mathbf{I}
\end{aligned}$$

Therefore, $L \cong \sum_{j=1}^{m} E_{L,j}(L)$. For any object A which satisfies $A \cong \sum_{j=1}^{m} E_{L,j}(A)$, let i be the isomorphism from $\sum_{j=1}^{m} E_{L,j}(A)$ to A, then the unique morphism is given by

$$\psi_R(i \circ \nu_1, \ldots, i \circ \nu_m).$$

It is easy to see the diagram commutes from (LEQ_j) and the uniqueness from (LCEQ). []

If we apply this theorem to the objects we defined in chapter 3, we get the following isomorphisms.

$$\begin{aligned} &\mathrm{nat} \cong 1 + \mathrm{nat} \\ &\mathrm{list}(X) \cong 1 + \mathrm{prod}(X, \mathrm{list}(X)) \end{aligned}$$

We can see the exact correspondence to domain theory. In domain theory, the domain of natural numbers and that of lists are defined as the minimal domains which satisfies the above isomorphisms.

By duality principle, we have the dual theorem of theorem 4.4.1.

Theorem 4.4.2: Let R be a computable right object declared as follows.

$$\begin{aligned} &\text{right object } R(X_1, \ldots, X_n) \text{ with } \psi_R \text{ is} \\ &\quad \alpha_{R,1} : R \to E'_{R,1}(R, X_1, \ldots, X_n) \\ &\qquad\qquad \cdots \\ &\quad \alpha_{R,m} : R \to E'_{R,m}(R, X_1, \ldots, X_n) \\ &\text{end object} \end{aligned}$$

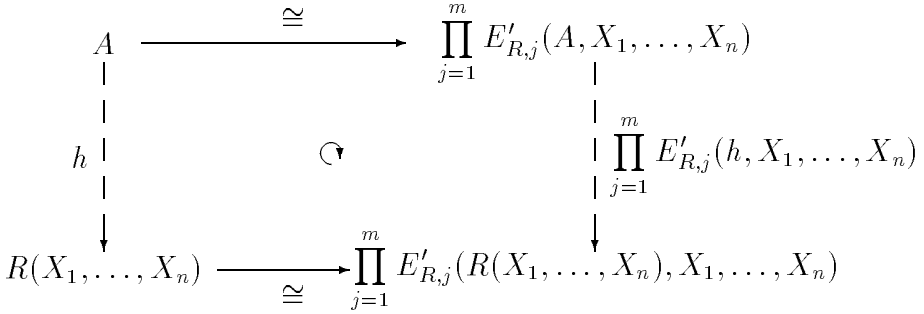
Then, the following isomorphism holds in any CSL structure which has R and products.

$$R(X_1, \ldots, X_n) \cong \prod_{j=1}^{m} E'_{R,j}(R(X_1, \ldots, X_n), X_1, \ldots, X_n)$$

where $\prod_{j=1}^{m}$ is the m-ary product. Furthermore, if A is an object which satisfies

$$A \cong \prod_{j=1}^{m} E'_{R,j}(A, X_1, \ldots, X_n),$$

there is a unique morphism h from A to $R(X_1, \ldots, X_n)$ such that the following diagram commutes.



Proof: By duality. []

We can see that the infinite list defined in subsection 3.3.6 is the maximal fixed point of the following domain equation.

$$\mathbf{inflist}(X) \cong X \times \mathbf{inflist}(X)$$

The next theorem states that productive objects define products.

Theorem 4.4.3: Let $P(Y_1, \ldots, Y_n)$ be a functor which is productive in Y_i. Then, there is a functor $F(Y_1, \ldots, Y_{i-1}, Y_{i+1}, \ldots, Y_n)$ such that

$$P(Y_1, \ldots, Y_n) \cong Y_i \times F(Y_1, \ldots, Y_{i-1}, Y_{i+1}, \ldots, Y_n)$$

Proof: First, note that it is easy to extend the theorem to productive functorial expressions by simply applying the theorem repeatedly. Let us prove the theorem by induction on the order of declaration of productive objects. Let the declaration of P to be as follows.

right object $P(Y_1, \ldots, Y_i, \ldots, Y_n)$ with ψ_P is
$\alpha_{P,1}: E_{P,1}(P, Y_1, \ldots, Y_{i-1}, Y_{i+1}, \ldots, Y_n) \to E'_{P,1}(Y_1, \ldots, Y_{i-1}, Y_{i+1}, \ldots, Y_n)$
$\ldots$
$\alpha_{P,j}: P \to E'_{P,j}(Y_1, \ldots, Y_{i-1}, Y_i, Y_{i+1}, \ldots, Y_n)$
$\ldots$
$\alpha_{P,m}: E_{P,m}(P, Y_1, \ldots, Y_{i-1}, Y_{i+1}, \ldots, Y_n) \to E'_{P,m}(Y_1, \ldots, Y_{i-1}, Y_{i+1}, \ldots, Y_n)$
end object

By induction hypothesis and from what we note at the beginning of the proof, there is a functor $F'(Y_1, \ldots, Y_{i-1}, Y_{i+1}, \ldots, Y_n)$ such that

$$E'_{P,j}(Y_1, \ldots, Y_n) \cong Y_i \times F'(Y_1, \ldots, Y_{i-1}, Y_{i+1}, \ldots, Y_n).$$

Since P is a computable object as well, $E_{P,k}(P, Y_1, \ldots, Y_{i-1}, Y_{i+1}, \ldots, Y_n)$ is productive in P. Therefore, from induction hypothesis there are functors

$$G_k(Y_1, \ldots, Y_{i-1}, Y_{i+1}, \ldots, Y_n)$$

such that

$$E_{P,k}(P, Y_1, \ldots, Y_{i-1}, Y_{i+1}, \ldots, Y_n) \cong P \times G_k(Y_1, \ldots, Y_{i-1}, Y_{i+1}, \ldots, Y_n).$$

Using exponentials, the above definition of P is essentially the same as

right object $P(Y_1, \ldots, Y_i, \ldots, Y_n)$ with ψ_P is
$\alpha_{P,1}: P \to \exp(G_1(Y_1, \ldots, Y_{i-1}, Y_{i+1}, \ldots, Y_n),$
$\qquad E'_{P,1}(Y_1, \ldots, Y_{i-1}, Y_{i+1}, \ldots, Y_n))$
$\ldots$
$\alpha_{P,j}: P \to Y_i \times F'(Y_1, \ldots, Y_{i-1}, Y_{i+1}, \ldots, Y_n)$
$\ldots$
$\alpha_{P,m}: P \to \exp(G_m(Y_1, \ldots, Y_{i-1}, Y_{i+1}, \ldots, Y_n),$
$\qquad E'_{P,m}(Y_1, \ldots, Y_{i-1}, Y_{i+1}, \ldots, Y_n))$
end object

From theorem 4.4.2, we have

$$P(Y_1,\ldots,Y_n) \cong Y_i \times F'(Y_1,\ldots) \times \prod_{\substack{k=1\\k\neq j}}^{m} \exp(G_k(Y_1,\ldots), E'_{P,k}(Y_1,\ldots)).$$

$F'(Y_1,\ldots)\times\prod_{\substack{k=1\\k\neq j}}^{m} \exp(G_k(Y_1,\ldots), E'_{P,k}(Y_1,\ldots))$ does not depend on Y_i. We have proved the theorem. □

From this theorem, we can always make the declaration of computable objects into an equivalent declaration to which we can apply theorem 4.4.2. For example, the declaration of the object for automata in subsection 3.3.7 was

right object $\mathrm{dyn}'(I,O)$ with univ' is
  $\mathrm{next}'\colon \mathrm{prod}(\mathrm{dyn}', I) \to \mathrm{dyn}'$
  $\mathrm{output}'\colon \mathrm{dyn}' \to O$
end object

to which we cannot apply theorem 4.4.2, but the above declaration is equivalent to the following one.

right object $\mathrm{dyn}'(I,O)$ with univ' is
  $\mathrm{next}'\colon \mathrm{dyn}' \to \exp(I, \mathrm{dyn}')$
  $\mathrm{output}'\colon \mathrm{dyn}' \to O$
end object

Then, from theorem 4.4.2 we can see $\mathrm{dyn}'(I,O)$ as the maximal fixed point of the following domain equation.

$$\mathrm{dyn}'(I,O) \cong \exp(I, \mathrm{dyn}'(I,O)) \times O$$

4.5 Reduction Rules for Full Evaluation

In section 4.1 we presented a set of reduction rules which can reduce any element to a canonical element. However, the notion of canonical element (definition 4.1.3) was quite weak (or sloppy), and the canonical elements we get out of reductions sometimes not acceptable as 'canonical'. We can define a more refined notion of canonical elements.

