JAIST Repository
https://dspace.jaist.ac.jp/

Title	代数仕様言語CafeOBJのための拡張可能な前処理系
Author(s)	浅羽，義之
Citation	
Issue Date	2003-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/1705
Rights	
Description	Supervisor:二木 厚吉，情報科学研究科，修士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

修 士 論 文

代数仕様言語CafeOBJのための拡張可能な前処理系の設計と実装

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

浅羽義之

2003年3月

修 士 論 文

代数仕様言語CafeOBJのための拡張可能な前処理系の設計と実装

指導教官　二木 厚吉教授

審査委員主査　二木厚吉 教授
審査委員　片山卓也 教授
審査委員　権藤克彦 助教授

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

110002 浅羽義之

提出年月: 2003 年 2 月

Copyright © 2003 by Yoshiyuki Asaba

概 要

本研究の目的は，代数仕様言語 CafeOBJ の構文の拡張を可能にし，仕様の記述を簡潔に行うことを支援することである．そのために，構文を拡張するための機構を提案する．

CafeOBJ は代数モデルにより形式的にソフトウェアの仕様を記述するための言語である．CafeOBJ は，強力なモジュール機構，型付け，パターンマッチ機能などを備えており，高い記述力を持つ．しかし，対象とするソフトウェアの問題によっては，仕様の記述が繁雑になる．そこで，構文を拡張することで仕様を簡潔に記述することで，仕様の可読性と保守性の向上が期待できる．しかし，現在の CafeOBJ システムではこのような構文の拡張を実現するのは難しい．

本研究では構文を拡張するための機構を Espresso と名付ける．Espresso は拡張構文を利用した仕様を入力とし，CafeOBJ を出力とする前処理系として擬似的な構文の拡張を実現する．拡張構文とその変換ルールを独立したモジュールとして記述し，複数のモジュールを組み合わせることでインクリメンタルな拡張をする．また，拡張モジュールの記述言語として CafeOBJ を利用する．しかし，このような構文拡張の実現には，複数の拡張構文の衝突など解決しなければならない問題がある．そこで本研究では，拡張構文の衝突を回避するために，拡張モジュールの利用ポリシーを導入する．以上の設計方針をもとに Espresso を実装した．また，実際に作成した Espresso を使っていくつか CafeOBJ の構文拡張の例題を作成した．そして，その拡張構文を使って銀行システムの仕様を記述し，仕様のコード量が減ることで仕様の可読性と保守性が向上したのを確認した．

目次

第1章　はじめに **1**
- 1.1　本研究の背景 1
- 1.2　本研究の目的 2
- 1.3　本論文の構成 2

第2章　背景 **3**
- 2.1　代数仕様言語 CafeOBJ 3
  - 2.1.1　モジュールシステム 4
  - 2.1.2　項書換えシステム 7
- 2.2　前処理系 7

第3章　拡張可能な前処理系 Espresso **9**
- 3.1　Espresso の設計方針 9
  - 3.1.1　モジュールによる拡張 9
  - 3.1.2　拡張モジュールの衝突 10
  - 3.1.3　パッケージ概念の導入 11
  - 3.1.4　ポリシーの導入 12
- 3.2　Espresso のアーキテクチャ 14
  - 3.2.1　基本モジュール 14
  - 3.2.2　モジュール管理機構 15

第4章　Espresso の実装 **17**
- 4.1　モジュールの実装 17
  - 4.1.1　基本モジュール 18
  - 4.1.2　拡張モジュール 22
- 4.2　ポリシーファイル 26
- 4.3　モジュール管理機構の構成 30
- 4.4　Espresso の実装のまとめ 35

第5章　例題の作成 **37**
- 5.1　銀行システムの仕様の概要 37

5.1.1 振舞仕様 37
5.1.2 射影演算 38
5.2 例題１：複数の述語を定義する拡張構文 39
5.2.1 拡張モジュール PREDS の作成 41
5.3 例題２：射影演算の等式記述のための拡張構文 43
5.3.1 拡張モジュール PROJ の作成 43
5.3.2 拡張構文の衝突 49
5.4 考察 51
5.4.1 PREDS モジュール 51
5.4.2 PROJ モジュール 52
5.4.3 パフォーマンス測定 53

第6章 関連研究 54

6.1 代数仕様言語 54
6.1.1 Maude 54
6.1.2 Larch 54
6.2 前処理系 55
6.2.1 EPP 55
6.2.2 Mix Juice 55
6.2.3 AspectJ 55
6.3 関連研究との比較 56

第7章 結論 57

7.1 まとめ 57
7.2 今後の課題 57

付 録A Espresso の使用方法 62

A.1 ファイル構成 62
A.2 動作環境 62
A.3 インストール 63
A.4 使用方法 63
A.4.1 コマンドオプション 63
A.4.2 環境変数 64
A.5 サンプルの実行 64

第1章　はじめに

本論文では，代数仕様言語 CafeOBJ の構文の拡張を可能にし，仕様記述を支援することである．そのために，CafeOBJ の構文を拡張するための機構を提案する．本章では，本研究の背景と目的を述べ，最後に本論文の構成について述べる．

1.1　本研究の背景

現在，様々な場面においてコンピュータ化が進んでおり，社会の中におけるソフトウェア位置付けは非常に重要である．そのため，ソフトウェアの信頼性向上は重要な課題である．その解決法として形式的手法が注目されている．

形式的手法とは数学的な厳密さでソフトウェアの仕様を記述し，様々な性質について検証するための手法である．代数仕様言語 CafeOBJ は実行可能な仕様言語として注目を浴びている．CafeOBJ は代数モデルにより形式的にソフトウェアの仕様を記述するための言語である．CafeOBJ は，強力なモジュール機構，型付け，パターンマッチ機能などを備えており高い記述能力を持つが，その反面，仕様の記述が煩雑なる場合がある．そのような煩雑さが仕様の可読性や保守性に与える影響は無視できない．この問題を解決する方法として CafeOBJ の構文拡張が考えられる．CafeOBJ の構文を拡張することで，仕様記述の煩雑さを軽減することが期待できる．

CafeOBJ はモジュール単位でソフトウェアの仕様を記述する．CafeOBJ では項を自由にユーザが定義できるが，モジュールの定義など CafeOBJ のトップレベルには，ユーザが容易に構文を拡張できるような機構を提供しておらず，その拡張は困難である．したがって，ユーザが構文を拡張したい場合，CafeOBJ システムを直接修正することになる．しかし，ユーザが CafeOBJ の構文を拡張したいという要求も多く，その容易な拡張機構が望まれる．

このような拡張機構は CafeOBJ に限らず，通常のプログラミング言語においても必要になってくる．構文の拡張はアドホックに拡張せず，拡張部分を 1 つの独立したモジュールとして実現し，これらのモジュールを組み合わせることでインクリメンタルに構文拡張できることが望ましい．しかし，このような構文拡張の実現には，複数の構文拡張の衝突などを解決しなければならない問題がある．

1.2 本研究の目的

本研究の目的は，代数仕様言語 CafeOBJ の構文の拡張を可能にし，仕様の記述を簡潔に行えることを支援することである．そのために，本研究では CafeOBJ の構文を拡張するための前処理系 Espresso を設計・実装する．ユーザが自由に構文を拡張するためには，Espresso 自体を拡張可能な構成で実現される必要があるので，拡張構文とその変換ルールを 1 つのモジュールとして記述し，それを組み合わせることで拡張する．拡張モジュールには 1 つの拡張構文を定義するのではなく，複数の拡張構文を定義することで，モジュールの記述の利便性を考慮する．また，モジュールの利用ポリシーを記述することでモジュールを組み合わせる際に起こる問題を回避する．また，Espresso を使った例題を作成し，本研究で提案する Espresso による構文拡張の有用性を示す.

1.3 本論文の構成

本論文の構成は次の通りである．2 章では CafeOBJ と前処理系について説明する．特に，本研究で利用する CafeOBJ のモジュールシステムについて説明する．3 章では，Espresso の設計方針とアーキテクチャについて述べ，インクリメンタルに構文を拡張でき，かつ，モジュールの衝突を回避できる機構を提案する．4 章では，提案したアーキテクチャの実現方法について詳しく説明する．5 章では，実際に Espresso を利用した例題を紹介する．6 章で，本研究と関連する研究を紹介し，本研究との比較を行なう．最後に 7 章でまとめと今後の課題について述べる．また，Espresso の使用方法を説明する．

第2章　背景

本章では，本研究の背景となる代数仕様言語 CafeOBJ と前処理系について説明する．

2.1　代数仕様言語CafeOBJ

代数的仕様記述法 (algebraic specification technique) は，抽象データ型の仕様を厳密に記述する方法として 1970 年代半ばに登場した．抽象データ型の考え方は，ある特定の種類のデータを操作する関数を一箇所に集め，そのデータの表現形態を外部から遮断し，そのデータを操作する関数の呼出し形態のみを知らせるものである．この方法は一般にデータ隠蔽と呼ばれ，関数定義を含めたデータと関数を一体とした実体を通常データ型または抽象データ型と呼ぶ．代数仕様言語では抽象データ型の数学モデルを**代数モデル**[14] と呼ぶ．データを台集合としデータを操作する関数を台集合上の演算と捉えることで，数学における代数の基本構造と一致することがわかる．

代数は通常 1 つの台集合上の演算に着目するような代数であるが，抽象データ型では複数のデータ型はいくつかの種類 (ソート) から構成されるので，代数仕様の世界では**多ソート代数**に拡張している．さらに，多ソート代数にソートを拡張して半順序な包含関係を定義可能な代数を**順序ソート代数**と呼ぶ．代数仕様はシステムをデータ型，つまり順序ソート代数としてモデル化した形式仕様であると言える．また近年においては，**隠蔽代数**という新たなモデルが提案されている．隠蔽代数はシステムをブラックボックスとみなし，システムの内部の構造ではなく外部からみたシステムの振舞いを扱う．隠蔽代数により抽象データ型だけでなく抽象状態機械を代数仕様で記述することも可能である．

