修 士 論 文

形式仕様の記述スタイルに関する研究

指導教官　二木厚吉 教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

杉山 智倫

1997 年 2 月 14 日

要旨

本稿では、特に仕様言語の直感的理解の容易さ、扱いやすさといった点に注目し、仕様言語を用いた形式仕様の記述スタイルについて研究を行った。

仕様言語とはソフトウェア開発の上流工程の仕様記述に用いる形式言語であり、厳密な数学モデルを基盤にもつ。この仕様言語を用いた仕様記述は、曖昧性のない厳密な数学的議論を行えるという利点を持つ反面、記述が難解になりやすく初学者には理解が難しく扱いにくいものである。また、仕様言語の持つ意味モデルから外れた概念の記述は総じて不自然になりやすいという欠点も持つ。現在、この仕様言語の開発に関して数多くの研究がなされ、多種多様な仕様言語が乱立している。仕様言語を用いた形式仕様の記述スタイルについても問題の解釈の方法やモデル化手法の工夫により、様々な記述例が紹介されている。

本稿では、最初に簡単に仕様言語の紹介を行い、代数仕様言語 CafeOBJ とモデル指向の仕様言語 Z 記法での実際の問題の記述の比較を通して、再度、ユーザーの立場から理想的な扱い易い形式仕様言語について見直しを進める。特に、代数仕様言語では記述が難解であると言われている状態遷移問題の仕様記述に関して、Z 記法と CafeOBJ を用いて様々な記述例を紹介し、仕様の明晰性・拡張性といった視点から、それらを比較する。状態遷移問題の仕様記述では、代数仕様の新しい潮流である Rewrite rule や Hidden sort を用いた手法と既存の手法との比較に重点を置く。

最後に、本研究の成果を踏まえ、CafeOBJ の書き換え規則やクラス宣言、Hidden sort を拡張した独自の手法を提案し、代数仕様言語を用いた状態遷移の自然な記述スタイルについていくつか模索する。

目 次

第 1 章

はじめに

ソフトウェアシステムの仕様記述の方法としては、従来から自然言語をベースに図や表を用いた方法が用いられてきたが、昨今では、より無矛盾性、完全性、非曖昧性に優れた仕様を作成可能である形式仕様の需要が増し、様々な仕様言語とそれに関連した研究が盛んに行われている。

形式仕様とは、形式手法の仕様作成段階で、厳密な数学モデルや論理体系に基づいた仕様言語を用いて作成された仕様のことである。形式仕様は、自然言語による仕様と比較して、より信頼性の高い仕様を作成することができ、仕様の機械的な検証も可能といったソフトウェア開発の効率面でも多くの利点を持つ。仕様言語としては、Z[11][12],VDM,Larch,OBJ[7] などが有名である。これらの仕様言語はその基盤となる数学モデルや論理体系により大別でき、Z,VDM はモデル指向言語、Larch,OBJ は代数仕様言語の代表格である。

しかし、仕様言語は上記のような優位点を持つ反面、実際の仕様の作成の現場では、仕様言語が厳密な数学的議論に基づいているため、要求仕様の数学モデルへの投射、いわゆるモデル化の作業が困難であったり、形式言語であるので記述に関して制約が多いといった不利点もあり、自然言語による仕様の作成では考えられない様々な困難な問題が発生する。

本研究では、現実の問題の実際の仕様記述を通して、仕様言語を用いた形式仕様の様々な記述スタイルについて比較検討を行った。具体的な比較のアプローチとしては、同様の問題に対して、異なる数学的基盤を持つ仕様言語 (Z,CafeOBJ) や異なるモデル化手法を用いて厳密な仕様記述を行い、その比較を行った。このような仕様言語の記述例の比較は、Heisel[1] によっても成されているが、特に本研究では、最近の代数仕様言語での新し

い潮流である書換え論理遷移規則 (Rewrite Rule) や Hidden Algebra を用いた手法について積極的に取り上げ、従来の手法との詳細な記述比較を試みた。仕様の比較の際には、特に仕様記述の直観的分かりやすさ (明晰性)、変更が容易であるといった扱い易さ (拡張性) に重点を置き比較を行った。Rewrite Rule とは、Meseguer ら [13] により提案された動的なオブジェクトの記述を意識したモデルであり、これを用いることで同期システムの記述を容易に行える。Hidden Algebra は終代数意味論に基づきオブジェクトの振舞に注目し仕様記述を行う代数仕様の新しい考え方で、Goguen,Malcolm[4] により提案され、現在、設計が進められているモデルである。

本研究では、様々な問題の記述比較を試みている。特に、代数仕様言語による記述が困難と言われる状態遷移の記述の問題に関しては大きく取り上げ、Hidden Algebra,Rewriting Rule などの最近の新しい技術を中心に５種類の記述スタイルで雇用代理店の仕様記述を行っている。その後の考察では実際の記述の際の明晰性、拡張性に注目して各々の記述スタイルの詳細な比較を行っている。最後の第 8 章では、比較結果を考慮し実験的試みとして、Rewrite rule とクラス宣言を用いた状態遷移記述の定式化や Hidden Algebra を拡張し複数の hidden sort を扱い仕様を具象化する手法を提案し、CafeOBJ を用いてより簡単に仕様の拡張・変更ができる記述スタイルを模索した。

本研究の狙いは、様々な手法による形式仕様の比較を通して、乱立する形式仕様作成法および仕様言語に求められる性質をユーザーの視点から再確認し、形式仕様および仕様言語に求められる要件を追究を進め、仕様言語の開発現場を刺激することにある。仕様言語を開発する際、私たちは、実際に仕様言語を用いて記述を行うのは人間であることを忘れてはならない。そういう意味で、言語の設計者は本当に人間の考えやすいモデル化手法に適合した仕様言語を言語の設計段階で考慮する必要があり、本研究では直観的で扱いやすい仕様記述法の追究に終始固執し研究を行ったきた。本研究が今後の形式仕様と仕様言語の設計の参考となり、仕様言語の更なる発展に役立てれば幸いである。

第 2 章

形式仕様

本章では、形式仕様の基本概念および、仕様言語、特に本研究で用いる代表的な仕様言語である Z 記法と CafeOBJ の説明を行う。

2.1 形式手法

私たちの生活は、コンピュータが浸透するに連れて快適になっているが、それと反比例するようにソフトウェアシステムは大規模化、複雑化の様相をみせ、そういったソフトウェアシステムの開発は実際にさらなる困難を余儀なくされている。

これに対し、昨今では、より効率良く信頼性の高いソフトウェアシステムを開発するために、数学的な厳密性を持った形式手法の重要性が増している。形式手法とは、ソフトウェアの仕様作成から最終的なプログラムコードの作成までを一貫して数学的議論に基づいた形式的な方法で行う手法である。形式手法はプログラム検証論から生まれたアイデアを利用する手法であり、現在もソウトウェア工学の一分野として研究が進められている。

形式手法の一連の工程の中で、その上流工程である仕様記述の段階で、矛盾のない完璧な仕様を作成するための形式仕様の役割は益々その重要性を増している。実際、ソフトウェア開発の上流工程において作成される仕様の品質が、完成したソフトウェアの品質の優劣に大きく影響し、仕様段階での早期のバグの発見は開発効率の向上に繋がる。形式手法の中で、形式仕様を作成する際に用いる形式言語が仕様言語である。

2.2 仕様言語

仕様言語とは、前述のように形式手法の上流工程でソフトウェアの仕様を作成する際に用いる形式言語のことである。仕様言語は、その基盤に必ず厳密な数学的手法を持つという点で、通常のプログラミング言語とは性質の異なるものである。

仕様言語としては、多種多様な数学モデル、論理体系に基づく言語が多く研究され提案されているが、その中でも、Z,VDM,Larch,OBJ などは実用化され有名である。ここでは、多くの仕様言語の中から特に、仕様言語の標準的な存在である Z 記法と、代数的手法に基づき仕様の機械的検証が可能である CafeOBJ について取り上げる。

2.2.1 Z 記法

Z 記法 (Z notation) は、ZF 集合論と一階述語論理を基盤とする形式仕様言語であり、1970 年代後半に Oxford 大学 Computing Laboratory(OUCL) の Programming Reserch Group を中心にして研究開発された。Z 記法は現在、国際的に ISO で標準化が行われている。

Z 記法では、集合 (set) を基本型として用いることにより、強い型付けが行われるている。Z 記法を用いて仕様を作成する際は、Z 記法の基本ライブラリである Z mathematical tool kit で定義してある集合論と一階述語論理上のプリミティブ (集合、関係、関数、列など) を用いて現実世界を類似のプリミティブに置き換えることでモデル化を行う。そのため、Z 記法はモデル指向言語と言われる。

また、Z 記法には仕様記述をより構造的かつ直観的にする枠組であるスキーマ (schema) を持つ。Z 記法では、このスキーマを構成単位として永続オブジェクトの記述も容易に行える。スキーマは人間が記述することを強く意識して作成された枠組であり、Z 記法の直観的な記述の手助けとなる。スキーマでは、定義したスキーマを他のスキーマに包含するといったスキーマ演算を行うこともできる。

一般に、Z 記法による仕様の検証は、基本的に手作業で行わなければならない。しかし、基本型等の型の整合性を確認するための Type Checker などのツール、また、Z 記法を実行できるサブセットなどははいくつか開発が進められている。

以上のような Z 記法の性質より、Z 記法は他の仕様言語に対して次のような優位点を持つと言える。

- 集合論や一階述語論理は、学校で学習する基本的な概念であり、初学者にも馴染みやすい。
- $\forall, \exists$ などの集合論と一階述語論理による豊かな表現力を持つ。
- スキーマで仕様の構造化ができ、直感的理解が容易である。
- スキーマを用いて事前状態と事後状態の記述によりオブジェクトの変化を容易に記述できる。

2.2.2 OBJ

OBJ は、UCLA で Gouguen によって開発された代数仕様言語である。OBJ は厳密に抽象データ型の考え方を順序ソート代数と等号論理による数学モデルで裏付けした代数型言語である。順序ソート代数は、抽象代数である多ソート代数を拡張したモデルで、ソートに包含関係 (順序関係) を定義できる。また、OBJ は順序ソート代数と等号論理を基盤としている一階の関数型言語としての一面を持ち、仕様をそのまま状態で機械的に実行可能であり、検証面で多くの利点を持つ仕様言語である。

OBJ では、パラメータ化機構が洗練されており、汎用的なパラメータ付きモジュールを定義することができる。これにより、より抽象的な再利用性の高い仕様を記述することが可能である。また、他のモジュールを輸入でき、モジュールの和をとることができるなど、モジュールを構成する方法も豊富にある。

OBJ を用いて記述を行う場合は、モデル化の際に Z 記法のように集合や類似のものを用いるのではなく、直接振舞をとらえる方程式の集合によって仕様記述を行う。この記述特性により代数仕様言語は性質指向であると言われる。

代数仕様言語の一番の優位点は、仕様の実行が可能であることである。OBJ でも等式を書き換え規則として解釈する項書換えシステム (TRS) により記号的に実行可能であり、仕様の機械的な検証が可能である。

2.2.3 CafeOBJ

CafeOBJ は等号論理を拡張した順序ソート書き換え論理に基づいた代数仕様言語であり、OBJ 言語の最新版である。CafeOBJ は情報処理振興事業協会 (IPA) 技術センターの

「実行可能な形式仕様言語 CafeOBJ の開発と評価」プロジェクトで研究開発中の仕様言語である。

CafeOBJ は従来の OBJ 言語と比較して、オブジェクト指向プログラミングの技法での記述を可能にするクラス宣言やレコード宣言、書き換え規則 (rewite rule) を導入し、システムの動的な振舞の記述も容易に行えるように拡張がなされている。また、代数仕様の新しい潮流である Hidden Algebra に基づく記述手法の導入も現在検討されている。CafeOBJ で記述した仕様は、OBJ の場合と同様に項書換えシステム (TRS) により機械的な検証が可能である。CafeOBJ では、module 単位で記述を行い、module は型宣言などを行う signature 部と公理 (等式やルール宣言) を記述する axioms 部からなる。

CafeOBJ は、以上のような特徴により、他の形式仕様言語に対し次のような優位点を持つ。

- モジュールのパラメータ化、モジュールの輸入の際の名前の付け替えが可能で、強力なモジュール化機構を持ち、抽象度・再利用性の高い仕様記述が可能である。
- 項書換えシステムにより、仕様をそのまま機械的に実行可能であり、仕様記述段階でプロトタイピング可能である。

また、他の代数仕様言語と比較して次のような優位点を持つ。

- 順序ソート代数に基づき、ソート間に包含関係が定義でき、例外処理や演算の多重継承を自然に扱える。
- 動的なシステムの変化を記述できる書き換え規則を持つ。
- クラス構文を持ちオブジェクト指向モデルの記述にも対応している。
- 演算の引数の位置を自由に指定できる。

