修 士 論 文

パラメータ化プログラミングにおける
モジュール変換に関する研究

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

浦上貴裕

1997 年 2 月 14 日

要 旨

ソフトウェアの規模や複雑さが増大し、一人の人間が一時に理解し把握できる限界を超えることを指したソフトウェア危機という言葉がある [15]。この危機を脱することを目的として考えられた概念が、プログラムをできるだけ互いに独立な部分に分割するモジュール化である。自然言語で仕様を記述する際のモジュール構造の与え方は、通常章や節に分ける形で実現している。この場合、章や節の間での参照による依存関係が仕様全体の構造を決定する。さらに全体の構造の見通しを良くするため、目次や索引が付けられる。目次や索引を利用することで、概略と詳細の関係での階層構造や、機能的な依存関係の形成、さらには機能の実装依存に代表される仕様のパラメータ化の実現などが可能となっている。

データ型は、データの集合とその上に定義された演算の集合をまとめたものである。このデータ型を抽象化したもの抽象データ型である。抽象データ型を記述単位とする代数仕様ではこれをモジュールとして定義する。本研究ではプログラムにモジュール構造を与える代表的な仕様記述法として代数的仕様記述法を採用した。この仕様記述法において、システムの仕様はモジュール (抽象データ型) の組み合わせとして形成される。モジュールを組み合わせるための機構としては、自然言語と同様、モジュールの参照構文をはじめ、パラメータ付きのモジュールを記述する構文などの実現が望まれる。しかし、パラメータ機構などの整備が進み、複雑にモジュールの参照や詳細化が可能になると、結果としてコード上から全体構造の見通しが付きにくくなる場合がある。

本稿ではこの様な定義の利用から複雑化した仕様の詳細化の支援を目的として、自然言語の仕様における目次や索引と同等の役割を果たすモジュールグラフの定義を行ない、モジュールグラフの利用によるプログラムの可読性の向上について検討を行なった。実際にはプログラムで用いられているデータ構造の挿替えをモジュールグラフ上で実現し、有効性の検討を行なった。

モジュールグラフの定義により、パラメータ付きモジュールや多重のパラメータ化を伴ったモジュールのグラフ化が可能となった。また検証の結果、仕様の可読性を高めることも確認できた。モジュールグラフを用いたデータ構造の挿替えの研究では、プログラムの保守作業と同時にモジュール構造の変更を確認できることが分かった。これにより、モジュールグラフは保守の際の安全性の確保に貢献すると言うことができる。

目 次

第 1 章

はじめに

この章では研究の背景と目的を明確にした後、関連研究の紹介を行ない、本研究が今まで行なわれてきた研究とどの様に差別化されるか説明する。

1.1 背景と目的

構造化プログラミングはダイクストラが提唱したプログラム作成法である。大きく複雑なプログラムは人間の知的管理の限界を超えるため信頼性や保守性が損なわれる。そのため小さく簡単なプログラムを作成し、これらを合成することで先に挙げた危険性を排除する考えから構造化概念が生まれたのである。この概念は 1970 年代以降のデータ抽象や goto なしプログラミングといったソフトウェア設計に関するの基本思想となっている [20]。

自然言語で記述された仕様は章や節へ細分化される。これは仕様にモジュール構造を与える方法であると考えることができる。また、仕様の全体構造を見通すために依存関係や詳細化の関係を示す目次や索引の添付を行なう。ここで章や節として記述されたモジュール構造をシステム全体の仕様に統合する機構としては、概略の記述中で詳細記述部分への参照付けが行なわれる詳細化の線での階層構造の定義や、概念上の依存関係の記述、さらには機能を実装に依存する旨を記述することにより得られるパラメータ化機構をあげることができる。

抽象データ型の仕様記述法はプログラムをモジュール化する概念であるということができるため、本研究には抽象データ型の仕様記述法を起源とする代数仕様言語を用いる。抽象データ型は、データの集合とその上に定義された演算の集合を形成するデータ型の抽象

である。抽象とは操作がどの様に定義されているかといった詳細の部分をブラックボックスとし、操作方法のみを明かす概念である。例えば、整数の抽象データ型では、1,2,3,... といった整数集合の要素と、整数を扱う四則演算の定義を行なう。利用者には数と演算の利用法が知らされ、演算の定義については伝えないのである。代数仕様 [14]~[17] では抽象データ型を記述単位として仕様の記述を行なう。代数仕様におけるシステムの仕様は、モジュール (抽象データ型) の組み合わせで形成される。モジュールを組み合わせるための機構としては、自然言語で記述された仕様の場合と同様である。

- ある抽象データ型が別の抽象データ型を参照する構文
- 抽象データ型の間の詳細化の関係を記述する構文
- 幾つかの抽象データ型を合成する構文
- パラメータ付きの抽象データ型を記述する構文
- 既存の抽象データ型を変形する構文
- 頻繁に使う抽象データ型をライブラリに取り込み、それを参照する構文

この様な形で現れている。そして、実際に言語の一部としてこれらの構文が組み込まれているのである。

本研究で用いる代数仕様言語 CafeOBJ [12][19] [24] では先にあげた様々な機構が言語の一部として組み込まれている。参照や詳細化の構文は輸入で実現しているし、パラメータ付きの抽象データ型はパラメータ付きモジュールの定義を可能とすることで対応している。抽象データ型の合成はこのパラメータ付きモジュールに対する特定化作業に見ることが可能である。代数仕様におけるパラメータ化プログラミングの研究は仕様の再利用性に注目が集まった 70 年代後半から高まる [5]。Clear で実現されたこの機構は仕様の再利用性を高めるのみでなく、仕様記述に要する期間に比較して仕様理解の時間が短縮されることからプログラミングや保守の時間短縮につながる概念である [7]。パラメータ化プログラミングの機構は CafeOBJ の前身である OBJ2 および OBJ3 で本格的に導入された。既存の抽象データ型を変形する構文として CafeOBJ にはモジュール和や名前替えの機構が備えられている。この機構により、一つのモジュールの多様な文脈での使用が可能となる。名前替えの適用の結果はモジュール内で定義されるソートや演算名の置換操作である。

自然言語で記述した仕様の記述単位は章や節というモジュールであり、代数仕様記述での記述単位は抽象データ型のモジュールであった。そして、自然言語上での参照関係を明確にする際には目次や索引を用意すれば良いことは先に触れた。代数仕様について考えた時にもモジュールの参照関係を明示する際に必要な構文は用意されているため、これらの構文が目次や索引の代わりになると言うことが可能である。ところが代数仕様に用意されたパラメータ付きの抽象データ型を記述する構文や既存の抽象データ型を変形する構文に着目すると、これらがモジュールの表現力を高める有効な機構である反面、複雑なモジュール構造を容易に記述することが可能となることがモジュール構造の可読性を損なわせる危険性を含んでいることが分かる。

本研究ではパラメータ化プログラミングや名前替えで失ったモジュール構造の可読性を再び取り戻し、使用の詳細化を支援することを目的としている。

1.2 図示による可視化の工夫

モジュールを図示することにより得られる可読性の向上について、過去には ADJ ダイアグラムによりモジュールの演算を表現する方法、及びプロトタイピング法におけるモジュールの構築過程を図示する研究などが行なわれている。ここで両者の研究について触れる。

1.2.1 ADJ ダイアグラム

ADJ ダイアグラムが導入されたのは 1976 年頃である。Goguen, Thatcher らにより抽象データ型の理解を容易にするための表現方法として開発された。ソートを示す円と、黒丸で示された演算とによりモジュール内で定義された演算の動作を示すことが可能となっている。正確に表現すると、演算はモジュール内でアリティ(定義域) からコアリティ(値域) への射として定義が行なわれるが、この射の関係を ADJ ダイアグラムは表現するのである。

例えば以下に示すモジュールのダイアグラムを用いた表現は図 1.1 のようになる。

```
module AUTOM {
  [ Input State Output ]
  op i : -> State
  op f : Input State -> State
  op g : State -> Output
}
```

モジュール AUTOM ではオートマトンの仕様の定義を行なっている。演算 i は、ソート State

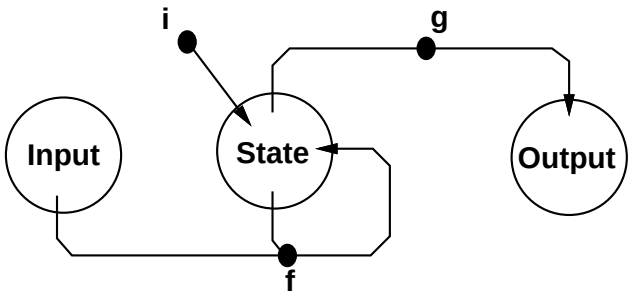


図 1.1: ADJ ダイアグラムによる表現

の基底項であり、f はアリティが Input , State でコアリティが Output となる演算。g はアリティが State で、コアリティOutput の演算である。実際にはこの演算に対する等式定義で、f は入力値と現在の状態の二つの引数により決定する値を出力することになるであろうし、g は現在の状態だけで決定する出力値を返すという具合に意味付けさられていくのである。

1.2.2 プロトタイピング法

岡田、二木らにより研究されたのが、空な記述から問題を完全に記述したプログラムに至るまでの形式的手法を与えるプロトタイピング法 [3][6][18] である。ここで用いるプロトタイプの語意は、建造物や船の模型のように最終性生物の属性のうちのいくつかを忠実に保持する一種のモデル、すなわち抽象化物として定義されている。

プロトタイピング法では、問題をプログラム化する過程において、この問題の細分化したものを忠実に保持したモデルとなる抽象化物の仕様化を行なう。この仕様化作業を繰り返し行なうことで問題の完全なプログラム化に至る方法を提案している。細分化した問題の仕様化を行なう際に決定オブジェクト、仮オブジェクト、共通下位オブジェクトをグラフ上で 3 種類に分けて表現し、プログラムの構築進度を表現している点も斬新であった。完全に問題を記述したプログラムでは、そこに現れる全オブジェクトが決定オブジェクトとして表示されることから、グラフによる進度の可視化が実現したのである。

1.3　本稿の構成

現在まで行なわれてきた図示による可視化の研究は ADJ ダイアグラムはモジュール内で定義された演算とソートの関係を示すものであり、プロトタイピング法におけるモジュールグラフは輸入によるモジュールの継承関係を示すものであった。本研究で定義を行なうモジュールグラフは輸入によるモジュールの継承関係に関する定義も行なうが注目しているのはパラメータ付きモジュールのグラフ化である。この点で今まで行なわれた研究と差別化を行なうことができると考える。

ここで、本稿の構成について説明する。

第 2 章では、本研究で利用する仕様記述言語の歴史的背景と言語基盤について説明を行なう。

第 3 章では、第 2 章で紹介した仕様記述言語におけるパラメータ化プログラミングの実際を紹介する。この章において、パラメータ化プログラミングにより得られる仕様の表現力の向上を理解することになる。表現力が高まるにつれ、コードの可読性が失われていく様子も示す。

第 4 章では、モジュールグラフの定義を行なう。本研究で利用する言語の様々なモジュール表現の説明と共に、対応するモジュールグラフを示す。

第 5 章では、第 4 章で定義したモジュールグラフを用いてデータ構造の挿替えを実践する。安直に記述した仕様を用意し、モジュールグラフでこの仕様を示す。グラフ上でデータ構造を挿替えてから、変更後のグラフを元にモジュールの再構築を行なう。

第 2 章

代数仕様と CafeOBJ

CafeOBJ は実行可能なサブセットを持つ仕様記述言語であり、意味論上の基盤として順序ソート等号論理の拡張である順序ソート書き換え論理を持つ。この章では CafeOBJ の言語基盤の説明と、1976 年に考案された OBJ から CafeOBJ へ行きつくまでの歴史的経緯を紹介する。

2.1 データ抽象と代数仕様

仕様書はプログラムの実装を行なうための設計書の意味を持つため、曖昧さが排除され明確かつ簡潔であることが要求される。ソフトウェアの要求仕様をまとめて文書化する手立てとして形式手法をあげることができる。この手法では形式言語により要求仕様の記述を行なう形式仕様記述を最低条件としている。形式言語を用いた結果、自然語や非形式的記述法による仕様書に比べ明確で曖昧さが排除されたものとなり、ソフトウェアの品質 (信頼性) 向上につながる。また、形式的な記述法を行なった結果の仕様全体をデータとして扱うことが可能であるため、後に文法的な誤りなどの形式的な検証や形式的手法を用いたコーディングが可能となるなどの利点を持つ。

形式仕様記述の手法には Z,VDM などの言語に代表される集合仕様、CCS,CSP などに代表されるプロセス代数などをあげることができる。OBJ,ACT や今回用いる CafeOBJ に採用されている手法は代数仕様 [14]~[17] [20][21] である。代数仕様は抽象代数にモデルを求める形式仕様記述のひとつの流派として次のような特徴を持つ。

- 抽象データ型の概念に厳密なモデルを提供する。

- 等号論理による簡潔な意味論を持つ。
- 項書換えシステムによる操作的な意味を与えることができる。

抽象代数は台集合とその上の演算とから構成され、演算の性質·振舞いによって規定される構造を持つ。従来の抽象代数は群のように一つの台集合上の演算に着目するような代数であったため、単一ソート代数と呼ばれる。しかし、抽象データ型では複数のデータ型の存在を自明とみるため、多ソート代数と呼ばれる。

ここで代数仕様と抽象データ型との関係について説明を加える。データ型を構成する集合の表現形態や演算の表現方法が抽象化され、集合や演算に要求される性質が表現と独立して記述されている様なデータ型を抽象データ型と呼ぶ。要するに抽象データ型とは、集合とその集合の上の演算とをひとまとめにしたものである。抽象データ型の特徴を次に列挙する。

- 高度なモジュール構造の実現が可能である
- 抽象度が高い
- 情報隠蔽の効果がある
- 問題の局所化がなされる

などである。これらの特徴をプログラミングの概念に生かすことで、メインテナンスや構造化などの手間を大幅に改善することが可能となるのである [14]。

抽象データ型の概念を形式的な記述法によりプログラミングに採用し、さらに抽象代数に基づく厳密なモデルとこのモデルに関する様々な解析手段を提供するのが代数仕様である。

2.2 代数仕様言語 CafeOBJ

本稿で仕様記述に用いる言語 CafeOBJ は、代数仕様言語と呼ばれる代数仕様に基づいた仕様記述言語である。OBJ は Goguen により 1976 年に考案され、77~79 年には単一ソート等号論理を基盤とした OBJ0 が UCLA にて Goguen と Tard らにより開発された。同時期に開発が行なわれた OBJT はエラー代数と呼ばれる理論を基盤にし、パラメータ機構の概念を一部採り入れた構成となっている。エラー代数は抽象代数に基づいた抽象データ型の理論を拡張し、エラーや部分関数を簡単かつ統一的に取り扱うために試験的に考案さ

れた理論である。OBJT におけるエラー代数の採用はその後 81~82 年に開発された OBJ1 でも基本的に踏襲された。このとき結合則及び交換則の下での書き換えや、緩やかな意味を持つセオリの導入などの拡張が行なわれた。

今回用いる代数仕様言語 CafeOBJ の言語基盤となっているのは OBJ1 の後の 84~85 年に SRI において、二木、Jouannaud らの手で開発された OBJ2 及び 87~90 年 Winkler, Meseguer, Goguen らによる開発の OBJ3 である。OBJ2 は OBJ1 のコードを基にした実装となっているが、この実装においてセオリ、ビューが実現するなど Clear で用いられていたパラメータ化プログラミングの概念の OBJ への本格的な採用が行なわれた。OBJ3 では OBJ2 で整備されたパラメータ付きモジュールの概念を更に洗練した実装が行なわれた。OBJ2,OBJ3 の論理基盤はどちらもソート間に包含関係を認める順序ソート代数である。順序ソート代数の書き換え規則に従い実行を行なう。

順序ソート代数は多ソート代数の拡張 (一般化) とみることができるため、多ソート代数上で議論されてきた定理や定義の多くは若干の変更により順序ソート代数上で検証することが可能となる。

代数仕様言語による形式的記述は抽象データ型の仕様記述法を起源としている。すなわち、記述しようとしている対象問題を多ソート代数とみなし、型分けされたいくつかの集合とそれ上に定義されたいくつかの演算を用いて項の書換えを行なうのである。

OBJ の最も大きな記述単位はモジュールとビューである。モジュールとはすなわちデータ抽象である。モジュールはさらにオブジェクトとセオリに大別される。これら 3 つの記述単位は以下の性質をもつ。

- オブジェクトは実行可能なコードを持ち、明確に規定された概念の表現を行なう。
- セオリは他のモジュールが満たすべき性質を表現し、仮パラメータとしてモジュールのインターフェイスを規定する。
- ビューは仮パラメータであるセオリと実パラメータとして作用する他のモジュールとの文法上、意味上の対応関係を与える。

モジュールは下位のモジュールを参照してその上に演算を定義することができる。この様に記述していくことで、参照関係による階層構造を形成することになる。モジュールを節点、参照関係を辺とする有向・非循環グラフを形成するのである。モジュール定義は次の 5 つの要素からなる。すなわち、1) モジュール名の定義、2) 参照モジュールの宣言、3) この

モジュールで用いるソート宣言 (ソート名、及び順序関係)、4) モジュールに属する演算子名とその型の宣言、5) 等式の詳細を記述。演算子の宣言を行なう際にはアリティとコアリティの関係を用いる。すなわち、新しい演算を定義する際には引数のソート列 (アリティ) とその演算による結果のソート (コアリティ) を明記するのである。

CafeOBJ 上でのモジュール定義の記法は上記 1)∼5) を用いて次のように記述する。

```
module MODULE {
  importing( ... )
  signature{ ... }
  axioms{ ... }
}
```

MODULE がモジュール名である。importing はモジュールの参照関係を記述する部分で、本来は参照方法に対する制約を持たせた protecting, extending,using の 3 種類の参照方法を記述する。

signature{...} の内部ではソートの定義と各ソートの包含関係、及びそのソートに対する演算を定義する。axioms{...} の内部が等式の詳細を記述する部分である。

OBJ の意味論的裏付けは、操作的には項書換えシステムによって、宣言的には順序ソート代数によって成されている。すなわちここで定義を行なった等式について、宣言的には等式そのものとして、操作的には左から右へという方向を持つ書き換え規則としてそれぞれ解釈するのである。操作的意味論に基づいた実行·検証は書き換え規則に従う。そのため、ここでいう検証とは順序ソート代数のモデルが存在するための条件を満たしていることを確認し、その上で項書き換えシステムとして停止性·合流性を確認することにほかならない。これ以外に検証によって示す必要があるのはパラメータ制約の充足と、参照に対する制約が守られていることである [21]。

宣言的意味論は記述した仕様に対する数学上のモデルを規定する。CafeOBJ ではこの数学的モデルとして抽象データ型による始代数モデルの導出を目的としている。そのため、ここで採用している宣言的意味論は始代数意味論である。始代数モデルは同型で閉じるという意味で一意に定まり、与えられたシグニチャと等式集合から得られる標準的モデルとなる。また始代数には以下のような性質がある。