CafeOBJ[3] は，OBJ3 の流れを汲む実行可能な代数言語である．OBJ3 は等式論理と順序ソート代数を意味定義の基礎として持つ．これに対し CafeOBJ は，複数の論理の組合せを仕様の意味モデルとして持つことができる．CafeOBJ が仕様の意味モデルとして持つことができる論理は，多ソート代数，順序ソート代数，隠蔽代数，書換え論理である．これらの複数の論理を任意の組合せにより開発対象のシステムに応じた意味モデルの選択が可能となる．

```
mod! NATPLUS {
  [ Nat ]  -- ソートの定義
  op 0 : -> Nat .          -- 0 引数の関数. つまり定数.
  op s_ : Nat -> Nat .     -- 1 引数の関数
  op _+_ : Nat Nat -> Nat -- 2 引数の関数
  vars M N : Nat . -- 変数の宣言
  eq M + 0 = M .    -- 等式
  eq M + s N = s(M + N) .
}
```

図 2.1: 自然数の定義例

2.1.1 モジュールシステム

ソフトウェア工学におけるもっとも重要な考え方の 1 つはモジュール構造である．これは大規模なプログラムを作成したり理解するためには，それをある規模の大きさの単位 (モジュール) に分解することが不可欠である，という認識にもとづいている．また，モジュール化は部品化を促し，再利用やデバッグを容易にする．代数仕様言語 CafeOBJ の仕様はこのモジュールを大きな記述単位とする．CafeOBJ は高度に洗練されたモジュールシステムを備えている．以下では，CafeOBJ が持つモジュールシステムの特徴について説明する．

モジュールの意味論

モジュールの定義にはきつい意味 (initial semantics) にもとづくモジュールと緩い意味 (loose semantics) にもとづくモジュールと意味を明示的に指定しないモジュール定義がある．きつい意味とは仕様に対して対応する意味がユニークになる．緩い意味とはシグニチャで規定された等式を満たすような任意の代数の族を表す．厳密な定義については [14] を参照されたい．きつい意味にもとづくモジュールの定義は module!もしくは mod!でモジュールを定義する．緩い意味にもとづくモジュールの定義は module*もしくは mod*でモジュール定義する．意味を明示しない場合については module もしくは mod でモジュールを定義する．図 2.1 に自然数と自然数上の+演算を定義した仕様を示す．

モジュールの輸入

CafeOBJ にはモジュールを再利用したり，モジュールを拡張するためにモジュールの輸入が可能である．例えば，モジュール M がモジュール M' を輸入するとは，M' の定義

をMが参照可能になることを意味する．この輸入関係は推移律を満たす．すなわちM' が輸入しているすべてのモジュールはMによって輸入される．モジュールの輸入はオブジェクト指向言語の継承に似ているが，利用方法に応じて3種類の制限が可能である．

- protecting または pr
  輸入されるモジュールで宣言されたソートに新たな要素を付け加えたり (非冗長)，ソートの既存の要素を同定したりしてはならない (非混同)．
- extending または ex
  輸入されるモジュールで宣言されたソートの既存の要素を同定してはならない (非混同)．
- using または us
  制限無し．つまり，輸入される仕様のコピーをモジュールに直接コピーするものと考える．

protecting 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無い．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 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 では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ものと考えることができる．

パラメータ付きモジュール

パラメータ付きモジュールはモジュールを実パラメータにとり，特定のモジュールを作り出すことが可能なモジュールである．つまり，フレームワークとして汎用的なモジュールをパラメータ付きモジュールとして定義しておき具体的なモジュールを与えることにより，モジュールの再利用性が格段に増す．例えば，ある型のリストを作成したい時は，リストのフレームワークをパラメータ付きモジュールで定義しておき，リストの要素を定義したモジュールを実パラメータとして与えることで，様々な要素のリストを作り出すことが可能となる．図2.2にパラメータ付きリストの例を示す．

パラメータ付きモジュールは (X :: TRIV) の部分がパラメータ宣言になる．TRIV モジュールは内部にただ1つのソートが宣言されているだけの簡単なモジュールである．X は仮パラメータとなる．実パラメータはこのモジュールの制約を満たすモジュールになる．つまり，1つ以上のソート宣言されているモジュールなら実パラメータにすることが可能である．では，次にパラメータを割り当てる方法について図2.3示す．

```
mod* TRIV {
  [ Elt ]
}
mod! LIST(X :: TRIV) {
  [ Elt < List ]
  op nil : -> List .    -- nil
  op _::_ : Elt List -> List -- cons
}
```

図 2.2: パラメータ付きモジュール

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

図 2.3: 自然数のリストモジュールを定義するためのビュー

パラメータを割り当てるにはビューを作る必要がある．ビューとは仮パラメータと実パラメータの対応関係を与えるものである．上の例では，TRIV モジュールのソート Elt を INT モジュールのソート Int へ対応付けを行っている．仮パラメータに実パラメータを束縛し新たなモジュールを作り出す操作を図 2.4 に示す．仮パラメータに渡すのはモジュールではなくビューを渡す．

モジュール表現

CafeOBJ のモジュールシステムでは複数のモジュールを統合して新たなモジュールを構成することができる．これを**モジュールの和**と呼ぶ．また，モジュールのソート名や関数記号を変更することが可能である．これを**モジュールの積**または**名前替え**と呼ぶ．図 2.5 では前に定義した NATPLUS モジュールにおけるソート名を Nat から NaturalNumber に変更し，関数 s を succ に変更する．モジュールの和や積を適当に組み合わせることによ

```
make INTLIST (LIST(X <= IntListView))
```

図 2.4: パラメータの束縛

```
NATPLUS * { sort Nat -> NaturalNumber, op s -> succ }
```

図 2.5: モジュールの積

```
NATPLUS> red s 0 + s s 0 .
-- reduce in NATPLUS : s 0 + s (s 0)
s (s (s 0)) : Nat
(0.000 sec for parse, 3 rewrites(0.000 sec), 5 matches)
```

図 2.6: red コマンド

り 1 つのモジュールを多様な文脈で使うことが可能になる．

2.1.2 項書換えシステム

代数仕様言語 CafeOBJ の操作的意味は項書換えシステム [15] によって与えられている．等式を左辺から右辺への書換え規則と解釈することで，代数仕様の意味は仕様が定義する項書換えシステムとなる．項は仕様で定義されたシグニチャから構成される記号列である．項書換えシステムでは有限個の書換え規則ですべての計算が定められるが，計算能力はどんな計算可能な関数も計算できるだけの一般性があり，また自動推論システムとしても使える．

CafeOBJ では図 2.6 のように red コマンドで項の**正規形**を求める．項の正規形とはどの書換え規則を用いてもそれ以上は簡約できない項のことである．代数仕様における項の値とは項の正規形であると言える．つまり値が等しいということは項の形が等しいこととみなす．

項書換えシステムは，代数仕様言語に限らず関数型言語，等式にもとづく言語，等式論理の自動証明，数式処理，プログラムの変換などにも応用されている．

2.2 前処理系

前処理系 (プリプロセッサ) とはソフトウェア開発ツールの 1 つで，プログラムをコンパイルもしくは実行する前に前処理を行なうプログラムのこと．プログラムを処理するプログラムであるとも言える (図 2.7)．前処理系として有名なものとしては C 言語の前処理系が有名である．

前処理系が行なうことは主に以下の 3 点が挙げられる．

- マクロ処理

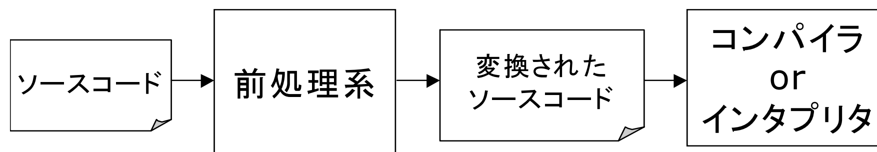

図 2.7: 前処理系

- ファイルの挿入
- 言語の拡張

マクロ処理とはマクロ定義や呼出しを含むテキストを入力とし，マクロ定義で指定されたパターンに照合する部分があれば，指定された方法に従って別の文字列に置き換えて出力をする．マクロ定義は，ふつう特別な文字または define や macro のようなキーワードを使って表示し，そのあとにマクロの名前と，定義を形成する本体を書く．たいていのマクロ定義では，仮引数すなわち値による置換を表わす記号が使用できる．マクロの使用は，マクロの名前と実引数からなる．マクロの使用が現われると，マクロ処理系はその本体中の仮引数を実引数で置き換え，マクロ使用自身を翻訳後の本体で置き換える．

ファイルの挿入は，それ指示した文のところに指定されたファイルを挿入する．C 言語の前処理系の場合だと include で指定したファイルを挿入する．

言語の拡張は古い言語に新しい制御の流れに関する機能やデータ構造化機能を加えることにより実現する．言語の拡張は主に組込みのマクロを用意し前処理系によってそれらが利用できるようにする．拡張の例としては C 言語を C++への拡張が有名である．初期の C++言語処理系は，C 言語のプリプロセッサとして実装されており，プリプロセッサが C++のソースコードを C 言語に変換していた．このように言語の拡張にも前処理系を使うことができる．また，最近では Java にアスペクト指向の概念を取り入れた AspectJ[8] や差分ベースのモジュール定義が可能な MixJuice[7] などもある．これらについては 6 章でもう一度説明する．

前処理系では，プログラムを実行もしくはコンパイル前に変換するため，元のプログラムと対応をとるのが難しいといった欠点や，デバッグがやりにくいという欠点がある．

第3章　拡張可能な前処理系Espresso

本章ではまず，拡張可能な前処理系Espresso[1]の概要について説明する．次にEspressoアーキテクチャについて説明する．

3.1　Espressoの設計方針

本研究ではCafeOBJの構文をインクリメンタルに拡張を可能とする機構を提案する．本研究では，以下のアプローチにもとづいてCafeOBJの構文拡張機構を実現する．

- 前処理系の導入によるCafeOBJの擬似的な構文拡張

言語の拡張方法には従来，前処理系による拡張方法や自己反映計算による拡張方法などがあるが，しかし，CafeOBJシステムは自由度の高い仕様の記述を許しているため，複雑な構文解析処理を行っている．そのため，現在のCafeOBJを本研究の目的に応じて再構成することは現実的ではない．本研究では，現在のCafeOBJシステムそのものの拡張ではなく，前処理系の導入によるCafeOBJの擬似的な構文拡張を目指す．具体的には，ユーザが前処理系に対してCafeOBJの構文拡張を定義し，前処理系が拡張構文をCafeOBJのコードに変換する．つまり，CafeOBJへの構文の追加は前処理系への構文の追加と等価になり，前処理系を拡張可能なアーキテクチャとして実現することになる．このような前処理系を実現するためのアプローチは以下である．また，本研究で提案する前処理系の概要図を図3.1に示す．