第 3 章

仕様言語の特性調査

本章では、仕様言語の基盤とする数学モデルの違いにより発生する記述特性の違いについて注目し、仕様言語の比較を行う。ここでは特に Z 記法と CafeOBJ を用いて、実際に簡単な問題の記述を行い、様々な視点からこれら２つの仕様言語の記述特性の比較を行う。Z 記法,CafeOBJ に関しては前章で説明しているが、簡単に表にまとめると次のような違いがある。

比較する仕様言語	CafeOBJ	Z 記法
背景	OBJ 言語を拡張 書き換え論理を導入	1970 後半に OUCL で開発 現在 ISO で標準化中
基盤とする数学モデル	順序ソート代数 書き換え論理	ZF 集合論 一階述語論理
実行系	項書換えシステム (TRS) で 機械的な検証が可能	完全な実行系はないが Type Checker などのツールあり
特徴	module で仕様を構成 性質指向	schema で構造化 モデル指向

Z 記法と CafeOBJ の仕様特性に違いをもたらす一番の原因は、これら２つの仕様言語の基盤となる数学モデルが異なることである。仕様の記述スタイルは大きくその仕様言語の基盤となる数学モデルに依存する。一般に、モデルベースの仕様言語と代数仕様言語では、ソフトウェアシステムの仕様への適用性が違うと考えられている。モデルベースの仕様言語は、抽象機械などの state based system(状態を持つ機械) の記述に適し、代数仕様言語は抽象データ型タイプの仕様に適していると言われている。

次の節以降では、Z 記法と CafeOBJ でのを簡単な例題の仕様の記述の比較を通して集合論モデルと代数モデルの差異、またそれに基づく記述特性の差異を明らかにしていく。記述の比較を行う問題としては本稿では次のようなものを扱う。

再帰的な問題 複雑なデータ構造の表現方法に関する比較。

抽象データ型 抽象度、再利用性の高い仕様記述についての比較。

状態の表現 動的なオブジェクトの表現方法に関する比較。

3.1 再帰的な問題の記述比較

まず、複雑な構造を持つオブジェクトをどのように表現できるかという点に着目し比較を行う。通常、オブジェクトの表現をそれぞれの仕様言語では次のようにモデル化する。

問題	CafeOBJ での表現	Z での表現
もの	ソートの要素 (項)	集合の要素
ものの集まり	ソート	基本集合

ここでは、特に再帰的なデータ構造の仕様の記述について比較検討する。複雑なデータ構造を表現する場合、無限再帰的な考え方は便利である。再帰的なデータ構造の問題としては、リストや木構造などが代表的である。

簡単なリストの記述

リストは再帰的なデータ構造を持つ問題の代表格である。リストは「要素とリストの組」として再帰的にデータ構造を与える考え方が自然である。

CafeOBJ によるリストの記述

CafeOBJ では、リストのデータ構造を汎用的なモジュールとして、次のような自然な再帰手法で定義できる。op は operation(関数) の定義を行う構文で、List を２つ引数にとり List を返す関数として定義する。次のモジュールは、再帰的なリスト構造を再帰的な考え方に自然な形で実現している。

```
module LIST [ X :: TRIV ] {
  [ Elt < List ]  -- Elt は TRIV のソートとして取り込まれる
  signature {
    op nil : -> List
    op __ : Elt List -> List
  }
}
```

代数仕様言語では、このように代数モデルの始代数的意味論に基づき、ものの集まり (ソート) を帰納的に定義できる。この定義方法を用いれば、再帰的なデータ構造を直観的に記述可能である。

Z 記法によるリストの記述

リストの表現に対し、Z 記法では列 (sequence) というプリミティブを準備している。Z 記法は集合論を基盤とし、基本型はすべて集合 (sets) として定義する。集合は要素の重複を許さない為、重複を許し順番をもつ要素の並びであるリストを表現するためには、写像を用いて定義した「列」の構造を用いる他はない。列は Z 記法では次のように定義されている。

$$\mathrm{seq}\, X == \{\, f : \mathbb{N} \pfun X \mid \mathrm{dom}\, f = 1..\#f \,\}$$

ここで$\pfun$は有限部分関数、$\#f$は集合 f の要素数を意味する。これは直感的には、{ 1 $\to$ A, 2 $\to$ B, 3 $\to$ B } といった各要素に索引をつけた写像の集合を表す。

木構造の記述

木構造は、「木は、ノードの情報と木のリストの組」として再帰的に表現できる。

CafeOBJ による木構造の記述

リストと同様に、CafeOBJ では木構造の再帰的な構造を崩さずに自然な記述が可能である。また、このモジュールはパラメータ化されており、汎用的な利用が可能である。

```
module TREE [ X :: TRIV ] {
 signature {
```

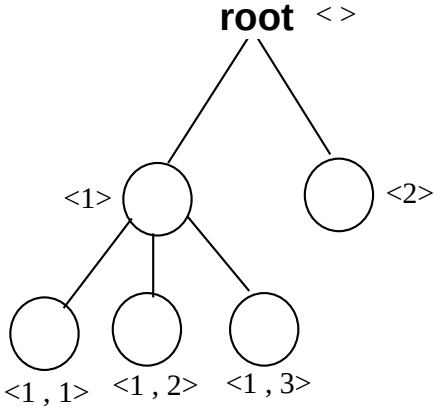
```
 [Elt < Tree < TreeList]   -- '<' はソートの順序関係を表す
 op <_,_> : Elt TreeList -> Tree  -- 木の定義
 op  nil  : -> TreeList
 op __ : TreeList TreeList -> TreeList {assoc} -- 結合則を満たす
 }
}
```

Z 記法による木構造の記述

Z 記法では、リストと同様に、木のノードに索引をつけて木構造を表現する。

$$
\begin{aligned}
&TREE[X] == \{\ f : \operatorname{seq}\mathbb{N}_1 \pfun X \mid \\
&\quad <> \in \operatorname{dom} f \\
&\quad \land\ (\forall\, path : \operatorname{seq}_1 \mathbb{N}_1 \mid path \in \operatorname{dom} f \bullet \\
&\qquad frontpath \in \operatorname{dom} f \\
&\qquad \land\ (lastpath \neq 1 \Rightarrow frontpath \\
&\qquad \frown < lastpath - 1 > \in \operatorname{dom} f))\}
\end{aligned}
$$

この仕様は直観的には分かりづらいが、次の図のような意味を表す。各ノードには、ノードの下に記述してあるような索引 (アドレス) が割り当てられる。



また、Z 記法では自由型 (FreeTypes) を用いて、

$$
\begin{aligned}
&[X] \\
&TREE ::= leaf\langle\langle X \rangle\rangle \\
&\qquad \mid node\langle\langle \operatorname{seq} TREE \rangle\rangle \\
&X == ....
\end{aligned}
$$

と定義できるが、この方法では基本型 X を必ず再定義する必要があり、完全な汎用性はない。実際に、この仕様は Z 記法の記述としては一般的なものではなく、普通、再帰的なデータ構造を表現する場合には列を用いる。

再帰的な問題の記述に関する整理

CafeOBJ は始代数的意味論に基づいており、集合を帰納的に定義することができる。よって、自然に構成子関数 (constructor function) を用いた再帰的な構造の記述が行える。この手法による記述は大変直観的であり、さらにパラメータ付きモジュールとして定義することで再利用性や抽象度の高い仕様の作成ができる。

Z 記法では基本型を集合を用いて定義しなければならないので、帰納法による自然な再帰的データ構造の表現ができない。そのため、列のような疑似的な手段で再帰の構造を表現せざるを得ない。自由型 (FreeTypes) を用いれば、BNF 記法のような直観的な記述法で記述可能であるが、自由型 X は汎用的なパラメータでなく、必ず X の実体である集合を他の場所で唯一に定義する必要がある。

3.2 抽象データ型で表現される問題の記述比較

抽象データ型は代数モデルの基本構造と一致するものであり、抽象データ型で表現される問題の記述は、代数仕様言語の最も得意とする問題である。一般に、抽象データ型で表現される問題として、スタックや待ち行列がある。ここでは、スタックの仕様を CafeOBJ と Z 記法で記述し比較を行う。

CafeOBJ によるスタックの記述

CafeOBJ でのスタックの記述は次のようになる。

```
module STACK [X :: TRIV] {
   [ NeStack < Stack ]
   signature {
     op empty : -> Stack
     op push : Elt Stack -> NeStack
```

```
    op pop : NeStack -> Stack
    op top : NeStack -> Elt
  }
  axioms {
     var S : Stack
     var E : Elt
     eq pop(push(E,S)) = S .
     eq top(push(E,S)) = E .
  }
}

view NAT-as-TRIV from TRIV to NAT {
 sort Elt -> Nat
}

module NAT-STACK {
 protecting (STACK [ X <= NAT-as-TRIV ])
}
```

CafeOBJ では、モジュールをパラメータ化することにより、汎用的なモジュールを記述することができる。この例では、view 宣言により、汎用的なモジュール STACK のパラメータにモジュール NAT を対応させ、自然数のスタックを記述している。view 宣言で NAT の代わりに別のモジュールを対応させれば、そのモジュール (代数) を構成する要素のスタックとなる。また、この仕様では、NeStack(Non Empty Stack) を Stack の下位ソートに定義し、pop と top の作業は NeStack に適応させている。このような記述を行うことで、空スタックに pop や top を適用することを回避している。この仕様は、データ型とその間の操作によって定義される抽象データ型そのものを表す。

Z 記法によるスタックの記述

Z 記法で抽象データ型を意識したスタックの仕様記述を行うには、

- 自由型 (FreeTypes) を用いてデータ型の定義を行う。
- 公理的記述 (axioms box) で操作の定義を行う。

というふうにすれば可能である。

自由型を用いたデータ型の定義

$$[X]$$

$$STACK ::= empty \mid push\langle\langle X \times STACK \rangle\rangle$$

公理的記述を用いた pop と top の定義

$$pop : STACK \rightarrow STACK$$
$$top : STACK \rightarrow X$$

$$\forall s : STACK;\ x : X \bullet$$
$$pop(push(x, s)) = s \land$$
$$top(push(x, s)) = x \land$$
$$push(x, s) \neq empty$$

Z 記法は、集合論と一階述語論理を基盤とするため表現力豊かな記述ができる。この仕様では、push(x,s) が空スタックではないことを明示することで、空スタックには pop,top は適用されない。

抽象データ型で表現される問題の記述に関する整理

抽象データ型による表現は代数モデルの基本構造と一致するもので、代数仕様言語の最も得意とする分野である。この問題は、CafeOBJ を用いれば大変自然で汎用性の豊かな記述を行える。Z 記法でこれらの問題の記述を行う際は、自由型を用いてデータ構造の定義を行い、操作は公理的記述 (axiomatic box) で定義を行えばよい。この方法を用いれば、CafeOBJ のように完全に汎用的なモジュールを作成することができない点を除いては、Z 記法でも CafeOBJ と同様の記述を行うことができると言える。

スタックの記述を行う場合、空スタックへ pop,top を適応しないように制約を与える必要があるが、CafeOBJ では、ソートの順序関係をうまく用いて記述し、Z 記法ではスタックの要素そのものに Z のプリミティブである $\neq$ を用いて条件を記述した。ここに、この２つの仕様言語の記述特性の違いが如実に現れている。

3.3 状態の表現方法の比較

この章では、動的なオブジェクトの記述を仕様言語を用いてどのように行えるかを見る。状態遷移を記述する問題としては有限オートマトン (FA) などがある。状態遷移の表現については次節で詳しく取り上げるが、ここでは簡単な旗 (FLAG) の状態表現の例を見る。旗は UP(上がっている) と DOWN(下がっている) の二つの状態を持つ。

CafeOBJ による状態の表現

CafeOBJ の前身である OBJ3 では状態を持つオブジェクトの概念がなく、等式を用いて状態の変化を記述しなければならなかった。ソートの要素で表現されるオブジェクトはデータオブジェクトを意味し、その変化を左右の項の等価関係を意味する等式を用いて記述する方法は、状態遷移の記述として自然ではないように思える。次の章以降で取り上げるが、現在の CafeOBJ では書換え論理に基づく遷移規則を用いて、より自然に FA や NFA のような動的振舞の記述を行うことができる。しかし、その場合も状態を始代数的意味論に基づく項として明示的に表現しなければならない。

FLAG の状態は次のような構成子関数 (constructor function) により生成される項として表現される。

```
module FLAG {
  [ FLAG ]
  op init : -> FLAG
  op up   : FLAG -> FLAG
  op down : FLAG -> FLAG
}
```

この仕様は、状態の表現のみであるが、状態遷移関数、様々な条件については、公理として等式を用いて定義する必要がある。そのためにも、状態は項として明示的に扱う必要がある。

Z 記法による状態の表現

Z 記法では、スキーマの中で入力、出力、事前状態、事後状態を簡単に記述でき、代数仕様言語より状態遷移を直感的に表現することができる。また、基本型の要素として状態

遷移の履歴を含まずに状態を表現できる。FLAG の状態表現は、簡単な至って簡単な定義で実現でき、それぞれの状態をスキーマの内部から参照できる。

$$FLAG ::= up \mid down$$

状態表現の記述に関する整理

代数仕様言語では、要素を始代数的意味論に基づく項として表現しなければならないため、状態の表現が窮屈である。一方、Z 記法ではスキーマ内で容易に状態およびその属性の参照ができるので状態を自然に表現することができる。