- 非冗長性 : 台集合の全ての要素が、与えられた定数·関数記号で表現できる
- 非混同性 : その代数で成立する基底等式はすべて与えられた等式から証明できる

OBJ での参照関係に対する制約は、他のモジュールの参照時のこれらの性質の守られ方によっている。すなわち、protecting では非冗長かつ非混同が守られた参照形態をとり、extending では非混同のみが守られるように参照され、using では制約の無いものとして定義されている。

OBJ3 から CafeOBJ への改良点としてはオブジェクト指向プログラミングの概念の導入があげられる。オブジェクト指向プログラミング実現のために構造を持ったデータ型を定義するレコード宣言が可能となっている。レコードは通常のソート同様半順序関係を与えることができ、また多重継承も可能となっている。更にレコード型を拡張したクラス定義も実現している。クラスは自律的で内部状態を持つような実体を定義する。クラス定義と、クラスへの多重継承や半順序関係の定義を実現したため、CafeOBJ を用いてオブジェクト指向プログラミングの方法論を用いた仕様記述が可能となっている。

第 3 章

パラメータ化プログラミング

パラメータ化プログラミング [7][8] はモジュールの汎用性や再利用性を高める非常に有効なプログラミングパラダイムである。この考えは Clear で実現し、OBJ に導入されたが、二木らにより実装された OBJ2 でセオリ、ビューが実現したことによりさらに強力なものとなった。OBJ3 ではこの仕組みが洗練された。本稿で用いる CafeOBJ にも OBJ3 のパラメータ化プログラミング機構が継承されている。

3.1 基本概念

パラメータ化プログラミングの概念から説明する。スタックやリストといったデータ構造を定義する場合、このデータ構造に格納される要素のソートを限定してモジュール設計を行なうことが利用形態によっては便利な場合がある。しかし、データ構造内で定義される一群の操作は格納されるソートに依存しないため、ソートを変更する度に同じデータ構造を定義し直すことは無意味である。そこでパラメータ化プログラミングでは、データ構造内で用いるソートとそのソートに関する演算など必要と思われる性質の代表例としてパラメータを用意する。データ構造の定義時にはこの代表ソートに対する操作の定義を行なう。後にこのデータ構造を利用する際には、ここで用意した仮パラメータに実パラメータ (例えば整数モジュール) を特定化する必要があるが、この様なモジュール定義により汎用的なデータ構造のモジュール定義が可能となる。モジュールの修正を行なう場合にも一つのモジュールを修正することで解決するため局所化が行なわれ、保守作業の効率化がなされるのである。

3.2 CafeOBJ のパラメータ化プログラミング

CafeOBJ の記述単位はモジュールとビュー宣言である。さらにモジュールはオブジェクトとセオリに分けることができる。オブジェクトが具体的なデータ型の代表である抽象データ型を意味するのに対し、セオリはオブジェクトの抽象であると考えることができる。CafeOBJ においてこれらを明記すべき時にはモジュール定義の際に `object{...},theory{...}` といった記述方法をとるが、普段は `module{...}` という宣言を用いた定義を行なう。要するに、定義方法として考えるとオブジェクトとセオリは区別されないのである。このため `module{...}` による定義を行なったモジュールはオブジェクトとセオリの双方の意味で使用することが可能となるのである。先に述べたようにオブジェクトの抽象がセオリでありこれらは共にモジュールであることから、セオリを更に抽象したセオリの定義も可能である。この様にモジュールの抽象化の階層構造を作り出すことが可能なのである。

ビューとはセオリモジュールからあるモジュールへのマップである。セオリを他のモジュールが満足するとはセオリ内で定義されたソートや演算に対応する定義がそのモジュール内で行なわれていて、これらを全て一対のものとして宣言できることをいう。満足の宣言を行なうためにビュー内では全てのマップを明記する。セオリとビューの利用がモジュールの表現能力を高めることからパラメータ化プログラミングの有効性が生まれるのである。パラメータ化プログラミングの説明に先立ち、整数のスタックの例をあげておく (図 3.1)。

この例題では整数の意味を記述したモジュール `INT` をモジュール `STACK-INT` 内で輸入することで参照可能にしている。スタックできる要素はそれゆえ整数に限定されている。同様に文字列や自然数を要素に利用する場合には各々のスタックを準備する必要がある。しかしそこで定義されるスタックの演算 `pop`,`push` などは対象ソートの字面は異なるが本質的には等しい記述となる。よって各ソートに対するスタックの定義は非常に冗長性の高い作業であるとみなせる。そこで、代表ソート `Elt` 対する演算定義を行ない、実行時には整数や文字列など適当なソートに特定化を行なう。

3.3 パラメータ付きモジュール

パラメータ付きモジュールの定義は図 3.2の形式で行なわれる。先に定義したパラメータ無しのスタックモジュール (図 3.1) と比較して明らかに異なっているのがモジュール `STACK` のヘッダに記述された `[X::TRIV]` なる部分である。これがパラメータ宣言である。`X` が仮

```
module STACK-INT {
  protecting(INT)  -- ←整数モジュールの (限定) 参照
  signature {
    [ NeStack < Stack ]
    op empty : -> Stack
    op push : Int Stack -> NeStack
    op pop : NeStack -> Stack
    op top : NeStack -> Int
  }
  axioms {
    var I : Int
    var S : Stack
    eq pop(push(I, S)) = S .
    eq top(push(I, S)) = I .
  }
}
```

図 3.1: 整数に限定したスタックの例

パラメータであり TRIV は仮パラメータに束縛する実パラメータへの制約を定めたモジュールの名前である。このモジュールをセオリ·モジュールと呼ぶ。制約条件を満たせば、実パラメータとしてパラメータ付きモジュールの束縛も可能である。

今回取り上げたセオリモジュール TRIV は、内部にただ一つのソートが宣言されているだけである非常に簡単なモジュールであるが、TRIV により実パラメータが制約をうけているために、仮パラメータ X に束縛され得るモジュールの内部では一つ以上のソート宣言が行なわれている条件が付けられる。逆にいえば、一つ以上のソート宣言が行なわれているモジュールであれば実パラメータに採用することができるのである。スタックで利用される要素についての要求は上記のような緩い制約によって実現されるが、二分木に採用する要素などでは大小関係が明確に規定された集合である必要があり、このためにきつい制約の記述されたセオリを用いることとなる。

先の図 3.1で定義したモジュールで整数の輸入を行なっていた部分をパラメータの輸入に代えた結果がパラメータ付きのモジュール STACK (図 3.2) である。

```
module TRIV {
  [ Elt ]
}

module STACK [ X :: TRIV ] {
  signature {
    [ NeStack < Stack ]
    op empty : -> Stack
    op push : Elt Stack -> NeStack
    op pop : NeStack -> Stack
    op top : NeStack -> Elt
  }
  axioms {
    var E : Elt
    var S : Stack
    eq pop(push(E, S)) = S .
    eq top(push(E, S)) = E .
  }
}
```

図 3.2: 汎用的に定義したスタック

3.4 ビュー

ビュー宣言は、モジュールがセオリを満たすことの宣言である。モジュール M とセオリ T とが以下の条件を満たす関係であるとき、モジュールがセオリを満たすという。

- セオリ T 内で定義されている各ソートに対応するソートがモジュール M 内でも定義されていること。これは $\phi_{sort} : S_T \to S_M$ なるマップの関係であると考えられる。
- セオリ T 内のサブソート関係が $s \leq s'$ であるときには先ほどのマップを用いて $\phi_{sort}(s) \leq \phi_{sort}(s')$ と記述できるようなサブソート関係がモジュール M 内でも定義されていること。
- セオリ T 内で演算が定義されている時には対応した演算がモジュール M 内でも定義されていて、T でのアリティ・コアリティの関係 $(s_1 \ldots s_n \to s)$ が M では、$\phi_{sort}(s_1) \ldots \phi_{sort}(s_n) \to \phi_{sort}(s)$ となる。

- セオリ T 内の等式がモジュール M 内で定義されている等式集合によって満たされていること。

ビューはパラメータのシグニチャからモジュールのシグニチャへのマッピングである。ビュー宣言内では、セオリのソートから実パラメータのソートへ、演算から演算への対応付けが行なわれているのである。例えば、本章で例題に用いている STACK の例では、セオリ·モジュール TRIV 内で定義されているソート Elt に何らかのソートを対応させることにより、「セオリ TRIV を満たしている」モジュールの宣言を行なうことになるのである。

TRIV を満たすモジュール INT へのビュー宣言は図 3.3のようにして行なわれる。TIVIEW はビュー名である。ここではセオリ TRIV からモジュール INT への対応関係を宣言しているため、頭文字を用いて命名している。本来モジュールがセオリを満足していることを示すには先に述べた条件を満たしていることの証明が必要なのであるが、この様にソート、演算、等式の各対応関係を明記するという記述方式をとるビュー宣言により、この証明を省くことが可能となっている。

```
view TIVIEW from TRIV to INT {
  sort Elt -> Int
}
```

図 3.3: ビュー宣言 (TRIV から INT へのマップ)

3.5 特定化

特定化とは、仮パラメータに実パラメータを束縛した特定例を得るための操作である。特定化を行なうことにより、パラメータ付きモジュールを実行可能なモジュールとしてプログラムに導入することが可能となる。

OBJ におけるパラメータの束縛はパラメータ付きのモジュールに実パラメータであるモジュールを組み込むことから、モジュール合成と考えることができる。この合成は一度の束縛で多くの関数を複雑に合成することが可能であるため、従来の関数型プログラミングにおける関数合成よりも強力であるといえる [9]。

パラメータ付きモジュール内のソートや演算はセオリモジュール内で宣言された仮の名前が付けられているわけであるが、束縛を行なう際には、それらをビューに従って実パラメータとなるモジュール内で宣言された名前に書き換え、また、共有する下位モジュールは一部だけコピーを行なう。この作業により、仮パラメータへの実モジュールの束縛が実現され、パラメータ付きモジュールの特定例を作り出すことができるのである。例えば、図3.2に示したパラメータ付きスタックモジュールの仮パラメータに対して、前節のビュー宣言を用いることで特定化を行なうことができ、この結果として、図3.1と同じ意味を持ったモジュールを特定例として得るのである。

パラメータ化プログラミングの有用性は、たとえば今回例題として用いている STACK で整数や文字列その他様々な要素を用いた特定例を作り出すことが可能となることから、汎用性や再利用性があると言われているが、それはこの特定化の機構が備わってこそ言えることなのである。汎用性や再利用性はパラメータ付きモジュールについてのみ言えることではない。ビュー宣言も予め作り貯めておくことで再利用性が生まれるのである。例えば、STACK 同様のセオリ TRIV を制約として用いて定義された他のモジュールを想定した時にも TIVIEW を用いた特定化を行なうことが可能なのである。モジュール STACK の特定化の例を図3.4に示す。

```
make STACKusingINT is STACK [ X <= TIVIEW ] endm
```

図 3.4: 整数へのスタック特定化

図3.4では、パラメータ付きモジュール、セオリ、ビューが予め宣言されたときに CafeOBJ の make コマンドを用いて行なわれる特定化の例を示したが、make コマンドの中でビュー宣言を行なう方法 (図3.5上) や、輸入しながら特定化する方法 (図3.5下) など様々な構文を用いても結果としては同じ意味を持ったモジュールを導出することができる。

パラメータ付きモジュール M を特定化せずに輸入する時、輸入するモジュール M' 自体がパラメータ化される。これを多重のパラメータ化と呼ぶ。M' に付加されるパラメータは M の仮パラメータであり、M' を特定化すると同時に M に実引数が渡されるのである。このとき特定化は多重に行なわれているため、この特定化を多重の特定化と呼ぶ。多重の特定化を行なうことにより、複雑な意味を持たせたモジュールの定義が可能になる。

```
make STACKusingINT2 is STACK
  [ X <= view from TRIV to INT { sort Elt -> Int }] endm

module STACKusingINT3 {
  protecting(STACK[ X <= view from TRIV to INT
                         { sort Elt -> Int }])
}
```

図 3.5: 様々な特定法

例えば、ある要素の列を降順に並べ替えるモジュールの定義を行なうには、列を定義したパラメータ付きモジュール SEQ を SEQ [X::TRIV] のようにパラメータ化したモジュールとして定義しておき、降順の並べ替えを定義するモジュール SORTING に特定化せずに輸入する。輸入されるモジュール SEQ の仮パラメータを、輸入するモジュール SORTING の仮パラメータとしておくことで、並べ替えを行なうモジュールの特定化により列のモジュールの仮パラメータにも等しい実パラメータが渡される。図 3.6に定義の概要を示す。

```
module TOSET {
  ... 全順序関係の定義
}
module SEQ [ X :: TRIV ]{
  ... 列の定義
}
module SORTING [ ELT :: TOSET ]{
  protecting(SEQ [ X <= view to ELT { sort Elt -> Elt }])
  ... 並べ替えの定義
}
```

図 3.6: 多重のパラメータ化

モジュール SEQ では要素の列を定義するために、ソートが少なくとも一つ定められたモジュールを実パラメータとして輸入したいという要求が書かれるが、SORTING で同じ要素を利用する際にはソートが定められていることに加えて、そのソートの要素を並べ替える

際に要素の大小関係が定義されている必要があるため、TOSET で定義された全順序関係を条件として要求している。SEQ の輸入構文内で定義されたビューはこの部分でのみ利用されるため「短命のビュー」またはビュー名の宣言が省略されることから「名無しのビュー」と呼ばれる。先に述べたように SEQ のパラメータへの制約は SORTING のそれと比べ緩いものである。内部構造として利用するパラメータ付きモジュールの制約が外部モジュールの制約と等しいか緩いことが多重のパラメータ化の制約についての必要条件となっている。これはビューの定義に照らして考えれば自明である。

全順序関係は図 3.7に示す定義で実現される [2]。

```
module TOSET {
  signature {
    [ Elt ]
    op _<_ : Elt Elt -> Bool
  }
  axioms {
    vars E1 E2 E3 : Elt
    eq (E1 < E1) = false .
    eq (E1 < E2) or (E2 < E1) or (E1 == E2) = true .
    eq (E1 < E2 and E2 < E3) implies (E1 < E3) = true .
  }
}
```

図 3.7: 全順序関係の定義

セオリ TOSET 内ではソート宣言のほかに演算定義がされている。そのため、ビューはソートの対応関係のほかに演算の対応関係が宣言される必要がある。CafeOBJ の組み込みモジュールでは、全順序関係が定義されている整数や文字列といったモジュールをもって特定例を作り出すことが可能である。

整数の場合、

```
view TOINT from TOSET to INT {
  sort Elt -> Int,
  op   _<_ -> _<_
}
```

であり、文字列の場合、

```
view TOSTR from TOSET to STRING {
  sort Elt -> String,
  op   _<_ -> string<
}
```

というビュー宣言を行なう。この様に文字列の全順序関係が string<で宣言されているなどビュー宣言に先駆けてセオリに対応するソートや演算のモジュール内での宣言を調べる必要がある。

言語要素の整備により多重のパラメータ化の記述が可能となりモジュールによる表現力が豊かになった。しかし、次に示すモジュール定義のヘッダ部分が示すように、複雑なモジュール宣言が可能となったことからモジュール自体の可読性の損失につながることがわかった。

```
module MAKE-PARENTS-SET [ FATHER MOTHER PARENTS :: TRIV ]{
  protecting (SET [ X <= view to NAME
    [ X <= view to FATHER { sort Elt -> Elt }]
    * { sort Name -> FatherName, op name -> Fname }
       { sort Elt -> FatherName }]
       * { sort Set -> FatherSet , sort SetSeq -> FatherSetSeq ,
            op nil -> nilFather })
  protecting (SET [ X <= view to NAME
    [ X <= view to MOTHER { sort Elt -> Elt }]
    * { sort Name -> MotherName, op name -> Mname }
       { sort Elt -> MotherName }]
       * { sort Set -> MotherSet , sort SetSeq -> MotherSetSeq ,
            op nil -> nilMother , op _-_ -> _del_ })
  protecting (SET [ X <= view to PARENTS { sort Elt -> Elt }]
    * { op _U_ -> _union_ })
  [ Elt.PARENTS Parents < SetSeq ]
  op nilName : -> Elt.PARENTS
  op parents : FatherName FatherName MotherName -> Parents
  op parents : MotherName FatherName MotherName -> Parents
  op make-parents : FatherSet MotherSet -> Set
  op getMname : FatherName MotherSet -> Elt.PARENTS
  vars -- ...  variables
  eq   -- ...  eqations
}
```

そこで、複雑な定義が行なわれたモジュールの可読性を高めるため、次の章ではモジュールグラフの定義を行なう

第 4 章

モジュールグラフ

この章ではモジュールグラフの定義を行なう。モジュールグラフはパラメータ付きプログラミングなどを伴って複雑化したモジュール構造の理解を容易にすること目的として提案するものである。

4.1 モジュールグラフの情報源

CafeOBJ での記述単位であるモジュールは次の 3 つの部分から構成される。

- ヘッダ : オブジェクト名、パラメータと制約モジュール名、参照するモジュール名と輸入の仕方、および名前替えを記述する部分。
- シグニチャ : 新しく定義されたソート、サブソートの関係、オペレーション定義の記述を行なう部分。CafeOBJ では `signature{...}` の括弧内に記述する。
- ボディー : 演算を記述する部分。CafeOBJ では `axioms{...}` の括弧内に記述する。

モジュールグラフはヘッダ部分を情報源として利用する。シグニチャ部分で宣言される演算の詳細については ADJ ダイアグラム [10][14] により図示することが可能であるし、本研究の目的を達成するために定義するモジュールグラフでは、モジュール間の輸入による参照関係と、パラメータ化されているモジュールがプログラムに含まれているような場合はパラメータの状態が一目で見てとれることが最重要であって、それ以外の情報は排除するべきであるという見地に立つからである。

モジュールのヘッダで宣言されるのはモジュール名、輸入するモジュールの名前と輸入の仕方、パラメータ化が施されている場合には仮パラメータ、及び実パラメータへの制約が記述されているモジュール (セオリ) 名と名前替えであった。これらヘッダ情報のグラフでの表現方法を以降の節で簡単な例題を用いて説明する。

4.2 モジュール

モジュールは基本的にモジュール名を実線の四角形の内部に記載することで示す (図 4.1)。

例外の発生は枝が交差する場合である。コード上で同じモジュールの定義を二度は行なわないということをグラフ上でも受け継ぐ。よって一枚のモジュールグラフ中で、実線で囲まれたモジュールは一箇所のみに出現することに決める。このため、既に定義されグラフとして利用されているモジュール (MODULE) と他のモジュール (MODULE') とを枝で結ぶ際に他の枝や節点との交差が生じる場合には、モジュール (MODULE) の他に四角形を破線で記述したグラフを用意し、結合に適当と思われる場所に配置する (図 4.2)。この約束はグラフの曖昧性の排除にもつながる。

パラメータ付きモジュールとして定義されている場合、特定例は確定モジュールとして扱うことができるが、特定化前のモジュール自体は不確定モジュールである。不確定モジュールと確定モジュールのグラフ上の違いについてはパラメータ付きモジュールのグラフ表現の節で説明する。

```
module MODULE {
  signature{ ... }
  axioms{ ... }
}
```