Definition 4.5.1: A canonical element is called *uncondition*, if it is generated by the following rule.

$$p ::= \mathbf{I} \mid \alpha_{L,j} \circ p \mid \psi_R(e_1,\ldots,e_m) \circ p \mid \psi_C(\ldots, e_j, \ldots, p_k, \ldots)$$

where R is not a unconditioned right object, C is a unconditioned object

right object $C(X_1,\ldots,X_n)$ with ψ_C is
  $\cdots$
  $\alpha_{C,j}\colon E_{C,j}(C, X_1,\ldots,X_n) \to E'_{C,j}(X_1,\ldots,X_n)$
  $\cdots$
  $\alpha_{C,k}\colon C \to E'_{C,k}(X_1,\ldots,X_n)$
  $\cdots$
end object

and if $E_{C,k}(C, X_1, \ldots, X_n)$ is simply C then the kth argument of ψ_C needs to be a unconditioned canonical element. []

For example,

$$\text{pair}(\text{succ}, \mathbf{I}) \circ \text{zero} \qquad \text{and} \qquad \text{pair}(\text{pi1} \circ \text{pair}(\text{succ} \circ \text{zero}, \text{nil}), \text{zero})$$

are canonical elements but not unconditioned. Their equivalent unconditioned canonical element is 'pair(succ $\circ$ zero, zero)'.

We can define reduction rules which only produce unconditioned canonical elements as result.

Definition 4.5.2: The form of reduction rules is

$$\langle e, p \rangle \twoheadrightarrow p'$$

where e is a CSL expression and p is a unconditioned canonical element whose domain is compatible with the domain of e.

1. FULL-IDENT
$$\langle \mathbf{I}, p \rangle \twoheadrightarrow p$$

2. FULL-COMP
$$\frac{\langle e_2, p \rangle \twoheadrightarrow p'' \qquad \langle e_1, p'' \rangle \twoheadrightarrow p'}{\langle e_1 \circ e_2, p \rangle \twoheadrightarrow p'}$$

3. FULL-L-NAT
$$\langle \alpha_{L,j}, p \rangle \twoheadrightarrow \alpha_{L,j} \circ p$$

4. FULL-R-FACT
$$\langle \psi_R(e_1, \ldots, e_m), p \rangle \twoheadrightarrow \psi_R(e_1, \ldots, e_m) \circ p$$
where R is not a unconditioned right object.

5. FULL-C-FACT
$$\frac{\langle e_j, p \rangle \twoheadrightarrow e'_j \quad \text{or} \quad e'_j \equiv e_j \circ E_{C,j}[p/C]}{\langle \psi_C(e_1, \ldots, e_m), p \rangle \twoheadrightarrow \psi_C(e'_1, \ldots, e'_m)}$$
where C is a unconditioned right object and e'_j is either the result of evaluating $\langle e_j, p \rangle$ or $e_j \circ E_{C,j}[p/C]$ depending of whether $E_{C,j}$ is simply C or not.

6. FULL-L-FACT
$$\frac{\langle e_j \circ E_{L,j}[\psi_L(e_1, \ldots, e_m)/L], p \rangle \twoheadrightarrow p'}{\langle \psi_L(e_1, \ldots, e_m), \alpha_{L,j} \circ p \rangle \twoheadrightarrow p'}$$

7. FULL-R-NAT

$$\frac{\langle E'_{R,j}[\psi_R(e_1,\ldots,e_m)/R] \circ e_j, \psi_P(\ldots,p,\ldots)\rangle \Rightarrow\!\!\!\!\Rightarrow p'}{\langle \alpha_{R,j}, \psi_P(\ldots,\psi_R(e_1,\ldots,e_m)\circ p,\ldots)\rangle \Rightarrow\!\!\!\!\Rightarrow p'}$$

In writing down this rule, $\psi_P(\ldots,\psi_R(e_1,\ldots,e_m)\circ p,\ldots)$ is rather inaccurate. It means picking up $\psi_R(e_1,\ldots,e_m)$ according to the occurrence of R in $E_{R,j}$. ψ_P's are nested as productive objects P's are in $E_{R,j}$. For example, the rule for 'pi1' of object 'prod' is

$$\frac{\langle p_1, \mathbf{I}\rangle \Rightarrow\!\!\!\!\Rightarrow p'}{\langle \mathrm{pi1}, \mathrm{pair}(p_1,p_2)\rangle \Rightarrow\!\!\!\!\Rightarrow p'}.$$

$E_{\mathrm{prod},1}$ is simply 'prod', so there is no ψ_C's. The rule for 'eval' of object 'exp' is

$$\frac{\langle e, \mathrm{pair}(\mathbf{I},p)\rangle \Rightarrow\!\!\!\!\Rightarrow p'}{\langle \mathrm{eval}, \mathrm{pair}(\mathrm{curry}(e),p)\rangle \Rightarrow\!\!\!\!\Rightarrow p'}.$$

Remember that $E_{\mathrm{exp},1}$ is 'prod(exp, X)'.

Let us call the new system FULL and the previous system defined in definition 4.1.7 LAZY. □

As we have proved theorem 4.3.1, we can easily show that FULL system is well-defined. In addition, we can show that the reduction in FULL system is stronger than that in LAZY system, that is,

Proposition 4.5.3: If $\langle e,p\rangle \Rightarrow\!\!\!\!\Rightarrow p'$ in FULL system, then $\langle e,p\rangle \Rightarrow c'$ in LAZY system. □

On the other hand, since a FULL reduction is nothing but the repeated application of LAZY reductions, we have the normalization theorem.

Theorem 4.5.4: For a unconditioned canonical element p and a CSL expression e whose domain is compatible with the codomain of p, there is a unconditioned canonical element p' such that

$$\langle e,p\rangle \Rightarrow\!\!\!\!\Rightarrow p'$$

in FULL reduction system. □

Chapter 5

Application of Categorical Data Types

In this chapter we see some applications of CDT and CPL. We have concentrated on category theory in the previous chapters and it is sometimes hard to relate our results to others if they are not familiar with category theory. The author is not claiming that it is better to use category theory in practice. Category theory is used as a guiding principle to see things without being obscured by inessentials. Therefore, once one establishes some results using category theory, it is very interesting to see what it means in other terms and we might get some deep insight.

In section 5.1, we will see an implementation of CPL. In section 5.2, we will examine the connection between CDT and typed lambda calculi and in section 5.3 we will propose a new ML which is obtained by combining the current ML and CPL.

5.1 An implementation of Categorical Programming Language

In chapter 4, we introduced a programming language CPL and its computation rules. A CPL system has been implemented using *Franz Lisp*. In the section, we will demonstrate the system and see some examples of reductions which it can manage.