3.4 仕様の構造化手法の比較

形式仕様で大規模な仕様を作成する際には、平坦な仕様では読みにくく、何らかの形で仕様を構造化して記述する必要がある。仕様の構造化は、CafeOBJ ではモジュール単位で行い、Z 記法ではスキーマ単位で行う。

CafeOBJ 仕様の構造化について

CafeOBJ のモジュールを構成する手法は豊富である。CafeOBJ では、各々のモジュールは各々１つの代数構造を形成する。ここで形成される代数構造は、loose semantics(ゆるい意味) での代数と tight semantics(きつい意味) での代数の両方が規定できる。また、パラメータ付きモジュールの定義を行え、view 宣言により他のモジュールをパラメータ部分に instantiate できる。また、モジュールの輸入手法も豊富にあり、輸入する際にモジュール要素の名前の付け替えもできる。モジュールの輸入に関しては３つの輸入モードがあり、次のような制限を持つ。

protecting 輸入するモジュールのソート上に新たな要素を付け加えたり、ソート上の既存の要素を同定してはならない。(非冗長・非混同)

extending 輸入するモジュールのソート上の既存の要素を同定してはならない。(非混同)

using なんら制限がない。

Z 仕様の構造化について

Z 記法ではスキーマを用いて仕様の構造化を行う。スキーマでは独特の記法やスキーマ演算を定義することにより、直感的に分かりやすい記述を可能にしている。また、スキーマ演算を用いてスキーマの操作や検証も可能である。

スキーマは下の例のように、宣言部 (上段, 型宣言など行う) と述語部 (下段, 公理の記述を行う) で構成する。宣言部では、他のスキーマの輸入の記述も行える。

$$
\begin{array}{|l}
\hline
Aplly \\
\Delta Agency \\
p? : Person \\
s? : Skills \\
\hline
p? \notin \mathrm{dom}\ cands \\
vacs' = vacs \\
cands' = cands \cup \{p? \mapsto s?\} \\
\hline
\end{array}
$$

実際、スキーマの枠組を用いない平坦な論理式で記述を行えば (これを線形形式という)、

$$
\begin{aligned}
Apply \mathrel{\widehat=} [&vacs, vacs' : Agency;\ cands, cands' : Agency; \\
&p? : Person;\ s? : Skills \mid \\
&\quad p? \notin \mathrm{dom}\ cands \land \\
&\quad vacs' = vacs \land \\
&\quad cands' = cands \cup \{p? \mapsto s?\} \\
&\quad]
\end{aligned}
$$

と直感的な理解が難しくなるので、スキーマの枠組の恩恵は偉大である。

また、仕様書から実際に動くプログラムへの移行に際して、スキーマは異なった抽象化レベルにおけるシステムの状態の間にある関係をとらえることにも使われる。

3.5 仕様言語の特性 (まとめ)

ここまで、いくつかの簡単な問題に対し、Z と CafeOBJ での実際の仕様記述を通して、それぞれの仕様言語の数学モデルの特性について考察してきた。

仕様言語により作成された仕様に関して論ずる場合、次のような点が論点と考えられる。本章では、これらの論点に基づき、仕様言語の比較を進める。

問題との親和性 問題の持つ構造をどれだけ保存できるか？

再利用性 モジュール性の高い仕様が書けるか？

仕様の明晰性 直観的に分かりやすい記述が可能か？

問題と数学モデルとの親和性の比較

仕様言語を用いて各々の問題の持つ構造を崩さずに表現できるかという点に着目し、CafeOBJ の基盤となる代数モデル (順序ソート代数) と Z 記法の基盤となる集合論モデルに関して記述結果を比較した。その結果は次の表に示す。

問題	代数モデル	集合論モデル
再帰的なデータ構造	◎	△
抽象データ型	◎	○
状態表現	△	◎

CafeOBJ では、何らかの構造的なオブジェクトを表現する問題に関しては、直観的な記述ができるが、状態が変化するオブジェクトの記述は苦手である。Z 記法では、状態を持つオブジェクトの表現は自然に行えるが、Z 記法のプリミティブでそのまま表記できないような構造を持つオブジェクトの表現を記述する際は工夫を要し、不自然な記述になりやすい。再帰的な問題の例題のように要素の重複の現れる問題を扱うためには、集合は要素の重複を許さない概念であるので、要素の索引をつけるための関数の定義を必要とする。

Z 記法では、仕様を予め準備されたプリミティブだけで記述しなければならない。この性質により初学者は容易に記述に馴染めるが、このプリミティブで記述できない問題に関しては記述が困難である。一方、CafeOBJ では等式だけで制約できる分、構造に関しては自由な表現ができる。しかし、仕様記述を１から始めなければならず慣れるまでは記述が難しい。また、Z 記法の∃など代数モデルでは表現しにくいものもいくつかある。

仕様の抽象性、再利用性の比較

仕様の再利用性、抽象性に関しては、代数仕様言語は大変優れている。OBJ は、loose semantics と tight semantics での代数をそれぞれ記述でき、パラメータ化することにより汎用的なモジュールを定義できる。一方、Z 記法では、基本型として自由型 (FreeTypes) を定義できるが、この自由型も必ず他の場所で定義される必要があり、抽象度の高い汎用的な仕様記述は得意ではないと言える。

仕様の明晰性の比較

仕様の直観的理解を容易にするものには、次のような要素があると考えられる。

- 問題の自然なモデル化
- 仕様の分かりやすい構造化
- 直感的な記法

問題の自然なモデル化に関しては前述したので省略し、以下の二項目に関して比較する。

仕様の分かりやすい構造化について

大規模な仕様を記述する場合は、仕様の分かりやすい構造化を行う必要がある。Z 記法では、直感的理解のしやすいスキーマという枠組がある。スキーマでは、名前付けの規則など記述方法が多く規定されている。また、大規模な仕様を記述する場合の段階的詳細化に関しても、異なる抽象化レベルにおけるシステムの間にある関係について、スキーマを用いて捉えることができる。CafeOBJ では、仕様言語自体豊富なモジュール化機構を持つが、大規模な仕様記述手法に関しては定まった記述方法、モジュール化手法が確立しておらず、個人の好みにより仕様の構成を行う。この場合、自由度の高いモジュールの構成が可能であるが、仕様が複雑化した場合や他人が初めてその仕様を読む場合、可読性の低い仕様になりがちである。

直感的な記法について

細かな記法の差異が人間の理解に強い影響をもたらすことはよくある。特に、演算子の前置・中置・後置や、記号の使い方が制約されている場合、思わぬ落し穴につまづくこと

がある。Z 記法は、プリミティブに関しては記法が定まっているが、その他の演算に関しては既存のプリミティブを用いて定義できる。しかし、既存のプリミティブの意味を再定義することは混乱に繋がるので推奨されず、実際問題として既存のプリミティブを用いて問題のモデル化の際に直観的に不自然な記述をしなければならないこともある。例えば、スキーマの中で事前状態、事後状態を「＝」で結ぶ記法は、「＝」が左右のオブジェクトの同値関係を意味することより、あまり直観的ではないように思える。CafeOBJ は、演算子の定義、記号の使い方の自由度が高く、各個人で演算の記法、性質を１から自由に作成でき、カスタマイズできる。

まとめ

本章では、Z 記法と CafeOBJ でいくつかの比較項目に対しを例題の記述を通して、仕様言語の記述特性についていくつか考察を行ってきた。

Z 記法と CafeOBJ との一番の違いは、Z 記法は仕様の機械的実行ができないが、CafeOBJ では機械的実行ができるということである。今回の仕様記述を通して、Z 記法は仕様を記述することに重点を置けるが、CafeOBJ は仕様記述の際に仕様の記述と実行 (検証) を考慮する必要があるという印象が強い。仕様記述の際にプロトタイピングできることは、大変便利であるが、仕様記述の本質を見失わないように注意が必要である。

また、どの比較項目に関しても言える Z 記法と CafeOBJ の大きな違いは、Z は既存のプリミティブを用いて記述を行い、CafeOBJ はすべてを自分で定義し制約のない自由度の高い仕様記述を行えるということである。この違いは考察の中でも述べたように、時には長所となり時には短所となり、仕様の記述者および読者に降りかかる。

CafeOBJ を用いて仕様記述を行えば、制約の少ない自由度の高い記述が可能である反面、初学者はその自由度の高さに困惑する。本研究では、直観的で扱い易い仕様の記述スタイルを追究する立場を取っている事情、初学者の立場から言わせてもらえれば、CafeOBJ に対し次のような課題 (要求) があるのではないかと考える。

- 状態と状態遷移の自然な表現方法および記述スタイル
- 大規模仕様のモジュールの効率的な構造化手法
- 直観的に扱いやすい基本ライブラリ、インターフェイス

第 4 章

state(状態遷移) とその分析方法

一般に、オブジェクト (もの) は普遍の属性をもつデータオブジェクトと状態を持つ (変化する属性を持つ) クラスオブジェクトの 2 種類に大別できる。state とは「状態」、すなわち、状態をもつオブジェクト (クラスオブジェクト) のことである。実際に仕様のモデルとなる現実世界には、この状態を持つオブジェクトが極めて多く、仕様記述を行う際にはこの状態を持つオブジェクト (state) を如何に扱うかという問題は大変重要な課題である。

本章では、state の問題の捉え方と事前状態・事後状態の差異に基づく state 問題の分析方法について説明する。

4.1 状態遷移の表現方法

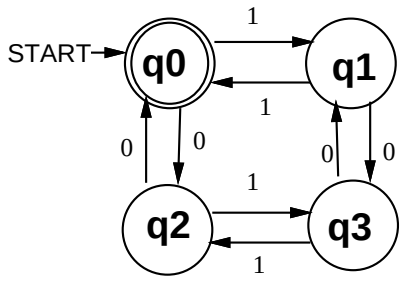
状態遷移の問題は一般に、

- 状態の集合
- 初期状態
- 状態遷移を行うためのいくつかの操作 (operation)

により定義される。

状態遷移を行う操作は入力値 (label など) と事前状態 (pre-state) を受け、事後状態 (post-state) と出力値を返す。state の表現方法としては、この定義に基づいたオートマトンが

代表的である。オートマトンは下の図のような状態遷移図 (transition diagram) を用いれば、さらに直感的に分かりやすく表現できる。下図は、同数の 0,1 を含む入力文字列を受理するオートマトンである。



4.2 state の問題の分析・整理

本稿では前述のように、事前状態と事後状態の差異に着目し状態遷移の記述を行う方法が状態遷移の自然なモデル化であると解釈する。そこで、現実世界の状態遷移の問題を以下のように分析して整理し、それぞれの項目がどのように仕様言語を用いて記述されるかを見る。

状態空間・状態属性 状態の変化するオブジェクトとその属性を把握する。

初期状態 状態および属性の初期状態を把握する。

状態遷移 状態遷移を行う operation に関して把握する。その入力値、出力値、事前条件、事後条件について把握する。状態遷移による属性の値の事前状態、事後状態の差異についても把握する。

事前条件 事前条件 (入力値、事前オブジェクトの満たすべき性質) を把握する。

事後条件 事後条件 (出力値、事後オブジェクトの満たすべき性質) を把握する。

入力 状態遷移を行う operation に必要なオブジェクトを把握する。

出力 状態遷移を行った後の副産物であるオブジェクトを把握する。

この項目を分かりやすいように、次のような表を用いて表す。

状態に関する整理

状態空間名	初期状態	属性	状態遷移を行う操作
名前１			

操作に関する整理

操作名	入力値	出力値	事前条件	事後条件
操作１				
操作２				

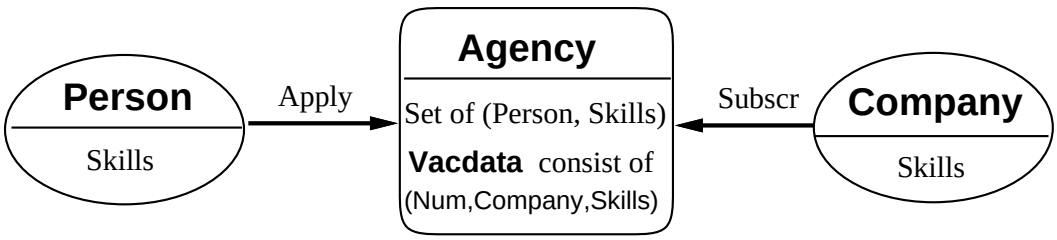
操作名	属性名	事前状態と事後状態の差異
操作１	属性１	
操作１	属性２	

4.3 例題: 雇用代理店 (Employment Agency)

state 記述の例として、次のような簡単な雇用代理店 (Employment Agency) の仕様記述を考える。

雇用代理店の要求仕様

雇用代理店は、求職者 (Person) と会社 (Company) 間の仲介を一手に行う代理店 (Agency) である。求職者は代理店に自分の名前 (Name) と技能 (Skills) を登録し、会社は会社名と募集する人材の技能 (Skills) を登録している。



