ADVANCES IN SOFTWARE SCIENCE AND TECHNOLOGY

Volume 2

JAPAN SOCIETY FOR SOFTWARE SCIENCE AND TECHNOLOGY

ADVANCES IN SOFTWARE SCIENCE AND TECHNOLOGY
VOLUME 2

JAPAN SOCIETY FOR SOFTWARE SCIENCE AND TECHNOLOGY

ACADEMIC PRESS, INC.

Harcourt Brace Jovanovich, Publishers

Boston San Diego New York
London Sydney Tokyo Toronto

Co-published for
Japan Society for Software Science and Technology
by Academic Press, Inc.
and
Iwanami Shoten, Publishers

This book is printed on acid-free paper. ∞

ACADEMIC PRESS, INC.
1250 Sixth Avenue, San Diego, CA 92101

United Kingdom Edition published by
ACADEMIC PRESS LIMITED
24–28 Oval Road, London NW1 7DX

Library of Congress Catalog Card Number: 90–660056

ISBN 0–12–037102–2
ISSN 1044–7997

AP Exclusive Sales Territory: United States, its territories and
dependencies, Canada, and Europe.
Iwanami Shoten Exclusive Sales Territory: Japan.
Nonexclusive sales rights throughout the rest of the world.

Printed in the United States of America

91 92 93 9 8 7 6 5 4 3 2 1

ADVANCES IN SOFTWARE SCIENCE AND TECHNOLOGY

VOLUME 2

JAPAN SOCIETY FOR SOFTWARE SCIENCE AND TECHNOLOGY
World Trade Center Building 7F, 2–4–1 Hamamatsu-cho
Minato-ku, Tokyo, 105 Japan

Contributors

Numbers in parentheses refer to the pages on which the authors' contributions begin.

Tsuneo Ajisaka (103), Department of Information Science, Kyoto University, Yoshida, Hon-machi, Sakyo, Kyoto, 606 Japan

Kunikazu Fujii (45), IBM Research, Tokyo Research Laboratory, IBM Japan, 5–11 Sambancho, Chiyoda-ku, Tokyo, 102 Japan

Yasunori Harada (153), Division of Information Engineering, Faculty of Engineering, Hokkaido University, N13–W8, Kita-ku, Sapporo, 060 Japan

Tsunetoshi Hayashi (197), Department of Computer Science and Systems Engineering, Faculty of Science and Engineering, Ritsumeikan University, 56–1 Tojiin-kitamachi, Kita-ku, Kyoto, 603 Japan

Ken Hirose (177), Department of Mathematics, Waseda University, 3–4–1 Ohkubo, Shinjuku-ku, Tokyo, 160 Japan

Hiroshi Horiguchi (123), Department of Mathematics, Tokyo Denki University, 2–2 Nishiki-cho Kanda, Chiyoda-ku, Tokyo, 101 Japan

Kazuaki Kajitori (123), Department of Mathematics, Tokyo Denki University, 2–2 Nishiki-cho Kanda, Chiyoda-ku, Tokyo, 101 Japan

Satoshi Kinoshita (61), Toshiba Corporation Research and Development Center, 1 Komukai-Toshiba-cho, Saiwai-ku, Kawasaki-shi, 210 Japan

Yoshihiro Matsumoto (103), Department of Information Science, Kyoto University, Yoshida, Hon-machi, Sakyo, Kyoto, 606 Japan

Kazufumi Mitani (153), Division of Information Engineering, Faculty of Engineering, Hokkaido University, N13–W8, Kita-ku, Sapporo, 060 Japan

Eiichi Miyamoto (153), Division of Information Engineering, Faculty of Engineering, Hokkaido University, N13–W8, Kita-ku, Sapporo, 060 Japan

Yoshio Ohno (13), Faculty of Science and Technology, Keio University, 3–14–1 Hiyoshi, Kohoku-ku, Yokohama, 223 Japan