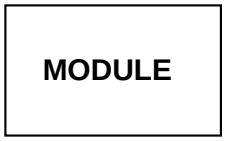


図 4.1: モジュール

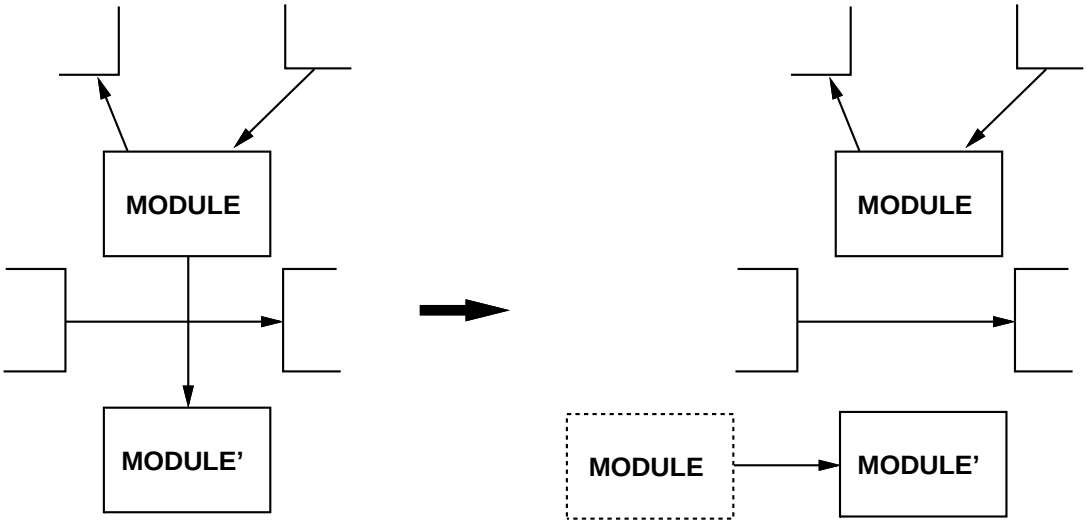


図 4.2: 枝の交差を避ける記法

4.3 輸入関係

モジュール M の内部において既存のモジュール M' 内で定義されているソート、演算などを参照するにはモジュール M' の輸入を行なう。この輸入により M' のコードが M の諸定義で使えるよう解放されたこととなるため、M' に予め輸入されていた全てのモジュールが M にも輸入されたことになるのである。このため、輸入によるモジュールの階層構造を作り出すことが可能となる [1]。

輸入を行なうにはその利用方法に対する制限に応じて protecting,extending,using の 3 種類があり、制限は各々次のようになっている。

- protecting : 輸入されるモジュールで宣言されたソート上に新たな要素を付け加えたり (非冗長)、ソート上の既存の要素を同定してはいけない (非混同)。
- extending : 輸入されるモジュールで宣言されたソート上の既存の要素を同定してはいけない (非混同)。
- using : このモードでの輸入には何ら制限が無い。using による輸入は、輸入されるコードのコピーがモジュール内に挿入されると考える。

`protecting` モードでは M' の宣言的意味論が M 内でそのまま保存される。このため、M'

で証明された意味論上の性質を用いる際に M 上で再度証明し直す必要がないのである。また、新たな規則の追加や E 戦略の再計算も不要であるため実装上では単純なコードの共用と考えることができる。

extending モードでは要素の追加が行なわれる可能性があるため、一部証明のやり直しが要求される。さらに、E 戦略の再計算が必要となる場合も生じる。

using モードは事実上、輸入されるモジュールコードのコピーを、輸入するモジュール内に記述するモードであると考えることができる。

この様に CafeOBJ では輸入モードについて、制約の持たせ方により細かな定義が行なわれている。グラフ上では、輸入されるモジュール M' から、輸入するモジュール M への実線の矢印を引くことで輸入関係を示すことに決める。さらに、輸入モードを明確に記述するため、この実線の矢印にモードの頭文字である p,e,u を添えることにする。

```
module MODULE {
  signature{ ... }
  axioms{ ... }
}

module MODULE1 {
  protecting(MODULE)
  signature{ ... }
  axioms{ ... }
}

module MODULE2 {
  extending(MODULE)
  signature{ ... }
  axioms{ ... }
}

module MODULE3 {
  using(MODULE)
  signature{ ... }
  axioms{ ... }
}
```

4.4 パラメータ付きモジュール

パラメータ付きモジュールのグラフ表現は、先にモジュールのグラフ表現のところで説明した通り、特定化前の不確定モジュールを描く場合と特定化後の確定モジュールを描く場合とで形状の異なるものとする。この記法を用いることにより、パラメータ付きモジュールをグラフ上で部品として用意することが可能となるため、グラフからモジュールの再利用を援助することにもつながる。特定例のみをグラフ化する場合では、用いられたパラメータ付きモジュールをコード上で見つけ出すのに手間がかかるからである。

ここでは不確定モジュールのグラフ表現を示す。確定･不確定モジュールのグラフに共通させるのは、仮パラメータの名前及び実パラメータへの制約のモジュール名を楕円で囲み

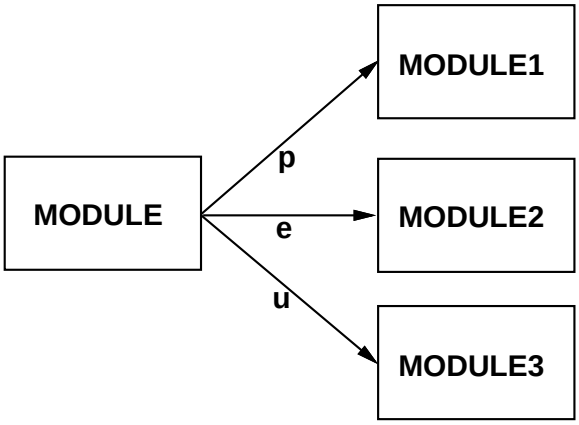


図 4.3: モジュールの輸入関係

(この楕円をパラメータ部と呼ぶ)、モジュールの上部に記述することである。制約を記述したモジュール (セオリ) もモジュールであるため、この不確定モジュールの外で定義されている必要がある。特定化を行ない確定したモジュールのグラフについては特定化の節で説明を加える。

不確定モジュールは角の丸い四角形で記述し、その内部にモジュール名を記述する。上部にはパラメータ部を記述する (図 4.4)。パラメータが多数ある際にはパラメータ部を並べて記入する。

```
module MODULE1 [ X :: THEORY ] {
  signature{ ... }
  axioms{ ... }
}
```

4.5 ビュー宣言

ビュー宣言はセオリから実パラメータであるターゲットモジュールへのマップの記述であるから、グラフ化を行なう際にはセオリ、ターゲットモジュール、ビュー名を菱形で囲んだビューの 3 者を用いる。図 4.5の様にセオリとターゲットモジュールとを結ぶ実線を引き、この実線からビューへの矢印を描くことで表現する。このとき、ビューから見て上部

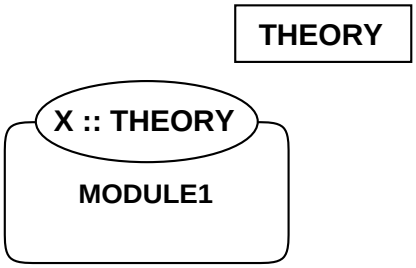


図 4.4: パラメータ付き (不確定) モジュール

にターゲットモジュールを記述し、下部にセオリモジュールを記述する。ここで利用するターゲットモジュールやセオリが他の部分で既に現れている場合にモジュールを囲む四角形を破線で描くのは、モジュールのグラフ表現の節で説明した通りである。

```
view VI1 from THEORY to MODULE2 {
  -- sort map
  -- operator map
}
```

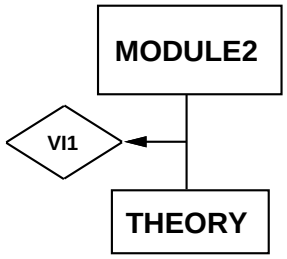


図 4.5: ビューのグラフ表現

ビュー宣言をグラフ化することにより、セオリとターゲットモジュールの組合せを一目で理解することが可能となるため可読性が生まれる。また、以降に行なわれる他の仕様構築の際にここで用いられているのと等しいセオリとターゲットモジュールの組を利用したビュー宣言を行なう場合には、このビューのグラフを見ることでビュー宣言の再利用を促すことが可能となるなどの再利用性にもつながるのである。基本的にはパラメータ付きモジュールのグラフ化による再利用の支援と同様の効果を見込むことが可能となる。

特定例として定義しようとしているモジュールの内部で直接にビュー宣言を行なうことにより、外部定義のビューを用いないような「名無しのビュー」が定義されることがある。名無しのビューを用いる場合にも、ここで定義を行なったビュー宣言のグラフを描くがその時に菱形で示したビューの内部にビュー名は記入しない。

4.6 特定化

特定化は仮パラメータに実パラメータを束縛する操作であるが、実パラメータがセオリの制約を満たしている上で束縛が可能となるということを示すため、特定化はを行なう際に直接仮パラメータに渡されるのはビュー宣言となっている。そこで、このモジュールグラフでの特定化の表現は、先に定義してあるパラメータ付きモジュールとビューとの組合せで実現する。

仮パラメータを記した楕円で囲んだ部分 (パラメータ部) へ向けて、菱形で記したビューから実線の矢印を引くのである。これにより、ビューで用いられたターゲットモジュールを実パラメータとして仮パラメータに束縛することを表すのである (図 4.6)。特定化もセオリモジュールを満たす実モジュールの輸入であると考えられるため、この実線の矢印は輸入の際に用いるものと同等の意味を持つと考えることができる。特定化の矢印であるという主張を必要とする場合には instantiation(特定化) の頭文字 i を矢印の横に添えることとする (図 4.6)。

```
MODULE1 [ X <= VI1 ]
```

これで、図 4.4に示した角の丸い四角形のパラメータ付き不確定モジュールが特定化され、確定モジュールとして記述されたことになる。この様に、確定モジュールは普段のモジュール同様四角形で示す。

特定化されたモジュールにもモジュール名を付けることが可能である。CafeOBJ では特定例に名前を付ける操作は (`protecting` モードでの) 輸入によって行なうが、モジュールグラフでは特定例自体が確定モジュールであるという考えから、このモジュールの構成要素を囲むように四角形を描き、内部の独立した場所にモジュール名を明記する。ここで、独立した場所とはモジュール構成要素のグラフに重ならないような場所を意味している。この様な記述を行なうことで、特定例の元となっているパラメータ付きモジュールやビュー、

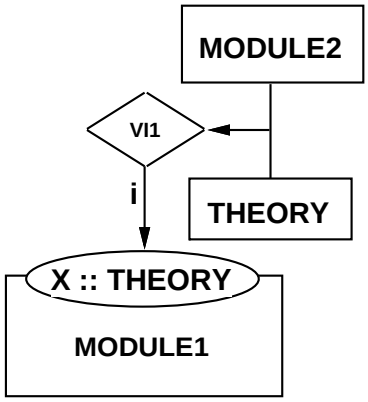


図 4.6: 特定化

さらには実パラメータまでを明確にすることを提案する (図 4.7)。この提案はモジュールの可読性を高め、更に保守性や拡張性の向上をも狙うものである。

```
make MODULE is MODULE1 [ X <= VI1 ] endm
```

ここで、前出の多重のパラメータ化 (図 3.6) のグラフ表現を考える。不確定モジュール SEQ を特定化せずに輸入することで、SORTING モジュール自体がパラメータ付きのモジュール、すなわち不確定モジュールとなる。よって、グラフ表現としては、SEQ は特定化はされていないが SORTING の内部構造として SORTING の仮パラメータに束縛されたことになるため、SEQ モジュールのは確定モジュールの表現形態をとる。SORTING は不確定モジュールの形状で示すこととなる。また、SEQ モジュールが SORTING モジュールの構成モジュールとなるため、これを内部に描き、対応するパラメータ部同士を矢印でつなぐのである。この矢印は、輸入の関係で描くと内側のモジュールから外側のモジュールへの向きになるが、ここではその反対の向きに記入する。後にこのモジュールが特定化される際の仮パラメータの束縛の流れは、外側のモジュール→内側のモジュールとなるからである。パラメータの束縛関係を可視化することで、不確定モジュールの扱われている部位の特定などが可能となる。すなわち、モジュールグラフによるコードの可読性の向上につながっていると考えられる。

また、モジュール SORTING 内部の輸入の部分に書かれているように、多重のパラメータ化を CafeOBJ のコードとして表現する際には、ビューを用いた仮パラメータから仮パラ

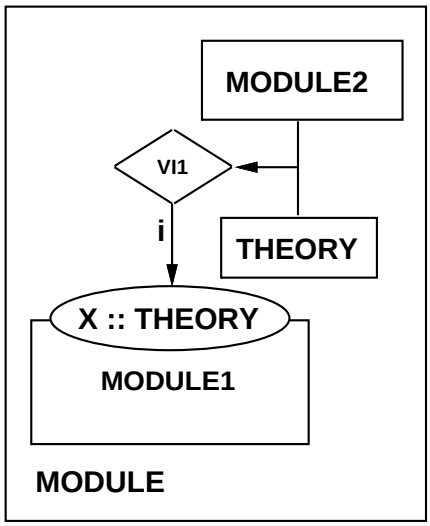


図 4.7: 特定例のモジュール表現

メータへの特定化を記述することとなっている。(SORTING では、SEQ の仮パラメータ X に対して SORTING の仮パラメータ ELT を束縛している。) ここで定義するモジュールグラフではこの関係にビューは書き込まない。その理由は、多重のパラメータ化を行なう時には、グラフ上外側に描かれたモジュールの要求する実パラメータへの制約が、構成要素として内部に取り込まれているパラメータ付きモジュールの要求する制約と等しいか、あるいはそれよりもきつい制約となるはずであるからである。もしも、内側の制約の方がきつくなっていると、外側のパラメータを束縛した実モジュールを内部のモジュールに束縛することができなくなり、モジュール構成要素が空となるのである。この様に、外側のモジュールと内側のモジュールでは制約のモジュール名は異なる可能性があるが、束縛されるモジュールは等しいものであるし、ビューを記入することによりモジュール内の簡潔さが損なわれることを避けるためにもビューの記入を行なわない方が良いと考えるのである。ここで定義した多重のパラメータ付きモジュールのグラフを図 4.8(a) に示した。同図の (b) には整数を定義したモジュール INT をもって (a) を特定化した例を示した。ここでも示した様に、特定化後は確定モジュールの記法を用いるのである。さらに (c) として、整数で特定化された SORTING のモジュール名を INT-SORTING とした例のモジュールグラフを描いた。

```
module SEQ [ X :: TRIV ]{
  ... Sequence
}

module SORTING [ ELT :: TOSET ]{
  protecting(SEQ [ X <= view to ELT { sort Elt -> Elt }])
  ... Sorting Elt sorted elements
}

view VI1 from TOSET to INT {
  sort Elt -> Int,
  op _<_ -> _<_
}

make INT-SORTING is SORTING [ ELT <= VI1 ] endm
```

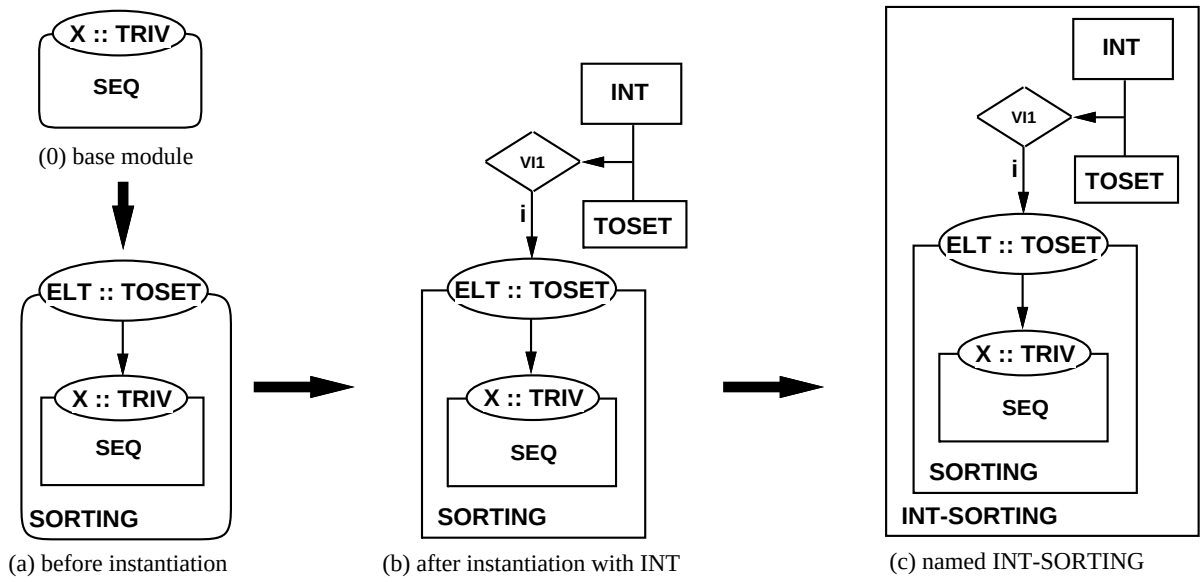


図 4.8: 多重の特定化のグラフ表現とその過程

4.7 名前替え

一枚のモジュールグラフ内に実線で描かれた同名のモジュールは一つしか存在しないように定義を行なってきた。しかし、CafeOBJ などでは、名前替えを行なうことで同一モジュールを、構造の等しい別のモジュールとして参照することが可能となっている。この様に名前替えにより一つのモジュールを数回参照する際には明示的に別モジュールであることを示す必要性がある。そこで、この場合に限り、等しいモジュールを実線で描くこととする。この時、名前替えがどの様に行なわれたかを示すことはモジュールグラフの本質ではないため、ここでは記入しない。

```
module MODULE1 {
  signature {
    [ E1 E2 ]
    op f : E1 -> E2
  }
  axioms { ... }
}
module MODULE2 {
  protecting(MODULE1 * { sort E1 -> Table1 , sort E2 -> Table2 ,
                         op f -> tablechange })
  protecting(MODULE1 * { sort E1 -> Score1 , sort E2 -> Score2 ,
                         op f -> scorechange })
  signature { ... }
  axioms { ... }
}
```