- 拡張構文とその構文をCafeOBJコードへ変換するためのルールのモジュール化
- 拡張モジュールをCafeOBJで記述
- 拡張モジュールの利用ポリシーの導入

3.1.1　モジュールによる拡張

モジュールとは複数の拡張構文とその変換ルールを記述した1つの独立したまとまり(モジュール)である．このようなモジュールを**拡張モジュール**と呼ぶ．また，拡張モジュー

[1] ExtenSible PREproceSSor for cafeObj

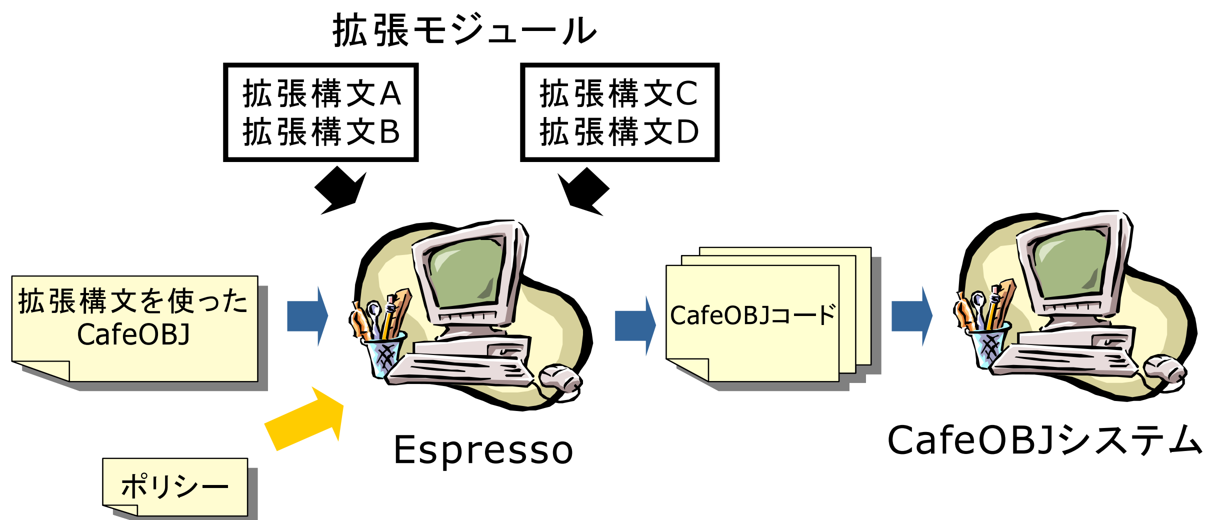

図 3.1: Espresso の全体像

ルは 1 つの拡張構文を定義するのではなく，モジュールの記述の利便性のために複数の拡張構文を定義できる．このようなモジュール化の利点として，再利用性と保守性の向上があげられる．具体的には，一度作成した拡張モジュールを他のユーザがそのまま利用したり，改良することが可能となる．

本研究では拡張モジュールを CafeOBJ を用いて記述する．そのことによって，次の利点が挙げられる．まず，Espresso のユーザは CafeOBJ のユーザであるので，モジュールを記述するための別の言語を覚えることなく，CafeOBJ を用いて CafeOBJ の構文を (擬似的に) 拡張することができる．次に，CafeOBJ の強力なモジュールシステムを利用することが可能である．前章で説明したように，CafeOBJ にはモジュールに関する様々な機能を備えており，その恩恵を受けることができる．また，CafeOBJ でモジュールを記述することは，op 宣言で項のパーザを記述し，等式で項の変換ルールを記述していることとみなすことができ，Espresso の設計方針とマッチする．さらに，CafeOBJ の項のパーザは任意の文脈自由文法を構文解析できるので幅広い構文の拡張が実現できる．

このように拡張構文をモジュール化することにより，ユーザは利用したい拡張構文を定義した拡張モジュールをただ組み合わせることでインクリメンタルな拡張が実現される．しかし，このようなモジュールを組み合わせることによる拡張には，モジュール間の不整合が生じる場合がある．次にこの不整合について説明する．

3.1.2 拡張モジュールの衝突

一般に，複数の拡張モジュールを組み合わせると，拡張モジュールが衝突して結果の予測が困難になる場合がある．ここで，**拡張モジュールの衝突**とはある拡張モジュールが他の拡張モジュールによって意図しない振舞いをすることである．このようなことが起こる

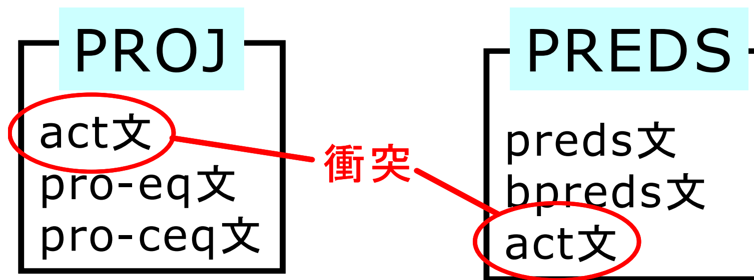

図 3.2: 拡張構文の衝突

のは，拡張モジュールを実装する時点において他のどのような拡張モジュールと組み合わせるか把握できず，他の拡張モジュールの影響を受けてしまうためである．

本研究では，拡張モジュールの衝突が起こるのは主に 2 つの形で起こると仮定する．

1. モジュール名の衝突
2. 拡張構文の衝突

1 は同じモジュール名で異なる機能を備えているモジュールが複数存在したとき，利用したいを他のモジュールによってモジュールを上書きしてしまい，参照したいモジュールの定義が消えてしまい，モジュールの振舞いが予測できなくなる．そのため，モジュール名の一意性を確保する必要がある．

2 は複数の拡張モジュール内において，同じ拡張構文が定義されていて変換ルールが異なる場合，どの拡張モジュールで定義した構文の変換ルールが適用されるか予測ができない．その結果，ユーザが意図していたものとは異なる結果が返るという問題がある．例えば図 3.2 の場合，PROJ モジュールと PREDS モジュールにいくつか拡張構文が定義されていて，act という拡張構文が 2 つのモジュールに定義されている．ユーザがこの 2 つの拡張モジュールを Espresso に組み込み，拡張構文を利用した仕様の中で act 文を使っていると，拡張構文の利用者はどちらのモジュールで定義されている変換ルールを使って変換されるか予測できない状況になる．そのため，構文が衝突しないように名前替えをしたり，変換ルールを指定するなどして衝突を回避する必要がある．

3.1.3 パッケージ概念の導入

既に述べたようにモジュール名が衝突した場合，意図しない動作が起こることがある．そのために，モジュール名の一意性を確保する必要がある．本研究では，これを解決するためにパッケージ機構を導入する．パッケージとはモジュールを 1 つのグループとしてまとめる 1 つの単位である．

パッケージの導入によりモジュールを階層的に名前を分けることが可能になる．なお，パッケージ名は Java[5] のように一意性を保つような名前規則があり，パッケージ名が衝突しないことを仮定する．このような前提をおくことで，モジュール名の衝突が起こらな

いことを仮定している．

3.1.4 ポリシーの導入

本研究では，拡張モジュールの利用に関するポリシーを導入する．本研究ではポリシーを拡張モジュールや拡張構文を用いた仕様記述とは独立して記述する方式を採用する．ポリシーでは主に以下の事を記述する．

- 利用する拡張モジュールの指定
- 拡張モジュールの衝突を回避するための記述

1つは利用する拡張モジュールの指定は，Espressoに組み込むことを指示するためのものである．もう1つの拡張モジュールの衝突を回避するための記述について詳しく説明する．

本研究では，既に説明したように拡張モジュールの衝突はモジュール名の衝突と拡張構文の衝突を想定しているが，モジュール名の衝突はパッケージの概念を導入することで一意性があることを仮定している．そこで，ポリシーでは主に拡張構文の衝突を回避することを記述する．本研究では拡張構文の衝突を回避するために，以下の2つの方法を採る．

- 拡張構文が衝突しないように，ユーザが拡張モジュールを直接修正することなく，拡張構文を変更できるようにする．
- コンテキストに応じて拡張構文の変換ルールを選択できるようにする．

拡張構文を変更するためには，CafeOBJの名前換えを行なう機能を利用する．名前換えをユーザが指定することで，衝突している拡張構文を違う構文に変更することができるので衝突が回避できる．また，ある拡張構文で定義されている変換ルールを別の構文で利用したいという場合においても，拡張構文の名前を変更することで利用可能になる．

次に，コンテキストに応じて拡張構文の変換ルールを選択できることにより，変換処理を細かく制御ができるようになる．拡張構文が衝突している場合でも，そのコンテストに応じて拡張構文の変換ルールを指定できるので衝突を回避できる． 拡張構文の名前換えとコンテキスト単位で変換ルールを切り換えることによりEspressoの入力となる仕様を変更せずに，独立に記述したポリシーを変更することにより，拡張構文の意味(変換ルール)を柔軟に変更できる．

コンテキスト

本研究では**コンテキスト**を変換する1つのブロック単位と定義する．具体的には，拡張構文を用いた仕様記述内に指定した範囲である．指定した範囲をコンテキストとし，コンテキストに応じて拡張構文の変換ルールを細かく指定できるようにする．そうすることに

```
#context FOO
mod! X {
  pr(NAT) .
  act f : Nat -> Nat .
  ops g h : Nat Nat -> Nat .
}
#end
#context BAR
mod! Y {
  pr(NAT) .
  act f : Nat -> Nat .
  ops g h : Nat Nat -> Nat .
}
#end
```

図 3.3: ローカルコンテキストの指定

より，拡張構文の衝突を解決でき，また，仕様を変更せずポリシーを書き換えるだけで，自由に構文の意味を変更できるようになる．コンテキストは次のように定義する．

コンテキストの単位は，基本的には 1 つの入力ファイルが 1 つのコンテキストとなる．入力ファイルが A, B, C とあった場合は，それぞれ異なるコンテキストとなる．また，さらに，1 つのファイル内でローカルなコンテキストも指定可能である．入力ファイル A のなかに，さらに FOO, BAR というコンテキストを細かく指定が出来る．ファイル内で指定されたコンテキストを**ローカルコンテキスト**と呼ぶ．ローカルコンテキストを指定するには入力ファイル内に，