When the system is started, it prints the following message and waits for user commands.

```
Categorical Programming Language (version 3)
cpl>
```

First, we have to declare some objects because the system does not know any objects when it is started. The very first object we declare is the terminal object. We use `edit` command to enter its declaration.

```
cpl>edit
| right object 1 with !
| end object;
right object 1 defined
cpl>
```

Note that user inputs are in *italic* font. We define products, exponentials and natural number object as well. The declarations are exactly the same as we presented in chapter 3 (except that to make output shorter we use '`s`' for successor and '`0`' for zero).

```
cpl>edit
| right object prod(a,b) with pair is
|   pi1: prod -> a
|   pi2: prod -> b
| end object;
right object prod(+,+) defined
cpl>edit
| right object exp(a,b) with curry is
|   eval: prod(exp,a) -> b
| end object;
right object exp(-,+) defined
cpl>edit
| left object nat with pr is
|   0: 1 -> nat
|   s: nat -> nat
| end object;
left object nat defined
cpl>edit
| left object coprod(a,b) with case is
|   in1: a -> coprod
|   in2: b -> coprod
| end object;
left object coprod(+,+) defined
cpl>
```

Each time we declare an object the system remembers its factorizer and natural transformations as well as the functor associated with. In the above transaction, '`prod(+,+)`' indicates that system recongnized '`prod`' as a covariant functor of two arguments whereas '`exp(-,+)`' indicates that '`exp`' is a functor which is contravariant in the first argument and covariant in the second. The variance is calculated as we formulated in section 3.2. The system can type CSL expressions using the rules in section 2.4. For example, we can ask the type of '`pair(pi2,eval)`'.

```
cpl>show pair(pi2,eval)
pair(pi2,ev)
    : prod(exp(*b,*a),*b) -> prod(*b,*a)
cpl>
```

where '`*a`' and '`*b`' are variables for objects, or we can see them as a kind of type variables in ML; '`pair(pi2,eval)`' is a polymorphic function in this sense.

As we have done in section 4.2, we can ask to the system to calculate '1+1' using '`simp`' command.

```
cpl>simp eval.pair(pr(curry(pi2),curry(s.eval)).pi1,pi2).pair(s.0,s.0)
```

```
s.s.0
    :1 -> nat
cpl>
```

Note that the composition 'o' is typed as '.'. The system applied reduction rules to get the following reduction:

$$\langle \texttt{eval.pair}(\ldots).\texttt{pair}(\texttt{s.0},\texttt{s.0}), \mathbf{I} \rangle \Rightarrow \texttt{s.s.0}.$$

We can see how the system deduced the reduction by enabling the trace mode.

```
cpl>set trace on
cpl>simp eval.pair(pr(curry(pi2),curry(s.eval)).pi1,pi2).pair(s.0,s.0)
0:eval.pair(pr(curry(pi2),curry(s.eval)).pi1,pi2).pair(s.0,s.0)*
1:eval.pair(pr(curry(pi2),curry(s.eval)).pi1,pi2)*pair(s.0,s.0)
2:eval*pair(pr(curry(pi2),curry(s.eval)).pi1,pi2).pair(s.0,s.0)
3[1]:pr(curry(pi2),curry(s.eval)).pi1*pair(s.0,s.0)
4[1]:pr(curry(pi2),curry(s.eval)).s.0*id
5[1]:pr(curry(pi2),curry(s.eval)).s*0
6[1]:pr(curry(pi2),curry(s.eval))*s.0
7[1]:curry(s.eval).pr(curry(pi2),curry(s.eval))*0
8[1]:curry(s.eval).curry(pi2).!*
9[1]:curry(s.eval).curry(pi2)*!
10[1]:curry(s.eval)*curry(pi2).!
11[1]:*curry(s.eval).curry(pi2).!
12:s.eval*pair(curry(pi2).!,pi2.pair(s.0,s.0))
13[1]:curry(pi2).!*
14[1]:curry(pi2)*!
15[1]:*curry(pi2).!
16:s.pi2*pair(!,pi2.pair(s.0,s.0))
17:s.pi2.pair(s.0,s.0)*id
18:s.pi2*pair(s.0,s.0)
19:s.s.0*id
20:s.s*0
21:s*s.0
22:*s.s.0
s.s.0
    :1 -> nat
cpl>
```

Each line has the following form:

$$\boxed{\text{step number}}\,[\,\boxed{\text{depth of computation}}\,]\!:\,\boxed{\text{expression}} * \boxed{\text{canonical element}}$$

It indicates the following reduction:

$$\langle\, \boxed{\text{expression}}\,,\, \boxed{\text{canonical element}}\,\rangle \Rightarrow \ldots$$

Step 0 denotes the reduction of

$$\langle \texttt{eval.pair}(\texttt{pr}(\texttt{curry}(\texttt{pi2}),\texttt{curry}(\texttt{s.eval})).\texttt{pi1},\texttt{pi2}).\texttt{pair}(\texttt{s.0},\texttt{s.0}), \mathbf{I} \rangle \Rightarrow \ldots. \qquad (+)$$

Step 1 is obtained from R-FACT rule (and R-COMP); the reduction (+) is the same as the reduction of

$$\langle \texttt{eval.pair}(\texttt{pr}(\texttt{curry}(\texttt{pi2}),\texttt{curry}(\texttt{s.eval})).\texttt{pi1},\texttt{pi2}),\texttt{pair}(\texttt{s.0},\texttt{s.0}) \rangle \Rightarrow \ldots.$$

Again from R-FACT, this reduction is the same as

$$\langle \mathtt{eval}, \mathtt{pair}(\mathtt{pr}(\mathtt{curry}(\mathtt{pi2}), \mathtt{curry}(\mathtt{s.eval})).\mathtt{pi1}, \mathtt{pi2}).\mathtt{pair}(\mathtt{s.0}, \mathtt{s.0}) \rangle \Rightarrow \ldots. \qquad (++)$$

which is step 2. From R-NAT, we have to calculate

$$\langle \mathtt{pair}(\mathtt{pr}(\mathtt{curry}(\mathtt{pi2}), \mathtt{curry}(\mathtt{s.eval})).\mathtt{pi1}, \mathtt{pi2}).\mathtt{pair}(\mathtt{s.0}, \mathtt{s.0}), \mathtt{prod}(\mathtt{exp}, \mathtt{a}), \mathtt{exp} \rangle \rightsquigarrow \langle \mathtt{curry}(\ldots), \ldots \rangle.$$

In order to do this, from R-NAT-F we have to calculate

$$\langle \mathtt{pr}(\mathtt{curry}(\mathtt{pi2}), \mathtt{curry}(\mathtt{s.eval})).\mathtt{pi1}, \mathtt{pair}(\mathtt{s.0}, \mathtt{s.0}) \rangle \Rightarrow \ldots \qquad (\dagger)$$

This reduction is carried out from step 3 to step 12 and we get

$$\langle \mathtt{pr}(\mathtt{curry}(\mathtt{pi2}), \mathtt{curry}(\mathtt{s.eval})).\mathtt{pi1}, \mathtt{pair}(\mathtt{s.0}, \mathtt{s.0}) \rangle \Rightarrow \mathtt{curry}(\mathtt{s.eval}).\mathtt{curry}(\mathtt{pi2}).!.$$

Note that from step 3 to 4 it did the reduction

$$\langle \mathtt{pi1}, \mathtt{pair}(\mathtt{s.0}, \mathtt{s.0}) \rangle \Rightarrow \mathtt{s.0},$$

and from step 6 to 7 it used L-FACT and did the reduction

$$\frac{\langle \mathtt{curry}(\mathtt{s.eval}).\mathtt{pr}(\mathtt{curry}(\mathtt{pi1}), \mathtt{curry}(\mathtt{s.eval})), \mathtt{0} \rangle \Rightarrow \ldots}{\langle \mathtt{pr}(\mathtt{curry}(\mathtt{pi2}), \mathtt{curry}(\mathtt{s.eval})), \mathtt{s.0} \rangle \Rightarrow \ldots}.$$

Therefore, (†) is

$$\langle \mathtt{pr}(\mathtt{curry}(\mathtt{pi2}), \mathtt{curry}(\mathtt{s.eval})).\mathtt{pi1}, \mathtt{pair}(\mathtt{s.0}, \mathtt{s.0}) \rangle \Rightarrow \langle \mathtt{curry}(\mathtt{s.eval}), \mathtt{curry}(\mathtt{pi2}).! \rangle,$$

and from R-NAT the reduction of (++) is the same as

$$\langle \mathtt{s.eval}, \mathtt{pair}(\mathtt{curry}(\mathtt{pi2}).!, \mathtt{pi2}.\mathtt{pair}(\mathtt{s.0}, \mathtt{s.0})) \rangle \Rightarrow \ldots.$$

The rest of the steps are done similarly.