Yasubumi Sakakibara (79), International Institute for Advanced Study of Social Information Science (IIAS-SIS), Fujitsu Limited, 140 Miyamoto, Numazu, Shizuoka, 410–03 Japan

Hiroyuki Sato (1), Department of Information Science, Faculty of Science, The University of Tokyo, 7–3–1 Hongo, Bunkyo-ku, Tokyo, 113 Japan

Masayuki Takeda (131), Department of Electrical Engineering, Kyushu University 36, Fukuoka, 812 Japan

Hozumi Tanaka (61), Department of Computer Science, Tokyo Institute of Technology, Ookayama 2–12–1, Meguro-ku, Tokyo, 152 Japan

Tsutomu Tayama (123), Department of Mathematics, Tokyo Denki University, 2–2 Nishiki-cho Kanda, Chiyoda-ku, Tokyo, 101 Japan

Jun'ichi Toyoda (45), The Institute of Scientific and Industrial Research, Osaka University, 8–1 Mihogaoka, Ibaragi-shi, Osaka, 567 Japan

Kuniaki Uehara (45), Department of Systems Engineering, Faculty of Engineering, Kobe University, Rokkodaicho, Nada-ku, Kobe, 657 Japan

Shin-ya Watanabe (153), Division of Information Engineering, Faculty of Engineering, Hokkaido University, N13–W8, Kita-ku, Sapporo, 060 Japan

Editor's Message

Ikuo Nakata
Editor-in-Chief

This is the second issue of *Advances in Software Science and Technology*, an annual journal published by the Japan Society for Software Science and Technology (JSSST). The Society was founded in 1983 as a professional society dedicated to the advancement of the science and technology of computer software.

Unparalleled progress in hardware technology has been a driving force in modern computer technology. It has dramatically improved the performance and reliability, increased the level of complexity and sophistication, and created numerous new applications for computer systems. Progress in software technology, on the other hand, has been much more conservative. By and large, the volume and the quality of current software production depend on the skills and dedicated craftsmanship of programmers. With ever-increasing demand for software production, our ability to build and use computer systems is now limited mainly by our ability to produce software.

Advancing software technology requires active research efforts toward scientific understanding of software systems, organized efforts to improve the current practice of software production, and drastic improvement of software education and training programs. The JSSST was founded to provide leadership, to promote and exchange ideas and experience, and to develop and organize concerted efforts in this direction.

The society has published a domestic bimonthly journal, *Computer Software*, since 1984. This contains original technical contributions that are refereed by the normal scientific review process. In addition, it contains survey papers, tutorials, conference reports, and miscellaneous articles. The journal covers a broad range of computer software. Topics featured in recent issues include algorithms, theory of programs, programming languages and methodology, operating systems, computer architecture, software engineering, artificial intelligence, and natural language processing.

Advances in Software Science and Technology is the second journal published by the JSSST. It is an annual publication with the same scope as *Computer Software*, and is intended to give international exposure to the activities of JSSST and to promote exchange of ideas and information among professionals and the public world-wide. Each issue of the journal contains original technical contributions as well as contributions that have appeared in previous issues of *Computer Software* in Japanese. The JSSST forms a special editorial committee for each issue of this journal; members of the committee for the second issue are listed in the front page together with those of *Computer Software*.

Like the previous issue of the journal, this issue describes a variety of activities, primarily in Japan. Software problems, however, are something we must all face

today; and international collaboration and exchange are absolutely necessary. We very much look forward to publishing contributions from a wide variety of authors in future issues.

Contents

Tutorial

Software Critique

Society News

Attachment of a First-Order Data Constructor and Its Application

Hiroyuki Sato

Summary. Today, categorical frameworks are widely used to represent datatypes in computer science. In order to provide simple and uniform representation, this article introduces a first-order data constructor. A first-order datatype is constructed as the left adjoint to the corresponding diagram. In **Set**, the framework implements the abstract datatype theory. In our approach, induction steps on an abstract datatype are separated from the induction basis. This provides a simpler representation of parameterized types than the universal algebra approach. Moreover, a first-order datatype is proved to be the initial solution of a certain domain equation. Using these constructions, we also apply conditional expressions to first-order datatypes.