#context コンテキスト名 ... #end

と記述することで囲まれた部分がローカルコンテキストとなる．また，どのローカルコンテキストにも属さない範囲については，**名前無しのコンテキスト**として定義する．なお，本研究ではコンテキストのネストについては認めない．では，図 3.3 にローカルコンテキストの具体例を示す．この場合，例えば，コンテキスト FOO には拡張構文 act の変換ルールを図 3.2 にある PROJ モジュールの変換ルールを指定し，コンテキスト BAR には PREDS モジュールで定義してある変換ルールを指定できる．このように，コンテキストを導入することで拡張構文が衝突している場合にも，コンテキストに応じて変換ルールを指定することができるので衝突を回避できる．そのことにより，ユーザが変換処理を細かく制御ができるようになる．

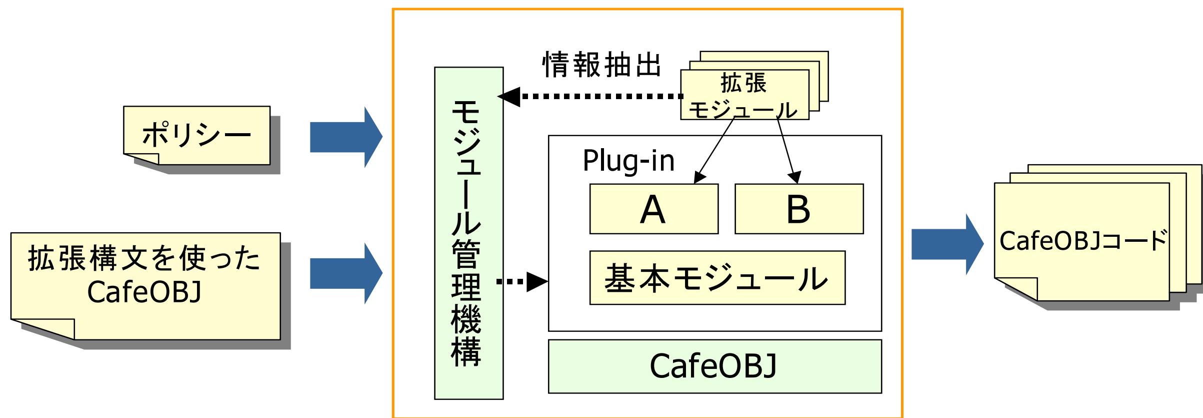

図 3.4: Espresso のアーキテクチャ

3.2 Espresso のアーキテクチャ

次に，前節で述べたアプローチをもとにした Espresso のアーキテクチャを説明する．本研究で提案するアーキテクチャを図 3.4 に示す．Espresso は基本モジュールと複数の拡張モジュールを組み合わせるためのモジュール管理機構，CafeOBJ から構成される．また，既に説明したように拡張モジュールの衝突を回避するためにポリシーの記述を導入する．

これらを用いて Espresso では CafeOBJ の構文拡張を以下のように実現する．Espresso はその初期状態において基本モジュールのみを持つ．ユーザは拡張モジュールを記述もしくは再利用し，それを Espresso に組み込むことで Espresso の拡張を行う．Espresso はユーザが指定したポリシーにもとづいて拡張モジュールを取り込みを行う．その際にモジュール管理機構によって，拡張構文の衝突を検出し，衝突を回避するための処理が行われる．Espresso を構成する基本モジュールとモジュール管理機構について詳しく説明する．

3.2.1 基本モジュール

基本モジュールとは Espresso のコアとなるモジュールのことである．基本モジュールは全ての拡張モジュールが輸入するモジュールである．基本モジュールでは CafeOBJ の文法 [10]，変換関数，内部参照関数，シンボルの操作をするための機能を定義している．文法の定義は CafeOBJ のサブセットを定義している．具体的にはシンタックスシュガーなどを取り除いた文法を定義している．**変換関数**とは基本モジュールの場合は，何も変換しないということを定義した関数である．つまり，基本モジュールで定義されている構文というのは変更がない事を意味する．そのため，Espresso の初期状態では CafeOBJ を受け取り CafeOBJ を返すだけの前処理系であるといえる．**内部参照関数**とは入力となる拡張構文を利用した仕様にどのような操作や等式が定義されているかなど拡張構文を用いた仕様で定義された情報を得るためのメタな参照である．**シンボルの操作**はユニークなシンボルの生成，シンボルの連結，文字列をシンボルに変換などを定義している．拡張モ

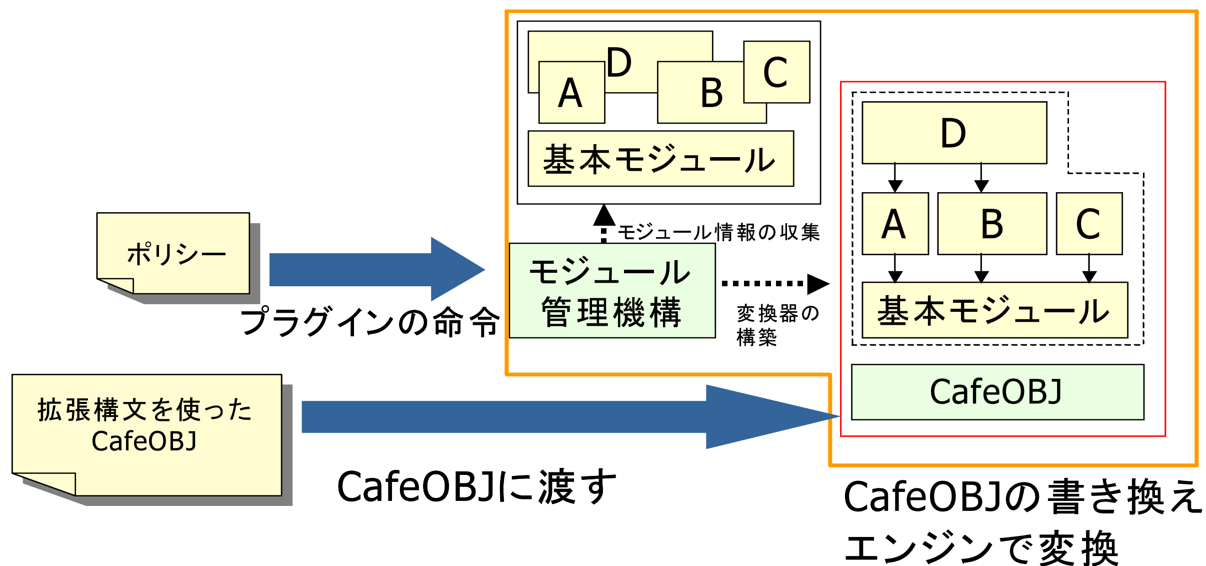

図 3.5: モジュール管理機構

ジュールはこれらの関数を利用して作成する．

3.2.2 モジュール管理機構

拡張モジュールを何も考えずに組み合わせると拡張の衝突が起こるので，衝突を回避するための機構が必要である．そのため，拡張モジュールを管理するモジュール管理機構が必要となる．例えば，ユーザが拡張モジュールに定義されている全ての拡張構文を把握しているならば，拡張構文の衝突は把握できる．しかし，現実にはユーザがすべてのモジュールの拡張構文を把握することは難しい．そのため，確実に衝突を発見しそれを警告することで，ユーザはポリシーを記述し，それに従って衝突を回避することが望ましい．この他にもモジュールをプラグインするためには様々な処理が必要になる．つまり，モジュール管理機構は拡張した前処理系を構築するための前処理を行う機構であるといえる．以下にモジュール管理機構の主な機能を挙げる．

- 拡張モジュールの情報管理
- 拡張モジュールの合成
- ポリシーに従った処理
- CafeOBJ システムとの通信

モジュール管理機構の一連の流れについて図 3.5 をもとに説明していく．モジュール管理機構は，ポリシーを入力とし，拡張モジュールの衝突を検出し，ユーザのポリシーに

従って衝突を回避するための処理を行う．そのため，どの拡張モジュールにどのような拡張構文が定義されているか (モジュール情報) を把握しておく必要がある．次に，ポリシーに従って拡張構文の名前換えやコンテキスト単位の変換ルールの指定を反映させて，Espresso に拡張モジュールを組み込むための準備をする．具体的には，CafeOBJ システムに拡張モジュールをロードし，それらをまとめた 1 つのモジュールを生成する．そして，基本モジュールと複数の拡張モジュールを組み合わせることによって構築された変換器を利用して，拡張構文を利用した仕様を CafeOBJ コードに変換する．変換処理は CafeOBJ の書換えエンジンを利用するので，モジュール管理機構によって CafeOBJ システムと通信して項として仕様を渡す．そして，項書換えシステムによって変換された結果を CafeOBJ システムから受けとる．また，変換処理の際に，モジュール管理機構はコンテキストの制御も行う．コンテキストの制御は主に入力ファイルからローカルコンテキストを調べ，コンテキストに応じて前処理系の変換処理を切り換える制御である．つまり，概念的には複数の変換器が作られ，コンテキストに応じて，変換器を使い分けることを行なう．

第4章　Espressoの実装

前章では，Espressoの設計方針とアーキテクチャについて説明した．本章では基本モジュール，拡張モジュール，およびモジュール管理機構の実装について説明する．また，ポリシーの記述方法についても説明する．

4.1　モジュールの実装

まず，モジュールに関する規則について説明する．モジュールはパッケージ単位にまとめられる．モジュール名を一意性を確保するために，**FQN**(完全限定名) を導入する．Espresso では FQN を以下のように定義する．

パッケージ名$モジュール名

そして，モジュールへの参照に関するルールは以下のように定義する．

同じパッケージに属さないモジュールへの参照は FQN による名前のみ参照を許す．同じパッケージに属すモジュールについては，FQN もしくはモジュール名のみでの参照を許す．

次にモジュールの名前とファイル名との関係について説明する．Espresso では拡張モジュールがどのファイルに定義されているか検索する必要がある．検索を効率的に行うために，モジュール名に対応したファイル名を付ける．また，パッケージ名はディレクトリ名と対応し，与えられたパスからパッケージ名とモジュール名を用いてファイルを検索する．ファイル名の名前規則は以下のようにする．

モジュール名.es

例えば，FOO というモジュールを定義したファイルは'FOO.es' というファイルとなる．このような名前がファイル名と名前が一致するモジュールを**パブリックモジュール**と呼ぶ．