この雇用代理店のその他の詳しい要求仕様は以下の通りである。

- １人の人間は同時に２つの登録を持てない。同じ名前を持つ人間は２人同時に登録されない。

- 会社は同時に複数の人材を募集できる。同じ技能を持つ人材でも複数募集できる。
- 仕事を探す人は skill を複数登録できる。(skills)
- 会社の求人登録には空き番号 (vacancy number,Vacno) が１つずつ割り振られる。
- ここでは、状態を変化させる操作としては、代理店のデータベースに人とその技能を付け加える Apply、会社の求人をデータベースに付け加え、求人番号を与える Subscr のみを考える。

4.3.1 雇用代理店の問題の分析・整理

4.2 節で示した状態遷移の分析方法で、雇用代理店の仕様は次のように整理できる。

状態空間 雇用代理店のデータベース (Agency)
属性 求職者リスト (PersonList) と求人リスト (CampanyList) を持つ
初期状態 状態の初期状態の構成を把握する。
状態遷移 求職者リストと求人リストに新しい登録を加える operation, apply,subscr。
事前条件 apply の際、入力値である新規登録者がまだ登録されてない。
事後条件 subscr の際の出力である求人番号が未登録の番号である。
入力 apply の際、人とその skill。subscr の際、会社と求める skill。
出力 subscr の際、ユニークな Vacno が割り当てられる。

状態に関する整理

状態空間名	初期状態	属性	状態遷移を行う操作
Agency	属性すべて empty	PersonList,CompanyList	apply,subscr

操作に関する整理

操作名	入力値	出力値	事前条件	事後条件
apply	Person,Skills	なし	Person が未登録	なし
subscr	Company,Skills	Vacno	なし	Vacno が未登録

操作名	属性名	事前状態と事後状態の差異
apply	PersonList	新求人者を登録 (Person,Skills)
apply	CompanyList	変化なし
subscr	PersonList	変化なし
subscr	CompanyList	新会社を登録 (Company,Skills,Vacno)

4.3.2 仕様の追加・変更項目

以上のような仕様で、各モデル化手法で一度仕様記述を行った後、仕様の変更として、PersonList の属性のデータに「求人者の性別 (sex)」を付け加える作業を行う。すなわち、apply は、Name と Sex と Skills を入力として取り、Agency にデータを登録する operation となる。表は以下のように変更される。

操作に関する整理

操作名	入力値	出力値	事前条件	事後条件
apply	Person,Skills,Sex	なし	Person が未登録	なし
subscr	Company,Skills	Vacno	なし	Vacno が未登録

操作名	属性名	事前状態と事後状態の差異
apply	PersonList	新求人者を登録 (Person,Skills,Sex)
apply	CompanyList	変化なし
subscr	PersonList	変化なし
subscr	CompanyList	新会社を登録 (Company,Skills,Vacno)

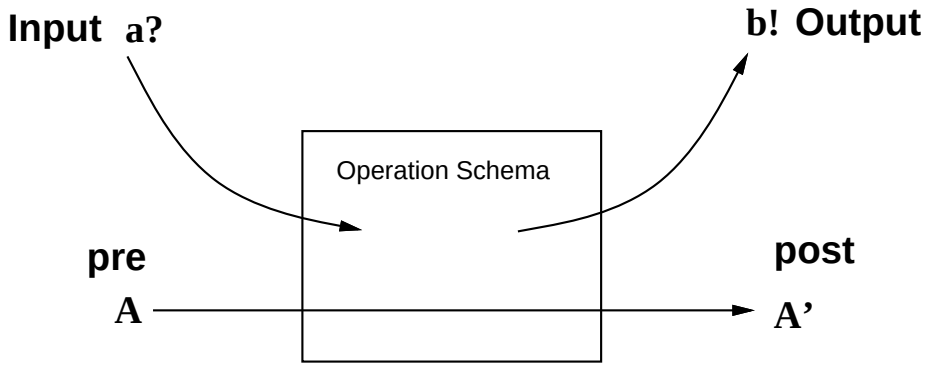
第 5 章

Z 記法による state の記述方法

本章では、Z 記法による state の記述手法と 4.3 節の例題:雇用代理店の実際の仕様記述をみていく。

5.1 Z 記法による state の記述方法

Z 記法は、一般に state の記述に適した仕様言語だと言われる。Z 記法では状態集合を基本型による定義を用いて、操作を事前状態と事後状態 (事前状態に dash をつけたもの) により簡単に記述できる。Z 記法では以下の図のように操作の前後での名前付け規則が決まっている。



Z 記法では、state を

- 状態スキーマ
- 初期状態スキーマ

- 操作スキーマ

の記述を行うことでモデル化が可能である。個々の state space は命令型プログラミング言語のように暗黙のうちモデル化されるようになる。このように、暗示的に state の表現を行う手法を implicit state approach という。

5.2 Z 記法による雇用代理店の仕様記述

基本集合の定義

基本型 (集合) として、Person、Skill、Company を定義し、Skills を Skill の巾集合、Vacno(vacancy number) を自然数の集合として定義する。

$$[Person, Skill, Company]$$
$$Skills == \mathbb{F}\, Skill$$
$$Vacno == \mathbb{N}$$

状態 (state space) の表現

雇用代理店のデータベースは、状態の変化するオブジェクトである。Z 記法では、状態を状態スキーマを用いて表現する。次の Agency は、雇用代理店のデータベースの状態空間を表現している。ここで、Vacdata を別の状態スキーマを用いて表現したのは、要求仕様では求人登録は重複を許すからである。(vacno により vacdata はユニークに決まる。)

Agency

$$persondata : Person \pfun Skills$$
$$campanydata : Vacno \pfun Vacdata$$

Vacdata

$$comp : Company$$
$$skills : Skills$$

初期状態の表現

次の Empty は、Agency の initial state を表現する。スキーマの宣言部にスキーマを記述することでスキーマ Agency の輸入を表現し、その中のオブジェクトである persondata と companydata がそれぞれ空集合のものが Agency の初期状態である。

Empty

$$Agency$$

$$persondata = \varnothing$$
$$companydata = \varnothing$$

操作の表現

Z 記法では操作を操作スキーマで表現する。ここで、?は入力変数、!は出力変数、「'」付きは事後状態のオブジェクト、「'」なしは事前状態のオブジェクトである。$\Delta Agency$は Agency と Agency' を輸入することを表現する記法である。スキーマ Agency' は明示的に定義されていないが、Agency の要素すべてに「'」をつけたものである。(スキーマの名前付け規則による)

Aplly

$$\Delta Agency$$
$$p? : Person$$
$$s? : Skills$$

$$p? \notin \operatorname{dom} persondata$$
$$companydata' = companydata$$
$$persondata' = persondata \cup \{p? \mapsto s?\}$$

Apply は persondata に新しいデータを加える操作である。新しいデータとして登録される Person は未登録である必要がある。

Subscr

$\Delta Agency$

$c? : Company$

$s? : Skills$

$n! : Vacno$

$persondata' = persondata$

$n! \notin \mathrm{dom}\ companydata$

$companydata' = companydata \cup \{n! \mapsto \mu\ Vacdata \mid comp = c? \land skills = s?\}$

Subscr は companydata に新しいデータを加える操作である。n!は求人番号の出力であり、vacdata に対してユニークに定まっている。

Z 仕様の変更

以上のような Z 仕様に属性として新たに性別を加える。この時、型宣言を付け加え、状態スキーマを以下のように変更しなければならない。

$Sex ::= male \mid female$

Agency

$persondata : Pdata \pfun Skills$

$campanydata : Vacno \pfun Vacdata$

Pdata

$person : Person$

$sex : Sex$

また、操作 apply を以下のように修正する。

Aplly

$\Delta Agency$

$p? : Person$

$s? : Skills$

$sex? : Sex$

$\{\mu\, Pdata \mid person = p?\} \notin \mathrm{dom}\, persondata$

$companydata' = companydata$

$persondata' = persondata \cup \{\mu\, Pdata \mapsto s? \mid person = p? \land sex = sex?\}$

Z 仕様では、このように最小限の仕様の変更で解決し、その他の変更を必要としない。

Z 記法による state 記述の考察

Z 記法では、state space と初期状態を状態スキーマとして表現し、状態を変化させる操作 (apply,subscr) をそれぞれ操作スキーマで表現している。操作スキーマでは、その state に対する操作を事前状態、事後状態の差異により記述する。Z 記法における操作スキーマの意味合いは、あるオブジェクトを入力として受け取り、あるオブジェクトを出力として生み出し、さらにあるオブジェクトを変化させることであり、そのすべてが簡潔に記述できる。事前条件、事後条件の記述は、構成要素に対して制約を与える状態不変式 (state invariant) を用いて、スキーマの中で表現力豊かに簡潔に記述することができる。

Z 記法では、スキーマ記法を用いることで、暗黙的に state space を表現することができる。これは state space の変更 (属性等の変更) が容易に行えるという利点がある。

第 6 章

CafeOBJ による state の記述方法

本章では、CafeOBJ を用いた state 記述の手法をいくつか採り上げ、それを用いて雇用代理店の仕様記述を実際に行う。採り上げる手法は、explicit state approach と Baumeister の手法、書き換え規則を用いた手法、Hidden Algebra に基づく手法の４手法である。

6.1 explicit state approach

通常、代数仕様言語で state の記述を行う際には、explicit state approach を用いることが一般的であり、昨今までは、この手法以外に state を記述する方法は提案されていなかった。explicit state approach とは、状態変化を促す operation に state を表現する項を引数として渡し、その項を関数的な方法で変化させる方法である。ここで state は始代数意味論に基づく項として明示的に表現されるので、この手法を explicit state approach と呼ぶ。

この手法による実際の state の記述方法は次のようになる。

- 状態空間は一つのソートで表現し、それぞれの状態はそのソートの要素として明示的に項で表現する。
- 初期状態は状態空間を表すソートの１つの要素 (項) である。
- 状態変化を行う操作は、事前状態を引数にとり、操作を加えた後の事後状態を返す構成子関数として表現する。
- 以上のように定義された state を、関数的に定義した観測関数を用いて観測する。

6.1.1 explicit state approach による雇用代理店の仕様記述

代数仕様言語 CafeOBJ を用いて、explicit state approach による雇用代理店の記述を紹介する。以下に掲げるコードが雇用代理店仕様の CafeOBJ コードである。このコードでは、state の記述を次のように対応づけている。

- 状態空間はソート Agency、それぞれの状態は Agency の要素として項として表現する。
- 初期状態は Agency の定数項である empty である。
- 状態変化を行う操作 apply と subscr は事前状態である Agency と入力値を引数にとり、事後状態の Agency を返す構成子関数として表現する。
- 以上のように作成された状態を観測関数 isPerson、skillsFor、isVacNo、vacFor を用いて観測する。
- 出力値である Vacno は、newVNo という観測関数を用いて観測する。

```
module AGENCY {
 -- 下準備 他のモジュールの輸入など
 import {
 protecting(SET[X <= view to SKILL { sort Elt -> Skill }]
            * { sort Set -> Skills })
 protecting(NAT * { sort Nat -> Vacno })
 }

 signature {
  -- 状態空間 "Agency" ------
  [Person, Company, Agency, Vacdata]
  -- 構成子関数 ---------------------------------
  op empty   : -> Agency    -- 初期状態
  op apply   : Agency Person Skills -> Agency
  op subscr  : Agency Company Skills -> Agency
  op vacdata : Skills Company -> Vacdata
  -- 観測関数 -----------------------------------------
```

```
 op isPerson  : Agency Person -> Bool
 op skillsFor : Agency Person -> Skills
 op isVacNo   : Agency Vacno -> Bool
 op vacFor    : Agency Vacno -> Vacdata
 op newVNo    : Agency Company Skills -> Vacno
}

axioms {
 var A : Agency    vars P P1 : Person
 var C : Company   vars S S1 : Skills
 var N : Vacno

 -- Agency に P1 のデータが登録されているかどうか? --
 eq isPerson(empty, P) = false .
 eq isPerson(apply(A, P, S), P1) = if P == P1
                                   then true
                                   else isPerson(A, P1) fi .
 eq isPerson(subscr(A, C, S), P) = isPerson(A, P) .

 -- P1 の Skills を取り出す（事前条件：P は他に登録されてない）--
 eq skillsFor(empty, P) = { nil } .
 eq skillsFor(apply(A, P, S), P1) = if P == P1 and isPerson(A, P1) == false
                                    then S
                                    else skillsFor(A, P1) fi .
 eq skillsFor(subscr(A, C, S), P) = skillsFor(A, P1) .

 -- N が Agency にすでに登録されているかどうか? --
 eq isVacNo(empty, N) = false .
 eq isVacNo(apply(A, P, S),N) = isVacNo(A, N) .
 eq isVacNo(subscr(A, C, S), N) = if N == newVNo(subscr(A, C, S), C, S)
                                   then true
                                   else isVacNo(A, N) fi .

 -- VacNo を指定し Agency より vacdata を取り出す（事後条件：N は未登録）--
 eq vacFor(empty, N) = vacdata({ nil }, C) .
 eq vacFor(apply(A, P, S), N) = vacFor(A, N) .
```

```
  eq vacFor(subscr(A, C, S), N) = if N == newVNo(subscr(A, C, S), C, S)
                                     and isVacNo(A, N) == false
                                  then vacdata(S, C)
                                  else vacFor(A, N) fi .

  --  新しいVacNoを割り当てる 初期値は0（出力の取得）--
  eq newVNo(empty, C, S) = 0 .
  eq newVNo(apply(A, P, S), C, S1) = newVNo(A, C, S1) .
  eq newVNo(subscr(A, C, S), C, S1) = if N == newVNo(subscr(A, C, S), C, S1)
                                      then N
                                      else s(newVNo(A, C, S1)) fi .
 }
}
```

explicit state approach 仕様の変更

以上のようなexplicit state approachの仕様に属性として新たに性別を加える。まず、signatureを以下のように変更する。

```
[ Person, Company, Agency, Vacdata, Sex ]
op apply : Agency Person Sex Skills -> Agency
```