1 Introduction

Today, categorical frameworks are widely used to represent datatypes in computer science. One standard framework is the cartesian closed category (ccc), which is exactly the typed λ-calculus [7]. Because of its simple machinery, ccc serves as the engine of a number of computation systems [1, 5]. It is, however, too weak to express some important concepts in computer science such as subtypes and abstract datatypes.

To increase its expressive power, we have taken various approaches. One is to add ad hoc objects, morphisms, and axioms. Another is to add systematically constructors of morphisms and objects such as equalizers and adjoints. The former corresponds, in λ-calculus, to δ-rules. This approach allows arbitrary discussion, but it destroys much of the effect of introducing the framework of category theory.

Systematic construction of objects and morphisms allows the simple and uniform extension of ccc, because we can still work in the framework of category theory. The expressive power of ccc-engines such as CAM [1] can systematically be strengthened along these lines. CPL [4] is a typical example. It uses the theory of (co-)algebras and provides a general framework. It defines objects and morphisms by using initiality or finality in a specially-structured category.

This article presents yet another categorical framework. First-order diagrams are defined and free constructions are discussed. Our approach is compatible with ccc-engines such as CAM. Our concern lies mainly in the categorical definition of datatypes, while CAM utilizes ccc as a reduction engine, not as a datatype definition scheme.

ISBN 0-12-037102-2

From this viewpoint, our approach resembles CPL. CPL gives a general framework for a datatype definition scheme that is closely connected with the way of computation. As regards its relation to computation, our approach is to first present a domain equation for a given diagram. By using this equation, we can define conditional expressions on the constructed type. In CPL, conditional expressions are defined on sum types. With our method, they can also be applied to abstract types. With this simple machinery, we obtain equally strong expressive power.

Another advantage of our approach is that we separate the induction steps from the induction basis. In the theory of universal algebra, the two are not treated separately. Our diagram exactly represents the induction step for the constructed type. This separation is suited, for example, to the definition of parameterized types.

In Section 3, first-order diagrams are defined. Datatype constructors are defined by means of first-order diagrams, and some examples are given in Section 4. The relation to existing theory is discussed in Section 5. Section 6 defines a domain equation for a datatype constructor. Section 7 compares our construction with CPL and interprets conditional expressions as an application of our construction.

2 Preliminaries

This section gives definitions and notations used in subsequent sections. Details are available elsewhere [9, 3].

Notation 1. Given a category $\mathcal{C}$, $o\mathcal{C}$ denotes its class of objects. The class of morphisms from object a to b is denoted by $\mathcal{C}(a, b)$.

Notation 2. To denote that $f \in \mathcal{C}(a, b)$, we sometimes write $dom(f) = a$ and $codom(f) = b$.

Notation 3. Given $f_0 : V_0 \longrightarrow W, \cdots, f_{n-1} : V_{n-1} \longrightarrow W$, we denote by $[f_i]_{0 \leq i < n}$ the naturally defined morphism to W from $\coprod_{0 \leq i < n} V_i$.

$$\begin{array}{ccc} V_i & \overset{injection}{\longrightarrow} & V_0 + \cdots + V_{n-1} \\ & \searrow & \downarrow [f_i]_{0 \leq i < n} \\ & f_i & W \end{array}$$

Definition 1 (adjoint)**.** *Let two categories $\mathcal{C}$ and $\mathcal{D}$, two functors $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{C}$ be given.*

A triple $\langle F, G, \varphi \rangle$ is an adjunction if φ is a natural isomorphism from $\mathcal{D}(-, G-)$ to $\mathcal{C}(F-, -)$.

We denote φ's a, b-th component $\mathcal{C}(a, Gb) \longrightarrow \mathcal{D}(Fa, b)$ by $\varphi_{a,b}$. We drop subscripts if they are clear from the context.