もし，ファイル名と対応がとれていない場合はエラーとなる．また，ファイル名と名前が対応しているモジュール以外にもモジュールが定義されている場合は，それらを**プライベートモジュール**と呼ぶ．前に定義した FQN はパブリックモジュールの FQN である．プライベートモジュールの FQN は次のように定義する．

パッケージ名$パブリックモジュール名::プライベートモジュール名

なお，現在の実装では外部のパッケージからプライベートモジュールを参照するのは禁止している．

これにより，モジュールの名前の衝突を完全に解決する．まず，前章で述べたようにパッケージ名の一意性を仮定しているので，パッケージの衝突は考慮しない．次にパブリックモジュールはファイル名と対応するので，同じディレクトリに同じ名前のファイルは作成できないので，衝突をすることは物理的に起こらない．次にプライベートモジュールについては，パブリックモジュールの一意性が示せて，かつ，同じファイル内に同じモジュール名の定義を禁止しているので，プライベートモジュールも衝突することは無い．これにより，モジュールの一意性を確保できる．

4.1.1 基本モジュール

基本モジュールはQIDSETとBASIC-SYNTAXモジュールから構成される．前者はシンボルを操作するためのモジュール定義であり，後者がCafeOBJの構文情報を定義したモジュールである．

QIDSET モジュール

まず，QIDSETモジュールについて説明する．QIDSETモジュールはBASIC-SYNTAXが輸入するモジュールである．つまり，すべての拡張モジュールが輸入するモジュールである．QIDSETモジュールはシンボルと文字列の操作などを定義したモジュールであり，変換ルールを記述するための補助的なライブラリ関数である．文字列を扱うモジュールはStringモジュール，シンボルを扱うモジュールはCHAOSモジュールとして組込みのモジュールとして定義してある．CHAOSモジュールにはシンボルを利用できるだけで，シンボルを操作するための関数は定義されていない．そこでQIDSETモジュールは，2つのモジュールを輸入し，不足している機能を追加している． QIDSETモジュールで定義した関数一覧を表4.1に示す.

シンボルの操作を実現するために，一部Common Lispのコードが含まれる．CafeOBJでは等式の右辺にCommon Lispのコードを記述することができ，いくつかのシンボルの操作などはCommon Lispを直接記述している．例えば，concat-symbol関数の場合は以下のような等式で実現している．

```
eq concat-symbol(I:Identifier, II:Identifier) =
  #! (let* ((s1 (symbol-name I))
            (s2 (symbol-name II))
            (str (concatenate 'string s1 s2)))
     (intern str)) .
```

関数名	機能
_ === _ : Identifier Identifier ->Bool	シンボルの同一名の判定.
_ _ : IdSet IdSet ->IdSet	シンボル列の構成子.
gensym : ->Identifier	ユニークなシンボルを生成.
concat-symbol : Identifier Identifier ->Identifier	2 つのシンボルの連結.
sortid2str : SortId ->String	ソート名の文字列を返す.
str2id : String ->Identifier	文字列をシンボルに変換.
id2str : Identifier ->String	シンボルを文字列に変換.
different : IdSet IdSet ->IdSet	2 つのシンボル列の差分を返す.
member? : Identifier IdSet ->Bool	シンボル列にあるシンボルがあるか判定.
make-var-symbol : Identifier SortId ->Identifier	シンボルとソート名から on-the-fly 変数宣言 (項に A:Nat という形式で現われる変数宣言) のシンボルを生成.
make-var : SortId ->Identifier	ソート名から on-the-fly 変数宣言のシンボルを生成. 変数名は gensym で生成されるシンボル.
get-var-symbol : Identifier ->Identifier	on-the-fly 変数宣言の変数名を返す.

表 4.1: QIDSET の関数一覧

```
[ Operation ]  --  op
[ Operation < OperationDecl ]
op (op _ : _ -> _ .) : Identifier SortIdSet SortId -> Operation .
op (op _ : -> _ .) : Identifier SortId -> Operation . -- constant
```

図 4.1: CafeOBJ による文法の記述

OperationDecl	::=	*Operation*
Operation	::=	**op** *Identifier* : *SortIdSet* -> *SortId* .
	|	**op** *Identifier* : -> *SortId* .

図 4.2: BNF による文法の記述

表 4.1 に定義してあるように，concat-symbol は 2 つのシンボルを引数とする．Common Lisp のコードの中で，等式の左辺にあるシンボルを表わすオブジェクトをそのまま使うことができる．let*構文で 1 引数目と 2 引数目のシンボルを文字列に変換したあとに 2 つの文字列を連結し，その文字列をシンボルに変換している．

BASIC-SYNTAX モジュール

次に BASIC-SYNTAX モジュールについて説明する．BASIC-SYNTAX モジュールは次の 3 つを定義したモジュールである．

- CafeOBJ の文法
- 変換関数
- 内部参照関数

BASIC-SYNTAX は出来る限り小さくするために**ops** や**vars** などといったシンタックスシュガーは BASIC-SYNAX から取り除き，拡張モジュールとして用意している．また，項の形を standard fix のみに限定している．CafeOBJ で CafeOBJ の文法を記述するには，次の規則で文法を記述できる．

- 非終端記号をソート名で表す.
- 文法の生成規則を op 宣言で記述.

例として CafeOBJ の op 宣言の文法を CafeOBJ と BNF で記述して対応を見る．図 4.1 が CafeOBJ で記述した文法であり，図 4.2 が BNF で記述した文法である．Operation,

関数名	意味
translate	トップレベルの変換関数.
translate-module	モジュールを変換する関数.
translate-view	ビューを変換する関数.
translate-module-element	モジュールの中身を定義する構文を変換する関数.
translate-view-element	ビューの中身を定義する構文を変換する関数.

表 4.2: 変換関数の一覧

OperationDeclが非終端記号になり，ソートの順序関係を用いることにより図4.2のBNFの1行目と対応が取れる．また，2行目と3行目のop宣言に定義した項の構成子をOperationに還元する文法の生成規則とみなすことで，BNFとCafeOBJによる文法の記述が容易に対応をとることができる．このようにして，他の構文についても記述した．他の構文の説明については省略する．

次に変換関数について説明する．変換関数は拡張構文を使った仕様を変換するための関数を指定する．BASIC-SYNTAXにおける各構文の変換ルールは「変換しない」というルールを定義している．変換関数については表4.2にまとめる．なお，拡張構文を定義するときは，それぞれの拡張構文に対して変換ルールを記述する．変換関数の呼び出し関係は次のようになる．まず，拡張構文を利用した仕様に対して，translate関数を適用する．次に，パースされた項がモジュールを定義しているシンボル列ならば，translate-moduleを呼び出す．ビューの場合ならば，translate-viewを呼び出す．そして，さらに，モジュールの内部をtranslate-module-element，ビューの内部をtranslate-view-elementと階層的に変換関数が呼び出されていく．拡張構文の変換ルールを記述する際は，その拡張構文がどの変換関数で呼び出されるか考慮し，その変換関数の中に変換ルールを記述する．

```
op translate-module-element : ModuleElement ModuleDecl -> ModuleElement .
eq translate-module-element(M:ModuleElement M':ModuleElement,
                  MD:ModuleDecl) = translate-module-element(M,MD)
                                   translate-module-element(M', MD) .
eq translate-module-element(O:Operation, M:ModuleDecl) = O .
eq translate-module-element(BO:BOperation, M:ModuleDecl) = BO .
eq translate-module-element(S:Sort, M:ModuleDecl) = S .
eq translate-module-element(H:HSortSet, M:ModuleDecl) = H .
```

最後に，内部参照関数について説明する．これらの関数は，モジュール名の取得や宣言されている等式を全て取得するなどといったメタな機能である．これらの関数はパターンマッチを利用することで記述する．以下にモジュール名を取得する関数の定義について示す．

```
op module-name : ModuleDecl -> Identifier { memo } .
eq module-name(mod! I:Identifier { ME:ModuleElement } ) = I .
eq module-name(mod! I:Identifier(P:ParameterList){ ME:ModuleElement })
                                        = I .
eq module-name(mod* I:Identifier { ME:ModuleElement } ) = I .
eq module-name(mod* I:Identifier(P:ParameterList){ ME:ModuleElement })
                                        = I .
```

関数の引数を項のパターン毎に定義していく．この例の場合では，モジュールを定義する構文のパターン毎にモジュール名を返す等式を記述している．また，CafeOBJ では一度簡約を行った結果をメモすることができる．op 宣言の属性の部分に memo を指定することで書換え結果をメモする．同じ引数を簡約する場合，簡約をせずにメモを参照することで効率的に実行することができる．このように module-name 関数やその他の内部参照関数は結果をメモすることで，効率を落すことなく必要な情報を参照できる．

4.1.2 拡張モジュール

拡張モジュールは拡張構文の定義，内部参照関数，変換ルールの記述をする．拡張モジュールは BASIC-SYNTAX もしくは BASIC-SYNTAX を輸入している拡張モジュールを extending モードで輸入する．また，拡張モジュールが属するパッケージ名を記述する．パッケージの指定は，

package パッケージ名

と指定する．

拡張構文の定義は，基本モジュールと同じ方法で op 文で文法を記述する． ここでは ops 文の拡張を例に拡張モジュールの定義について見ていく．ops 文は同じアリティとコアリティな関数を 1 度に複数宣言するための構文である．この構文は CafeOBJ に組込みで持っている構文であるが，op 文を 1 つずつ書くことと同じ意味 (シンタックスシュガー) なので基本モジュールを軽くするために取り除いている．拡張構文の宣言を図 4.3 に示す．また，拡張された構文を加えた BNF を図 4.3 に示す．

ops 文の場合，op 文を拡張した構文なので OperationDecl のサブソートとして，Operations というソートを宣言した．ops を使った構文は，Operations という非終端記号に還元されることを意味している．その他の拡張構文を定義したい場合，表 4.3 に示したソートのサブソートとして，ある非終端記号を表すソートを宣言し拡張する．例えば，同じソートを表わす変数を一度に複数宣言するための構文 vars を追加したい場合は，VariableDecl のサブソートとして，あるソート (Variables など) を定義する．そして，拡張構文を定義する際に，拡張構文を用いた項は宣言したソートに還元されることを定義する．

もし，ある構文の優先順位を定義したい場合は prec 属性を追加することで実現できる．

```
[ Operations ]  --  ops
[ Operations < OperationDecl ]
op (ops _ : _ -> _ .) : IdSet SortIdSet SortId -> Operations .
op (ops _ : -> _ .) : IdSet SortId -> Operations . -- constant
```

図 4.3: ops 構文の拡張

OperationDecl	::=	*Operation* | *Operations*
Operation	::=	**op** *Identifier* : *SortIdSet* -> *SortId* .
	|	**op** *Identifier* : -> *SortId* .
Operations	::=	**ops** *IdSet* : *SortIdSet* -> *SortId* .
	|	**ops** *IdSet* : -> *SortId* .