ここで、構成子関数であるapplyを変更したので、applyの記述の現れる場所は全て変更する必要がある。これは、公理の等式すべて、またその事前条件・事後条件を記述した条件部分すべてにまで及ぶ。構成子関数に変更を加える場合は、少しの変更により、この仕様のほとんどを書き換える必要がある。

explicit state approach の考察

explicit state approachでは、以上のように個々のstateは、状態集合(ソート)の1つの要素(項)として表現し、操作的な意味合いを持つ状態変化を行うoperationは、それぞれの項の構成子(constructor)として表現される。表現した項は、様々な観測関数により観測される。stateの属性へのアクセス、出力へのアクセスはすべて観測関数により行われる。これは余り直観的な表現方法ではないような気がする。この手法の利点としては、記述した仕様が項書換えシステムにより即実行可能であり、機械的な検証を行うことである。

しかし、この手法による問題点として、以下のようなことが考えられる。

- 公理部分で同様の操作記述を繰り返し行わなければならないことがある。(事前条件と事後条件は観測関数毎に記述されているので、同様の事前条件、事後条件を繰り返し記述しなければならない)
- 項として state が明示的に表現されているため、既存の状態遷移問題に新しい操作を付け加えるときや状態に新しい属性を加えるとき、矛盾がないかどうか、その都度観測関数等をすべて確認する必要がある。
- そのようにして最終的に作成された仕様は明晰性の低いものになりがちである。

6.2 implicit state approach

implicit state approach とは、explicit state approach とは対比的に、Z 仕様のように状態空間を暗示的に取り扱う手法である。代数仕様言語による implicit state approach での状態の表現方法はいくつか提案されているが、ここでは Baumeister により [3] で提案された代数仕様言語 Larch を用いて Z のスタイルで state の仕様を記述する手法を CafeOBJ のコードに置き換えて紹介する。(Larch、CafeOBJ はどちらも代数仕様言語であるが、Larch は多ソート代数と一階等式論理に基盤を持ち、CafeOBJ は順序ソート代数と書き換え論理に基盤を持つ。順序ソート代数は多ソート代数の拡張であり、書き換え論理は等式論理の拡張であるので、この置き換えは容易に行える。)

6.2.1 Baumeister の手法

Baumeister は、state の解釈を以下のように考えることで、代数仕様言語を用いて、Z のスタイルで state の仕様記述を行うことを可能にしている。

- 個々の状態は同一の signature からなるΣ代数である。
- 状態空間は、抽象データ型 (abstruct datatype) とする。ここでは状態と同一の signature からなるΣ代数の集合 (同型な代数) である。
- 初期状態は、それぞれの状態の要素がそれぞれの要素の初期状態と等しいΣ代数である。

- 状態変化を行う操作も抽象データ型として与える。操作の適用により変化するものは、state の構成要素 (属性) の解釈だけである。

例題: カウンター

Baumeister の手法をカウンターの記述例を用いて説明する。CafeOBJ は実行可能な仕様言語であるが、この仕様は CafeOBJ において実行可能なコードではない。

```
module COUNTER {
  extending(NAT)
  op c : -> Nat   -- 定数項 c は、カウンターの状態を表す
}

module ZERO {
  using(COUNTER)
  eq c = 0 .    -- 初期状態 c=0
}

-- Z の delta にあたる pushout function
module DELTACOUNTER {
  using(COUNTER)
  using(COUNTER * { op c -> c' })
}

module INC {
  using(DELTACOUNTER)
  eq c' = s (c) .
}

module DEC {
  using(DELTACOUNTER)
  ceq s (c') = c :if  not c' == 0 .
}
```

この仕様では、state は定数項 c で表現され、c の解釈の変更によりカウンターの状態遷移を表現している。具体的には、状態空間は、代数 COUNTER と同じ signature を持つ

Σ 代数の集合であり、初期状態は COUNTER のΣ代数 ZERO である。状態変化を促す操作 INC,DEC は、２つのΣ代数それぞれの項 c の間の解釈を与える。c は事前状態のカウンターの state を c' は事後状態のカウンターの状態を表現している。このような c の解釈の変更により、代数仕様言語を用いて、Z のスタイルで仕様を記述することが Baumeister の手法である。

理論的な裏付け

この手法の基盤となる Relation as Abstract Datatypes Approach に関しては Baumeister により文献 [2] で詳説してある。ここでは簡単に紹介することにする。

[2] では、一階述語論理と等式論理などの base institution 間の関係を institution を用いて定義してある。この事実は、より直感的には、集合論の二項関係を２つの代数間の関係へと対応づけできるということを表す。すなわち、ソート $s_1 \ldots s_m$ の変数 $x_1 \ldots x_n$ からなる state は、state invariant を充足する、多ソート signature $\Sigma =< S, Op >, S = \{s_i \mid i = 1 \ldots m\}, Op = \{x_i : s_i \mid i = 1 \ldots n, s_i \in S\}$ からなる代数の集合として見ることができる。

6.2.2 Baumeister の手法による雇用代理店の仕様記述

次に実際に Baumeister の手法を用いた Employment Agency の記述を見る。まず、状態空間は次のように与える。PRODUCT,FUNCTION は、各々、順序対と関数の集合を規定する CafeOBJ のモジュールとして別に定義する。personlist と companylist は状態の属性を意味する。

```
module AGENCY {
  [Person, Skills, Company]
 import {
  protecting(SET[X <= view to SKILL { sort Elt -> Skill }]
             * { sort Set -> Skills })
  protecting(PRODUCT[Company, Skills] * { sort Prod -> Vacdata})
  protecting(NAT * { sort Nat -> Vacno })
  using(FUNCTION[Person, Skills] * { sort Rel -> PersonList })
  using(FUNCTION[Vacno, Vacdata] * { sort Rel -> CompanyList })
 }
 signature {
  op personlist : -> PersonList
```

```
  op companylist  : -> CompanyList
 }
}
```

また、初期状態は、personlist と companylist が空集合である次のような代数である。

```
module EMPTY {
  using(AGENCY)
  axioms {
   personlist = { } .
   companylist = { } .
  }
}
```

次に、状態変化を行う操作を記述する準備として、Z 記法のスキーマのΔに相当する DELTAGENCY を定義する。

```
module DELTAAGENCY {
  using(AGENCY)
  using(AGENCY * { op personlist -> personlist' }
               * { op companylist -> companylist' })
}
```

Apply,Surscr はこの DELTAAGENCY を用いて次のように記述できる。update-PL と update-CL は、各々Personlist と companylist に要素を付け加える関数を意味する。isPerson はisVacNo は explicit state approach で定義したものと同様のものである。

```
module APPLY {
  using(DELTAAGENCY)
  signature {
  op p-input : -> Person
  op s-input : -> Skills
  }
  axioms {
  cq campanylist' = campanylist :if not isPerson(p-input) .
  cq personlist' = update-PL(personlist, p-input, s-input)
     :if not isPerson(p-input) .
  }
}
```

```
module SUBSCR {
  using(DELTAAGENCY)
  signature {
  op c-input : -> Company
  op s-input : -> Skills
  op n-out   : -> Vacno
  }
  axioms {
  cq personlist' = personlist :if not isVacNo(n-out) .
  cq companylist' = update-CL(companylist, n-out, [c-input, s-input])
      :if not isVacNo(n-out) .
  }
}
```

Baumeister の手法による仕様の変更

Baumeister の手法による仕様に属性として新たに性別を加える。仕様の変更箇所は、まず、状態空間を表すモジュール AGENCY に、ソート Sex を加え、新たに順序対 Pdata を定義し、PersonList を Pdata から Skills への関数の集合へと変更する。

```
[Person, Skills, Company, Sex]
  protecting(PRODUCT[Person, Sex] * { sort Prod -> Pdata})

  using(FUNCTION[Pdata, Skills] * { sort Rel -> PersonList })
```

また、モジュール APPLY を、Sex の追加に対応するように変更する。update2 は、Person, Sex,Skills をとり update する関数へと作り直す必要がある。

```
module APPLY {
  using(DELTAAGENCY)
  signature {
  op p-input : -> Person
  op s-input : -> Skills
  op sex-imput : -> Sex
  }
  axioms {
  cq companylist' = companylist :if not isPerson(p-input) .
```

```
  cq personlist' = update-PL2(personlist, p-input, sex-input, s-input)
     :if not isPerson(p-input) .
  }
}
```

Baumeister の手法による仕様の変更も、このように Z 記法と同様に最小限の変更で解決する。

Baumeister の手法による state 記述の考察

Baumeister の手法を用いる利点は、Z 記法の state 記述のスタイルで代数仕様を作成することができることである。これにより、事前状態と事後状態の差異により容易に状態の記述を行うことができる。また、Baumeister の手法による state 記述の問題点としては次のようなことが考えられる。

- 状態の要素と事前事後状態間の関係を記述するための論理を新たに取り付けないといけない。
- 機械的に検証するためには、新たに実行可能なモジュールを作成するなど多くの準備が必要である。

6.3 書き換え規則を用いた手法

6.3.1 書換え論理

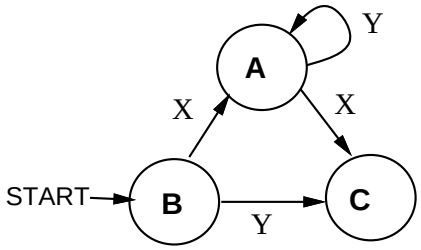
CafeOBJ では基盤となる論理体系として書換え論理を採用している。書換え論理については、Meseguer により文献 [13] に詳説されている。書換え論理ではオブジェクトの動的な変化を表す規則として書き換え規則 (rewrite rule) を定義してある。この書き換え規則を用いれば、同期システムの記述も容易に行うことができる。書き換え規則は、CafeOBJ の中でルール宣言として実現されている。

ルール宣言

CafeOBJ では、オブジェクトの変化の言明を書き換え規則に基づくルール宣言の構文を用いて記述可能である。ルール宣言は CafeOBJ では次のような記法で記述される。

```
rule  事前オブジェクト => 事後オブジェクト
crule 事前オブジェクト => 事後オブジェクト
  :if 事前条件 and 事後条件
```

これを用いれば、状態変化の記述をより直感的に記述できる。例えば、次のようなオートマトンの仕様は以下のようなコードになる。




```
module AUTOMATON {
   signature{
   [ State ]
   ops A B C : -> State
   }
   axioms{
   rule [Y]: A => A .
   rule [X]: B => A .
   rule [Y]: B => C .
   rule [X]: A => C .
   }
}
```

6.3.2 書き換え規則を用いた雇用代理店の仕様記述

書き換え規則は、事前状態から事後状態への変化の言明を直感的に記述できる。書き換え規則を用いて、ここでは次のように状態遷移問題の表現を行う。

- 状態空間をソート Agency、状態をそのソートの要素として項で表現する。状態の属性は構成子関数 addPerson,addCompany の引数に含ませる。
- 初期状態は、状態空間の表すソートの初期状態にあたる１つの要素 (定数項) として表現される。

- 状態遷移はルール宣言を用いて表現する。その際、apply,subscr は事前状態と入力を引数に取り事後状態を返す操作として定義する。
- 出力は、その状態の構成子関数の引数に含ませる。

状態遷移関数 apply,subscr とルール宣言を用いて記述した。apply は、入力として Person と Skills をとり、事前状態 A を事後状態 A' へと遷移させる。subscr も同様に定義できる。ルール宣言は以下のように用いる。

```
crule operation 名 (事前状態, 入力値 1, 入力値 2) => 事後状態
      : if 事前条件 and 事後条件 .
```