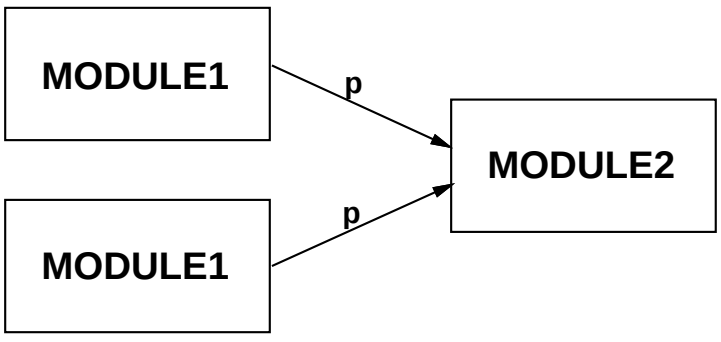


図 4.9: 名前替えによる多重輸入のグラフ

特定化されたパラメータ付きモジュールの名前替えを伴う利用であれば、パラメータの付かないモジュールの名前替えの場合と同様に描くことができる。しかし、輸入時に特定化を行なわない多重のパラメータ化と同時に、名前替えを伴って等しいモジュールを数回利用する場合には、内部モジュールとして同型のモジュールが実線で描かれることとなる。次に示すコードは、セオリに THEORY モジュールを指定したモジュール MODULE1 が、MODULE2 内で多重のパラメータ化と名前替えにより二通りに利用される例である。また、これに対応させたモジュールグラフを図 4.10 として記述した。

```
module MODULE1 [ X :: THEORY ] {
  signature {
    [ E1 E2 ]
    op f : E1 -> E2
  }
  axioms { ... }
}
module MODULE2 [ ELT1 :: THEORY, ELT2 :: THEORY ] {
  protecting(MODULE1 [ X <= view to ELT
    { sort mapping , operation mapping }
    * { sort E1 -> Table1 , sort E2 -> Table2 ,  -- Renaming 1
        op f -> tablechange })
  protecting(MODULE1 [ X <= view to ELT
    { sort mapping , operation mapping }
    * { sort E1 -> Score1 , sort E2 -> Score2 ,  -- Renaming 2
        op f -> scorechange })
  signature { ... }
  axioms { ... }
}
```

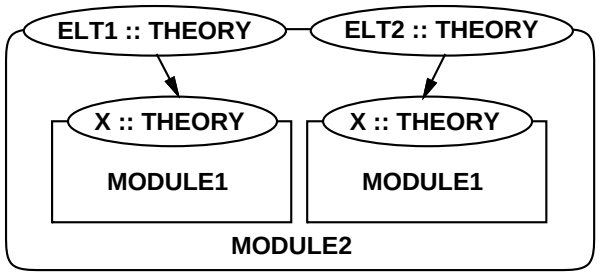


図 4.10: 多重のパラメータ化と名前替えによる多重輸入のグラフ

4.8 モジュールのグラフ化

この節でモジュールグラフによりモジュール定義の可読性が高まることの確認を行なう。前章の最後に示した複雑で可読性が損なわれたモジュール宣言の例をグラフ化を行なう。ここで再度コードを掲載する。

```
module MAKE-PARENTS-SET [ FATHER MOTHER PARENTS :: TRIV ]{
  protecting (SET [ X <= view to NAME
    [ X <= view to FATHER { sort Elt -> Elt }]
    * { sort Name -> FatherName, op name -> Fname }
       { sort Elt -> FatherName }]
       * { sort Set -> FatherSet , sort SetSeq -> FatherSetSeq ,
            op nil -> nilFather })
  protecting (SET [ X <= view to NAME
    [ X <= view to MOTHER { sort Elt -> Elt }]
    * { sort Name -> MotherName, op name -> Mname }
       { sort Elt -> MotherName }]
       * { sort Set -> MotherSet , sort SetSeq -> MotherSetSeq ,
            op nil -> nilMother , op _-_ -> _del_ })
  protecting (SET [ X <= view to PARENTS { sort Elt -> Elt }]
    * { op _U_ -> _union_ })
  signature{ ... }
  axioms{ ... }
}
```

このモジュールのヘッダ部分よりグラフ化に必要な情報を抽出する。

- モジュール名は MAKE-PARENTS-SET である。
- パラメータ付きモジュールである。パラメータは FATHER, MOTHER,PARENTS である。セオリは全パラメータについて TRIV である。
- 次のモジュールが輸入されている。
  1. 輸入されるモジュールはパラメータ付きモジュール SET である。
     - SET の仮パラメータはパラメータ付きモジュール NAME に特定化される。
     - NAME の仮パラメータは MAKE-PARENTS-SET の仮パラメータ FATHER に特定化される。
     - NAME は名前替えされる。
     - SET は名前替えされる。

2. 輸入されるモジュールはパラメータ付きモジュール SET である。
   - SET の仮パラメータはパラメータ付きモジュール NAME に特定化される。
   - NAME の仮パラメータは MAKE-PARENTS-SET の仮パラメータ MOTHER に特定化される。
   - NAME は名前替えされる。
   - SET は名前替えされる。
3. 輸入されるモジュールはパラメータ付きモジュール SET である。
   - SET の仮パラメータは、PARENTS に特定化される。
   - SET は名前替えされる。

モジュール MAKE-PARENTS-SET 内部で、パラメータ付きモジュール SET と NAME を名前替えと共に参照していることが分かった。上の情報をまとめた結果、図 4.11に示すモジュールグラフを描くことができた。コードと比較してモジュール構造の可読性が向上したことが分かる。

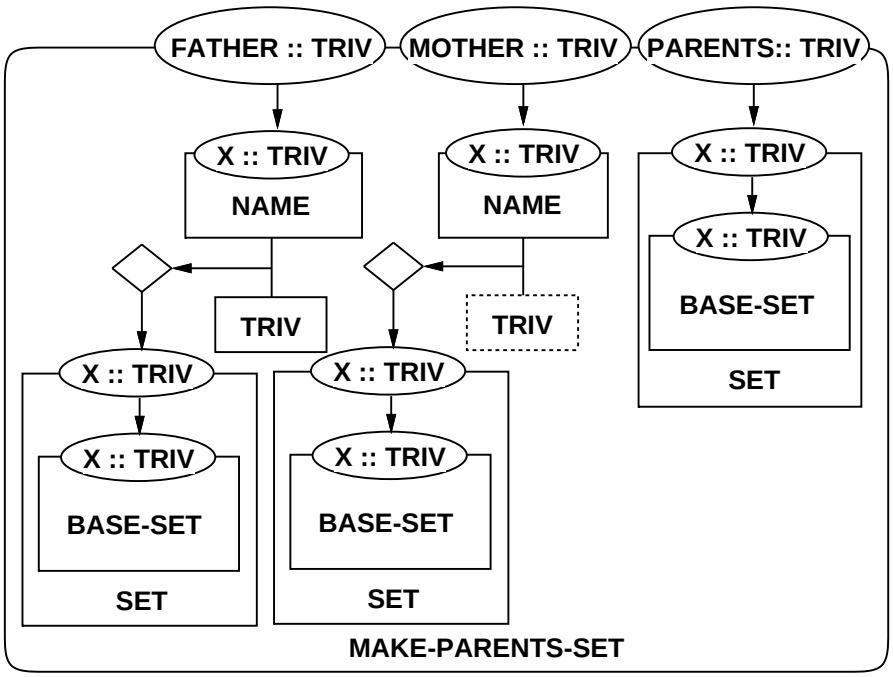


図 4.11: MAKE-PARENTS-SET モジュールのグラフ化

第 5 章

例題による検討

モジュールグラフを用いた仕様詳細化の可能性について検討する。本研究ではモジュールグラフのプログラムの改良時に与える影響について調査を行なった。実際にはデータ構造の挿替えのモジュールグラフ上での実現に注目した。

挿替え前後のモジュールグラフを比較して、データ構造以外のモジュール間で結合関係が変更されないようにプログラムの書き換えを実行することに注意した。結合関係に変更が生じるということはプログラムの構造変化につながり、安全性が失われると考えたからである。

5.1 テーブル検索

テーブルサーチの仕様を定義する。仕様への要求を以下に示す：
テーブルはデータ構造内にインデックス (INDEX) と値 (VAL) からなる二項組を格納している。渡された検索キーと等しいインデックスが格納されている二項組を探し出し、組になっている値を取り出す。

5.1.1 仕様とモジュールグラフによる表示

上記の要求を満たす仕様の構築を考える。まずはリストをデータ構造として採用することにより線形探索によるデータ検索の方法を考える。
今回定義する仕様においてインデックス (INDEX) と値 (VAL) に用いるソートはテーブルの最終的な用法が決定するまで不明である。よって、この二つの値を INDEX,VAL なる仮パラ

メータとする。またこれら仮パラメータに対する当面の制約は少なくとも一つのソートが宣言されていることである。そこで制約のモジュールとしては TRIV を用いておく。これで「あるソートのインデックス」及び「あるソートの値」の二項組を定義できる。二項組を要素に持つリスト (2TUPLE-LIST) を定義する。空リストは nil とし、二項組は CafeOBJ の混置記法を用いて次の演算で実現する。

```
<_,_> : Elt.INDEX Elt.VAL -> 2Tuple
```

定義中の_で示されている部分とアリティとの対応関係により引数の配置を決定するのが CafeOBJ の混置記法である。ここで Elt.INDEX は仮パラメータ INDEX の制約モジュール TRIV 内で定義されているソートを表している。後に特定化を行なう際にビューで記述されたソート Elt にマップされる実パラメータのソートが Elt.INDEX に置換されるのである。Elt.VAL も同様の意味を持つ仮のソートである。

```
module 2TUPLE-LIST[ INDEX :: TRIV , VAL :: TRIV ]{
  [ 2Tuple , TuList ]                                  -- ソート定義
  op nil : -> TuList { constr }                        -- 空リスト
  op <_,_> : Elt.INDEX Elt.VAL -> 2Tuple { constr } -- タプルの定義
  op list : 2Tuple TuList -> TuList                    -- タプルリスト
}
```

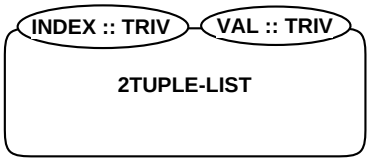


図 5.1: リストを用いたタプル

list はタプルとリストを取りリストを返す演算である。この演算でタプルを要素としたリスト構造の定義を行なっている。演算 list は 2Tuple と TuList を定義域のソートの系列とし、値域のソートは TuList とする。

タプルリストの検索を行なうモジュール TABLE を定義する。検索はインデックスと与えられた検索キーとの等価性比較に依るため、CafeOBJ 組み込みの二項関数_==_を用いる。この関数_==_は 2 つの基底項を各々正規形に還元し、構造上の同一性の判定を行なう。

```
module TABLE[ INDEX :: TRIV , VAL :: TRIV ]{
  -- タプルリストの輸入(多重パラメータ)
  protecting(2TUPLE-LIST[ INDEX <= view to INDEX { sort Elt -> Elt } ,
                          VAL   <= view to VAL  { sort Elt -> Elt } ])
  signature {
    -- テーブル検索の演算定義
    op _[_] : TuList Elt.INDEX -> Elt.VAL
  }
  axioms {
    vars I I' : Elt.INDEX
    var V : Elt.VAL
    var L : TuList
    -- テーブル検索
    eq list(< I, V >,L) [ I' ] = if I == I' then V else L [ I' ] fi .
  }
}
```

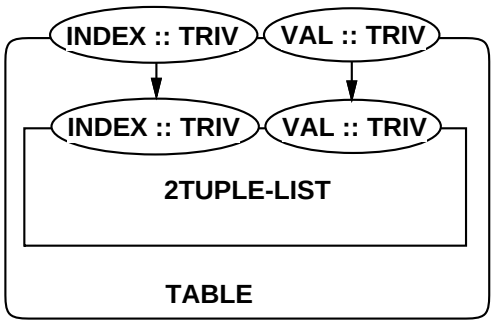


図 5.2: テーブルの検索モジュール

等価性の二項関数_==_の比較時には、対象となる 2 つの項のソートが等価であることが必要条件である。このため TABLE で用いるソートは 2TUPLE-LIST で定義されている二項組の要素と等しくする。モジュール TABLE の内部モジュールとして 2TUPLE-LIST を利用しているため、多重の特定化後は双方のモジュールで等しい実パラメータのモジュールを輸入し、そこで宣言されているソートの参照が可能となる。この機構により等価性の二項関数に現れるソートが別のものとなる危険性を排除することが可能となるのである。

テーブル検索の方針にはリストの線形探索を用いている。ここではリストの先頭要素のタプルのインデックス値と検索キーとを比較し、等しければタプルの値を返す操作を定義

した。等価でない場合はリストの次の要素について比較を行なう。この操作をリストの要素がなくなるまで繰り返す。今回の定義ではリスト内に検索キーと等しい要素がない場合の動作は未定義としている。プロトタイピング法では代表となる操作の定義がなされていれば良いため、現在のところ定義の不要な操作については手を付けないことにした。

```
TABLE[ INDEX <= view to STRING { sort Elt -> String } ,
       VAL   <= view to INT    { sort Elt -> Int    } ]
}
```

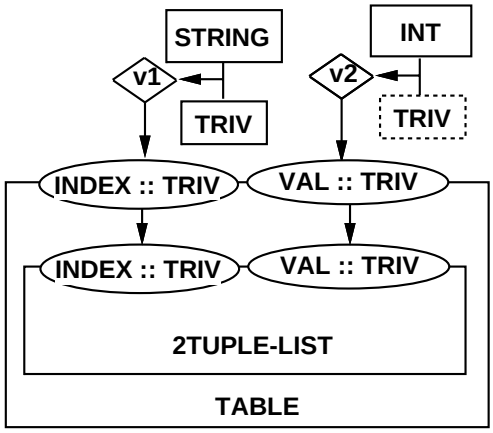


図 5.3: テーブルの特定化

図5.3にTABLEの特定例を示した。この特定例ではINDEXに文字列を、VALに整数を各々用いる。文字列の集合とそれに対する演算の定義を行なっているモジュールSTRINGと、整数の集合とそれに対する演算の定義を行なっているモジュールINTとを各仮パラメータに束縛している。ここで出来上がった特定例のモジュールをTABLEStrIntとする。

この特定例を用いて、定義したモデルの CafeOBJ 上での実行による検証が可能である。次に示す検証例に示すように書き換え結果として期待通りの項を得ることができた。これにより、このモデルは正しく定義されていると考えることができる。

```
    StrIntTABLE> red list(< "r" , 32 >,
                          list(< "a" , 2 >,
                            list(< "w" , 92 >, nil))) [ "a" ] .
    -- reduce in StrIntTABLE :
```

```
     list(< "r" , 32 >,list(< "a" , 2 >,list(< "w" , 92 >,
       nil))) [ "a" ]
2 : NzNat
(0.017 sec for parse, 6 rewrites(0.033 sec), 7 match attempts)

StrIntTABLE> red list(< "c" , 1 >,
                  list(< "a" , 2 >,
                    list(< "f" , 3 >,
                      list(< "e" , 4 >, nil)))) [ "o" ] .
-- reduce in StrIntTABLE :
     list(< "c" , 32 >,list(< "a" , 2 >,list(< "f" , 92 >,
       list(< "e" , 3 >,nil)))) [ "o" ]
nil [ "o" ] : Int
(0.017 sec for parse, 12 rewrites(0.017 sec), 18 match attempts)
```

実行結果について触れる。1~8 行目までが最初の実行例で 10 行目からは次の例である。StrIntTABLE>は CafeOBJ のカレント·モジュールを表したプロンプトである。red は reduce の略であり、以降に記述した項の簡約を要求するコマンドである。続く list(···) ["a"] が与えた項である。項の終端にはピリオドを付加する。この下の 3 行は処理系の解釈が表示された部分であり、続いて示されているのが簡約結果である。結果に続く括弧で括られた行は簡約に要した時間や書き換え回数などを表している。

最初の例では< "r" , 32 >, < "a" , 2 >, < "w" , 92 >, nil のタプルリストから "a"を検索し、これと組になっている値 2 が出力されたことになる。次の例では< "c" , 1 >,< "a" , 2 >,< "f" , 3 >,< "e" , 4 >,nil 内の"o"を検索するが、リスト内に存在しなかったため空リストに対する検索の状態 (エラーを定義していない) で終了している。

出来上がったプログラムのモジュールグラフによる表現を図 5.4 (0) に示す。

図 5.4(0) からデータ構造の変更を考える。この変更による検索の実行効率の向上を図るため二分木を採用する。モジュールグラフ上でのデータ構造の変更過程を (0)~(3) に示す。

- (0)~(1):TABLE の内部構造として用いた 2TUPLE-LIST を二分木構造に変更。このため、INDEX への制約は厳しいものへと変更になっている。
- (1)~(2):INDEX への制約変更を受けて TABLE の仮パラメータへの制約を変更。
- (2)~(3):不確定モジュール 2TUPLE-TREE を TABLE の内部構造に採用。モジュール TABLE を二分木に対応させ bTABLE とする。

ここで図示した変更過程を元に bTABLE,2TUPLE-TREE の構築を行なう。

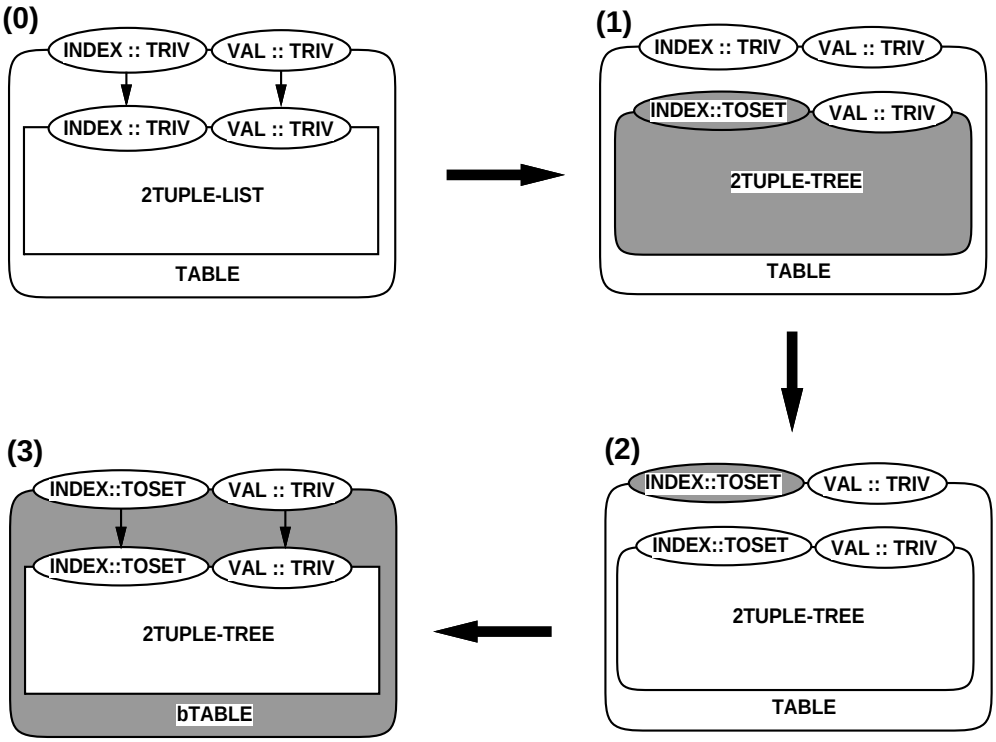


図 5.4: テーブル検索プログラムのモジュールグラフ

現段階で変更を加えることが明確になっている部分をグラフ上で示し、それに従い順次変更を加えていくのが今回提案するモジュールグラフを用いた詳細化である。あるモジュールに変更を加えた場合、このモジュールの参照、意味の付加を行なうモジュールにこの変更の影響が現れる。モジュールグラフでは多重のパラメータ化によるモジュールの階層構造を包含関係として示すため、内部に記載されているモジュールに変更が行なわれた際には、そのモジュールを含んでいる外部モジュールの変更確認の必要性を示すこと可能なグラフであることが重要である。