図 4.4: ops 構文を拡張した BNF

ソート名	ソート名の意味
ModuleDecl	モジュール宣言を表すソート.
SortDecl	ソート宣言を表すソート.
OperationDecl	関数のシグニチャの宣言を表すソート.
EquationDecl	等式を記述するための構文を表すソート.
TransitionDecl	trans 文を拡張するためのソート.
VariableDecl	変数宣言の構文表すソート.
ImportDecl	輸入を宣言する構文を表すソート.
ModuleElement	上のどれにも該当しないモジュールの内部を定義する構文を追加するためのソート.
ViewDecl	ビューを宣言する構文を表すソート.
ViewElement	ビューの内部を定義するためのソート.

表 4.3: 拡張構文を定義するための非終端記号一覧

```
op (ops _ : _ -> _ .) : IdSet SortIdSet -> SortId { prec: 20 } .
```

prec の値は 0〜127 まで設定でき，数字が小さい方が優先順位が高いことを意味する．

内部参照関数の定義は，BASIC-SYNTAX に定義されている内部参照関数を再利用して記述する．BASIC-SYNTAX には，拡張構文も含めて関数のシグニチャを定義する構文を使った文を全て返すなどといったものを定義している．ops はこの BASIC-SYNTAX に定義されている内部参照関数を用いる．ops の例で実際に作成した例を以下に示す．

```
op get-ops : ModuleElement -> ModuleElement { memo } .
op get-ops-in : ModuleElement -> ModuleElement { memo } . -- 補助関数
eq get-ops(M:ModuleElement) = get-ops-in(get-op(M)) .
ceq get-ops-in(O:OperationDecl) = O if O :is Operations .
ceq get-ops-in(O:OperationDecl) = no-elem if not(O :is Operations) .
ceq get-ops-in(O:OperationDecl M:ModuleElement) =
  O get-ops-in(M) if O :is Operations .
ceq get-ops-in(O:OperationDecl M:ModuleElement) =
  get-ops-in(M) if not(O :is Operations) .
eq get-ops-in(no-elem) = no-elem .
```

get-ops 関数は ops で宣言した文を全て返すための内部参照関数である．get-ops-in は get-ops の補助関数である．get-ops の等式を記述するときは，BASIC-SYNTAX に定義されている get-op 関数を用いて，関数のシグニチャを定義した宣言を全て集める．補助関数で集められたものから:is 関数で，ソートが ops 文を表わす Operations か判定して ops 文を集める．最後の 1 行は何もみつからなかったときの例外を表す．

内部参照関数のシグニチャを op 文で宣言するときには，standard な形，つまり，_を含まない op 宣言をすることとする．Espresso では，拡張構文を mixfix による op 宣言，内部参照関数と変換関数を standard による op 宣言と区別する．Espresso では拡張構文の衝突を検出するときには，このルールをもとに拡張モジュールから探し出す．拡張構文の検出についてはモジュール管理機構のところで詳しく説明する．拡張構文の内部参照関数を定義するための，BASIC-SYNTAX に定義されている内部参照関数について表 4.4 にまとめる．拡張構文の内部参照関数は表 4.4 に定義されているものを使って，一度それぞれ宣言したものを集めて，補助関数を定義して:is 関数で取捨選択する．

次に変換ルールの記述を説明する．変換ルールを記述するには，基本モジュールで定義された関数を使って，等式で変換ルールを記述する．ops の変換ルールの記述は図 4.5 のように記述する．

変換ルールはパターンマッチを使って記述する．変換ルールを記述するには等式を用いる．具体的には，等式の左辺に変換関数とその引数に変換したい構文を記述し，右辺に

関数名	機能
get-op	関数のシグニチャを宣言しているものを集める.
get-sort	ソートを宣言しているものを集める.
get-var	変数を宣言しているものを集める.
get-eq	等式を宣言しているものを集める.
get-op-map	関数の名前換えを宣言しているものを集める.
get-sort-map	ソートの名前換えを宣言しているものを集める.

表 4.4: 内部参照関数を定義するための関数一覧

```
var I : Identifier . var IS : IdSet .
var S : SortId . var SS : SortIdSet .
eq translate-module-element(ops I : SS -> S .) = (op I : SS -> S .) .
eq translate-module-element(ops I IS : SS -> S .) = (op I : SS -> S .)
                          translate-module-element(ops IS : SS -> S .) .
eq translate-module-element(ops I : -> S .) = (op I : -> S .)
eq translate-module-element(ops I IS : -> S .) = (op I : -> S .)
                          translate-module-element(ops IS : -> S .) .
```

図 4.5: ops 構文の変換ルールの記述

<table>
<tr><td>top</td><td>::=</td><td>policy | top policy</td></tr>
<tr><td>policy</td><td>::=</td><td>import | syntax | reduce | context</td></tr>
<tr><td>import</td><td>::=</td><td>import ident</td></tr>
<tr><td>syntax</td><td>::=</td><td>syntax * { decllist }</td></tr>
<tr><td>decllist</td><td>::=</td><td>decl | decllist , decl</td></tr>
<tr><td>decl</td><td>::=</td><td>sortmap | opmap</td></tr>
<tr><td>sortmap</td><td>::=</td><td>sort ident | hsort ident</td></tr>
<tr><td>opmap</td><td>::=</td><td>op opmaplexlist | bop opmaplexlist</td></tr>
<tr><td>opmaplexlist</td><td>::=</td><td>opmaplex | opmaplexlist opmaplex</td></tr>
<tr><td>opmaplex</td><td>::=</td><td>ident | opmap_paren</td></tr>
<tr><td>opmap_paren</td><td>::=</td><td>(opmaplexlist)</td></tr>
<tr><td>reduce</td><td>::=</td><td>reduce ident</td></tr>
<tr><td>context</td><td>::=</td><td>context file :: localcontext :: opdecl -> ident</td></tr>
<tr><td>file</td><td>::=</td><td>file , ident | ident | *</td></tr>
<tr><td>localcontext</td><td>::=</td><td>localcontext , ident | ident | *</td></tr>
<tr><td>identlist</td><td>::=</td><td>identlist ident | ident</td></tr>
<tr><td>ident</td><td>::=</td><td>[^\(\)\[\] \n\t]+</td></tr>
</table>

表 4.5: ポリシーファイルの文法

変換される結果を記述する．ops 文はシグニチャを定義するための拡張構文であるので，translate-module-element 関数で拡張構文を変換する．変換ルールは ops の後に続くシンボル列を参照し，複数の op 文を生成する．

4.2 ポリシーファイル

本研究では，ポリシーを記述するための簡易記述言語を設計し，そのパーザを実装した．ポリシーの文法を表 4.5 のように定義する．ポリシーファイルには以下の 4 つについて記述する．

- 組み込む拡張モジュールの指定
- 拡張構文の名前換え
- コンテキストに応じた変換ルールの指定
- トップレベルの変換関数の指定

組み込む拡張モジュールの指定

組み込む拡張モジュールの指定について説明する．Espresso では，拡張モジュールを組合せることにより Espresso を拡張するので，どの拡張モジュールを利用するかを指定する必要がある．それを指定するために，import 文を用いてプラグインする拡張モジュール名を FQN で記述する．例えば，FOO$BAR というモジュールを利用する場合は，

```
import FOO$BAR
```

と記述する．また，あるパッケージに属するモジュールを全て利用したい場合は'*' を指定することにより利用可能になる．例えば，FOO というパッケージに属するモジュールを全て利用したい場合は,

```
import FOO$*
```

と指定することも可能である．

拡張構文の名前換え

拡張構文が衝突している場合，その構文の変換ルールがどのように適用されるか予測ができないので，そのような危険な状況を回避する必要がある．そのため，構文の名前換えとコンテキストに応じた変換ルールの切り替えを実現し，状況に応じて拡張の衝突に対処する．まず，拡張構文の名前換えの指定について説明する．拡張構文の名前換えの指定は syntax 文を用いて記述する．例えば，拡張構文 ops を my-ops という構文に変更したい場合は,

```
syntax System$OPS * { op (ops _ : _ -> _ .) -> (my-ops _ : _ ->_ .)}
```

と指定する．また拡張モジュールで定義した非終端記号 (ソート名) を変更することも可能である．

```
syntax System$OPS * { sort Operations -> MyOperations }
```

コンテキストに応じた変換ルールの指定

コンテキストに応じた変換ルールの指定は context 文で指定する．context 文では，コンテキストと拡張構文を指定し，その変換ルールが定義してあるモジュール名を指定する．

前章で説明したように，コンテキストの単位は 1 つの入力ファイルをコンテキストの単位とし，さらにファイル内でローカルコンテキストを定義できる．例えば，input というファイルの FOO というローカルコンテキストにおいて，拡張構文 act は Projection$PROJ で定義されている変換ルールを適用することを意味する．

```
#context input :: FOO :: (act _ : -> _ .) -> Projection$PROJ
```

```
import System$OPS
import System$VARS
context * :: FOO :: (act _ : -> _ .) -> Projection$PROJ
context * :: BAR :: (act _ : -> _ .) -> System$PROJ
```

図 4.6: コンテキストを指定したポリシーファイル

ここで定義しているローカルコンテキスト FOO は入力ファイル input の中の FOO だけに対してのポリシーである．もし入力ファイル input1 というファイルにもローカルコンテキスト FOO が定義されていても，それは異なるコンテキストとなる．もし，2 つのコンテキストを同じコンテキストとして扱いたい場合は，“,” でファイル名を追加することで指定できる．

```
#context input, input1 :: FOO :: (act _ : -> _ .) -> Projection$PROJ
```

また，全ての入力ファイルの FOO を同じコンテキストとして扱いたい場合は，“*” で指定することで適用できる．

```
#context * :: FOO :: (act _ : -> _ .) -> Projection$PROJ
```