It is inconvenient to write down the definition of the addition every time we want to add something. Therefore, the system has the facility to give names to morphisms. For example, we can name the addition function '`add`' and use it as follows:

```
cpl>let add=eval.pair(pr(curry(pi2),curry(s.eval)).pi1,pi2)
add : prod(nat,nat) -> nat defined
cpl>simp add.pair(s.0,s.s.0)
s.s.s.0
    :1 -> nat
```

We can define the multiplication and factorial functions.

```
cpl>let mult=eval.prod(pr(curry(0.!),curry(add.pair(eval,pi2))),id)
mult : prod(nat,nat) -> nat defined
cpl>let fact=pi1.pr(pair(s.0,0),pair(mult.pair(s.pi2,pi1),s.pi2))
fact : nat -> nat defined
cpl>simp mult.pair(s.s.0,s.s.s.0)
s.s.s.s.s.s.0
    :1 -> nat
cpl>simp fact.s.s.s.s.0
s.s.s.s.s.s.s.s.s.s.s.s.s.s.s.s.s.s.s.s.s.s.s.s.0.!
    :1 -> nat
cpl>
```

Let us next define the object for lists.

```
cpl>edit
| left object list(p) with prl is
|   nil:1->list
|   cons:prod(p,list)->list
| end object;
left object list(+) defined
cpl>edit
| let append=eval.prod(prl(curry(pi2),
|                          curry(cons.pair(pi1.pi1,eval.pair(pi2.pi1,pi2)))),
|              id);
append : prod(list(*a),list(*a)) -> list(*a) defined
cpl>let reverse=prl(nil,append.pair(pi2,cons.pair(pi1,nil.!)))
reverse : list(*a) -> list(*a) defined
cpl>let hd=prl(in2,in1.pi1)
hd : list(*a) -> coprod(*a,1) defined
cpl>let hdp=case(hd,in2)
hdp : coprod(list(*a),1) -> coprod(*a,1) defined
cpl>let tl=case(in1.pi2,in2).prl(in2,in1.pair(pi1,case(cons,nil).pi2))
tl : list(*a) -> coprod(list(*a),1) defined
cpl>let tlp=case(tl,in2)
tlp : coprod(list(*a),1) -> coprod(list(*a),1) defined
cpl>let seq=pi2.pr(pair(0,nil),pair(s.pi1,cons))
seq : nat -> list(nat) defined
cpl>
```

The morphism '`seq`' returns a list of length n for a given natural number n such that the list consists of the descending sequence of natural numbers, $n-1, n-2, \ldots, 2, 1, 0$. We can try it in the system.

```
cpl>simp seq.s.s.s.0
cons.pair(s.pi1,cons).pair(s.pi1,cons).pair(0,nil).!
    :1 -> list(nat)
cpl>
```

The result dose not look like the sequence of 2, 1 and 0, but this is because our definition of canonical element (definition 4.1.3) is weak. We can ask the system to reduce an element to unconditioned canonical elements (see definition 4.5.1) using reduction rules listed in definition 4.5.2.

```
cpl>simp full seq.s.s.s.0
cons.pair(s.s.0.!,cons.pair(s.0.!,cons.pair(0.!,nil.!)))
    :1 -> list(nat)
cpl>
```

Now, it looks more like the sequence of 2, 1, and 0. We may continue to do some more reductions about lists.

```
cpl>simp hd.seq.s.s.s.0
in1.s.s.0.!
    :1 -> coprod(nat,1)
cpl>simp hd.nil
in2.!
    :1 -> coprod(*a,1)
cpl>simp hdp.tl.seq.s.s.s.0
in1.s.0.!
```

```
    :1 -> coprod(nat,1)
cpl>simp full append.pair(seq.s.s.0,seq.s.s.s.0)
cons.pair(s.0.!,cons.pair(0.!,cons.pair(s.s.0.!,cons.pair(s.0.!,cons.
    pair(0.!,nil.!)))))
    :1 -> list(nat)
cpl>simp full reverse.it
cons.pair(0.!,cons.pair(s.0.!,cons.pair(s.s.0.!,cons.pair(0.!,cons.
    pair(s.0.!,nil.!)))))
    :1 -> list(nat)
cpl>
```

where '`it`' denotes the result of the immediately-preceding reduction.

Let us next experiment with infinite lists.

```
cpl>edit
| right object inflist(a) with fold is
|   head: inflist -> a
|   tail: inflist -> inflist
| end object;
right object inflist(+) defined
cpl>let incseq=fold(id,s).0
incseq : 1 -> inflist(nat) defined
cpl>simp head.incseq
0
    :1 -> nat
cpl>simp head.tail.tail.tail.incseq
s.s.s.0
    :1 -> nat
cpl>let alt=fold(head.pi1,pair(pi2,tail.pi1))
alt : prod(inflist(*a),inflist(*a)) -> inflist(*a)
cpl>let infseq=fold(id,id).0
infseq : 1 -> inflist(nat)
cpl>simp head.tail.tail.alt.pair(incseq,infseq)
s.0
    :1 -> nat
cpl>
```

where '`incseq`' is the infinite increasing sequence 0, 1, 2, 3, 4, ..., and '`infseq`' is the infinite sequence of 0s. We can merge two infinite lists by '`alt`' which picks up elements alternatively from the two infinite lists.

5.2 Typed Lambda Calculus

In this section, we will investigate connection between CPL and typed lambda calculi. Lambda calculi were invented to mathematically formalize the notion of computation. Typed lambda calculi (first order) are an important part of lambda calculi and are studied in various ways. Usually a typed lambda calculus starts with a fixed number of ground types and allows only $\rightarrow$ as type constructors. For example, [Stenlund 72] treats natural numbers and ordinals, and [Troelstra 73] deals with one level higher ordinals. An interesting question is "What kind of types can be added to lambda

calculi?" Natural numbers, ordinals, lists, ... We will show in this section that any data types we can define in CPL can be added into typed lambda calculi.

We are to define a typed lambda calculus. As CPL does not have any ground objects to start with, our lambda calculus does not have any ground types either. Instead it has two ways of constructing types, one corresponding to forming left objects and the other corresponding to forming right objects.

Definition 5.2.1: The syntax of our lambda calculus is given as follows.

1. An enumerable set *TVar* of type variables. $\rho, \nu, \ldots \in \mathrm{TVar}$.

2. The set *Type* of types is defined by the following rules.

$$\frac{\rho \in \mathrm{TVar} \qquad \rho \in \Gamma}{\Gamma \vdash \rho \in \mathrm{Type}} \qquad \frac{\emptyset \vdash \sigma \in \mathrm{Type} \qquad \Gamma \vdash \tau \in \mathrm{Type}}{\Gamma \vdash \sigma \to \tau \in \mathrm{Type}}$$

$$\frac{\Gamma \cup \{\ \rho\ \} \vdash \sigma_1 \in \mathrm{Type} \qquad \ldots \qquad \Gamma \cup \{\ \rho\ \} \vdash \sigma_n \in \mathrm{Type}}{\Gamma \vdash \underline{\mu}\rho.(\sigma_1, \ldots, \sigma_n) \in \mathrm{Type}}$$

$$\frac{\Gamma \cup \{\ \rho\ \} \vdash \sigma_1 \in \mathrm{Type} \qquad \ldots \qquad \Gamma \cup \{\ \rho\ \} \vdash \sigma_n \in \mathrm{Type}}{\Gamma \vdash \overline{\mu}\rho.(\sigma_1, \ldots, \sigma_n) \in \mathrm{Type}}$$

   We use $\sigma, \tau, \ldots$ for the meta-variables of Type. $\underline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)$ corresponds to left objects and $\overline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)$ corresponds to right objects.