To express that the triple $\langle F, G, \varphi \rangle$ is an adjunction, we use the following diagram:

$$\begin{array}{ccc} a & \overset{\eta_a}{\longrightarrow} & GF(a) \\ & \searrow & \downarrow G\varphi_{a,b}(f) \\ & f & G(b) \end{array} \qquad \Bigg| \qquad \begin{array}{c} F(a) \\ \downarrow \varphi_{a,b}(f) \\ b \end{array}$$

In the above figure, $\eta_a = \varphi^{-1}(id_{F(a)})$. We denote morphisms that are defined by using φ by means of naturally defined morphisms.

3 First-Order Diagrams and Data Constructors

Definition 2. *A (first-order) diagram D is a triple $\langle S_D, T_D, A_D\rangle$ such that*[1]*:*

1. S_D is the set of *sorts*.
2. T_D is the set of *operators*. Every operator $f \in T_D$ is associated with a unique arity n_f, a natural number, and a finite sequence $L_f = (s^f_0, \ldots, s^f_{n_f-1}, s^f_{n_f})$ of sort of length $n_f + 1$. The *sort* of f denotes this sequence. $f\colon L$ indicates that the sequence L is the sort of f. We say that the *domain* of f is $(s^f_0, \ldots, s^f_{n_f-1})$ and that its *codomain* is $s^f_{n_f}$.
3. A_D is the set of *axioms*. Axioms are written in equalities of *terms*. Terms are constructed by operators in T_D and projections together with compositions.

Notation 4. We write $s \in L$ to denote that the sequence L contains s as a component.

Definition 3. *Let $\mathcal{C}$ be a category with finite products, and $D = \langle S_D, T_D, A_D\rangle$ a diagram. The category $\mathcal{C}^D$ is defined as follows:*

1. $o(\mathcal{C}^D)$ is the whole class of translations F that send a sort $s \in S_D$ to an object $F(s)$ of $\mathcal{C}$ and an operator $f\colon (s^f_0, \ldots, s^f_{n_f-1}, s^f_{n_f}) \in T_D$ to a morphism $F(f)$ with domain $F(s^f_0) \times \cdots \times F(s^f_{n_f})$ and codomain $F(s^f_{n_f})$.

$$(s^f_0, \ldots, s^f_{n_f-1}) \quad \xrightarrow{f} \quad s^f_{n_f}$$

$$\begin{array}{ccc} F(s^f_0) \times \cdots \times F(s^f_{n_f-1}) & \xrightarrow{F(f)} & F(s^f_{n_f}) \\ \Big\downarrow{\scriptstyle a_{s^f_0} \times \cdots \times a_{s^f_{n_f-1}}} & & \Big\downarrow{\scriptstyle a_{s^f_{n_f}}} \\ G(s^f_0) \times \cdots \times G(s^f_{n_f-1}) & \xrightarrow{G(f)} & G(s^f_{n_f}) \end{array}$$

   This translation F is naturally extended to the translation $\tilde{F}$ on *terms*. $\tilde{F}$ translates the composition and projection to the composition and projection in the category $\mathcal{C}$.
2. For $F, G \in o(\mathcal{C}^D)$, a morphism $a \in \mathcal{C}^D(F, G)$ is a family $\{a_s\colon F(s) \to G(s) | s \in S\}$ of morphisms of $\mathcal{C}$ such that $a_{s^f_{n_f}} \circ F(f) = G(f) \circ (a_{s^f_0} \times \cdots \times a_{s^f_{n_f-1}})$. The composition $b \circ a$ of two morphisms $a = \{a_s\colon F(s) \to G(s) | s \in S\}$ and $b = \{b_s\colon G(s) \to H(s) | s \in S\}$ is defined as $\{b_s \circ a_s\colon F(s) \to H(s) | s \in S\}$.
3. Let $t_0 = t_1 \in A_D$. $F \in o(\mathcal{C}^D)$ must satisfy $\tilde{F}(t_0) = \tilde{F}(t_1)$ in $\mathcal{C}$.