現在変更の必要が明らかなのは 2TUPLE-LIST とそのセオリであるが、グラフの包含関係に従い、モジュール変更の波紋は 2TUPLE-LIST→TABLE の順に広がっていくと考えれば良いのである。

線形探索では CafeOBJ にあらかじめ組み込まれている等価性比較 (_==_) を用いてインデックスの値と検索キーとを比較していた。よってここで求められる制約はソートが一つ以上宣言されていることであり、緩い制約の TRIV を用いていた。しかし、二分木では検索キーと対象の木の要素の大小比較を行ない、比較結果により左右の検索対象木を決定する。そのために全順序関係が定義されている値を用いる必要性が生まれる。そこで、仮パラメータ INDEX のセオリに全順序関係の意味を記述したモジュール TOSET を用いることに決めた。内部で参照しているモジュールの制約がきついものとなったことを受けて、TABLE モジュールにもセオリ変更が必要となった。これらの変更を元に演算等を書き換えたモジュール定義を行なう。

```
module TOSET {
  signature {
    [ Elt ]
    op _<_ : Elt Elt -> Bool
  }
  axioms {
    vars E1 E2 E3 : Elt
    eq (E1 < E1) = false .
    eq (E1 < E2) or (E2 < E1) or (E1 == E2) = true .
    eq (E1 < E2 and E2 < E3) implies (E1 < E3) = true .
  }
}
module 2TUPLE-TREE[ INDEX :: TOSET , VAL :: TRIV ]{
  [ 2Tuple , TuTree ]
  op nil : -> TuTree { constr }
  op <_,_> : Elt.INDEX Elt.VAL -> 2Tuple { constr }
  op tree : 2Tuple TuTree TuTree -> TuTree          -- 二分木の定義
}
```

CafeOBJ では基本的に全てのモジュールにモジュール BOOL が輸入されるため、そこで定義されている and,or,xor,not,implies の各演算と、BOOL が輸入しているモジュール内で定義された true,false の各項の利用が許される。モジュール TOSET ではこれらの言語要素を用いている。

次に、2TUPLE-LIST を挿替えた結果 TABLE に必要となった変更をもとに演算定義を書き直す。現実には、

```
eq list(< I, V >,L) [ I' ] = if I == I' then V else L [ I' ] fi .
```

を二分木に対応させるべく変更を加える。

ceq という条件付きの等式定義により、検索キーとタプルのインデックスとの比較を行なう。等価な場合はそのタプルの値を出力し、等価でない場合は大小比較の結果により検索キーが大きければ右の木へ、小さければ左の木へ検索対象の木を変更する。木の要素が空になるまでこれらの操作を行なう。という具合に二分木検索の定義を行なう。この結果、等式定義は次のように書き換えられた。

```
ceq tree(< I', V >,LEFT,RIGHT) [ I ] = V :if I == I' .
ceq tree(< I', V >,LEFT,RIGHT) [ I ] = LEFT [ I ] :if I < I' .
ceq tree(< I', V >,LEFT,RIGHT) [ I ] = RIGHT [ I ] :if I' < I .
```

ceq は、:if 以降が成立する場合に限り前半の等式の右辺が左辺に書き換わる。という宣言である。

```
module bTABLE[ INDEX :: TOSET , VAL :: TRIV ] {
  protecting(2TUPLE-TREE[ INDEX <= view to INDEX { sort Elt -> Elt ,
                                                   op _<_ -> _<_ } ,
                          VAL   <= view to VAL  { sort Elt -> Elt } ])
  signature {
    op _[_] : TuTree Elt.INDEX -> Elt.VAL
  }
  axioms {
    vars I I' : Elt.INDEX
    var  V : Elt.VAL
    vars LEFT RIGHT : TuTree
    ceq tree(< I', V >,LEFT,RIGHT) [ I ] = V :if I == I' .
    ceq tree(< I', V >,LEFT,RIGHT) [ I ] = LEFT [ I ] :if I < I' .
    ceq tree(< I', V >,LEFT,RIGHT) [ I ] = RIGHT [ I ] :if I' < I .
  }
}
```

リスト構造を二分木の構造に変更した結果、TABLE モジュール内部の検索部分への変更が必要になった。モジュール内部の他の変更箇所は名前替えにより対処可能となっている。

特定例については、セオリの変更に伴い特定化に用いるビューの部分に変更を加える必要がある。

```
module bTABLEStrInt {
  protecting (bTABLE[
               INDEX <= view to STRING { sort Elt -> String ,
                                         op   _<_ -> string< } ,
               VAL   <= view to INT    { sort Elt -> Int     } ])
}
```

TOSET で定義されている演算_<_ に対するマップとして string<を宣言する。この演算は文字列の順序関係を定義する演算であり、モジュール STRING で定義がされている。

最終的にモジュール TABLE のデータ構造をリストから二分木へ挿替えることを目的として 2TUPLE-LIST から 2TUPLE-TREE へのモジュールの変更を行なったが、こ変更の波紋はモジュール全体に広がったといえる。

5.2 データ構造の要素数の算出

データ構造に格納されている要素数を計算する操作に必要な仕様定義を以下で行なう。

5.2.1 仕様とモジュールグラフによる表示

集合の要素数を数える仕様を定義する。要素数計算を行なうモジュールを定義する前に、まず集合の定義を行なう。

汎用性のある集合を定義するため、要素のソートはパラメータとする。要素の列にベキ等則の属性を持たせることで集合の基本形態とする。列のソートを SetSeq とした。集合定義を実現する演算を以下に示す。

```
{_} : SetSeq -> Set
```

CafeOBJ で演算にベキ等則のなどの属性を与える場合、演算定義の後ろに {,} で括った宣言を付加する。ベキ等則を満たす演算の定義には {idem} を付加する。この他に assoc, comm, id: という宣言により各々結合則, 交換則, 単位元の存在を満たす演算の定義が可能である。単位元は id: nil と宣言することで nil と記述した項を単位元とした定義が可能である。ただし nil が項である旨の定義が先に必要である。定義した演算記号が構成子であることを宣言するには constr を用いる。集合定義のモジュール BASE-SET ではここで述べた宣言を全て利用している。

```
module BASE-SET [ X :: TRIV ] {
  signature{
    [ Set Elt < SetSeq ]
    op {_} : SetSeq -> Set { constr }
    op nil : -> SetSeq  { constr }
    op _,_ : SetSeq SetSeq -> SetSeq
                    { assoc comm idem idr: nil constr }
  }
}
```

[Set Elt < SetSeq] はソート宣言である。ここで用いる<はソートの順序付けに用いられる。集合ソート Set と要素のソート Elt とを共に要素の列 SetSeq の下位ソートに位置付けているのは、集合の要素として集合を持たせることを可能にするためである。集合

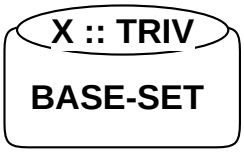


図 5.5: 集合の基本定義

の基本構造を定義したモジュール BASE-SET に対し、集合要素数の計算を行なう演算を定義する。

定義するモジュール名を SET-SIZE とする。このモジュールでは BASE-SET に格納された要素数の計算の際に整数の足し算を行なうため、整数を定義したモジュール INT の輸入を行なう。集合要素は一旦列となる定義を BASE-SET で行なったため、要素数の計算は列の長さを計算する方法と等しい方針を用いる。集合が入れ子になっている場合には再帰的に集合要素数を計算する。

```
module SET-SIZE [ X :: TRIV ] {
  protecting(BASE-SET [ X <= view to X { sort Elt -> Elt }])
  protecting(INT)
  signature{
    op size : SetSeq -> Int
  }
  axioms {
    var E : Elt
    var S : Set
    vars SS SS2 : SetSeq
    eq size(nil) = 0 .
    eq size({ E }) = 1 .
    eq size({ S }) = size(S) .
    eq size({ E , SS }) = 1 + size({ SS }) .
    eq size({ S , SS }) = size(S) + size({ SS }) .
  }
}
```

仕様の正当性を実行により検証するため、特定例の導出を行なう。ここでは集合の要素として文字列を用いる。文字列は CafeOBJ の組み込みモジュール STRING として定義してある。

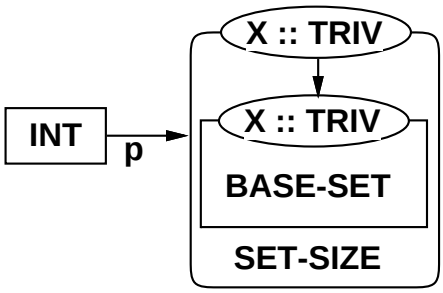


図 5.6: 集合の要素数算出のモジュール

```
module Str-SET-SIZE {
  protecting (SET-SIZE [ X <= view to STRING { sort Elt -> String }])
}
```

定義した文字列集合の要素数を算出する例を示す。ここで得られた結果は期待通りである

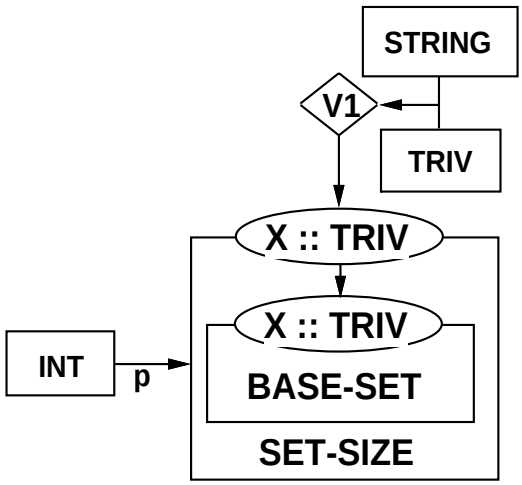


図 5.7: 文字列集合の要素数算出のモジュール

と考えることができる。よって、この仕様のモデルは正しいと判断することができる。

```
Str-SET-SIZE> red size({"a" , "b" , "c" , nil}) .
-- reduce in Str-SET-SIZE :
     size({ ("a" , "b" , "c" , nil) })
3 : NzNat
```

```
(0.017 sec for parse, 6 rewrites(0.017 sec), 23 match attempts)

Str-SET-SIZE> red size({ {"c" , "a" , "f" , "e" , nil } ,
                         {"o" , "b" , "j" , nil } , nil }) .
-- reduce in Str-SET-SIZE :
     size({ ({ ("c" , "a" , "f" , "e" , nil) } ,
              { ("o" , "b" , "j" , nil) } , nil) })
7 : NzNat
(0.017 sec for parse, 18 rewrites(0.067 sec), 71 match attempts)
```

最初に挙げた例では、集合は {"a" , "b" , "c" , nil} である。nil は空の要素であるため要素数には入らない。このため、この集合の要素数は 3 である。次の例では集合が要素となっている。{ "c" , "a" , "f" , "e" , nil } の要素数 4 と、{ "o" , "b" , "j" , nil } の要素数 3 とを足し併せて、この集合全体の要素数は 7 となる。以上で文字列を要素にとる集合の要素数の算出を行なうプログラムの定義が完了した。モジュールグラフでモジュール SET-SIZE を示し、データ構造の挿替えについての検討を行なう。図 5.8(0) がここで定義した仕様のグラフ表現である。

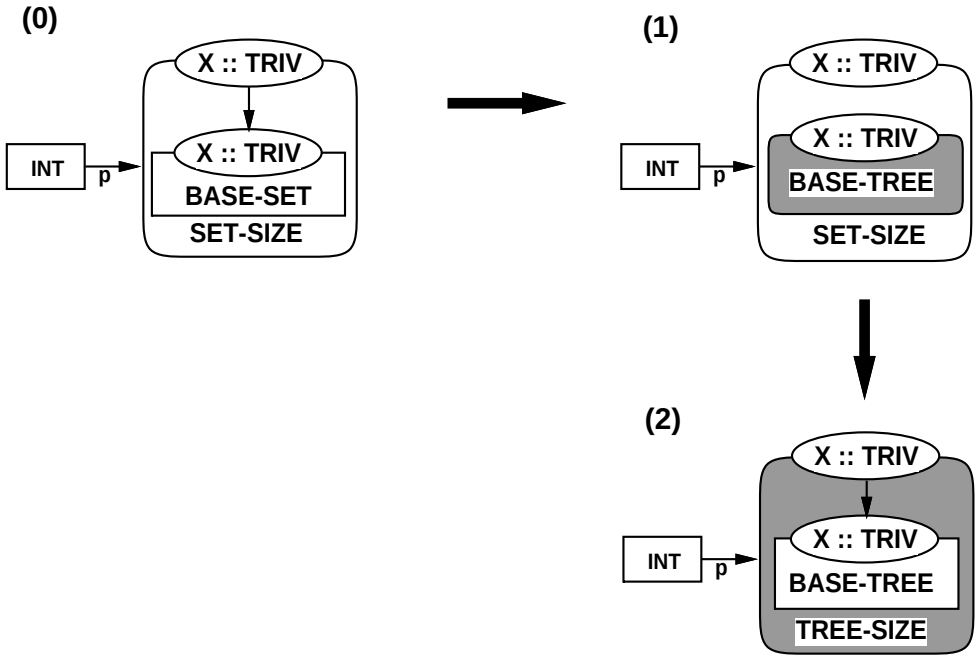


図 5.8: データ構造の要素数を算出するプログラムのモジュールグラフ

ここで用いた集合は要素として集合を利用することができるような工夫を行なっている。

そこで他のデータ構造に挿替えを行なう際にもデータ構造自体を要素として持つことが可能となる構造にすることで、変更箇所を最小限にとどめる。今回はデータ構造を一般の木に挿替える。図 5.8(1) に示すように BASE-SET に代わる BASE-TREE の定義を経て、データ構造を一般の木へ挿替える。この図で示した変更に従いコードの変更を進める。

集合の演算定義から一般の木の演算定義への変更を以下に示す。まず集合の演算定義は次の様に行なっていた。

```
op {_} : SetSeq -> Set { constr }
```

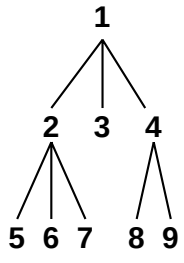
これを下のように変更する。

```
op node_[_] : Elt TreeSeq -> Tree { constr }
```

この変更で一般の木を実現する。node に名札と部分木のリストが続く。空の部分木リストを nil と定義することで、木を node 1 [node 0 [nil]] のように記述する。

```
module BASE-TREE [ X :: TRIV ] {
  [ Tree < TreeSeq ]
  signature {
    op node_[_] : Elt TreeSeq -> Tree { constr }
    op nil : -> TreeSeq  { constr }
    op _,_ : TreeSeq TreeSeq -> TreeSeq
                    { assoc comm id: nil constr }
  }
}
```

ここで定義した一般の木によるデータの表現例を図 5.9に示す。なお、ここでは整数の特定例を用いている。




```
node 1 [ node 2 [ node 5 [ nil ] ,
                  node 6 [ nil ] ,
                  node 7 [ nil ] ] ,
         node 3 [ nil ] ,
         node 4 [ node 8 [ nil ] ,
                  node 9 [ nil ] ] ]
```

図 5.9: 一般の木でのデータ表現

データ構造の集合 BASE-SET から一般の木 BASE-TREE への挿替えに伴う SET-SIZE への変更の必要性を確認する。モジュール SET-SIZE では要素数の計算の等式は再帰的な計算の宣言となっている。データ構造の変更により、ここに変更が必要であることが分かる。そこで等式宣言を一般の木に対応させる変更を行なった。変更結果を以下の TREE-SIZE に示す。

```
module TREE-SIZE [ X :: TRIV ] {
  protecting(BASE-TREE [ X <= view to X { sort Elt -> Elt }])
  protecting(INT)
  signature {
    op size : TreeSeq -> Int
  }
  axioms {
    var E : Elt
    var T : Tree
    vars TS TS2 : TreeSeq
    eq size(nil) = 0 .
    eq size(node E [ nil ]) = 1 .
    eq size(node E [ nil ] , TS ) = 1 + size(TS) .
    eq size(node E [ TS ] ) = 1 + size(TS) .
    eq size(node E [ TS ] , TS2 ) = 1 + size(TS) + size(TS2) .
  }
}
```

等式定義の部分以外は名前替えによる変更にとどめることができた。今回は要素数の計算が可能であれば仕様の要求を満すことになる。このため、パラメータの制約はデータ構造の挿替えによらず緩い TRIV を用いている。

特定例は集合を用いた場合同様に文字列を要素として持たせる。

```
module Str-TREE-SIZE {
  protecting(TREE-SIZE [ X <= view to STRING { sort Elt -> String }])
}
```

特定化例に実際の項を入力し、書き換えの結果を先の実行例と比較することで等しい結果となっていることの検証を行なった。

```
    Str-TREE-SIZE> red size(node "a" [ node "b" [ nil ] ,
                                       node "c" [ nil ] ]) .
    -- reduce in Str-TREE-SIZE :
          size(node "a" [ (node "b" [ nil ] ,
                           node "c" [ nil ]) ])
```

```
3 : NzNat
(0.017 sec for parse, 5 rewrites(0.033 sec), 13 match attempts)

Str-TREE-SIZE> red size(node "c" [ node "a" [ node "f" [ nil ] ,
                                               node "e" [ nil ]] ,
                                     node "o" [ node "b" [ nil ] ,
                                               node "j" [ nil ]]]) .
-- reduce in Str-TREE-SIZE :
     size(node "c" [ (node "a" [ (node "f" [ nil ] ,
                                   node "e" [ nil ]) ] ,
                         node "o" [ (node "b" [ nil ] ,
                                   node "j" [ nil ]) ]) ])
7 : NzNat
(0.033 sec for parse, 13 rewrites(0.033 sec), 35 match attempts)
```

先の実行例と比較して、与える項の記述の仕方や簡約に要する時間などに違いはあるが、結果は等しい。よって正しいモデルの定義が行なわれたといえる。

5.3 二つのグループから別のグループを作り出す例

二つのデータ構造に格納されたデータ群のマージをとる仕様の作成である。詳細を述べると、父親の名前が格納されているグループと母親の名前が格納されているグループとから両親のグループを作り出すというものである。父親と母親の名前のデータ表現は演算 name により実現する。両親のデータ表現は演算 parents で実現することとする。

5.3.1 仕様とモジュールグラフによる表示

あらかじめ名前の表現法を定義する必要がある。ソート Name を用意し、姓・名の 2 引数を取ってソート Name を返す演算の定義を行ない、モジュール名を NAME とする。姓・名はパラメータとしておく。演算 name の定義域は仮パラメータのソートとし、値域は Name とする。

```
module NAME [ X :: TRIV ] {
  [ Name ]
  op nilNam : -> Elt
  -- name(姓, 名)
  op name : Elt Elt -> Name
}
```