ここでは、いくつかのその他の観測関数は省略している。

```
module AGENCY {
 import {
 protecting(SET[X <= view to SKILL { sort Elt -> Skill }]
            * { sort Set -> Skills })
 protecting(NAT * { sort Nat -> Vacno })
 }

 signature {
  -- 状態空間 "Agency" ------
  [Person, Company, Agency, Nat < Output]
  -- 構成子関数 --------------------
  op empty   : -> Agency
  op o-empty : -> Output
  op addPerson  : Agency Person Skills Output -> Agency
  op addCompany : Agency Company Skills Vacno Output -> Agency
  -- 状態遷移関数 ------------------
  op apply   : Person Skills Agency -> Agency
  op subscr  : Company Skills Agency -> Agency
  -- その他の関数 ------------------
  op isPerson  : Agency Person -> Bool
  op skillsFor : Agency Person -> Skills
  op isVacNo   : Agency Vacno  -> Bool
  op vacFor    : Agency Vacno  -> vacdata
```

```
  op newVNo     : Agency  -> Vacno
   }

 axioms {
  var A  : Agency     vars P P1 : Person
  var C : Company     var S  : Skills
  var N : Vacno       var O  : Output

  -- 状態遷移関数の記述
  crule apply(P, S, A) => addPerson(A, P, S, o-empty)
        :if (not isPerson(A, P)) and true .
  crule subscr(C, S, A) => addCompany(A, C, S, newVNo(A),newVNo(A))
        :if true and (not isVacNo(newVNo(A))) .

  -- Agency にその Person のデータが既に登録されているかどうか --
  eq isPerson(empty, P) = false .
  eq isPerson(addPerson(A, P, S, O), P1) = if P == P1
                                    then true
                                    else isPerson(A, P1) fi .
  eq isPerson(addCompany(A, C, S, N, O), P) = isPerson(A, P) .

  --  新しいVacNo を割り当てる 初期値は0 --
  eq newVNo(empty) = 0 .
  eq newVNo(addPerson(A, P, S, O)) = newVNo(A) .
  eq newVNo(addCompany(A, C, S, N, O)) = s(N) .
 }
}
```

書き換え規則を用いた手法による仕様の変更

この仕様に属性として人間の性別を追加する。まず、ソートにSexを追加し、構成子関数addPersonの引数にSexを追加する。また、状態遷移の式の入力にsexを追加し書き直す。

```
[Person, Company, Agency, Nat < Output, Sex]
op addPerson  : Agency Person Sex Skills Output -> Agency
```

```
op apply   : Person Sex Skills Agency -> Agency

  crule apply(P, SEX, S, A) => addPerson(A, SEX, P, S, o-empty)
        :if (not isPerson(A, P)) and true .
```

構成子関数の変更に伴い、観測関数に現れる addPerson の部分に機械的に引数 SEX を追加する必要がある。しかし、事前条件、事後条件の記述が観測関数に混ざってない分、変更箇所は、explicit state approach の場合よりは少なくて済む。

書き換え規則を用いた手法の考察

以上のように書き換え規則を用いれば、状態変化の記述がより直感的に記述可能である。この手法を用いれば、状態遷移を事前状態と事後状態の差異により記述可能である。ルール宣言の中で、事前条件・事後条件は条件部として、入力は状態遷移関数の引数として、出力は事後状態の要素の一部として１つのルール宣言の中に記述できる。しかし、状態の表現を項を用いて行い、属性と出力については、状態の構成子の中に含ませているため、この仕様に変更を加える際の作業は Z 仕様と比べて面倒である。

6.4 Hidden Algebra に基づく手法

Hidden Algebra については、文献 [4] で詳説してある。Hidden Algebra に基づく hidden sort を用いた手法は、オブジェクトの振舞に注目して仕様の記述を行う終代数的意味論に基づいたアプローチである。hidden sort を用いることでオブジェクトのデータ構造に依存しない仕様記述が可能である。

6.4.1 hidden sort を用いた state のモデル化

この手法により次のように state を表現する。

- 状態空間は hidden sort として解釈を与える。各々の状態はこのソートの要素と解釈する。
- 初期状態は、hidden sort の中の初期状態の意味を持つ１つの要素である。

- 状態遷移を行う操作は hidden sort と入力を引数にとり hidden sort を返す method 関数として表現する。
- 状態の構成要素は観測関数を用いて観測する。この観測関数を attribute と呼ぶ。

例題:FLAG

最初に hidden sort を用いた手法の例として FLAG の仕様を見ていく。FLAG では、旗の状態が変化し、hidden sort で表現する対象となる。ここでは、旗のデータの表現には依存せずに、旗の振舞を method 関数 (up,down,rev) と attribute(up?) のみで定義する。

```
module FLAG {
  signature {
    hidden[ Flag ]
    ops (up_)(down_)(rev_) : Flag -> Flag  -- method
    op up?_ : Flag -> Bool                 -- attribute
  }
  axioms {
    var F : Flag .
    eq up? up F = true .
    eq up? down F = false .
    eq up? rev F = not up? F .
  }
}
```

このコードでは、state space を hidden sort Flag として定義している。状態 (Flag) は、method 関数により変化するが、このアプローチでは、その実体は項などにより明示しない立場をとっていることに注意する。method 関数は構成子関数とは性質が異なり、オブジェクトの振舞により記述される関数である。例えば、この仕様の method 関数 UP は、Flag で示されるオブジェクトの実体が (自分で) 旗を上げたことを意味する。状態の観測は観測関数 (UP?) により行う。この FLAG の仕様の場合、その状態を見極めるには、Flag の大外の method 関数 (直前に行われた操作) のみ考慮すれば、それで十分である。

6.4.2 hidden sort を用いた雇用代理店の仕様記述

hidden sort を用いた手法で雇用代理店の仕様を記述する。

```
module AGENCY {
  import {
    protecting(SET[X <= view to SKILL { sort Elt -> Skill }]
               * { sort Set -> Skills })
    protecting(NAT * { sort Nat -> Vacno })
    }

  signature {
    hidden [ Agency ]   -- Agency は状態を表す hidden sort
    [ Person Company ]  -- visible sort
    -- initial state    -- hidden の場合は、考えなくてもよい
    -- method 関数
    op apply  : Person Skills Agency -> Agency
    op subscr : Company Skills Agency -> Agency

    -- attribute 関数
    op isPerson  : Agency Person -> Bool
    op pskill？  : Person Agency -> Skills
   }

  axioms {
   var A:Agency              var P:Person
   vars C C1:Company         vars S S1:Skills
   var N:Vacno

   -- attribute
   eq isPerson(empty, P) = false .
   eq isPerson(apply(P, S, A), P1) = if P == P1
                                     then true
                                     else isPerson(A, P1) fi .
   eq isPerson(subscr(C, S, N, A), P) = isPerson(A, P) .

   eq pskill?(empty, P) = { nil } .
   eq pskill?(apply(P, S, A), P1) = if P == P1 and isPerson(A, P1) == true
                                     then S
                                     else pskill?(A, P1) fi .
```

```
   eq pskill?(subscr(C, S, N, A), P) = pskill?(A, P1) .
  }
}
```

hidden sort を用いた手法による仕様の変更

この仕様に属性として性別を加える。この場合、hidden sort Agency の要素の構造は隠蔽されているので変更する必要はない。変更する箇所は、Sex を visible sort として加えることと、method 関数 apply の引数の Sex を加え、apply を用いて記述している attribute を機械的に変更することである。

```
[ Person Company Sex ] -- visible sort
op apply : Person Sex Skills Agency -> Agency
-- attribute
   eq isPerson(empty, P) = false .
   eq isPerson(apply(P, SEX, S, A), P1) = if P == P1
                                     then true
                                     else isPerson(A, P1) fi .
   eq isPerson(subscr(C, S, N, A), P) = isPerson(A, P) .
...
```

hidden sort を用いた手法の考察

hidden sort を用いた手法で記述された仕様は、Agency が hidden sort で記述してある点を除けば explicit state approach の仕様と違いがないように見える。しかし、Hidden Algebra による状態遷移のモデル化は method 関数によりオブジェクトの振舞を規定したものであるので、explicit state approach による仕様とは随分性質の異なるものである。

hidden sort を用いた手法では、状態のデータ構造を考えずに仕様を記述できる。これにより、属性の追加削除の際に頭を迷わす必要がなくなり、状態遷移関数を中心にした仕様の記述ができる。

hidden sort を用いた手法の難点としては、機械的な検証を行うため、attribute を記述する際に、複雑な関数型プログラミングのようなコードを作成しなければならない点がある。このコードは弟三者にとって可読性の低いコードになりやすい。また、仕様の変更の際は、explicit state approach と同様に結果的に多くの箇所の変更を必要とする。

第 7 章

まとめ: state の仕様記述の比較と考察

本章では、前章で行った様々な手法による state 記述に関しての考察を行う。

7.1 比較結果と考察

仕様の明晰性の比較

4.2 節の問題の分析に基づく細かい明晰性の比較の結果を表に示す。ここではまず、各々の手法による仕様記述の中でのモデル化が問題の構造を崩さずに自然になされているかという点に着目し仕様の明晰性の分析を行った。明晰性は人間的な曖昧な問題であり、個人個人により捉え方に差があると考えられる。今回の分析結果は筆者の主観的な判断である点は拭いきれないが、考察において何故そう評価したかを細かく記述した。また、各々の表現方法による利点不利点についても検討した。

比較項目・比較表

状態空間・状態の記述　状態空間の表現が自然に分かりやすくモデル化できる。
状態属性の記述　状態の属性の表現が自然に分かりやすくモデル化できる。
状態遷移関数の記述　状態遷移関数の表現が自然に分かりやすくモデル化できる。
pre,post 状態による記述　object の事前・事後状態の差異により状態遷移が記述できる。
事前条件の記述　状態遷移の事前条件の記述が容易に行える。
事後条件の記述　状態遷移の事後条件の記述が容易に行える。
入力の記述　入力の記述が直観的に分かりやすく行える。
出力の記述　出力の記述が直観的に分かりやすく行える。

	Z	CafeOBJ			
比較項目	schema	explicit	implicit	rule	hidden
状態空間・状態の記述	◎	○	△	○	◎
状態属性の記述	◎	△	◎	○	△
状態遷移関数の記述	○	△	△	◎	○
pre,post 状態による記述	◎	×	◎	◎	×
事前条件の記述	◎	△	◎	◎	△
事後条件の記述	◎	△	◎	◎	△
入力の記述	◎	○	◎	◎	◎
出力の記述	◎	○	◎	○	○

考察

Z 記法でのスキーマを用いた仕様記述は、事前・事後状態の差異による記述が容易に行える。状態遷移関数を操作スキーマという箱のような枠組で表現するのは、やや直観的な表現ではない気がするが、操作スキーマを用いることで状態遷移の全ての要素はその中で分かりやすく記述できる。

状態遷移の概念を一番直観的に表現できるのは、オブジェクトの変化の言明である書換え規則を用いた手法である。この手法では、状態遷移関数を事前状態と事後状態へ変化させる関数として直観的に記述可能である。この手法では状態を始代数に基づく項として表現しているため、属性と出力を構成子関数の引数に含ませて表現している。これにより、

状態は属性と出力を引数に持つ項として表現され、状態遷移の度に最外にこの構成子関数が加わる形となる。状態は状態遷移により過去の履歴を持たずに変化するオブジェクトであるので、この状態表現はやや直観的でないと評価している。

implicit state approach に基づく Baumeister の手法は、状態空間と状態遷移関数を抽象データ型と解釈しており代数手法として難しいモデル化を必要とする。しかし、この解釈により Z 記法のスキーマと同様の記述スタイルを代数手法で実現できる。

explicit state approach による記述は、状態を項として表現するために、状態遷移関数を構成子関数で表現する。これは仕様の機械的な実行を意識したモデル化であり、状態遷移のモデルとしては不自然である。この手法での仕様の記述スタイルでは、事前状態、事後状態での記述は行われていない。

Hidden Algebra に基づく手法は、終代数意味論に基づく全く新しい記述へのアプローチである。この手法では仕様をオブジェクトの振舞により記述する。この捉え方から、状態遷移は状態の実体となるオブジェクトの振舞であると解釈する。この手法は従来の手法とは全く異なる問題の捉え方であり、事前状態・事後状態の差異による記述の明晰性の比較項目では、単純に比較できない。この手法による状態遷移の記述については弟 8 章で再検討する。

追加・変更の容易さ (拡張性) についての比較

ここでは仕様の変更・拡張 (追加や削除) の容易さに着目し、各手法の分析を行う。比較項目、比較結果は次に示す通りである。

比較項目・比較表

属性の変更　属性値の変更、属性の追加、削除が容易に行える。
状態遷移関数の変更　状態遷移を行う操作の変更、追加、削除が容易に行える。
事前・事後条件の変更　事前条件・事後条件の変更、追加、削除が容易に行える。
入力の変更　入力の変更、追加、削除が容易に行える。
出力の変更　出力の変更、追加、削除が容易に行える。