[1] In fact, D can be regarded as a category with generators S_D, T_D, and axioms A_D. $\mathcal{C}^D$ is then a category of product-preserving functors from D to $\mathcal{C}$. However, we prefer the notation of this definition because the important point is that D is a collection of generators, not that it can be regarded as a category.

Definition 4. *First-order objects for a diagram D in a category $\mathcal{C}$ are defined as objects of $\mathcal{C}^D$.*

Notation 5. We write $\{(F(s)), (F(f))\}_{s\in S_D, f\in T_D}$ instead of $F \in o\mathcal{C}^D$. If the set of operators is empty, we simply write $\{(F(s))\}_{s\in S_D}$.

Definition 5. *For a diagram $D = \langle S_D, T_D, A_D\rangle$, its underlying diagram is defined by $\bar{D} = \langle S_D, \emptyset, \emptyset\rangle$,*

Definition 6. *Given a diagram D, the underlying functor $U_D\colon \mathcal{C}^D \longrightarrow \mathcal{C}^{\bar{D}}$ is defined in the usual way: let $F \in o(\mathcal{C}^D)$. $U_D(F)$ is the translation that sends $s \in S_D$ to $F(s)$.*

We come to the definition of a data constructor:

Definition 7. *A first-order data constructor of a diagram D is a pair $\langle F_D, rec_D\rangle$ such that the triple $\langle F_D, U_D, rec_D\rangle$ is an adjunction.*

4 Examples

This section gives some examples and shows that the definitions in the last section are reasonable. In the rest of this section, we fix an arbitrary category $\mathcal{C}$ closed under finite products.

4.1 Lists

Let the diagram D_L be $\langle S_L, T_L, \emptyset\rangle$ with $S_L = \{s_0, s_1\}$ and $T_L = \{t\colon (s_0, s_1, s_1)\}$

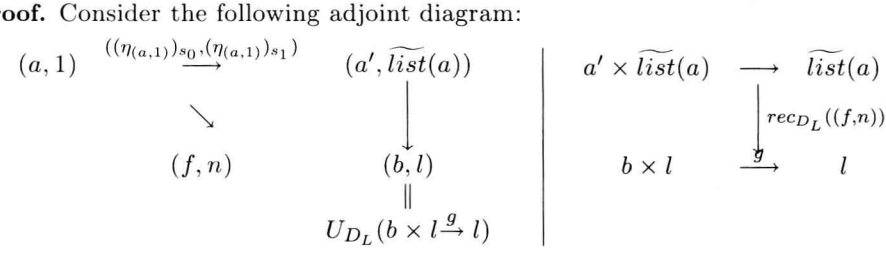
Notation 6. We write $a_0 \times a_1 \xrightarrow{c} a_1$ to denote $\{(a_0, a_1), (c)\} \in o(\mathcal{C}^{D_L})$. Let α be a morphism of $\mathcal{C}^{D_L}$. It is also denoted by $(\alpha_{s_0}, \alpha_{s_1})$.

Let the first order data contructor $\langle F_{D_L}, rec_{D_L}\rangle$ exist.

Notation 7. We denote $U_{D_L}F_{D_L}(\{(a, 1)\})$ by $(a', \widetilde{list}(a))$.

Lemma 1. $a' \cong a$

Proof. Consider the following adjoint diagram:



We construct another left adjoint F'_{D_L} by using, instead of $\eta_{(a,1)}$ and rec_{D_L}, $(id_a, (\eta_{(a,1)})_{s_1})$ and $rec'_{D_L} = ((rec_{D_L})_{s_0} \circ (\eta_{(a,1)})_{s_0}, (rec_{D_L})_{s_1})$ respectively. By easy calculation, $\langle F'_{D_L}, U_{D_L}, rec'_{D_L}\rangle$ can be proved to be an adjunction. Therefore, $(a', \widetilde{list}(a)) \cong (a, \widetilde{list}(a))$. This means that $a' \cong a$. ∎

From the previous lemma, we can choose the datatype constructor $\langle F_{D_L}, rec_{D_L}\rangle$ such that $U_{D_L}F_{D_L}(a, 1) = (a, X)$ for some $X \in o\mathcal{C}$. We write $list(a)$ for this X.