3. An enumerable set *Var* of variables. $x, y, z, \ldots \in \mathrm{Var}$.

4. The set *Term* of terms and their types are defined by the following rules.

$$\frac{x \in \mathrm{Var} \qquad x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

$$\frac{\Gamma \cup \{\ x : \sigma\} \vdash t : \tau}{\Gamma \vdash \lambda x^{\sigma}.t : \sigma \to \tau}$$

$$\frac{\Gamma \vdash t_1 : \sigma \to \tau \qquad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash t_1 t_2 : \tau}$$

$$\Gamma \vdash C_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),i} : \sigma_i[\underline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)/\rho] \to \underline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)$$

$$\Gamma \vdash J_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau} : (\sigma_1[\tau/\rho] \to \tau) \to \ldots \to (\sigma_n[\tau/\rho] \to \tau) \to \underline{\mu}\rho.(\sigma_1, \ldots, \sigma_n) \to \tau$$

$$\Gamma \vdash D_{\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),i} : \overline{\mu}\rho.(\sigma_1, \ldots, \sigma_n) \to \sigma_i[\overline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)/\rho]$$

$$\Gamma \vdash P_{\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau} : (\tau \to \sigma_1[\tau/\rho]) \to \ldots \to (\tau \to \sigma_n[\tau/\rho]) \to \tau \to \overline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)$$

   $C_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),i}$ is the i-th constructor of $\underline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)$ and $J_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau}$ is the generalized iterator for it. $D_{\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),i}$ and $P_{\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau}$ are the dual pairs. []

We have the usual reduction rules α and β and two *delta* rules. We write $a \triangleright b$ for the term a reducing to the term b by one step reduction and write $a \overset{\star}{\triangleright} b$ for a reducing to b by some steps. The two delta rules are:

$$J_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau} a_1 \ldots a_n (C_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),i} b) \triangleright a_i (\sigma_i[J_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau} a_1 \ldots a_n/\rho] b)$$

and

$$D_{\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),i} (P_{\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau} a_1 \ldots a_n b) \triangleright \sigma_i[P_{\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau} a_1 \ldots a_n/\rho](a_i b)$$

where $\sigma[t/\rho]$ is a term of type $\sigma[\tau/\rho] \to \sigma[\upsilon/\rho]$ when the type of t is $\tau \to \upsilon$ and is defined as follows.

1. If ρ does not appear in σ, then $\sigma[t/\rho] \equiv \lambda x^{\sigma}.x$.
2. $\rho[t/\rho] \equiv t$.
3. $(\sigma_1 \to \sigma_2)[t/\rho] \equiv \lambda x^{\sigma_1 \to \sigma_2[\tau/\rho]}.\lambda y^{\sigma_1}.\sigma_2[t/\rho](xy)$.
4. $\underline{\mu}\nu.(\sigma_1,\ldots,\sigma_n)[t/\rho] \equiv J_{\underline{\mu}\nu.(\sigma_1[\tau/\rho],\ldots),\underline{\mu}\nu.(\sigma_1[\upsilon/\rho],\ldots)} a_1 \ldots a_n$
   where $a_i \equiv \lambda x^{\sigma_i[\tau/\rho][\underline{\mu}\nu.(\sigma_1[\upsilon/\rho],\ldots)/\nu]}.C_{\underline{\mu}\nu.(\sigma_1[\tau/\rho],\ldots),i}(\sigma_i[\underline{\mu}\nu.(\sigma_1[\upsilon/\rho],\ldots)/\nu][t/\rho]x)$.
5. $\overline{\mu}\nu.(\sigma_1,\ldots,\sigma_n)[t/\rho] \equiv P_{\overline{\mu}\nu.(\sigma_1[\upsilon/\rho],\ldots),\overline{\mu}\nu.(\sigma_1[\tau/\rho],\ldots)} a_1 \ldots a_n$
   where $a_i \equiv \lambda x^{\overline{\mu}\nu.(\sigma_1[\tau/\rho],\ldots)}.\sigma_i[\overline{\mu}\nu.(\sigma_1[\tau/\rho],\ldots)/\nu][t/\rho](D_{\overline{\mu}\nu.(\sigma_1[\tau/\rho],\ldots),i} x)$.

It looks very complicated but this is the faithful translation of $\sigma[t/\rho]$ as σ being a functor.

Let us see some types we can define in our lambda calculus.

Example 5.2.2: The empty type can be defined as $\emptyset \equiv \underline{\mu}\rho.()$, and one point type can be defined as $1 \equiv \overline{\mu}\rho.()$. We denote the element of 1 as $* \equiv P_{1,1\to 1}\lambda x^1.x$. []

Example 5.2.3: The product of two types, σ and τ can be defined as $\sigma \times \tau \equiv \overline{\mu}\rho.(\sigma,\tau)$. We have two projections.

$$\pi_1 \equiv D_{\sigma\times\tau,1} \quad : \sigma \times \tau \to \sigma \qquad\qquad \pi_2 \equiv D_{\sigma\times\tau,2} \quad : \sigma \times \tau \to \tau$$

If a is a term of type σ and b is a term of type τ, we can define a term $\langle a,b\rangle$ of type $\sigma \times \tau$.

$$\langle a,b\rangle \equiv P_{\sigma\times\tau}(\lambda x^1.a)(\lambda x^1.b)* \quad : \sigma \times \tau$$

We have the following reduction.

$$\pi_1\langle a,b\rangle \equiv D_{\sigma\times\tau,1}(P_{\sigma\times\tau,1}(\lambda x.a)(\lambda x.b)*) \triangleright (\lambda x.x)((\lambda x.a)*) \overset{\star}{\triangleright} a$$

Similarly, we can show that $\pi_2\langle a,b\rangle \overset{\star}{\triangleright} b$. []

Example 5.2.4: Dually, the coproduct of σ and τ is defined as $\sigma + \tau \equiv \underline{\mu}\rho.(\sigma,\tau)$. Two injections are defined as follows.

$$\iota_1 \equiv C_{\sigma+\tau,1} \quad : \sigma \to \sigma + \tau \qquad\qquad \iota_2 \equiv C_{\sigma+\tau,2} \quad : \tau \to \sigma + \tau$$

$J_{\sigma+\tau,\nu}$ satisfies the following reductions.

$$J_{\sigma+\tau,\nu} ab(\iota_1 c) \equiv J_{\sigma+\tau,\nu} ab(C_{\sigma+\tau,1} c) \triangleright a((\lambda x.x)c) \triangleright ac$$

$$J_{\sigma+\tau,\nu} ab(\iota_2 c) \overset{\star}{\triangleright} bc \qquad []$$

Example 5.2.5: Let us define the natural numbers in our lambda calculus. The definition of type is

$$\omega \equiv \underline{\mu}\rho.(1, \rho).$$

Zero and the successor function are defined by

$$0 \equiv C_{\omega,1}* \quad : \omega \qquad \mathrm{s} \equiv C_{\omega,2} \quad : \omega \to \omega$$

J gives us almost the ordinary well-known iterator but its type is

$$J_{\omega,\sigma} \quad : (1 \to \sigma) \to (\sigma \to \sigma) \to \omega \to \sigma.$$