	Z	CafeOBJ			
比較項目	schema	explicit	implicit	rule	hidden
属性の変更	○	△	○	△	◎
状態遷移関数の変更	◎	△	◎	◎	△
事前事後条件の変更	◎	△	△	◎	△
入力の変更	◎	○	◎	◎	○
出力の変更	◎	◎	◎	◎	◎

考察

Z 記法のスキーマを用いた手法は、属性の変更は状態スキーマの中で、その他の変更は操作スキーマの中で容易に行える。状態スキーマの内部では属性の型を Z 記法のプリミティブを用いて明示的に定義する必要がある。implicit state approach も拡張性に関しては Z 記法と同様の結果であるが、事前・事後条件は各々の等式やルールの条件部で記述する必要があり、変更の際には複数箇所の変更を必要とする。

explicit state approach は、前章でも見たように変更の際に非常に多くの労力を要する。これに対し、explicit state approach と同様に状態を項として表現する書き換え規則を用いた手法では構成子関数の変更に係わる属性の変更以外は容易に行える。これは、状態遷移に係わる全ての要素を状態遷移を表現する書き換え規則の中で記述可能なためである。

Hidden Algebra に基づく手法では、属性と出力は外部から観測関数 (attribute) で観測するので、この関数の変更のみで済み容易に拡張できる。method 関数の変更に関係する状態遷移関数と入力の変更を行うには、観測関数の関連場所すべてを変更する必要がある。また、事前事後条件に関しては各々観測関数の中に散りばめられているので、変更の際に厄介である。

7.2 state 記述に関する考察

本研究では、様々な state の記述方法の比較を実際に同一の問題 (雇用代理店) の記述を通して行った。本節では、仕様言語に求められる性質として、明晰性、表現力、再利用性、その他、仕様の検証手法を含めた総合的な比較の結果を以下の表に示す。

比較項目・比較表

明晰性　分かりやすい直観的な仕様記述が行える。
拡張性　仕様の変更・拡張のしやすい仕様が作成できる。
表現力　表現力豊かな仕様記述が行える。
再利用性　再利用性の高い仕様が作成できる。
検証手法　検証が容易で検証手法が豊富である。

	Z	CafeOBJ			
比較項目	schema	explicit	implicit	hidden	rule
明晰性	◎	△	◎	○	◎
拡張性	◎	△	○	△	○
表現力	◎	○	○	○	○
再利用性	○	◎	○	◎	◎
検証手法	△	◎	△	◎	◎

考察

この表で示すように、今回の研究で用いた手法では状態遷移の仕様記述に関しては、どの手法も長所と欠点を合わせ持っている。implicit state approach をとる Baumeister の手法と Z 記法のスキーマによる記述では明晰性・拡張性の高い仕様を記述ができるが、検証時にかかる負担が大きい。explicit state approach や hidden sort を用いた手法では、検証手法は充実しているが、可読性の低く拡張の難しい仕様になりやすい。その中でも、書き換え規則を用いた手法は、直観的な状態遷移の記述と検証が比較的容易に行える。

7.3 state 記述の潮流と今後の展望

本研究では、Z 記法で標準的である事前状態と事後状態の差異による状態変化の記述を代数仕様言語 CafeOBJ の様々な記述スタイルでどのように記述できるかを見てきた。

代数仕様言語を用いて状態遷移問題を記述する際は、explicit state approach を用いるしか有効な手法がなかったのだが、近年の研究で代数間の射を用いて、Z 記法のスタイルで仕様を記述する手法 (Baumeister の手法) や書換え規則を用いる手法、代数仕様の新しい概念である Hidden Algebra を用いた手法が提案され、様々な記述スタイルで状態遷移問題を記述できるようになった。

それぞれの記述スタイルについては弟６章で解説したが、なかでも特に書換え規則を用いた手法を用いれば、自然な事前状態と事後状態の記述で状態遷移問題を代数的手法で記述でき、この手法は Z 記法のスキーマを用いた手法と同等の記述力を持つことを本稿では実際の仕様記述を通して示した。

代数仕様言語を用いて状態遷移問題を記述する際、一番の問題となるのは状態の表現方法である。現在は状態の属性や出力を構成子関数の引数とする項として表現する方法が普通であるが、この方法では仕様の変更を行う場合に、その項を含む公理全ての変更が必要になる。これに対し、Hidden Algebra に基づく手法を用いれば状態のデータ構造に依存しない仕様を記述できる。この手法は現在設計段階であり、有効な例題が少ないのだが、書き換え規則や順序ソートを Hidden Algebra に採り入れる研究も行われており、今後の研究成果が期待される。

第 8 章

CafeOBJ による拡張・変更の容易な新しい state 記述手法の提案

本章では、4.2 節で行った事前状態、事後状態による問題の整理に基づき、state の記述方法を明確に定式化し、さらに、状態属性などの状態遷移の構成要素の変更が容易にできるような記述手法を模索する。

8.1 クラス宣言と書き換え規則を用いた state の記述手法

弟 6 章で書き換え規則を用いた手法を紹介した。この手法の唯一の欠点は、状態の表現を項で行うため、状態の表現が不自然であり、出力、属性を構成子関数の引数として表現したため、変更が容易に行えないということであった。本節では、クラス宣言を用いて、state space をクラスで表し、これらの変更が容易に行えるように書き換え規則による手法を拡張した。また、4.3 節での分析の方法で整理した状態遷移の問題に関しては、機械的に仕様作成が可能なように記述スタイルの定式化を試みた。

8.1.1 CafeOBJ のクラス宣言について

CafeOBJ では、クラス宣言の構文がある。これは状態の変化するオブジェクトとその属性を体系的に記述するのに便利である。構文は、

```
class クラス名 {
```

属性名１ ： 属性１のソート
属性名２ ： 属性２のソート
. . . . . .
}

となり、これによりクラスオブジェクトの定義ができる。この定義により、クラスオブジェクトのインスタンスが次のように自動的に生成される。

< オブジェクト識別子 ： クラス名 |
属性名１ = 属性値１,
属性名２ = 属性値２,
. . . . . >

8.1.2 クラス宣言を用いた state の記述方法の定式化

state をより自然に形式的に記述するために、クラス宣言と書き換え規則を用いて、以下のような状態遷移の記述形式を提案する。

(状態の表現)
class 状態クラス名 {
状態属性名１ ： ソート
状態属性名２ ： ソート
. . . . . .
出力値 ： ソート
}

(状態の遷移の表現)
rule operation 名 (事前状態, 入力値１, 入力値２) => 事後状態 .

crule operation 名 (事前状態, 入力値１, 入力値２) => 事後状態
:if 事前条件 and 事後条件 .

すなわち、整理した各々の状態遷移の要素は以下のように定式的に記述できる。

- 状態はクラスオブジェクトとして、その属性はクラスの属性として定義する。
- 初期状態は属性が全て初期値であるインスタンスである。
- 状態遷移関数は、入力値と事前状態を引数にとり、事後状態を返す関数として定義する。
- 事前条件は crule の条件部に書く。(状態遷移が発生する条件として)
- 事後条件も crule の条件部に書く。(状態遷移が発生する条件として)
- 入力値は状態遷移関数の引数に記述する。
- 出力値は Output として状態の持つ属性として記述する。

8.1.3 クラス宣言を用いた雇用代理店の記述

以上のような記述手法で、4.3 節の雇用代理店の記述を行ってみた。

```
module AGENCY {
 import {
 protecting(SET[X <= view to SKILL { sort Elt -> Skill }]
            * { sort Set -> Skills })
 protecting(NAT * { sort Nat -> Vacno })
 }

 signature {
 [ Person, PersonSet, Company, CompanySet, ObjectId, Output ]

  -- 構成子関数 ----------------------------------
  op p-empty : -> PersonSet  -- 属性 PersonSet の初期値
  op c-empty : -> CompanySet -- 属性 CompanySet の初期値
  op o-empty : -> Output     -- 「出力なし」を意味する出力値
  op addPerson  : PersonSet Person Skills -> PersonSet
  op addCompany : CompanySet Company Skills Vacno -> CompanySet

  -- 状態はクラス宣言を用いて表現する。
  class Agency {
```

```
    p-set : PersonSet
    c-set : CompanySet
    out   : Output
 }

 -- 状態遷移関数は、入力値と事前状態を引数に取り、
 -- 事後状態を返す関数である。
 op apply   : Person Skills Agency -> Agency
 op subscr  : Company Skills Agency -> Agency

 -- others ------------------------------------------
 -- 条件判断などのために準備した関数、述語類
 op isCand    : PersonSet Person -> Bool
 op isVacno   : CompanySet Vacno -> Bool
 op newVNo    : CompanySet  -> Vacno
  }

axioms {
 var PS : PersonSet  CS : CompanySet
 vars P P1 : Person  O  : Output
 var C : Company   var S  : Skills
 var N : Vacno     vars Id Id' : ObjectId

 -- 書き換え規則により状態遷移を表現する。

 -- apply -------------------------------------------------
 crule apply(P, S, < Id : Agency |
                          p-set = PS,
                          c-set = CS,
                          out   = O >)
  =>  < Id' : Agency |
              p-set = addPerson(PS, P, S),
              c-set = CS,
              out   = o-empty. >)
 -- 事前・事後条件
 :if (not isCand(PS, P)) and true .
```

```
  -- subscr ---------------------------------------------------
  crule subscr(C, S, < Id : Agency |
                           p-set = PS,
                           c-set = CS,
                           out   = 0 >)
   =>  < Id' : Agency |
               p-set = PS,
               c-set = addCompany(CS, C, S newVNo(CS))
               out = newVNo(CS) > .
  -- 事前・事後条件
  :if true and (not isVacno(newVNo(CS))) .

  -- others ---------------------------------------------------

  -- Agency にその Person のデータが既に登録されているかどうか --
  eq isCand(p-empty, P) = false .
  eq isCand(addPerson(PS, P, S),  P1)
            = if P == P1 then true
                         else isCand(PS, P1) fi .

  -- Agency にその Vacno のデータが既に登録されているかどうか --
  eq isVacNo(c-empty, N) = false .
  eq isVacNo(addCompany(CS, C, S, N), N1)
     = if N1 == newVNo(addCompany(CS, C, S, N))
             then true
             else isVacNo(CS, N) fi .

  --  新しいVacNo を割り当てる 初期値は0 --
  eq newVNo(c-empty) = 0 .
  eq newVNo(addCompany(CS, C, S, N)) = if N == newVNo(CS)
                                       then s(newVNo(CS))
                                       else N fi .
 }
}
```

state space にクラスを用いる利点は、状態とその属性を自然な形でモデル化できることがある。また、各状態属性が別々のソートで定義できるため、弟 6 章での書き換え規則を用いた手法で問題であった属性の変更も最小限の変更で可能となる。この記述では出力を状態の属性と同列に扱い、状態の持つ要素と解釈したが、このモデル化は大変自然なモデル化であると言える。また、4.3 節のような形で整理された問題は、この記述スタイルを用いて定式的に記述可能である。

8.2 Hidden Algebra に基づく明晰性・拡張性の高い state 記述手法の模索

本節では、Hidden Algebra に基づく手法での明晰性・拡張性の高い state 仕様記述の可能性を探る。

8.2.1 複数の hidden sort を用いて仕様を具象化する方法

第６章では、Hidden Algebra に基づく手法として、雇用代理店全体の状態 (Agency) を hidden sort として扱い、Agency の振舞により仕様記述する手法で記述を行った。しかし、この手法を貫き通すと、問題が複雑になるにつれて仕様が必要以上に長文になり、可読性の低いものになると考えられる。

本節では、第６章の仕様をもとに、Hidden Algebra 仕様の hidden sort の具象化を行う手法を模索する。hidden sort を具象化する利点は、１つの hidden sort から複数の hidden sort へ具象化を行うことで method 関数を適応する要素が明確になり、仕様の拡張が容易になることである。

一段階の具象化

まず、Agency のそれぞれの属性 (PersonList,CompanyList) を hidden sort として定義し、一段階具象化した仕様記述を行ってみた。

state の属性	hidden sort
initial state	各々の属性が initial 状態である状態
状態遷移を行う操作	hidden sort を引数に取り hidden sort を返す method 関数
入力	method 関数の引数
出力	観測関数を用いて観測する
事前条件	各観測関数毎に記述
事後条件	各観測関数毎に記述