図 5.10: 名前モジュールのグラフ表現

用いるデータ構造の選定であるが、まずは安直に要求を満たすことを考えて集合を用いることとする。集合の基本定義は前節で定義した BASE-SET を用いる。このモジュールでは集合に対する操作の定義が一切なされていない。よって集合演算を行なう今回の例を実現するには BASE-SET の上に新たな演算を付け加える必要がある。演算を付け加えるにはモジュールの輸入を行なうが、BASE-SET の仮パラメータが特定されていない状態であるため、多重のパラメータ化となる。集合の演算定義を加えたモジュールのモジュール名を SET とする。

今回定義するモジュール MAKE-PARENTS で必要な集合の操作は、集合内に特定の要素が含まれることを調べる述語と両親集合を作る際に用いると思われる和集合及び差集合の演算である。そこで、集合のモジュール上にこれらの演算を定義する。内包性確認の述語は_in_という二項関数の形で実現する。等式定義は線形検索の要領で行なう。また和･差集合も定義域･値域に集合のソートをとる二項演算として定義する。

```
module SET [ S :: TRIV ] {
  protecting (BASE-SET [ X <= view to S { sort Elt -> Elt }])
  signature {
    [ Set , Elt < SetSeq ]
    op _in_ : Elt Set -> Bool                -- Element Search
    op _U_ : Set Set -> Set { assoc comm }   -- Union
    op _-_ : Set Set -> Set                  -- Difference
  }
  axioms {
    vars E E1 E2 : Elt
    vars ES1 ES2 : SetSeq
    vars S1 S2 : Set
    eq E in { nil } = false .               --  集合要素の存在
    eq E in { E1 , ES1 } =                          --|
       if E == E1 then true else E in { ES1 } fi .  --|
    eq { nil } U S1 = S1 .                   --    和集合
    eq { ES1 } U { ES2 } = { ES1 , ES2 } .           --|
    eq { ES1 } - { nil } = { ES1 } .         --    差集合
    eq { nil } - { ES1 } = { nil } .                 --|
    eq { E , ES1 } - { ES2 } =                       --|
       if E in { ES2 } then { ES1 } - { ES2 }        --|
              else { E } U ({ ES1 } - { ES2 }) fi . --|
  }
}
```

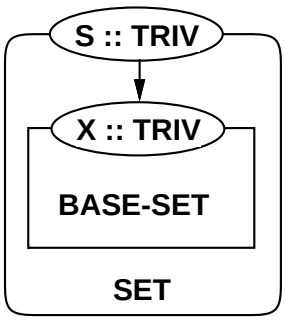


図 5.11: 集合モジュールのグラフ表現

集合モジュール SET を用いて、父親の名前が格納されている集合と母親の名前が格納されている集合から名字の等しいものを捜し出し、両親の名前の格納された集合を作り出す

仕様の定義を行なう。両親を表現するため、parents として両親の姓・父親の名・母親の名の 3 引数から両親の名前を格納した項を返す演算の定義を行なった。

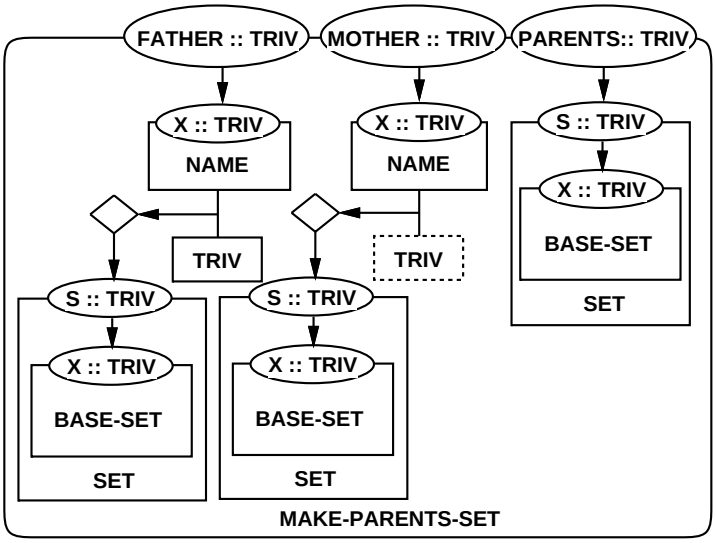


図 5.12: 両親集合を作り出すモジュールのグラフ表現

```
module MAKE-PARENTS-SET [ FATHER MOTHER PARENTS :: TRIV]{
  protecting(SET [ S <= view to
    NAME [ X <= view to FATHER { sort Elt -> Elt }]
      * { sort Name -> FatherName , op name -> Fname }
         { sort Elt -> FatherName}]
       * { sort Set -> FatherSet , sort SetSeq -> FatherSetSeq
           op nil -> nilFather })
  protecting(SET[ S <= view to
    NAME [ X <= view to MOTHER { sort Elt -> Elt }]
      * { Renaming to define Mother-SET same as Father . }
  protecting(SET[ S <= view to PARENTS { sort Elt -> Elt }])
  signature {
    [ Parents < Elt.PARENTS ]
    op parents : Elt.FATHER Elt.FATHER Elt.MOTHER -> Parents
    op parents : Elt.MOTHER Elt.FATHER Elt.MOTHER -> Parents
    op MKparents : FatherSet MotherSet -> Set
    op getMfi : Elt.FATHER MotherSet -> Elt.MOTHER
    op getMfa : Elt.FATHER MotherSet -> Elt.MOTHER
  }
  axioms {
    var Sf : FatherSetSeq ; var Sm : MotherSetSeq
```

```
        vars Ffa Ffi : Elt.FATHER ; vars Mfa Mfi : Elt.MOTHER
        eq MKparents({nilF},{nilM}) = {nil} .
        eq MKparents({nilF},{Mname(Mfa,Mfi),Sm}) =
             {parents(Mfa,nilFNam,Mfi) } U MKparents({nilF},{Sm}) .
        eq MKparents({Fname(Ffa,Ffi),Sf},{nilM}) =
             {parents(Ffa,Ffi,nilMNam) } U MKparents({Sf},{nilM}) .
        eq MKparents({Fname(Ffa,Ffi),Sf},{ Sm }) =
             { parents(Ffa,Ffi,getMfi(Ffa,{ Sm })) } U
                 MKparents({Sf},{Sm}
                    Mdef {Mname(getMfa(Ffa,{Sm}),getMfi(Ffa,{Sm}))}) .
        eq getMfi(Ffa,{ nilM }) = nilMNam .
        eq getMfi(Ffa,{Mname(Mfa,Mfi),Sm}) =
             if Ffa == Mfa then Mfi else getMfi(Ffa,{Sm}) fi .
        eq getMfa(Ffa,{ nilM }) = nilMNam .
        eq getMfa(Ffa,{Mname(Mfa,Mfi),Sm}) =
             if Ffa == Mfa then Mfa else getMfa(Ffa,{Sm}) fi .
      }
    }
```

モジュールグラフ (図 5.12) からも明らかであるが、このモジュールでは父親集合と母親集合とから両親集合を作り出すために、これらに用いる集合をそれぞれ輸入により用意している。このとき集合の演算名をそのまま用いたのでは 3 者で同名の演算を参照することになり定義が曖昧なものとなる。そこで輸入を行なう際に名前替えを利用してこの曖昧性を排除する。

モジュールグラフでは名前替えを伴った輸入を行なう際には、同一のモジュールをそれぞれ実線で記入することでこの意味を表明することにしていた。ゆえに図 5.12では SET などのモジュールが実線で 3 度描かれているのである。

父親集合と母親集合とは名前の集合であるため、SET モジュールの仮パラメータにパラメータ付きモジュール NAME を特定化している。モジュール NAME 内の演算も名前替えにより曖昧性排除がされる。コード上でこの作業はデータ構造の輸入を行なう protecting(...) の括弧内で一度に記述されているため名無しのビューが用いられている。図中でビューの菱形の内部にビュー名が記述されていないのはそのためである。

次に仮パラメータを全て文字列に特定化する宣言をおこなう。

```
module MAKE-Str-PARENTS-SET {
  protecting (MAKE-PARENTS-SET
       [ FATHER <= view to STRING { sort Elt -> String },
         MOTHER <= view to STRING { sort Elt -> String },
         PARENTS <= view to STRING { sort Elt -> String}])
}
```

ここではデータ構造として集合を用いたが、この仕様からの二分木を用いた仕様の導出を考える。一旦モジュールグラフへ変換して二分木を用いたモデル作成時の変更箇所の特定を行なう。

モジュールグラフ (図 5.13) の (0) が MAKE-PARENTS-SET のモジュールグラフによる表現である。この図から二分木による実現 (5) に至るまでの変更について説明する。

- (0)~(1):データ構造を二分木に変更するためモジュールグラフ上のデータ構造部分を書き換える。集合の基本構造を示したモジュール BASE-SET を用いていることから二分木も基本構造を記述したモジュール BASE-TREE を用意する。
- (1)~(2):データ構造部分の二分木化を終了。仮パラメータへの制約は全順序関係を記述したセオリ TOSET となる。
- (2)~(3):二分木に格納される要素は全順序関係を持つデータ集合となる。このため、ビューで用いられるセオリも TRIV よりもきつい制約の TOSET となる。
- (3)~(4):モジュール NAME の仮パラメータへの制約も同様に変更が必要になる。
- (4)~(5):MAKE-PARENTS-SET を MAKE-PARENTS-TREE と改名し、仮パラメータへの制約を変更する。

上記で終えたモジュールの構造の変更を元に MAKE-PARENTS-SET 内の等式を二分木構造へ対応させる。これで MAKE-PARENTS-TREE の実体を得る。

二分木の定義は BINARY-TREE で行なう。仕様の定義上、列のモジュールを利用する。BASE-SEQ に対する多重のパラメータ化を行なう。

```
module BINARY-TREE[ELT :: TOSET] {
  extending (BASE-SEQ[X <= view to ELT { sort Elt -> Elt }])
  [ BinaryTree ]
  signature {
    op makeBinTree_ : Seq -> BinaryTree
    op flatten_ : BinaryTree -> Seq
    op nilTree : -> BinaryTree
    op binTree : Elt BinaryTree BinaryTree -> BinaryTree
    op insert_to_ : Elt BinaryTree -> BinaryTree
    op delete_from_ : Elt BinaryTree -> BinaryTree
  }
  axioms {
    var L : Seq
    vars E E' : Elt
    vars T T' : BinaryTree
    eq makeBinTree(nilSeq) = nilTree .
    eq makeBinTree(E L) = insert E to (makeBinTree L) .
    eq insert E to nilTree = binTree(E, nilTree, nilTree) .
    eq insert E to binTree(E', T, T') =
        if E < E' or E == E'
          then binTree(E', (insert E to T), T')
          else binTree(E', T, (insert E to T')) fi .
    eq delete E from T = makeBinTree(drop(E,flatten T)) .
    eq flatten nilTree = nilSeq .
    eq flatten(binTree(E, T, T')) =
           ((flatten T) $ (E (flatten T'))) .
  }
}
```

集合で仕様化を行なった上記プログラムからデータ構造を二分木に挿替えたことにより、仮パラメータへの制約であるセオリがただ一つのソート宣言を要求する TRIV から、全順序関係を定義した TOSET へ変更になった。BINARY-TREE の内部モジュールとして参照している BASE-SEQ のセオリを示していないが、内部モジュールのセオリは外部のセオリと等しいか緩い制約であることが要求される。きつい制約を持つモジュールを内部構造として参照するモジュール定義は、ビュー宣言の定義から不可能となっている。実際に BASE-SEQ のセオリは TRIV である。

BINARY-TREE の利用による MAKE-PARENTS-SET のモジュール構造の変化をモジュールグラフで示すことができた。このモジュール構造を元に MAKE-PARENTS-TREE の構築を行なった。

```
module MAKE-PARENTS-TREE [ FATHER MOTHER PARENTS :: TOSET ] {
  protecting(BINARY-TREE[ ELT <= view to
    NAME [ X <= view to FATHER { sort Elt -> Elt , op _<_ -> _<_ }]
      * { sort Name -> FatherName , op name -> Fname }
         { sort Elt -> FatherName , op _<_ -> name< }]
       * { sort BinaryTree -> FatherTree , sort Seq -> FatherSeq
           op nilTree -> nilFather , op binTree -> binFTree })
  protecting(BINARY-TREE[ ELT <= view to
    NAME [ X <= view to MOTHER { sort Elt -> Elt , op _<_ -> _<_ }]
      * { Renaming to define Mother-SET same as Father . }
  protecting(BINARY-TREE[ ELT <= view to PARENTS
       { sort Elt -> Elt , op _<_ -> _<_ }])
  signature {
    [ Parents < Elt.PARENTS ]
    op nilName : -> Elt.PARENTS
    op parents : Elt.FATHER Elt.FATHER Elt.MOTHER -> Parents
    op parents : Elt.MOTHER Elt.FATHER Elt.MOTHER -> Parents
    op MKparents : FatherTree MotherTree -> BinaryTree
    op MKparentsSeq : FatherTree MotherTree -> Seq
    op getMfi : Elt.FATHER MotherTree -> Elt.MOTHER
    op getMfa : Elt.FATHER MotherTree -> Elt.MOTHER
  }
  axioms {
    vars Ffa Ffi : Elt.FATHER ; vars Mfa Mfi : Elt.MOTHER
    vars Fr Fl Ftree : FatherTree ; vars Mr Ml Mtree : MotherTree
    eq MKparents(Ftree , Mtree) =
         makeBinTree(MKparentsSeq(Ftree , Mtree)) .
    eq MKparentsSeq(nilFather,nilMother) = nilSeq .
    eq MKparentsSeq(nilFather,binMTree(Mname(Mfa,Mfi),Ml,Mr)) =
         parents(Mfa, nilName, Mfi)
             ( MKparentsSeq(nilFather, Ml) $
               MKparentsSeq(nilFather, Mr)) .
    eq MKparentsSeq(binFTree(Fname(Ffa,Ffi),Fl,Fr),nilMother) =
         parents(Ffa, Ffi, nilName)
             ( MKparentsSeq(Fl, nilMother) $
               MKparentsSeq(Fr, nilMother)) .
    eq MKparentsSeq(binFTree(Fname(Ffa,Ffi), Fl, Fr),
                    binMTree(Mname(Mfa,Mfi), Ml, Mr)) =
         parents(Ffa, Ffi, getMfi(Fname(Ffa, Ffi),Mtree))
             ( MKparentsSeq(Fl, getTree(Fname(Ffa,Ffi),
                 delete Mname(Ffa,getMfi(Fname(Ffa,Ffi),Mtree))
                   from Mtree))
             $ MKparentsSeq(Fr, getTree(Mname(Ffa,Ffi),
                 delete Mname(Ffa,getMfi(Fname(Ffa,Ffi),Mtree))
                   from Mtree))) .
```

```
        ... equations
      }
    }
```

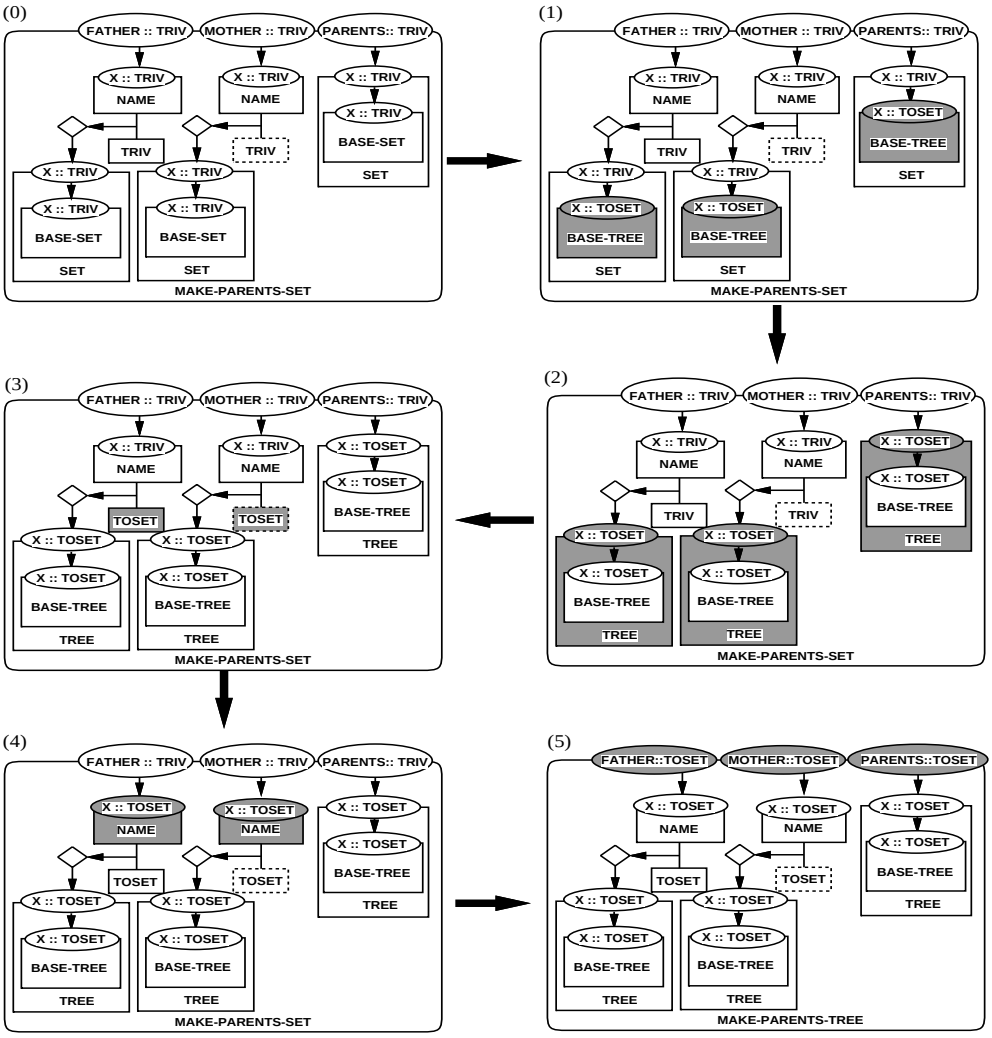


図 5.13: 二つのグループから別のグループを作り出すプログラムのモジュールグラフ

5.4 検索データベース

氏名で検索を行なうデータベースの定義を行なう。このデータベースには住所、氏名、電話番号が格納されていることとする。

5.4.1 仕様とモジュールグラフによる表示

データベースを関係データベースとして定義する。住所、氏名、電話番号を各々インデックス付きデータの集合として定義し、インデックスによる参照が可能となるように仕様の構築を行なう。

第一に定義するのはインデックス付きのデータ要素である。集合の定義は前出の BASE-SET 及び SET を用いる。集合で扱う要素をインデックス付きデータとするための定義をおこなう。

```
module BASE-ELT {
  [ Elt ]
  op nil : -> Elt {constr}
}
module INDEXed-DATA [ INDEX :: TRIV , VAL :: BASE-ELT ] {
  [ Data ]
  op _[_] : Elt.VAL Elt.INDEX -> Data
}
```