We can define the ordinary one by this $J_{\omega,\sigma}$ as follows.

$$\tilde{J}_\sigma \equiv \lambda x.\lambda y.\lambda n.J_{\omega,\sigma}(\lambda z.x)yn \quad : \sigma \to (\sigma \to \sigma) \to \omega \to \sigma$$

It satisfies the usual reductions:

$$\tilde{J}_\sigma ab0 \overset{\star}{\triangleright} J_{\omega,\sigma}(\lambda z.a)b(C_{\omega,1}*) \triangleright (\lambda z.a)((\lambda x.x)*) \overset{\star}{\triangleright} a$$

and

$$\tilde{J}_\sigma ab(\mathrm{s}n) \overset{\star}{\triangleright} J_{\omega,\sigma}(\lambda z.a)b(C_{\omega,2}n) \triangleright b(J_{\omega,\sigma}(\lambda z.a)bn) \approx b(\tilde{J}_\sigma abn)$$

where $\approx$ is the equivalence relation generated by $\overset{\star}{\triangleright}$. Using $\tilde{J}_\sigma$, we can define all the primitive recursive functions. For example, the addition function can be define as

$$\mathrm{add} \equiv \lambda n.\lambda m.\tilde{J}_\omega msn \quad : \omega \to \omega \to \omega. \qquad \square$$

Example 5.2.6: As [Stenlund 72] and [Troelstra 73], we can define the type for ordinals by $\Omega \equiv \underline{\mu}\rho.(1, \omega \to \rho)$. We only check whether our definition of the iterator coincides with the ordinary one.

$$\Omega \equiv \underline{\mu}\rho.(1, \omega \to \rho)$$

$$0_\Omega \equiv C_{\Omega,1}* \quad : \Omega$$

$$\sup \equiv C_{\Omega,2} \quad : (\omega \to \Omega) \to \omega$$

$$J_{\Omega,\sigma} \quad : (1 \to \sigma) \to ((\omega \to \sigma) \to \sigma) \to \Omega \to \sigma$$

$$J_{\Omega,\sigma}(\lambda x.a)b0_\Omega \triangleright (\lambda x.a)((\lambda x.x)*) \overset{\star}{\triangleright} a$$

$$\begin{aligned} J_{\Omega,\sigma}(\lambda x.a)b(\sup t) &\triangleright b((\omega \to \rho)[J_{\Omega,\sigma}(\lambda x.a)b/\rho]t) \\ &\equiv b((\lambda y.\lambda z.J_{\Omega,\sigma}(\lambda x.a)b(yz))t) \triangleright b(\lambda z.J_{\Omega,\sigma}(\lambda x.a)b(tz)) \end{aligned} \qquad \square$$

Example 5.2.7: Finally, the type for finite lists can be defined by

$$L_\sigma \equiv \underline{\mu}\rho.(1, \sigma \times \rho)$$

with

$$\text{nil} \equiv C_{L_\sigma,1} * \quad : L_\sigma \qquad\qquad \text{cons} \equiv C_{L_\sigma,2} \quad : \sigma \times L_\sigma \to L_\sigma$$

$$J_{L_\sigma,\tau} \quad : (1 \to \tau) \to (\sigma \times \tau \to \tau) \to L_\sigma \to \tau$$

whereas the type for infinite lists can be defined by $I_\sigma \equiv \overline{\mu}\rho.(\sigma, \rho)$ with

$$\text{head} \equiv D_{I_\sigma,1} \quad : I_\sigma \to \sigma \qquad\qquad \text{tail} \equiv D_{I_\sigma,2} \quad : I_\sigma \to I_\sigma$$

$$P_{I_\sigma,\tau} \quad : (\tau \to \sigma) \to (\tau \to \tau) \to \tau \to I_\sigma$$

$$\text{head}(P_{I_\sigma,\tau} abc) \overset{*}{\triangleright} ac \qquad\qquad \text{tail}(P_{I_\sigma,\tau} abc) \overset{*}{\triangleright} P_{I_\sigma,\tau} ab(bc) \qquad \square$$

After finishing this section, the author is communicate with [Mendler 86] where recursive types are introduced into first-order and second-order typed lambda calculi. He uses least fixed points and greatest fixed points as we do, but their recursion combinator R has a different type from ours.

$$\frac{M : (\rho \to \tau) \to \sigma \to \tau}{R_{\sigma,\tau}(M[\underline{\mu}\rho.\sigma/\rho]) : \underline{\mu}\rho.\sigma \to \tau}$$

The author cannot give a clear connection between our iterator and his. In addition, he takes fixed points over a single type expression and, therefore, he needs some basic type constructors like 1 and $+$, whereas in our lambda calculus there are no basic type constructors.

5.3 ML and Categorical Programming Language

We might say that ML is based on (first order) typed lambda calculi as we might say that LISP is based on untyped lambda calculi. The type structure of ML depends on the version of ML we are talking about. If we are talking about the original ML developed with LCF [Gordon, Milner and Wordsworth 79], it had some base types, product, disjoint sum, integer, etc. , and had ability to introduce new types via recursively defined type equations. For example, the data type for binary trees whose leaves are integers were defined as

```
absrectype btree = int + (btree # btree)
    with leaf n = absbtree(inl n)
    and node(t1,t2) = absbtree(inr(t1,t2))
    and isleaf t = isl(repbtree t)
    and leafvalue t = outl(repbtree t)
    and left t = fst(outr(repbtree t))
    and right t = snd(outr(repbtree t));;
```

Here, we needed the coproduct type constructor '+' as a primitive. We could not do without it, whereas 'int' can be defined in terms of others primitives (ML has it as a primitive type just because of efficiency).

At the next evolution of ML which yielded the current Standard ML [Milner 84, Harper, MacQueen and Milner 86], we discovered that the coproduct type constructor is no longer needed as a primitive. Standard ML has a 'datatype' declaration mechanism by which the coproduct type constructor can be defined.

```
datatype 'a + 'b = inl of 'a | inr of 'b;
```

A datatype declaration lists the constructors of the defining type. An element of ''a + 'b' can be obtained by either applying 'inl' to an element of ''a' or applying 'inr' to an element of ''b'. We can define the data type for binary trees in Standard ML as follows.

```
datatype btree = leaf of int | node of btree * btree;
```

The symbol '|' is just like '+', but we shifted from the object level of the language to the syntax level. Note that we no longer need the separate definition of 'leaf' or 'node'. We can define the other functions using case statements.

```
exception btree;

fun isleaf t = case t of
                 leaf _ => true
               | node _ => false;

fun leafvalue t = case t of
                    leaf n => n
                  | node _ => raise btree;

fun left t = case t of
               leaf _ => raise btree
             | node(t1,t2) => t1;

fun right t = case t of
                leaf _ => raise btree
              | node(t1,t2) => t2;
```

We got rid of the coproduct type constructor from the primitives, but Standard ML still needs the product type constructor. From a category theoretic point of view, we can sense asymmetry in the type structure of Standard ML. Let us remember that CPL (or the lambda calculus defined in section 5.2) needs neither the coproduct type constructor nor the product type constructor as a primitive. We should be able to introduce the symmetry of CPL into ML. Let us proceed to the next stage of the ML

evolution and define Symmetric ML.