ローカルコンテキストについても違う名前のローカルコンテキストを同じコンテキストとして扱うこともできる．例えば，ローカルコンテキスト FOO と BAR があった場合，同じコンテキストとして扱いたい場合は，“,” でローカルコンテキストを追加する．

```
#context * :: FOO, BAR :: (act _ : -> _ .) -> Projection$PROJ
```

また，名前無しコンテキストを含めて全てのローカルコンテキストを指定したい場合は “*” で指定する．

```
#context * :: * :: (act _ : -> _ .) -> Projection$PROJ
```

なお，このようにファイル名とローカルコンテキスト名に “*” が指定された場合，全ての入力ファイルのコンテキストで衝突している拡張構文の変換ルールを指定することができる．

では次にコンテキストに応じた変換ルールを指定した例として，図 4.6 をポリシーとし，図 4.7 を拡張構文を利用した仕様とする．Projection$PROJ と System$PREDS には act 文が定義されていて，PROJ に定義されている変換ルールは，act を bop に変換し，PREDS に定義されている変換ルールは，act を op に変換するものとする．図 4.8 が変換結果になる．この場合，図 4.7 のコンテキスト FOO の中のモジュール X では act が bop に変換され，コンテキスト BAR の中のモジュール Y では act が op に変換されているのが確認できる．このように，コンテキストを細かく分けることで，変換処理を細かく指定できる．

Espresso ではコンテキストに応じた変換ルールの指定において，1 つのコンテキストにおいて同じ拡張構文に複数の変換ルールを指定した場合はエラーになる．つまり，以下の

```
#context FOO
mod! X {
  pr(ABC) .
  act f : F -> G .
}
#end
#context BAR
mod! Y {
  pr(ABC) .
  act f : F -> G .
}
#end
```

図 4.7: コンテキスの例

```
mod!  X {
  pr( ABC ) .
  bop f : F -> G .
}
mod!  Y {
  pr( ABC ) .
  op f : F -> G .
}
```

図 4.8: コンテキストに応じた変換ルールを指定した変換結果

ようにグローバルコンテキストで指定した変換ルールをローカルコンテキストで違う変換ルールを指定することを禁じている．

```
#context * :: * :: (act _ : -> _ .) -> Projection$PROJ
#context * :: FOO :: (act _ : -> _ .) -> System$PREDS
```

また，コンテキストに応じた変換ルールを指定する際に，ポリシーで拡張構文の名前換えを指定した構文に対して，コンテキスト文に指定することもできる．例えば，拡張構文 act を my-act に名前換えすることを記述したポリシーに対して，context-a に拡張構文 my-act の変換ルールを指定することができる．以下のポリシー記述に示す．

```
syntax Projection$PROJ * {op (act _ : _ -> _ .) -> (my-act _ : _ -> _ .)}
context * :: context-a :: (my-act _ : _ -> _ .) ->  Projection$PROJ
```

しかし，反対に名前換えする前の拡張構文をコンテキストに指定するとエラーになる．

```
syntax Projection$PROJ * {op (act _ : _ -> _ .) -> (my-act _ : -> _ .)}
context * :: context-a :: (act _ : _ -> _ .) -> A
```

この場合，拡張構文 act を my-act に変更することを宣言しているので，拡張モジュール PROJ の中には拡張構文 act は定義されていない．このように，syntax 文と context 文がポリシーに記述されていた場合，syntax 文を先に評価する．

変換関数の指定

トップレベルでの変換するための関数はデフォルトでは translate 関数で変換を行うが，これを変更することが可能である．そうすることで，図 4.2 関数に定義されているもの以外の変換関数をユーザがカスタマイズすることができる．例えば，ユーザが独自に変換関数 my-translate を作成し，これをもとに変換したい場合は，

```
reduce my-translate
```

と指定することで変更できる．

4.3 モジュール管理機構の構成

Espresso のコアであるモジュール管理機構について説明する．前章でモジュール管理機構のアーキテクチャを説明をした．そのアーキテクチャをもとに Ruby[11] で実装した．モジュール管理機構のクラス構成を表 4.6 にまとめる．また，モジュール管理機構は以下のことを行う．

- 利用する拡張モジュールの情報収集

クラス名	機能
Espresso	Espresso のメインクラス
ModuleParser	拡張モジュールをパースし，情報を収集するためのクラス
PolicyParser	ポリシーファイルをパースするためのクラス.
DependChecker	モジュールの検索と依存関係を作成するクラス.
PackageManager	全てのパッケージを管理するクラス.
PackageInfo	パッケージ情報を管理するクラス.
ModuleInfo	拡張モジュールの情報を管理するクラス.
TSort	トポロジカルソートを行うクラス.
ResultParser	CafeOBJ システムの出力した結果をパースし，結果をプリティプリントするクラス.
Reduce	CafeOBJ システムと通信するためのクラス.

表 4.6: モジュール管理機構を構成するクラス一覧

- 拡張の衝突の検出
- 拡張モジュールのロード
- トップレベルとコンテキストモジュールの生成
- CafeOBJ とのプロセス間通信
- CafeOBJ からの結果を解析

モジュール情報の収集

まず，モジュール管理機構は前節で説明したポリシーファイルを読み込み必要なモジュール， 拡張構文の名前換え，コンテキスト情報，変換関数について情報を集める．そのため，ポリシーファイルのパーザを作成し，ポリシーを解釈する．これは，PolicyParser によって実現される．次に，必要な拡張モジュールから次の 5 つの情報を収集する．

1. パッケージ名
2. モジュール名
3. 輸入するモジュール
4. ソート名
5. op 宣言

そのため，拡張モジュールをパースするパーザを作成した．実際にはこのパーザを ModuleParser クラスとして実装した．これらの情報は，拡張構文の衝突の検出やモジュールの依存関係の参照など様々な場所で参照する．

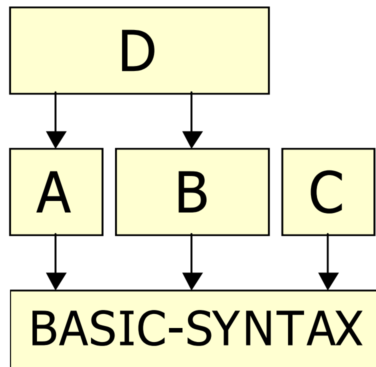

図 4.9: 依存関係のグラフ

拡張構文の衝突の検出

拡張構文の衝突は拡張モジュールで定義されている拡張構文を参照し，同じ構文が複数の拡張モジュールにおいて定義されているものを検出する．しかし，拡張構文の定義は，op 宣言によるものであり，変換関数の定義や内部参照関数もまた op 宣言で定義される．そこで Espresso では，変換関数と内部参照関数は standard な形で宣言されているという前提を置く．つまり，拡張構文は_がある関数のシグニチャとする．そうすることで，拡張構文の検出を容易に実装できる．拡張構文の衝突を検出するためには，mixfix 宣言の名前とそのアリティをもとに衝突を検出する．そのためにまず，mixfix 宣言の名前が同じものを集めて，さらにアリティのソートが一致した場合は拡張構文の衝突が起こっていると判断し警告を出力する． Espresso の実装では，衝突を検出した場合は以下のように警告を出力する．

```
warning!! conflict op ( act  _ : -> _ . ) : at Projection$PROJ, System$PREDS,
```

この例では，Projection$PROJ と System$PREDS で拡張構文が衝突していることを警告している．

拡張モジュールのロード

拡張モジュールを CafeOBJ で利用するには，一度 CafeOBJ システムにロードする必要がある．Espresso では，依存関係をもとに自動的に CafeOBJ システムへロードする．依存関係とは輸入関係を表しており，その輸入関係をもとに依存関係のグラフを作成する．依存関係のグラフは図 4.9 のようになる．そして，作成した依存関係のグラフを用いて，トポロジカルソート [13] を行ない，ソートした順序で CafeOBJ システムニ拡張モジュールをロードしていく．

ロードする際には，拡張モジュールの名前や依存関係で指定されているモジュール名を FQN に変換し，package 文を消す必要がある．package というキーワードは CafeOBJ の文法ではないので，そのままだとエラーになる．

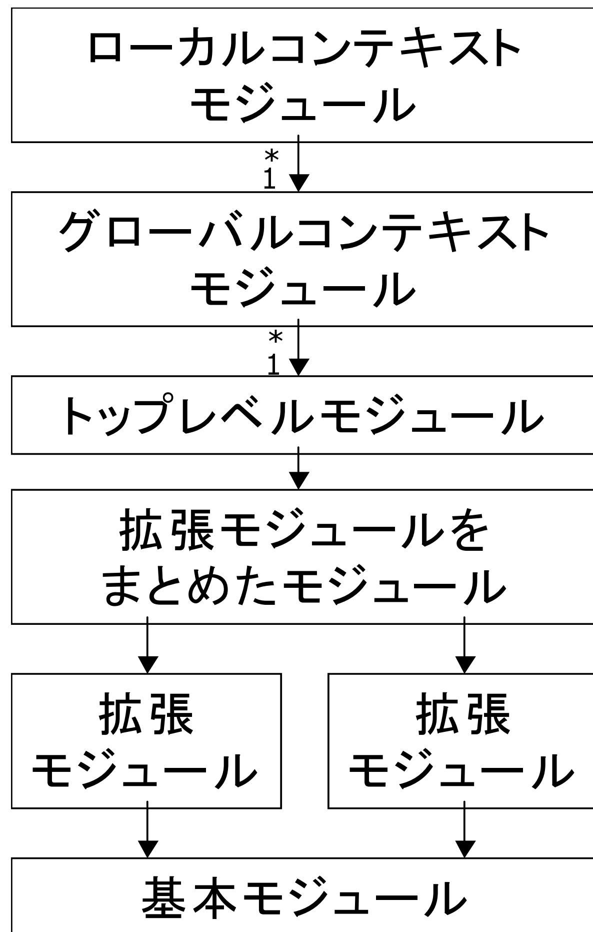

図 4.10: モジュールのレイヤ構造

トップレベルとコンテキストモジュールの生成

モジュール管理機構はトップレベルのモジュール，コンテキストに応じたモジュールを生成する必要がある． モジュールのレイヤ構造を図 4.10 に示す．

まず最初に，全ての拡張モジュールを取り込んだモジュールを生成する．そのために，トポロジカルソートによって得られた依存関係を検索し，参照されないモジュールを集める．図 4.9 のモジュール構造の場合では，C，D が集められる．図 4.9 のモジュール関係をもとに生成されるトップレベルモジュールは，次のようなモジュールが自動生成される．

```
mod TOP-MODULE {
  ex(C) .
  ex(D) .
}
```