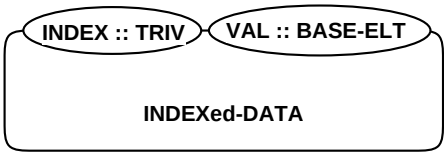


図 5.14: インデックス付きデータのモジュールグラフ

BASE-ELT は、要素に対して制約を与えるセオリ·モジュールである。ここで要求しているのはソートと定数項が一つずつ定義されていることである。この制約を架されるのは値の集合となる仮パラメータ VAL である。再帰的な検索の際に終了条件を示す基底項が必要となるための制約である。

住所のデータ要素を定義する。集合要素の値にはパラメータ付きモジュール INDEXed-DATA

で用いられている制約 BASE-ELT が求める基底項の存在が必要となる。住所は文字列により記述されることに決め、CafeOBJ の組み込みモジュール CHAOS 内で定義されている文字列 Identifier をアリティとする演算定義を行なった。インデックスは依然パラメータとしておく。ここで定義した住所を示す演算にインデックスを付加したデータが住所集合の要素となる。

```
module ADDRESS-BASE {
  protecting(CHAOS)
  [ Identifier , Address ]
  op nil : -> Address
  op address : Identifier -> Address
}
module ADDRESS [ INDEX :: TRIV ] {
  protecting(INDEXed-DATA
             [ INDEX <= view to INDEX { sort Elt -> Elt },
               VAL   <= view to ADDRESS-BASE
                 { sort Elt -> Address , op nil -> nil }]
             * { sort Data -> AddressKey })
}
```

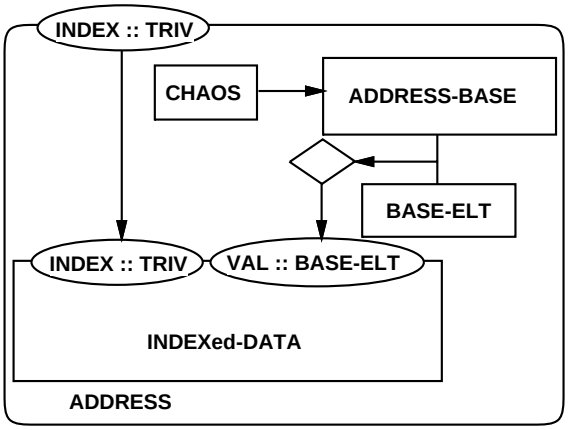


図 5.15: 住所のデータ要素のモジュールグラフ

ADDRESS-BASE で住所となる要素に用いる演算の定義を行なった。モジュール ADDRESS の定義により、ADDRESS-BASE の要素にインデックスを付加することでデータ集合の内部構造として利用可能としている。また、ここで利用している INDEXed-DATA モジュールは名

前替えを行なう参照形態をとるため、モジュールグラフ内の他所での出現の有無に関わらずここでの参照は実線で記入する。

住所のデータ要素と同様、名前の定義にも基底項を nil とする。名前も文字列で表現されるため CHAOS 内で定義されている文字列 Identifier をアリティとする演算定義を行なった。演算 name へは Identifier の項による姓と名を渡す。ここで定義した演算 name にインデックスを付加したデータが名前を格納する集合の要素となる。

```
module NAME-BASE {
  protecting(CHAOS)
  [ Name ]
  op nil : -> Name
  op name : Identifier Identifier -> Name
}
module NAME [ INDEX :: TRIV ] {
  protecting(INDEXed-DATA
             [ INDEX <= view to INDEX { sort Elt -> Elt } ,
               VAL   <= view to NAME-BASE
                 { sort Elt -> Name , op nil -> nil }]
             * { sort Data -> NameKey })
}
```

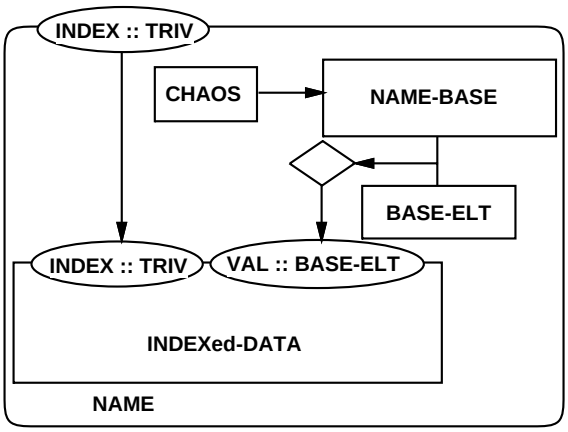


図 5.16: 名前のデータ要素のモジュールグラフ

図 5.15と図 5.16とを比較して明らかなように、これらのモジュールは輸入を行なうモジュールやパラメータの構造が等しいものとなっている。

電話番号は整数の 3 引数により市街局番を含む実現を行なうのが現実的であるが、市街局番の先頭に 0 が来る可能性を安直に解決するため、文字列を用いた定義を行なった。基底項のデータ定義を行ない、演算は Identifier を引数にとる定義とした。tel(03-1234-5678) という形式の記述となる。ここで定義した演算にインデックスを付加したものが電話番号を格納する集合の要素となる。

```
module TEL-BASE {
  protecting (CHAOS)
  [ Identifier , Tel ]
  op nil : -> Tel
  op tel : Identifier -> Tel
}
module TEL [ INDEX :: TRIV ] {
  protecting(INDEXed-DATA
             [ INDEX <= view to INT { sort Elt -> Int },
               VAL   <= view to TEL-BASE
                 { sort Elt -> Tel , op nil -> nil }]
             * { sort Data -> TelKey })
}
```

関係データベースの定義を行なう。後の拡張生を考慮して、実態を格納するデータ集合の他にインデックスのデータ集合を用意する。データの検索効率が良いだけでなく、今回用いなかった性別や年齢といった項目の追加時に各データの定義とインデックス集合の格納場所を付加するのみの変更で終了し、拡張性があると考えるからである。インデックス集合の要素の定義を行なう。この要素は住所、氏名、電話番号の順に INDEX の実パラメータと等しいモジュールの束縛を行なう定義とする。

```
module INDEX-TABLE [ ADD-I NAM-I TEL-I :: TRIV ] {
  [ Table ]
  -- [ Address Index ] [ Name Index ] [ Tel Index ]
  op [_][_][_] : Elt.ADD-I Elt.NAM-I Elt.TEL-I -> Table
}
```

以上で定義を行なった各集合の要素を用いてデータ検索の仕様定義を行なう。様々なデータ集合を扱うことが可能となるようにパラメータ付きの集合を各要素で特定化し、名前替えしながら輸入する。この特定化により住所の集合、名前の集合、電話番号の集合、及びインデックスデータの集合を実現したことになる。図 5.18に示す。集合のモジュールなどの

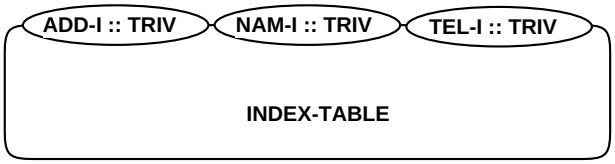


図 5.17: インデックスモジュールのグラフ

名前替えによる参照を実線で描いて強調している。インデックスモジュールの仮パラメータと各データ要素のソートとの関連付けを読みとるにもモジュールグラフが有利であることが分かる。

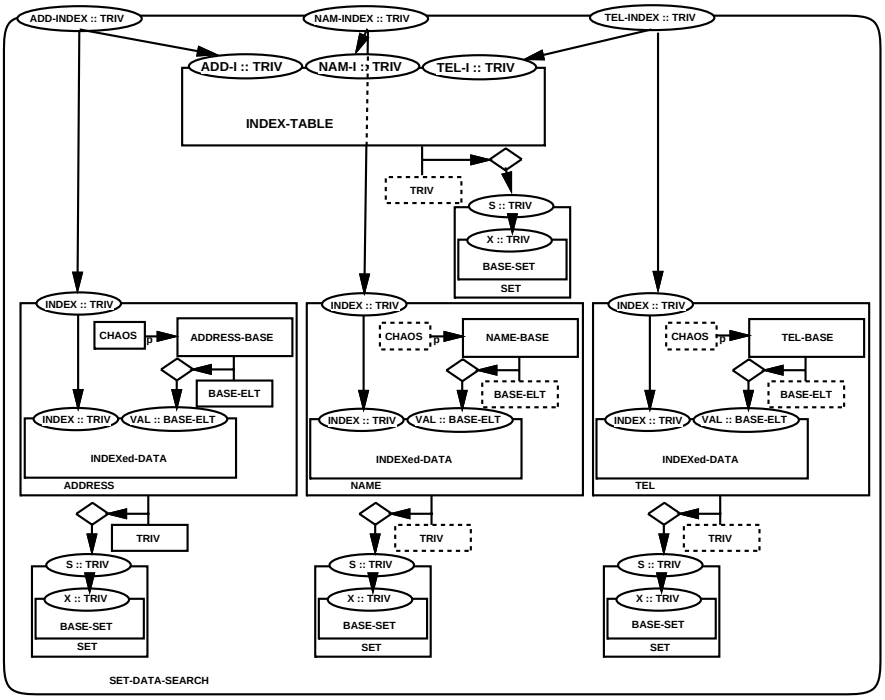


図 5.18: 集合による関係データベース検索モジュールのグラフ表現

```
module SET-DATA-SEARCH [ ADD-I NAM-I TEL-I :: TRIV ] {
  protecting(SET[X <= view to ADDRESS
    [INDEX <= view to ADD-I{..}] { sort Elt -> AddressKey } ]
      * { sort Set -> AddressSet , sort SetSeq -> AddressSeq })
  protecting(SET[ X <= view to NAME [INDEX <= view to NAM-I{...}]
    { map and renaming same as ADDRESS }])
  protecting(SET[ X <= view to TEL [INDEX <= view to TEL-I {...}]
    { map and renaming same as ADDRESS }])
  protecting(SET [ X <= view to INDEX-TABLE
    [ ADD-I <= view to ADD-I { sort Elt -> Elt }, ... ]
    { map and renaming same as ADDRESS }])
  [ OneData BaseData ]
  op [_,_,_,_] : AddressSet NameSet TelSet TableSet -> BaseData
  op Address=_Name=_Tel=_ : Address Name Tel -> OneData
  op searchByName : BaseData Name -> OneData
  op getAddressIndex : TableSet Int -> Int
  op searchAddress : AddressSet Int -> Address
  op getTelIndex : TableSet Int -> Int
  op searchTel : TelSet Int -> Tel
  vars I I1 I2 I3 : Int
  var A : Address , var AST : AddressSet
  var ASQ : AddressSeq
  vars Nb SN : Name , var NSQ : NameSeq
  var T : Tel , var TST : TelSet , var TSQ : TelSeq
  var TaST : TableSet , var TaSQ : TableSeq
  eq searchByName([ AST,{(Nb [ I ]),NSQ },TST,TaST ],SN ) =
      if Nb == SN then
            Address= searchAddress(AST,getAddressIndex(TaST,I))
            Name= Nb Tel= searchTel(TST,getTelIndex(TaST,I))
         else searchByName([ AST,{ NSQ },TST,TaST ],SN ) fi .
  eq getAddressIndex({ [ I1 ][ I2 ][ I3 ] , TaSQ },I) =
      if I2 == I then I1 else getAddressIndex({ TaSQ } , I) fi .
  eq searchAddress({(A [ I ]) , ASQ },I1) =
      if I == I1 then A  else searchAddress({ ASQ },I1) fi .
  eq getTelIndex({ [ I1 ][ I2 ][ I3 ] , TaSQ },I) =
      if I2 == I then I3 else getTelIndex({ TaSQ } , I) fi .
  eq searchTel({(T [ I ]) , TSQ },I3) =
      if I == I3 then T  else searchTel({ TSQ },I3) fi .
}
```

定義したモジュールSET-DATA-SEARCHのインデックスに整数を利用した特定例を作り出すことにより、CafeOBJ 上で実行による仕様の正当性の検証が可能となる。この検証のための特定例をSET-DATA-SEARCH-INTとして定義し、モジュールグラフ化した (図5.19)。

集合を用いて定義した検索データベースのモジュールグラフからデータ構造を一般の木

に挿替えることを考える。図5.19に示した挿替え前後のモジュール構造から一般の木への変更を行なう。集合を用いた関係データベースの定義では保守作業の効率化を考えインデックスを格納する集合を定義した。しかし、一般の木の各データには、前後の関係を示すインデックスの格納が求められる。例えば、住所、名前、電話番号の順番で格納される木の住所データには後ろに続く名前へのインデックスが付き、名前データには住所データと電話番号データへのインデックスが付くことが現実的な設計となると考えられるのである。このため、図5.19で提案したモジュールの変更は実際のモジュール構築に用いることができない。そこで、モジュールグラフからINDEX-TABLEを削除した上でコード化を行なう。

モジュールグラフを用いて仕様に対する考察を加えた上で仕様の改良に着手することが可能となることもグラフによる可読性の向上の現れであると考えることができる。

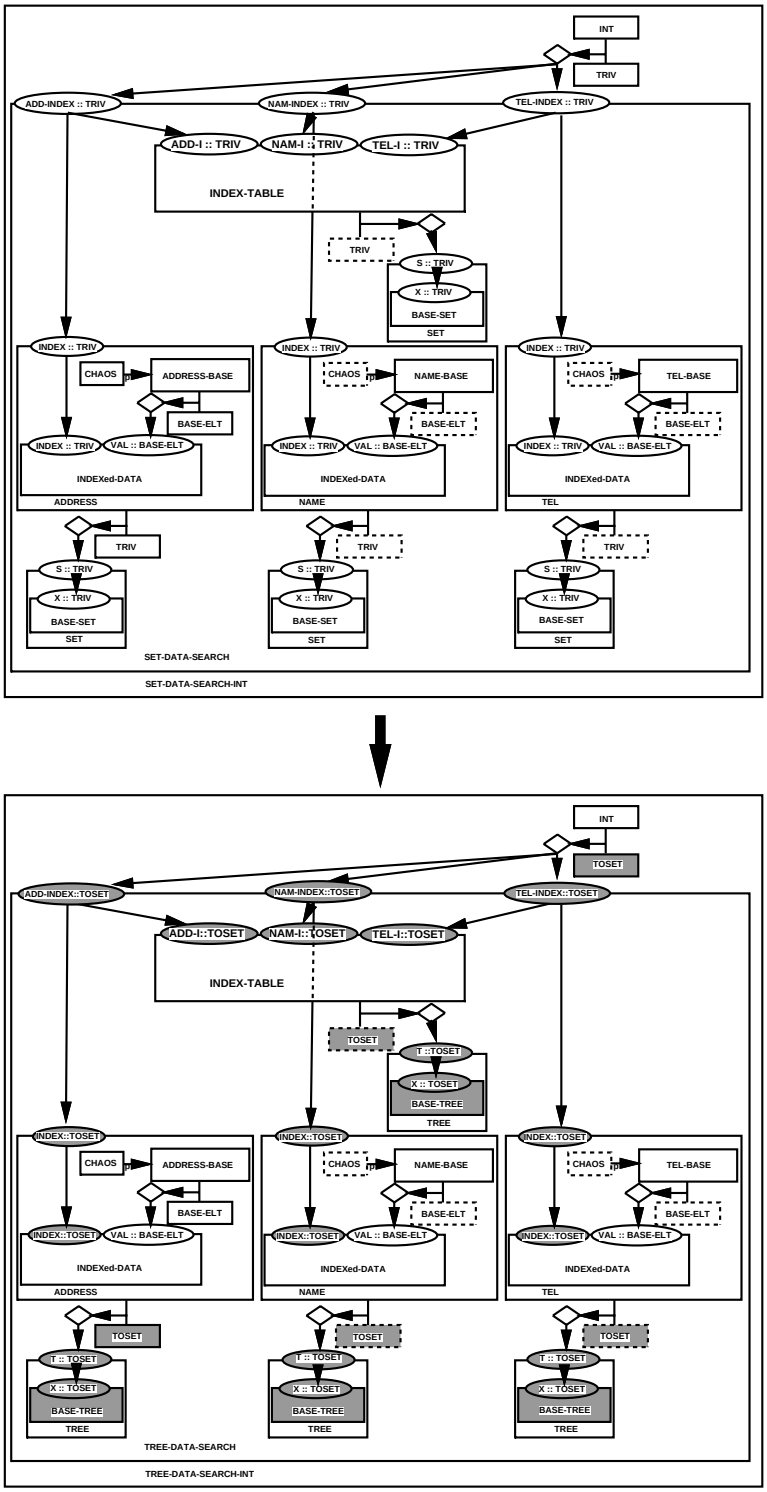


図 5.19: 検索データベースのデータ構造挿替え

第 6 章

考察

本研究ではモジュールグラフの定義に関する研究を行なってきた。本稿で紹介したグラフ定義に行きつくまでには様々な工夫を重ねている。そこで、この章では最初にモジュールグラフが今回紹介した形式に落ち着くまでの変遷を説明する。

6.1 グラフ定義の変遷

初期のモジュールグラフはモジュールを楕円で記述していた。継承関係を示す破線の矢印には輸入であることを表現するために importing の文字を添えた。制約モジュールを記述してパラメータ付きモジュール本体の楕円に向けた矢印結ぶことでパラメータ付きモジュールと定義した。特定化にはビュー宣言を行なっているモジュール内には角の丸い四角形に囲まれたビュー名を書いて、実際にビューを利用している場所には実パラメータと制約モジュールを矢印で結び、間にビューを記述した。以下に記載する並べ替えモジュールをグラフ化したものを図 6.1 に示す。初期のグラフでは特定化を行なう際の実モジュールの束縛の仕方が表現上曖昧になっている。特に多重の特定化のグラフ表現は可読性の優れたものとは言い難い。図 6.1を説明する。TRIV の制約の付いた SEQ を輸入して TOSET の制約の付いた SORTING が定義してある。SORTING 内で定義されたビューによりこれらのモジュールは仮パラメータ同士が特定化されている。すなわちこれが多重のパラメータ化を表現している。INT-SORTING はモジュール内で INT から SORTING の制約 TOSET へのビューを宣言している。これで SORTING の INT による特定化が行なわれ、このモジュールを輸入している INT-SORTING モジュールの定義が終了した。となるわけである。

```
module SEQ [ X :: TRIV ]{
  ... Sequence
}

module SORTING [ ELT :: TOSET ]{
  protecting(SEQ [ X <= view to ELT { sort Elt -> Elt }])
  ... Sorting Elt sorted elements
}

view VI1 from TOSET to INT {
  sort Elt -> Int,
  op _<_ -> _<_
}

make INT-SORTING is SORTING [ ELT <= VI1 ] endm
```