	Primitives	Declaration Mechanism
ML	`->`, `unit`, `#`, `+`	`abstype`
Standard ML	`->`, `unit`, `*`	`datatype`
Symmetric ML	`->`	`datatype`, `codatatype`
CPL		`left object`, `right object`

ML Evolution

Remember that datatype declarations correspond to left object declarations. We list constructors for types. In order to get rid of the product type constructor from primitives, we should have a declaration mechanism which corresponds to the right object declaration mechanism. Its syntax is

`codatatype` *TypeParam TypeId* `=`
Id `is` *TypeExp* `&` ... `&` *Id* `is` *TypeExp*`;`

A codatatype declaration introduces a type by listing its destructors. The product type constructor can be defined as follows.

```
codatatype 'a * 'b = fst is 'a & snd is 'b;
```

where '`fst : 'a * 'b -> 'a`' gives the projection function to the first component and '`snd : 'a * 'b -> 'b`' gives the projection function to the second component. If the declaration is recursive, we do not take the initial fixed point of the type equation but the final fixed point. This is firstly because of symmetry and secondly because the initial fixed points are often trivial. Because of this, we can define infinite objects by codatatype declarations. For example, the following declaration gives us the data type for infinite lists.

```
codatatype 'a inflist = head is 'a & tail is 'a inflist;
```

If we took the initial fixed point, we would get the empty data type.

Obviously we have destructors for co-data types because we declare them, but how can we construct data for co-data types? We had `case` statements for data types, so we have 'merge' statements as dual. Its syntax is

`merge` *Destructor* `<=` *Exp* `&` ... `&` *Destructor* `<=` *Exp*

For example, the function '`pair`' which makes a pair of given two elements can be defined as follows.

```
fun pair(x,y) = merge fst <= x & snd <= y;
```

As a more complicated example, we might define a function which combines two infinite lists together.

```
fun comb(l1,l2) = merge head <= head l1
                        & tail <= comb(l2,tail l1);
```

It is now clear that, if elements of co-data types are just records and 'merge' creates records after evaluating expressions, this 'comb' function never terminates because it tries to sweep the entire infinite lists which cannot be done in finite time. We need lazyness in the evaluation mechanism. An element of 'inflist' is a record of two components but each component is a closure whose computation leads to a value. A 'merge' statement creates a record consisting of these records. Therefore, the declaration of 'inflist' is not like

```
datatype 'a inflist = something of 'a * 'a inflist;
```

but is closer to

```
datatype 'a inflist = something of (unit -> 'a) *
                                   (unit -> 'a inflist);
```

and 'head', 'tail and 'comb' are like

```
fun head(something(x,l)) = x();

fun tail(something(x,l)) = l();

fun comb(l1,l2) = something(fn () => head l1,
                            fn () => comb(l2,tail l1));
```

Note that, as we use pattern matching to declare functions over data types, we can also use it to declare functions over co-data types. For example, an alternative definition of 'comb' may be

```
fun head comb(l1,_) = head l1
  & tail comb(l1,l2) = comb(l2,tail l1);
```

Conclusions

We have looked at a categorical approach to the theory of data types. The goal of this thesis was to develop CPL (Categorical Programming Language) which is a programming language in a categorical style and which has a categorical way of defining data types.

CSL (Categorical Specification Language) was actually developed later than CDT (Categorical Data Types) and CPL. At first, CDT was given its semantics without depending on CSL. We could have carried out the thesis without CSL, but CSL provides the syntactic materials for CDT and CPL so we would have still needed those parts. CSL is very much like an ordinary algebraic specification language, but it is not trivial in two senses: the treatment of functors and the treatment of natural transformations. Functors are very similar to functions but variances make them special and interesting. Natural transformations are essentially polymorphic functions, so if there had been a specification language for polymorphic functions, we might not have needed to struggle for developing CSL. It might be interesting to investigate what *polymorphic algebraic specification languages* can be.

CSL is equational. Much of category theory can be presented equationally so that CSL is good enough in this sense, but presenting categorical concepts equationally loses half of the essential meaning. For example, although the adjoint situation can be explained equationally, its essence is something more. This is why, the author believes, there are so many equivalent forms of defining the adjoint situation. Therefore, it is nice to have a specification language which can naturally express categorical concepts. Sketches [Barr and Wells 85] are more categorical than equations, so it might be an idea to use sketches in CSL.

CDT is the heart of the thesis. It was developed after the author first studied category theory and tried to express categorical definitions in algebraic specification languages. As we have seen in chapter 3, algebraic specification languages can express categorical definitions but not naturally. CDT succeeded to define some basic categorical definitions like products, coproducts, exponentials, natural numbers and so on more naturally, but it cannot define, for example, pullbacks or more complicated categorical concepts. One of the suggestions to extend CDT is to allow equations inside the CDT

declarations. In this way, we may define pullbacks as follows:

$$
\begin{array}{l}
\text{right object pullback}(f\colon A \to C, g\colon B \to C) \text{ with pbpair is} \\
\quad \pi_1\colon \text{pullback} \to A \\
\quad \pi_2\colon \text{pullback} \to B \\
\quad \text{where} \\
\quad\quad f \circ \pi_1 = g \circ \pi_2 \\
\text{end object}
\end{array}
$$

The declaration should be read as follows:

1. For any morphisms $f\colon A \to C$ and $g\colon B \to C$, pullback(f, g) is an object and it is associated with two morphisms

$$\pi_1\colon \text{pullback}(f,g) \to A \qquad \text{and} \qquad \pi_2\colon \text{pullback}(f,g) \to B$$

   such that $f \circ \pi_1 = g \circ \pi_2$.
2. For any morphisms $h\colon D \to A$ and $k\colon D \to B$ such that $f \circ h = g \circ k$, there exists a unique morphism pbpair$(h, k)\colon D \to$ pullback(f, g) such that

$$\pi_1 \circ \text{pbpair}(h,k) = h \qquad \text{and} \qquad \pi_2 \circ \text{pbpair}(h,k) = k$$

Note that 'pullback' is no longer a simple functor but takes two morphisms. We can similarly define pushouts, equalizers, co-equalizers and so on. In fact, we can define any finite limit or colimit. Since limits and colimits are something to do with diagrams, it seems natural to introduce the declaration mechanism of diagrams. For example, we may have a diagram consisting of three objects and two morphisms as follows:

$$
\begin{array}{l}
\text{diagram el is} \\
\quad \text{objects } A, B, C \\
\quad \text{morphisms } f\colon A \to C, g\colon B \to C \\
\text{end diagram}
\end{array}
$$

Then, 'pullback' can be regarded as taking an 'el' diagram as its parameter, and it is a functor from the category of 'el' diagrams. This extension is becoming very much similar to the parametrization mechanism in algebraic specification languages. Diagrams correspond to so-called loose specifications, and object declarations correspond to parametrized specifications (or procedures in CLEAR's terminology) which take a specification which matches as a parameter and return a new specification. It is very interesting to investigate the possibility of CDT with equations along this line as a first class specification language. CDT we presented in this thesis was bounded by the restriction of computability. If we introduce equations, it becomes increasingly difficult to connect them to computing. If we had 'pullback' in CDT, we would have to prove $f \circ k = g \circ h$ before using pbpair(k, h). Therefore, the programming would involve some proving.

CDT and CSL are essentially one sorted systems (here sort = category), and some would like to extend them to many sorted systems. We could have extended them here, but since our main goal in this thesis was to understand data types, we were

interested in only one category, the category of data types, and so CDT and CSL were single sorted. Our approach is very close to that of domain theory which mainly deals only one category, the category of domains. On the other hand, algebraic specification methods deal with many categories. Each specification is associated with a category. However, they are still related in some sense because they all are algebras over the category of sets (or some other underlying category). As we mentioned above, if we extend CDT with the diagram declarations, we will have to deal with a lot of different categories of diagrams, and it will be interesting to find out what F, G-dialgebras can give us in this context.