この解釈により雇用代理店の仕様記述は次のようになる。

```
module AGENCY {
  import {
    protecting(SET[X <= view to SKILL { sort Elt -> Skill }]
               * { sort Set -> Skills })
    protecting(NAT * { sort Nat -> Vacno })
    }

  signature {
    hidden [ PersonList CompanyList ]   -- hidden sort の宣言
    [ Person Company ]  -- visible sort
    -- initial state    -- hidden の場合は、考えなくてもよい
    -- method 関数
    op apply  : Person Skills PersonList -> PersonList
    op subscr : Company Skills CompanyList -> CompanyList

    -- attribute 関数
    op isPerson  : PersonList Person -> Bool
    op pskill？  : Person PersonList -> Skills
   }

  axioms {
   var  PL   :PersonList     var P     :Person
   vars C C1 :Company        vars S S1 :Skills
   var  N    :Vacno

   -- attribute
```

```
    eq isPerson(empty, P) = false .
    eq isPerson(apply(P, S, PL), P1) = if P == P1
                                       then true
                                       else isPerson(PL, P1) fi .
    eq isPerson(subscr(C, S, N, PL), P) = isPerson(PL, P) .

    eq pskill?(empty, P) = { nil } .
    eq pskill?(apply(P, S, PL), P1) = if P == P1 and isPerson(PL, P1) == true
                                       then S
                                       else pskill?(PL, P1) fi .
    eq pskill?(subscr(C, S, N, PL), P) = pskill?(PL, P1) .
  }
}
```

この具象化された仕様では、当初のものと比較して、各々の attribute 関数の適応する属性要素が明確になるため、状態属性や出力の変更箇所を少なく抑えることができる。また、method 関数についても同様で対応する hidden sort が明確になるので、不必要な記述を必要としない。

この仕様の具象化においては、hidden sort と metohd 関数の定義、変数の定義の書き換えを行った以外は全く変更を要しなかった。もし、順序ソートの概念が hidden sort の世界に存在し、PersonList を Agency の下位ソートに宣言できれば、全く変更なしに済んでしまうだろう。

8.2.2 クラス宣言を用いて複数の hidden sort を定義する手法

Hidden Algebra を用いる手法の利点は、状態のデータ構造に左右されずに仕様の記述が行えるということである。ここでは、8.1 節のクラス宣言と書き換え規則を用いた手法の状態属性を hidden sort で定義できるように拡張する。この記述方法により、本来、状態のデータ構造を形成するものとして表現されていた属性や出力の変更も容易に行えるようになる。

クラス宣言を用いて複数の hidden sort を定義する手法の定式化

この手法は基本的に 8.1.2 節の定式化を基盤としている。まず、状態空間をクラスで定義し、各々の状態属性を hidden sort で定義する。method 関数としては状態遷移を明示

的に表現する状態クラスを引数にとるものと、hidden sort の振舞を規定するものの２種類を準備する。属性、出力などは attribute 関数により参照する。

```
(hidden sort の定義)
hidden [ 状態属性１ 状態属性２ ... ]

(状態の表現)
class 状態クラス名 {
   状態属性名１ : hidden-sort
   状態属性名２ : hidden-sort
   .....
   }

-- method1 --------------
  -- 状態遷移関数 -------
op メソッド名１ : 事前状態クラス　入力値１ ... -> 事後状態クラス
op メソッド名１ : 状態属性１ 入力値１ ... -> 状態属性１
...

-- method2 --------------
...

-- attribute ------------
op 述語名１　 : hidden-sort データ１ データ２ -> Bool
op 述語名２　 :  ...

op 観測関数名１ : hidden-sort データ１ データ２ -> データ
op 観測関数名２ :  ...

-- axioms ---------------

-- 状態遷移の記述
crule メソッド名（事前状態クラス 入力値１　...）=> 事後状態クラス
 :if （事前条件）and（事後条件）.

-- attribute 関数の記述
```

...

クラス宣言を用いて複数の hidden sort を定義する手法による雇用代理店の記述

以上の手法に基づく雇用代理店の仕様記述は次のようになる。

```
module AGENCY {
  signature {
  hidden [ PersonList CompanyList ]
  [ Person Company Cdata ObjectId, Nat < Output ]

  class Agency {
   p-list : PersonList
   c-list : CompanyList
   }

  -- 構成子関数 ----------------------
  op o-empty : -> Output
  op p-empty : -> PersonList
  op noPerson : -> Person
  op cdata : Company Skills Vacno -> Cdata

  -- method of apply ---------------------------
  op apply  : Agency Person Skills -> Agency
  op apply  : PersonList Person Skills -> PersonList

  -- method of subscr --------------------------
  op subscr : Agency Company Skills -> Agency
  op subscr : CompanyList Company Skills -> CompanyList

  -- attribute -----------------------
  op out      : Agency -> Output
  op isPerson : PersonList Person -> Bool
  op getPdata : PersonList Person -> Skills
  op isVacno  : CompanyList Vacno -> Bool
```

```
op getCdata : CompanyList Vacno -> Cdata
op newVNo : CompanyList -> Nat
}

axioms {
  var PS    : PersonList      var CS : CompanyList
  vars P P1 : Person          var S  : Skills
  var O     : Output          var CD : Cdata
  var N     : Vacno           var A  : Agency
  vars Id Id' : ObjectId      var C  : Company

-- axioms of apply
crule apply(<Id : Agency |
                  p-list = PS,
                  c-list = CS >,
                  P, S)
 => <Id' : Agency |
          p-list = apply(PS, P, S),
          c-list = CS >
 :if not isPerson(PS, P) and true  .

-- axioms of subscr
crule subscr(<Id : Agency |
                   p-list = PS,
                   c-list = CS >,
                   C, S)
 => <Id' : Agency |
           p-list = PS,
           c-list = subscr(CS, C, S) >
 :if true and not isVacno(newVNo(CS)) .

-- attribute ----------
eq out(apply(A)) = o-empty .
eq out(subscr(< Id : Agency |
                     p-list = PS,
                     c-list = CS >))
```

```
      = newVNo(CS) .

  rule isPerson(apply(PS, P1, S), P) => isPerson(PS, P) .
  eq isPerson(apply(PS, P, S), P) = true .
  eq isPerson(p-empty, P) = false .

  rule getPdata(addPerson(PS, P1, S), P) => getPdata(PS, P) .
  eq getPdata(addPerson(PS, P, S), P) = S .
  eq getPdata(p-empty, P) = noPerson .

  ..................
  }
}
```

この記述スタイルを用いれば状態の要素すべての追加・削除などが簡潔に変更できる。また、状態はクラス (状態属性は hidden sort)、状態遷移は書換え論理の遷移規則で表現してあり、これは大変自然なモデル化であると言える。しかし、現時点では CafeOBJ のクラス宣言は hidden sort を用いて宣言することをサポートしていないので、この記述スタイルはあくまで実験的試みに過ぎない。

しかし、この記述スタイルは明晰性・拡張性に優れており大規模な仕様を記述する際には大変便利な記述スタイルになると期待される。また、クラス宣言に hidden sort を用いる方法は、Hidden Algebra による仕様記述の段階的詳細化 (refinement) の面から見ても大変興味深いものであるし、この手法により定式的に状態遷移問題の記述が可能ならば、階層的な状態遷移問題など現在問題とされている様々なタイプの状態遷移問題の記述を容易に行える可能性を秘めている。

第 9 章

結論

本研究では、様々な仕様言語の記述スタイルについて取り上げ、実際の問題の仕様記述を通して各々の記述スタイルの比較検討を行った。

まず最初に、Z 記法と CafeOBJ による簡単な問題の仕様記述の比較により、異なる数学的基盤を持つ仕様言語同士の記述特性の比較を行った。Z 記法は集合論と一階述語論理のプリミティブを用いてモデル化を行うのに対し、CafeOBJ では直接等式やルール宣言を用いて自由なモデル化手法により仕様を記述する。この記述方法の違いは第３章でも述べたように、時には長所となり時には短所となり仕様記述者に降りかかる。

次に、代数仕様言語では記述が困難であると言われる状態遷移問題に対し、Z 記法と CafeOBJ で可能な４つの記述スタイルで雇用代理店の仕様記述を行い、明晰性、拡張性に関して記述の厳密な比較を行った。これまで代数仕様言語を用いた事前状態と事後状態の差異に基づいた有効な記述例は示されてなかったが、6.3 節で行った書換え規則を用いた手法を用いれば、代数仕様言語で Z 仕様と同等の状態遷移の記述が可能であることを記述スタイルの比較を通して示した。さらに、8.1 節ではクラス宣言と書き換え規則を用いて状態遷移問題の定式化を行った。この方法を用いれば、Z 記法での状態遷移の記述スタイルと同等の記述力を持つ CafeOBJ の記述スタイルを作成できる。

最後に代数仕様の新しい考え方である Hidden Algebra を用いて明晰性・拡張性の高い仕様が作成できるか模索した。Hidden Algebra はオブジェクトのデータ構造に依存しない仕様記述手法であり、これを用いれば状態を始代数に基づく項ではなく、そのデータ構造が隠蔽されたものとして状態を表現できる。8.2.2 節では本稿で提案する記述スタイルの最終形として、クラス宣言と書き換え規則を用いた手法をさらに拡張し、クラス属性を

hidden sort で定義する手法を提案した。しかし、現段階で CafeOBJ のクラス宣言はクラス属性として hidden sort を記述することをサポートしていないので、この手法は実験的試みに過ぎない。しかし、もし、この記述法が可能になれば、状態遷移を含む大規模な仕様を作成する際、大変強力な記述スタイルになると推測される。

本研究の今後の課題としては、次のようなことを考えられる。

- 各々の手法で記述した仕様の詳細な検証手法の比較調査
- クラス宣言を用いて複数の hidden sort を定義する手法の理論的裏付け
- 複数の hidden sort および段階的詳細化による階層的な状態構造を持つ状態遷移問題の記述可能性の調査

第 10 章

謝辞

本研究を進めるにあたり、指導していただいた二木厚吉教授に深く感謝致します。また、有益な助言をしていただいた渡部卓雄助教授、緒方和博先生、Razvan Diaconescu 先生に御礼を申し上げます。最後に、研究に関する議論につきあっていただいた言語設計学講座の諸氏に感謝致します。

参考文献

[1] Maritta Heisel,"Specification of the Unix File System: A Comparative Case Study ",pp475-488,Lecture Notes in Computer Science 936 (July 1995).

[2] Hubert Baumeister,"Relation as abstruct datatypes: An institution to specify relations between algebras"In TAPSOFT'95, volume 915 of LNCS, pages 756-771, Arhus, Denmark,(May 1995)

[3] Hubert Baumeister,"Using Algebraic Specification Language for Model-Oriented Specifications",Technical Report MPI-I-96-2-003, Max-Planck-Institut fuer Informatik, (Feb 1996)

[4] Joseph Goguen and Grant Malcolm, "A hidden agenda",(1996)

[5] Razvan Diaconescu,"Behavioural Rewriting Logic: semantic foundations & proof theory", http://ldl-www.jaist.ac.jp:8080/cafeobj/abstracts/hsrwl.html(1996)

[6] Joseph Goguen and Razvan Diaconescu,"Towards an algebraic semantics for the object paradigm." In Harmut Ehrig and Fernando Orejas, editors, Recent Trends in Data Type Specification, volume 785 of LNCS, pages 1-34,Springer(1994).

[7] Joseph A Gougen, Timothy Winkler, Jose Meseguer, Kokichi Futatsugi and Jean-Pierre Jouannaud,"Introducing OBJ",Technical Report SRI-CSL-92-03,Computer Science Laboratory, SRI International(March 1995)

[8] Joseph A Goguen,"Theorem Proving and Algebra",MIT, http://www-cse.ucsd.edu/users/goguen/pubs/index.html.

[9] Joseph A Goguen, Grant Malcolm,"Algebraic Semantics of Imperative Programs",MIT Press(1996).

[10] Razvan Diaconescu and Kokichi Futatugi,"Logical semantics for cafeobj", Technical Report IS-RR-96-0024S, Japan Advanced Institute for Science and Technology(1996).

[11] J.M.Spivey,"The Z Notation: A Reference Manual 2nd edition.",Prentice Hall, (1992)

[12] Ben Potter,Jane Sinclair,and David Till:"An Intoroduction to Formal Specification and Z",Prentice Hall (1991). (邦訳：田中武二監訳:" ソフトウェア仕様記述の先進技法-Z 言語", トッパン (1993)

[13] Meseguer, J.,"Conditional Rewriting Logic: Deduction, Models and Concurrency ",Proc. 2nd International CTRS Workshop, Lecture Notes in Computer Science 516, pp.64-91 (1991)

[14] J.A.Goguen and R.Burstall. "Institutions: Abstract model theory for specification and programming",Journal of the Association for Computing Machinery,39(1):95-146 (Jan 1992)

[15] 二木厚吉, "代数モデルの基礎", コンピュータソフトウェア Vol.13,No.1, pp.3-22(1996).

[16] "CafeOBJ マニュアル", http://www.ipa.go.jp/STC/CafeP/cafe-jp.html,IPA (1996).

[17] 中川 中, 谷津弘一, 本間毅寛, "CafeOBJ への誘い" ,Technical report,IPA (1996).

[18] 浜口正孝," 形式仕様言語 OBJ によるオブジェクト指向仕様記述", 北陸先端科学技術大学院大学修士論文 (1995).

[19] 飯田周作," 並行オブジェクト指向モデルに基づく並行分散システムの形式仕様作成法", 北陸先端科学技術大学院大学修士論文 (1996).

[20] 海野浩," 形式仕様のライブラリに関する研究", 北陸先端科学技術大学院大学修士論文 (1996).