Notation 8. We denote $F_{D_L}(\{(a,1)\})$ by $a \times list(a) \overset{cons_a}{\longrightarrow} list(a)$. We write nil_a for $(\eta_{(a,1)})_{s_1}$.

A simple calculation gives the following proposition:

Proposition 1. *Given* $b \times l \overset{g}{\longrightarrow} l$, $f: a \longrightarrow b$ *and* $n: 1 \longrightarrow l$:

1. $rec_{D_L}(f,n)_{s_0} = f$.
2. $rec_{D_L}(f,n)_{s_1} \circ nil_a = n$.
3. $rec_{D_L(a,1),\{(b,l),(g)\}}(f,n)_{s_1} \circ cons_a = g \circ (f \times rec_{D_L(a,1),\{(b,l),(g)\}}(f,n)_{s_1})$.

Example 1. Let us follow the previous construction for $\mathcal{C} = \mathbf{Set}$. An object $\{(S_0,S_1),(T)\}$ of $\mathbf{Set}^{D_L}$ consists of two sets, S_0 and S_1, and a function $T: S_0 \times S_1 \rightarrow S_1$. This corresponds to the translation that sends s_0 to S_0, s_1 to S_1, and t to T.

In **Set**, we have a left adjoint F_{D_L} (see the next section). $list(S_0)$ is the set of finite lists composed of elements of S_0, that is, $\{\langle a_0,\ldots,a_{n-1}\rangle | n \geq 0, a_i \in S_0 (0 \leq i < n)\}$. $cons_{S_0}: S_0 \times list(S_0) \rightarrow list(S_0)$ is the usual cons that sends $a_0 \in S_0, \langle a_1,\ldots,a_{n-1}\rangle \in list(S_0)$ to $\langle a_0,a_1,\ldots,a_{n-1}\rangle$. The morphism $nil: 1 \rightarrow list(S_0)$ corresponds to the empty list $\langle\rangle \in list(S_0)$. The previous proposition shows that rec_{D_L} in **Set** is the usual list induction schema.

An advantage of this method is that we can construct data structures similar to list types in the same way.

Let us follow the **Set** example again. When F_{D_L} is applied to $\{S_0, I\}$, where I is an arbitrary set, we have a structure that closely resembless *list* in the above example. $F_{D_L}(\{(S_0,I)\})_{s_1}$ is isomorphic to $I \times list(S_0) = \{(i,\langle a_0,\ldots,a_{n-1}\rangle) | i \in I\}$. There are as many *nil*s as I, that is, $\{(i,\langle\rangle) | i \in I\}$. This datatype represents an I-indexed list of the set S_0. We can use the same induction step for $list(S_0)$. When $I = 1$, we have only one *nil*, which we have already investigated. The only difference is that as the induction basis, we consider every case for every *nil*.

A typical example of an *I-indexed list* is the recursion on dotted-cons ($a_0\ a_1\ \ldots\ a_{n-1}.a_n$) in Lisp. It has the same recursion schema as *list* though a_n need not be nil. This difference lies only in whether $I = 1$ in our approach. We have the same induction schema, rec_{D_L}, though the induction basis is different from that of *list*.

We have this advantage because our diagram is an abstraction of induction steps only. This makes our construction simple and flexible. A comparison with CPL is studied in later sections.

4.2 Tree

Let D_T be $\langle S_T, T_T, \emptyset\rangle$, where $S_T = \{s_0\}$ and $T_T = \{t: (s_0,s_0,s_0)\}$.

Notation 9. We denote $F_{D_T}((a))$ by $\{(tree(a)),(maketree_a)\}$ or by $tree(a) \times tree(a) \overset{maketree_a}{\longrightarrow} tree(a)$. We denote $\eta_a: (a) \longrightarrow (tree(a))$ by $makenode_a$.