CPL is a functional programming language without variables. It may look like FP proposed by John Backus because FP also has no variables. However, CPL is based on category theory and it has an ability to declare data types by means of CDT. CPL does not need any primitives to start with. One of the reasons for not having variables is that category theory is abstract in the sense that objects are simply points and only their outer behaviour is concerned. However, we could have variables for morphisms. For example, we might want to have

$$\text{twice}(f) \stackrel{\text{def}}{=} f \circ f$$

which takes a morphism $f\colon A \to A$ and returns a morphism of $A \to A$. The current CPL system cannot handle it and we have to write it like

$$\text{twice} \stackrel{\text{def}}{=} \text{eval} \circ \text{pair}(\pi_1, \text{eval})$$

which is a morphism from $\exp(A, A)$ to $\exp(A, A)$. This definition is not self-explanatory. It is evident that we need morphism variables in CPL for easier use. Note that $\text{twice}(f)$ can simply be a macro because definitions can never be recursive.

We proposed in chapter 5 to make CPL more like an ordinary functional programming language. It has datatype declarations as well as co-datatype declarations. It is left for the future to actually implement the language. It is interesting to see how to handle (or represent) lazy data types.

Since CPL is an applicative language and has the possibility of executing programs in parallel as well as the possibility of partial evaluation, some kind of special hardware can be invented to execute CPL programs fast.

The future plan of CDT and CPL would be to extend CDT to cope with equations and to develop a total programming environment in which users can define things categorically, reason (or prove) their properties categorically, execute some programs categorically.

Bibliography

[Arbib and Manes 75] Arbib, M. A. and Manes, E. G. (1975): *Arrows, Structures, and Functors — The Categorical Imperative —*. Academic Press.

[Arbib and Manes 80] Arbib, M. A. and Manes, E. G. (1980): The Greatest Fixed Points Approach to Data Types. In *proceedings of Third Workshop Meeting on Categorical and Algebraic Methods in Computer Science and System Theory*, Dortmund, West Germany.

[Barr and Wells 85] Barr, M. and Wells, C. (1985): *A Series of Comprehensive Studies in Mathematics* Volume 278: *Toposes, Triples and Theories*. Springer-Verlag.

[Burstall and Goguen 77] Burstall, R. M. and Goguen, J. A. (1977): Putting Theories Together to Make Specification. In *Proceedings of 5th International Joint Conference on Artificial Intelligence*. pp. 1045–1058.

[Burstall and Goguen 80] Burstall, R. M. and Goguen, J. A. (1980): The Semantics of Clear: A Specification Language. Internal Report CSR-65-80, Department of Computer Science, University of Edinburgh.

[Burstall and Goguen 81] Burstall, R. M. and Goguen, J.A. (1981): An Informal Introduction to Specifications using Clear. In *The Correctness Problem in Computer Sciences*, Academic Press, pp. 185–213.

[Burstall and Goguen 82] Burstall, R. M. and Goguen, J.A. (1982): *Algebras, Theories and Freeness: an Introduction for Computer Scientists.* Internal Report CSR-65-80, Department of Computer Science, University of Edinburgh.

[Burstall and Lampson 84] Burstall, R. M. and Lampson, B. (1984): A Kernel Language for Abstract Data Types and Modules. In *Lecture Notes in Computer Science*, Volume 173, pp. 1–50.

[Curien 86] Curien, P-L. (1986): *Categorical Combinators, Sequential Algorithms and Functional Programming*. Research Notes in Theoretical Computer Science, Pitman.

[Dybjer 83] Dybjer, P. (1983): Category-Theoretic logics and Algebras of Programs. Ph. D. thesis, Department of Computer Sciences, Chalmers University of Technology. University of Gothenburg.

[Goguen and Burstall 83] Goguen, J. A. and Burstall, R. M. (1983): Introducing Institutions. In *proceedings of Logic of Programming Workshop*.

[Goguen and Tardo 79] Goguen, J. A. and Tardo, J. (1979): An Introduction to OBJ: A Language for Writing and Testing Software Specifications. In *Specification of Reliable Software*, IEEE, pp. 170–189.

[Goguen, Thatcher and Wagner 78] Goguen, J. A., Thatcher, J. W. and Wagner, E. G. (1978): An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types. In *Current Trends in Programming Methodology*, prentice-Hall, pp. 80–149.

[Gordon, Milner and Wordsworth 79] Gordon, M. J., Milner, A. J. and Wordsworth, C. P. (1979): *Edinburgh LCF. Lecture Notes in Computer Science*, Volume 78.

[Goldblatt 79] Goldblatt, R. (1979): *Topoi: The Categorical Analysis of Logic.* Studies in Logic and Foundation of Mathematics, Volume 98, North-Holand.

[Harper, MacQueen and Milner 86] Harper, R., MacQueen, D. and Milner, R. (1986): *Standard ML.* LFCS Report Series, ECS-LFSC-86-2. Department of Computer Science, University of Edinburgh.

[Kelly 72] Kelly, G. M. (1972): Many-Variable Functorial Calculus I. In *Lecture Notes in Mathematics*, Volume 281, Springer-Verlag, pp. 66–105.

[Lambek and Scott 86] Lambek, J. and Scott, P. J. (1986): *Introduction to Higher-Order Categorical Logic. Cambridge Studies in Advanced Mathematics*, Volume 7.

[Lawvere 63] Lawvere, F. W. (1963): Functorial Semantics of Algebraic Theories. In *Proceedings of the National Academy of Science*, Volume 50, pp. 869–872.

[Lehmann and Smyth 81] Lehmann, D. and Smyth, M. (1981): Algebraic Specification of Data Types – A Synthetic Approach –. *Mathematical System Theory*, Volume 14, pp. 97–139.

[Mac Lane 71] Mac Lane, S. (1971): *Categories for the Working Mathematician. Graduate Texts in Mathematics 5*, Springer-Verlag.

[Manes 76] Manes, E. G. (1976): *Algebraic Theories. Graduate Texts in Mathematics 26*, Springer-Verlag.

[Martin-Löf 79] Martin-Löf, P. (1979): Constructive Mathematics and Computer Programming. Paper presented in 6th International Congress for Logic, Methodology and Philosophy of Science.

[Mendler 86] Mendler N. P. (1986): *First- and Second-Order Lambda Calculi with Recursive Types.* Technical Report TR 86-764, Department of Computer Science, Cornell University.

[Milner 84] Milner, R. (1984): *The Standard ML Core Language.* Internal Report CSR-168-84, Department of Computer Science, University of Edinburgh.

[Parasaya-Ghomi 82] Parasaya-Ghomi, K. (1982): Higher Order Abstract Data Types. Ph. D. Thesis, Department of Computer Science, UCLA.

[Plotkin 81] Plotkin, G. D. (1981): *A Structural Approach to Operational Semantics.* Technical Report DAIMI FN–19, Computer Science Department, Århus University.

[Rydeheard 81] Rydeheard, D. E. (1981): Application of Category Theory to Programming and Program Specification. Ph. D. thesis, University of Edinburgh.

[Rydeheard and Burstall 86] Rydeheard, D. E. and Burstall, R. M. (1986): *Computational Category Theory.*

[Scott 76] Scott, D. (1976): Data Types as Lattices. *SIAM Journal of Computing*, Volume 5, pp. 552–587.

[Smyth and Plotkin 82] Smyth, M. B. and Plotkin, G. D. (1982): The Category-Theoretic Solution of Recursive Domain Equations. *SIAM Journal of Computing*, Volume 11.

[Stenlund 72] Stenlund, S. (1972): *Combinators, λ-Terms and Proof Theory.* D. Reidel, Dordrecht.

[Stoy 77] Stoy, J. E. (1977): *The Scott-Strachey Approach to Programming Language Theory.* MIT Press.

[Tait 67] Tait, W. (1967): Intentional Interpretation of Functionals of Finite Type I. *Journal of Symbolic Logic*, 32, pp. 198–212.

[Troelstra 73] Troelstra, A. S. (1973): *Mathematical Investigation of Intuitionistic Arithmetic and Analysis. Lecture Notes in Mathematics*, Volume 344, Springer-Verlag.

Declaration

This thesis has been written by myself, and the work is my own.

Edinburgh, 1 June 1987

Tatsuya Hagino