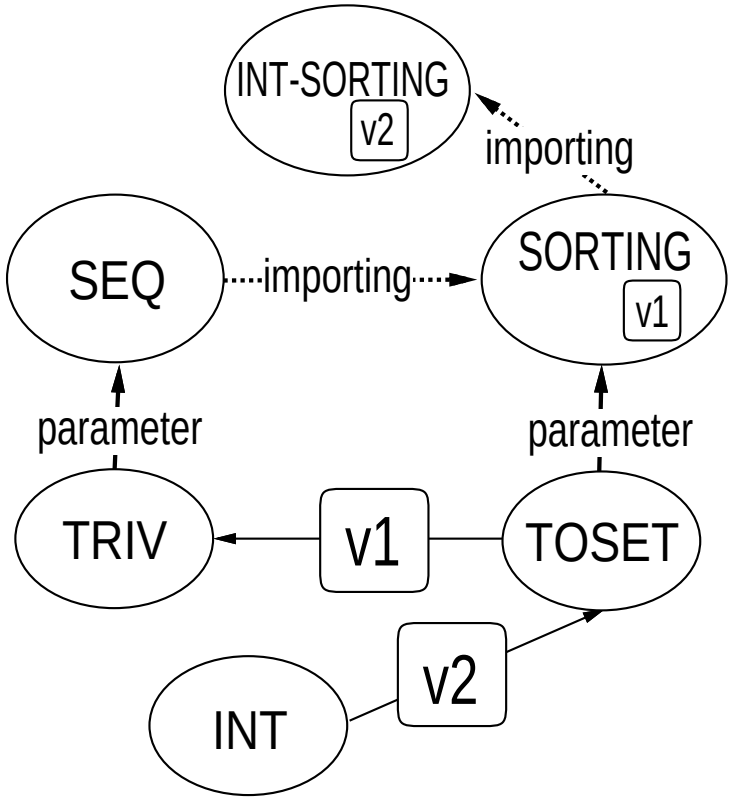


図 6.1: 初期のグラフ

初期のグラフ定義では曖昧な部分が多かった。先ずモジュールの記述回数についてである。制約のモジュールはオブジェクトモジュールの定義と同様、楕円で記述するが、これでは等しいモジュールが幾度となく出現することになり曖昧である。また、ビュー宣言に関するグラフ表現も正確ではなかった。ビューは制約からターゲットモジュールへのマップであるという表現としてはここでの定義で十分である。しかし言語の定義では、ビューをモジュール定義の外で個別に定義可能となっている。このグラフ定義ではビューの個別定義を表現できない。名無しのビューに対して適当な名前付けを行なう点も不自然であった。特定化と特定例の違いの表現も不可能であるし、破線の多用も作図の不便さを増大させている。

現在のモジュールグラフはこれら曖昧性の排除について研究した結果として定義したものである。モジュールの出現回数については名前替えを用いている際には別モジュールとみることが可能であるため、回数制限を設けてはいない。しかし通常のモジュールの出現回数は確実に 1 回と決めた。ビュー宣言は個別に定義が可能とした。ビューの形状を菱形に決めモジュール宣言の形状と明確に区別できるようにしたことから曖昧性の排除にもつながった。名無しのビューは菱形の内部を無記名とすることで宣言に忠実なグラフ化が行なえるようになった。また、特定化はビューとパラメータ部のグラフ上での結合であると決め、特定例は特定化したモジュール群を内部に持つモジュール定義であると明確に区別した。モジュールの形状を確定前後で変化させることも曖昧性の排除に貢献する。破線を使用しないグラフ記述のために、輸入やパラメータに用いる矢印の種類を統一した。パラメータ部に制約モジュール名のみでなく仮パラメータ名を記述すると決めたことからコードとグラフとの対応関係を掴みやすくなったことも可読性の向上に貢献している。

6.2 グラフ化の有効性

モジュールグラフの定義には以下の性質の向上を念頭においてきた。モジュールグラフの利用により各性質へ与えた影響について考察する。

- プログラムの可読性
- プログラムの保守性
- 各モジュールの独立性

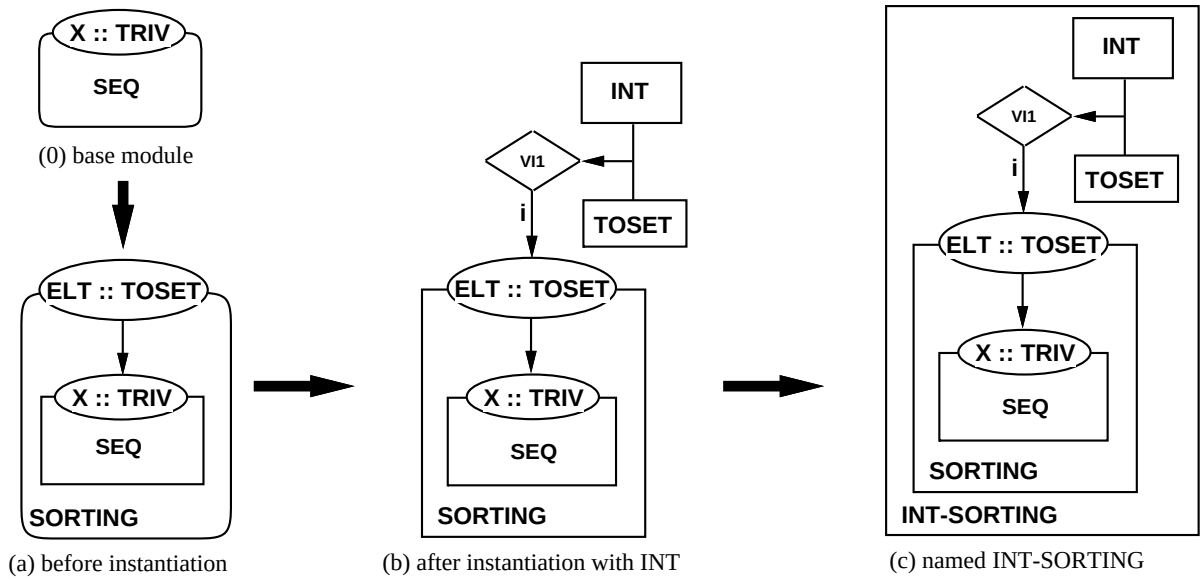


図 6.2: 現在のグラフ

- プログラムの安全性
- プログラムの拡張性
- モジュールの再利用性

6.2.1 可読性

今回定義を行なったプログラムとして最も複雑なモジュール構造を持っているのが検索データベースを定義したSET-DATA-SEARCHである。このモジュール定義を行なった節に示したように、プログラムのコードとしてはSET-DATA-SEARCHモジュールの他にINDEXed-DATAをはじめ 10 個のモジュールが用いられている。さらに名前替えを行なっているため実質的な参照モジュール数はこれより多いものとなる。モジュールグラフを利用しない場合、これらのモジュールの参照関係の把握を行なうには多くの時間を要することとなる。

例としてSET-DATA-SEARCH-INTモジュールから全体のモジュール理解を行なう作業の大まかな流れを見ていく。図6.3に示すのがモジュール理解の流れである。この図の上の階層からモジュール理解は始まる。(1) では、SET-DATA-SEARCH-INTモジュールの内部を読み、ここで特定化されているモジュールがSET-DATA-SEARCHであることを知る。モジュー

ル SET-DATA-SEARCH 内を読み、SET モジュールへ ADDRESS、NAME、TEL、INDEX-TABLE 各モジュールが多重のパラメータ化と名前替えを伴って利用されていることを理解するのが (2) である。各モジュール内からそのモジュールの内部要素を知るのが (3) であるが、この階層ではデータ要素の 4 種類のモジュールと集合の基礎の定義を行なったモジュールの計 5 つのモジュール内を参照することになる。この様に内部参照を繰り返し行なうことからモジュールの関連性を理解することが可能である。しかし多重の特定化や、名前替えの及ぶ範囲などの理解には更に時間を要する。

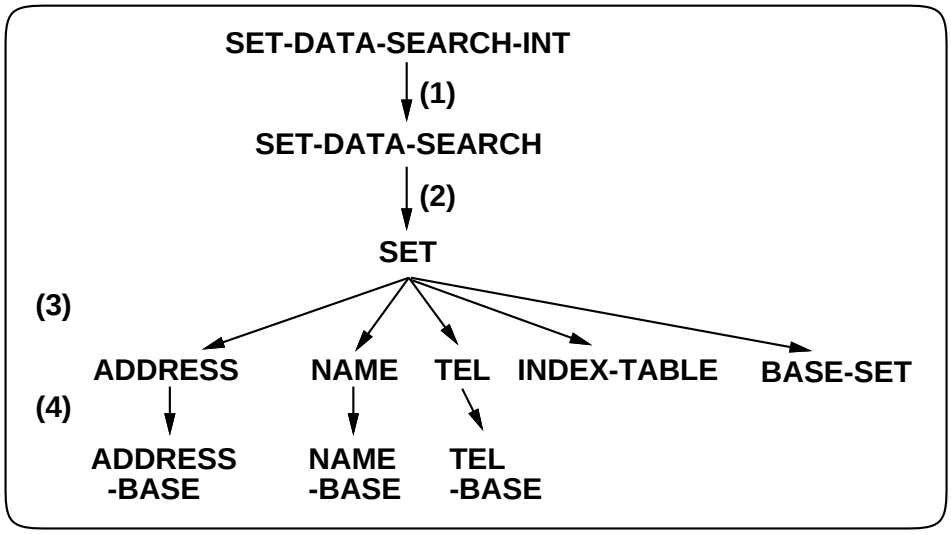


図 6.3: モジュール理解の流れ

また 3 重以上の多重の特定化が行なわれるような場合には、多重のパラメータ化構造や特定化における実モジュールの束縛関係を理解するためにコードを追うことは非常に困難である。この様に一目でモジュールの参照関係や多重の特定化の結合関係を把握することが可能となる点をみてもモジュールグラフによるプログラムの可読性の向上を言うことができる。

モジュールグラフからプログラムの動作を読みとるという意味での可読性について考える。モジュール定義のヘッダ部分に記述されている情報からモジュールの動作を読みとることは不可能であるため、この意味での可読性はないことが分かる。モジュールグラフはモジュール定義の際に記述するヘッダ部分を用いて構築するという定義に基づいているため、これは自明である。モジュール名がそのモジュールの意味を代表した部分であるが、演

算定義などを考慮せずにモジュールの動作を理解することは不可能なのである。

6.2.2 保守性

モジュールの可読性が向上したことにより、プログラムの保守性が向上したと言うことができる。プログラムを改良、再構築する際には、プログラムに用いられているモジュール群の関連性の理解を行なった後に改良部分の特定を行なう動作が伴うからである。今回例題として行なったモジュールの構築、再構築の操作をここで振り返る。データ構造の挿替えにより生まれる変更必要箇所をモジュールグラフ上で特定することからこの作業を始めることが可能となったのは、モジュールグラフによるプログラムの可読性により生じた保守性の現れであると考えることができるからである。

6.2.3 安全性と拡張性

不確定モジュールは角の丸い四角形で表現するという形状的な変化を用いていることにより、実行可能なモジュール (確定モジュール) とそうでないモジュールとをグラフ上で明確に区別することが可能となっている。この視覚的な区別により確定モジュールを独立な単位モジュールと考えることが可能となる。この単位モジュールは、他のプログラム構築の際にブロックとして再利用することが可能となるのである。また個々の独立したモジュールを組み合わせてプログラムを形成することはシステムの安全かつ容易な拡張を可能にする。モジュールグラフ上で確定モジュールの結合状況を視覚的に確認することが可能となることから、コードで記述されたプログラムを示すのに比較してこの安全性と拡張性の証明を行なうことが容易であることも言える。

6.2.4 再利用性

コードとしてプログラムを眺めている場合には、そこで輸入される整数や文字列のモジュールのようなモジュールの再利用を考えることは容易である。同様に特定化前のパラメータ付きモジュール単体での再利用も容易に思い浮かぶ。しかし、多重にパラメータ化されたモジュールはモジュール構造が難解になるほど特定化後の確定モジュールのイメージが困難になるため再利用性を想像することは難しい。モジュールグラフはこの様に複雑なモジュール構造を持ったモジュール群のまとまった形での再利用を支援する。今までモ

ジュールプログラミングの利点として提唱されてきた再利用性の意味をモジュールグラフによりさらに高めることが可能となるのである。

第 7 章

今後の課題

7.1 等式の変換

モジュールグラフはモジュール定義のヘッダ部分を解析した結果の図示であるため、データ構造の挿替えに伴う等式の変更をグラフから示唆することは不可能であった。モジュール構造を示すことが今回の最大の目的であったことからもこの必要性は表面化していない。事実、モジュールグラフ上でのデータ構造の挿替えではモジュール構造の変更を示すにとどまり、等式の変更はプログラマに任せる結果となっている。今後の課題としてデータ構造の挿替えに伴う等式の変換について形式的手法の開発が望まれる。

7.2 グラフの利用

モジュール構造の可読性を考慮してモジュールグラフを利用してきたため、今回の研究では大きなシステムへの利用例の表示を避けた。現実にはデータ構造を用いて定義したパラメータ付きモジュールの特定例が他のモジュールで参照されるような場面もある。この様なモジュールの関係を図示する際に複雑な内部構造をグラフ上で示していると、紙面的な問題やグラフの誤記載、可読性損失といった危険性の増加につながる。これを踏まえ、今後のモジュールグラフの利用形態として次のことを提案する。内部構造の複雑なモジュールは別の部分にグラフを記載しておき、グラフ上ではモジュール名を括弧で括った通常モジュールとするのである。例えば、グラフ上に [MAKE-PARENTS-SET] なるモジュールの記載があった場合には他の場所 (ページ) にモジュール MAKE-PARENTS-SET の詳細を定義した

モジュールグラフを見ることができるという具合にするのである。この様なグラフ上の詳細化の関係を用いることで紙面からグラフの整理が可能となり、先ほど挙げた危険性の増加を抑える効果を生むと考えられる。

第 8 章

謝辞

本研究を行なうにあたり、指導して頂いた二木厚吉教授に深く感謝致します。また、有益な助言をして頂いた渡部卓雄助教授、緒方和博助手、菅原太郎氏、天野憲樹氏、飯田周作氏、海野 浩氏、田中 哲氏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頂いた言語設計学講座の諸氏に感謝致します。

参考文献

[1] Futatsugi, K., Okada, K., A Hierarchical Structuring Method for Functional Software Systems, *Proc. 6th International Conference on Software Engineering*, IEEE Computer Society, pp393–402, September 1982.

[2] Futatsugi, K., Goguen, J., Meseguer, J., Okada, K., Parameterized Programming in OBJ2, *Proc. 9th International Conference on Software Engineering*, (Monterey, California), IEEE Computer Society, pp51–60, March 1987.

[3] Futatsugi, K., Goguen, J., Meseguer, J., Okada, K., Parameterized Programming and its Application to Rapid Prototyping in OBJ2, *Japanese Perspectives in Software Engineering*, edited by Yoshihiro Matsumoto and Yutaka Ohno, Addison-Wesley, pp77–102, 1989.

[4] Futatsugi, K., Structuring and Derivation in Algebraic Specification / Programming Language Systems, *Journal of Information Processing*, Vol. 14, No. 2, pp153–163, 1991.

[5] Goguen, J., Parsaye-Ghomi, K., Algebraic Denotational Semantics using Parameterized Abstract Modules, *Proc., International Conference on Formalizing Programming Concepts*(Peniscola,Spain) edited by J. Diaz and I. Ramos, Springer, Lecture Notes in Computer Science, Volume 107, pp292–309, 1981.

[6] Goguen, J., Meseguer, J., Rapid Prototyping in the OBJ Executable Specification Language, *Proc., Rapid Prototyping Workshop*(Columbia, Maryland)1982. Also in *Software Engineering Notes*,ACM Special Interest Group on Sofware Engineering, Volume7, Number5, 1982, pp75–84.

[7] Goguen, J., Parameterized Programming, *IEEE Transactions on Software Engineering*, Volume SE-10, Number 5, pp528-543, September 1984. preliminary version in *Proceedings, Workshop on Reusability in Programming*, ITT, pp138–150, 1983.

[8] Goguen, J., Principls of Parameterized Programming, *Software Reusability, Volume I: Concepts and Models*, edited by Ted Biggerstaff and Alan Perlis, Addison-Wesley(ACM, Frontier Series), pp159–225, 1989.

[9] Goguen, J., Winkler, T., Messeguer, J., Futatsugi, K., Jouannaud, J., P., Introducing OBJ, Technical Report, SRI–CSL–92–03,Computer Science Laboratory, SRI International, 1992. To appear In J.A.Goguen editor Application of Algebraic Specification using OBJ, Cambridge University Press.

[10] Goguen, J., Malcom, G., Algebraic Semantics of Imperative Programs, MIT Press, Cambridge, Massachusetts, 1996.

[11] Goguen, J., Theorem Proving and Algebra, MIT Press, Cambridge, Massachusetts, 1996.

[12] Diaconescu, R., Futatsugi, K., Logical Semantics for CafeOBJ, JAIST Research Report, IS-RR-96-0024S, 1996.

[13] 石畑清, アルゴリズムとデータ構造, 岩波講座, ソフトウェア科学, No.3, 岩波書店, 1989.

[14] 稲垣康善, 坂部俊樹, 抽象データタイプの代数的仕様記述法の基礎 (1), 多ソート代数と等式論理, 情報処理, Vol.25, No.1, Jan. 1984.

[15] 稲垣康善, 坂部俊樹, 抽象データタイプの代数的仕様記述法の基礎 (2), 抽象データタイプ, 情報処理, Vol.25, No.5, May 1984.

[16] 稲垣康善, 坂部俊樹, 抽象データタイプの代数的仕様記述法の基礎 (3), 抽象型構成子のモデルと始代数意味論, 情報処理, Vol.25, No.7, July 1984.

[17] 稲垣康善, 坂部俊樹, 抽象データタイプの代数的仕様記述法の基礎 (4), 終代数意味論に基づく抽象型構成子の仕様記述、実現ならびにその検証, 情報処理, Vol.25, No.9, Sep. 1984.

[18] 岡田康治, 二木厚吉, プロトタイピング技法の形式化の試み, 情報処理学会「プロトタイピングと要求定義」シンポジウム, 1986.

[19] 中川 中, 谷津弘一, 本間毅寛, CafeOBJ への誘い, Technical report, IPA, Internet publication http://www.ipa.go.jp/STC/CafeP/invit-cafe-jp.html, 1996.

[20] 中川 中, 谷津弘一, 本間毅寛, 形式手法への誘い –代数仕様とその周辺–, Technical report, IPA, Internet publication
http://www.ipa.go.jp/STC/CafeP/invit-formal-jp.html, 1996.

[21] 中川 中, 抽象データ型、形式仕様、代数仕様と OBJ, Technical report, IPA, Internet publication http://www.ipa.go.jp/STC/CafeP/intro2-jp.html, 1996.

[22] エイホ, A.V., ホップクロフト, J.E., ウルマン, J.D., (大野義夫 訳), データ構造とアルゴリズム, 情報処理シリーズ 11, 培風館, 1987.

[23] セジウィック, R., (野下, 星, 佐藤, 田口 訳), アルゴリズム, 第 3 巻＝グラフ・数理・トピックス, 近代科学社, 1993.

[24] http://ldl-www.jaist.ac.jp:8080/cafeobj/, CafeOBJ user's manual.