Proposition 2 (tree induction)**.** *For* $f: b \times b \longrightarrow b$ *and* $g: a \longrightarrow b$, *we have the following tree induction schema:*

1. $rec_{D_T((a)),\{(b),(f)\}}(g)_{s_0} \circ makenode_a = g$.
2. $rec_{D_T((a)),\{(b),(f)\}}(g)_{s_0} \circ maketree_a =$
   $f \circ (rec_{D_T((a)),\{(b),(f)\}}(g)_{s_0} \times rec_{D_T((a)),\{(b),(f)\}}(g)_{s_0})$.

Example 2. When $\mathcal{C}$ is **Set**, $tree(a)$ represents the set of binary trees whose nodes are elements of a. As in the case of *list*, rec_{D_T} represents the tree induction schema.

4.3 Natural Number

Let $D_N = \langle S_N, T_N, \emptyset \rangle$ be a diagram where $S_N = \{s_0\}$ and $T_N = \{t: (s_0, s_0)\}$. It is easily proved that $\mathcal{N} \xrightarrow{suc} \mathcal{N} = \{(\mathcal{N}), (suc)\} = F_{D_N}(1)$ is the natural number object. The same discussion appears in Goldblatt [3]. As expected,we denote $\eta_1 : 1 \longrightarrow \mathcal{N}$ by 0. We have the following induction schema as expected.

Proposition 3 (Induction on Natural Number). *For $x: 1 \longrightarrow a$ and $y: a \longrightarrow a$,*

1. $rec_{D_N(1),\{(a),(y)\}} \circ 0 = x$.
2. $rec_{D_N(1),\{(a),(y)\}} \circ suc = y \circ rec_{D_N(1),\{(a),(y)\}}$.

The above three examples use diagrams whose axiom set is empty. A well-known example that has a non-empty axiom set is *stack*.

4.4 Stack

Let D_S be $\langle S_S, T_S, A_S \rangle$, where $T_S = \{t_0: (s_0, s_1, s_1), t_1: (s_1, s_0), t_2: (s_1, s_1)\}$, $S_S = \{s_0, s_1\}$, and $A_S = \{t_1 \circ t_0 = \pi_1, t_2 \circ t_0 = \pi_0\}$. Here, π_i means the projection map to the i-th component.

This diagram defines a *stack*. We denote $F_{D_S}(a)$ by $\{(stack(a)), (push_a, pop_a, top_a)\}$.

5 $\mathbf{Set}^D$

The last section showed that several useful datatypes can be defined by using diagrams. The next problem is to determine how first-order datatypes can be defined.

Theorem 1. *Every diagram has a corresponding data constructor in* **Set**.

Proof. We use the following proposition [6].

Proposition 4. *Let* **A** *be finitary algebraic. Then, the forgetful functor $U: \mathbf{A} \longrightarrow$* **Set** *is monadic.*

Finitary algebraic categories, which have their forgetful functor to **Set**, have been intensively studied, e.g., [8]. In computer science, a finitary algebraic theory on **Set** is called a Σ-algebra [2]. In **Set**, the situation is rather trivial. We can utilize the theory of universal algebra.

In **Set** the category of first-order objects for a diagram is finitary algebraic. Therefore, by the above proposition, its forgetful functor is monadic. In particular, it has the left adjoint, that is, a first-order data constructor. ∎

The above theorem means that the theory of first order data constructors has a **Set**-model. This makes the following definition significant.

Definition 8. *Let $\mathcal{D} = \{D_i\}_{i \in I}$ be a set of diagrams $D_i (i \in I)$. Consider a ccc-based system that adds first-order data constructors for each D_i as constructors of objects and morphisms. We denote it by $ccc + \mathcal{D}$.*

A category $ccc + \mathcal{D}$ corresponds to, in a type theory, typed λ-calculus over some abstract datatype theory.

The following results immediately from the definition.

Corollary 1. *For an arbitrary set $\mathcal{D}$ of first-order data constructors,* **Set** *is a model of $ccc + \mathcal{D}$.*