次に構文の名前換えを考慮したトップモジュールを生成する．名前換えは CafeOBJ の

機能を用いて行う．拡張モジュールを1つにまとめたモジュールに対して，ポリシーに記述した名前換えを行う．例えば，ポリシーに拡張モジュールAのfという拡張構文をgに変更する場合は，以下のようなトップレベルのモジュールを生成する．

```
mod TOP-MODULE' {
  ex(TOP-MODULE * { op (f).A -> g }) .
}
```

コンテキストの指定がある場合はさらに**グローバルコンテキストモジュール**を生成する．グローバルコンテキストモジュールは "*" で指定されたコンテキストの指定を反映したモジュールである．このモジュールはトップレベルのモジュールを輸入し，ポリシーに応じた変換ルールを適用するモジュールを生成する．グローバルコンテキストモジュールは，"@@GLOBALCONTEX@@" というモジュール名でモジュールを生成する．例えば，Sysmte$OPS と System$VARS というモジュールで拡張構文 ops が衝突している場合，

```
context * :: * :: (ops _ : -> _ .) -> System$VARS
```

このようなポリシーが与えられた時，次のようなグローバルコンテキストモジュールを生成する．

```
mod @@GLOBALCONTEXT@@ {
  ex(TOP-MODULE * { op (ops _ : -> _ .).System$OPS ->
                    (undef-syntax ops _ : _ .) }) .
}
```

CafeOBJ のモジュールシステムでは，モジュールに定義した関数を取り除くということはできないので，内部的に名前換えをすることで，指定した変換ルール以外を無効にし，指定した変換ルールが適用されるようにした．例の場合では，ops 文を定義している System$VARS 以外の拡張モジュール，つまり，System$OPS に対して，ops 文を内部的に名前換えすることで，System$VARS の変換ルールが適用されるようになる．拡張構文を無効にするためには，undef-syntax というキーワードを拡張構文の先頭につけることで無効にする．そのため，Espresso では undef-syntax ではじまる拡張構文の定義を禁止している．

ポリシーにローカルコンテキストを指定している場合は，**ローカルコンテキストモジュール**を生成する．ローカルコンテキストモジュールはグローバルコンテキストを輸入して作成する．ローカルコンテキストモジュールは，"CONTEXT@ローカルコンテキスト名" というモジュール名になる．ローカルコンテキストモジュールはグローバルコンテキストモジュールと同様に，指定された変換ルール以外の変換ルールを無効にするために，指定したモジュール以外で定義されている拡張構文を名前換えする．では，コンテキストモジュールの生成される例とを以下に示す．

```
context * :: * :: (vars _ : -> _ .) -> System$VARS
```

```
context * :: FOO :: (ops _ : -> _ .) -> System$OPS
```

このようなポリシーが与えられた時，次のようなローカルコンテキストモジュールを生成する．

```
mod CONTEXT@FOO {
  ex( @@GLOBALCONTEXT@@ * { op ( ops _ : -> _ . ).System$VARS ->
                              ( undef-syntax ops _: -> _ .) } ) .
}
```

拡張構文を使用した仕様はトップレベルモジュール，グローバルコンテキストモジュール，ローカルコンテキストモジュールのいずれかの上で CafeOBJ の項書換えシステムを用いて変換される．コンテキストを制御するために，入力となる仕様を解析し，コンテキストに応じて変換するためにモジュール切り替える．コンテキストを切り換えるためには，CafeOBJ の select コマンドを用いる．なお，本研究では，実装上の理由によりコンテキストについて以下の制限を設けている．ローカルコンテキストの指定は任意の範囲ではなく，ModuleDecl または ViewDecl に属する構文を使っている範囲となる．つまり，CafeOBJ のモジュール宣言やビュー宣言などであり，モジュール定義内にローカルコンテキストを指定すると正しく変換されない．

Espresso では，CafeOBJ システムと通信するためにパイプを用いている．パイプを通して，モジュール管理機構から CafeOBJ システムに select コマンドを送る．また，Espresso では，CafeOBJ の項書換えシステムを利用して仕様の変換をするので，パイプを通して入力の仕様を項として渡し，CafeOBJ の reduce コマンドで変換する．これらの処理は Reduce クラスに実装した．

変換結果の取り出し

CafeOBJ の項書換えシステムで変換された仕様を取り出す必要がある．CafeOBJ で変換された結果は図 4.11 のように表示される．モジュール管理機構はこの変換された仕様を取り出す必要がある．現在の方法ではアドホックな方法を採っている．ある程度，決まった形式で出力されるので，その結果をパースするパーザ，ResultParser クラスを実装した．変換された仕ようは図 4.11 の下の部分だけであり，その他の部分は必要がないので取り除く．そして，構文解析した結果をプリティプリントする．

4.4 Espresso の実装のまとめ

前章のアプローチをもとに Espresso の実装について説明した．Espresso は基本モジュール，拡張モジュール，モジュール管理機構，CafeOBJ で構成した．また，ポリシーを入力とすることで，拡張構文の衝突の回避を実現した．最後にこれらの機能と形式について

```
@@GLOBALCONTEXT@@> -- reduce in @@GLOBALCONTEXT@@ : translate(mod! X
                                          { (pr ( NAT
                                             ) .
                                                ops (f g)
                                                : Nat
                                                -> Nat
                                                .)
                                          })
mod! X { (pr ( NAT ) . op f : Nat -> Nat . op g : Nat
                                          -> Nat
                                          .) } : Module
(0.010 sec for parse, 6 rewrites(0.000 sec), 48 matches)
```

図 4.11: CafeOBJ の変換結果

名称	機能	形式
ポリシー	拡張モジュールの利用ポリシーを記述	ポリシー記述用言語
モジュール管理機構	拡張モジュールのプラグインを制御する機構	Ruby プログラム
基本モジュール	CafeOBJ のサブセットの文法情報の定義	CafeOBJ コード
拡張モジュール	基本モジュールを基に拡張構文を定義	CafeOBJ コード

表 4.7: Espresso の構成一覧

表 4.7 にまとめる.

第5章　例題の作成

前章では，ops の拡張構文を例に拡張モジュールについて説明した．本章では実際に CafeOBJ には無い構文を拡張する例題について説明する．そこで，実際に銀行システムの仕様記述を例にいくつか拡張構文を追加している．

5.1　銀行システムの仕様の概要

ここでは，銀行システムの仕様を記述するために，まず，振舞仕様と射影演算について説明する．そして，振舞仕様による銀行システムの例を見ていく．そして，1 つ前の例題で作成した複数の述語を定義する拡張構文と，ここで拡張する構文を利用して拡張構文を用いた銀行口座の仕様記述を試みる．また，衝突の例を示すために PREDS モジュールに以下の拡張構文を追加する．

```
-- Bool 値以外を返す関数を定義するための構文
var I : Identifier . SS : SortIdSet . S : SortId .
op (act _ : _ -> _ .) : Identifier SortIdSet SortId -> Action .
op (act _ : -> _ .) : Identifier SortId -> Action .
-- act の変換ルール
eq translate-module-element((act I : SS -> S .), M:ModuleDecl) =
                              (op I : SS -> S .) .
eq translate-module-element((act I : -> S .), M:ModuleDecl) =
                              (op I : -> S .) .
```

これは，述語以外の操作を act と名付け，それを op に変換するだけの構文である．

5.1.1　振舞仕様

隠蔽代数 [4] にもとづいて記述される仕様を**振舞仕様**と呼ぶ．隠蔽代数ではシステムをブラックボックスとみなして内部の構造ではなく，外部からみた振舞いを扱う．

システムの状態空間は**隠蔽ソート**によって表現される．この隠蔽ソート上にシステムの状態を遷移させる**操作演算**(Action)，およびシステムの状態を観測する**観測演算**(Observation) が定義される．観測の結果は抽象データ型で表わされ，隠蔽ソートと明示的に区別する

```
mod* ACCOUNT {
  pr(INT) .
  *[ Account ]*
  bop balance : Account -> Nat . -- observation
  bops deposit withdraw : Nat Account -> Account . -- action
  var N : Nat .  var A : Account .
  eq balance(deposit(N, A)) = balance(A) + N .
  ceq balance(withdraw(N, A)) = balance(A) - N if N <= balance(A) .
  ceq balance(withdraw(N, A)) = balance(A) if balance(A) < N .
}
```

図 5.1: 銀行口座の振舞仕様

ために**可視ソート**と呼ばれる．このように隠蔽代数では，データを表わす可視ソートとシステムの状態を表わす隠蔽ソートの 2 種類のソートを扱う．操作演算と観測演算を**振舞演算**と呼ぶ．以下に，振舞仕様を用いた銀行口座の仕様を図 5.1 示す．

隠蔽ソートを定義するには，*[ソート名]*と記述する．例では，Account というソート名が隠蔽ソートになる．振舞演算を定義するには，bop で定義する．銀行口座の例では，balance を観測演算として定義し，deposit および withdraw を操作演算として定義している．振舞仕様では，操作演算によって状態がどのように変化するかを観測演算による観測結果により定義する．

また，振舞の等価性を等式で記述するために振舞等式が CafeOBJ に用意されている．CafeOBJ では beq を用いて等式を定義する．CafeOBJ による振舞仕様の記述の詳細については [10] を参照されたい．

5.1.2 射影演算

隠蔽代数として記述したシステムをいくつか集めて合成することにより，大きなシステムを記述することができる．いくつかのコンポーネントを合成して 1 つの合成されたシステムを合成システムと呼ぶ．また，合成システムの状態空間から個々のサブシステムへの状態空間に射影する操作を**射影演算**(Projection) と呼ぶ．CafeOBJ では振舞演算として定義する．

射影演算を使った例として 3 つの銀行口座を合成したシステムの仕様を図 5.2 に示す．それぞれの口座には，deposit と withdraw というアクションが定義されている．3 つの口座を合成した銀行システムは，それぞれの銀行口座のアクションを呼出す deposit1，deposit2，deposit3 と withdraw1，withdraw2，withdraw3 が定義されている．また，銀行システムから個々の口座の状態を取り出すために，account1，account2，account3 という射影演算

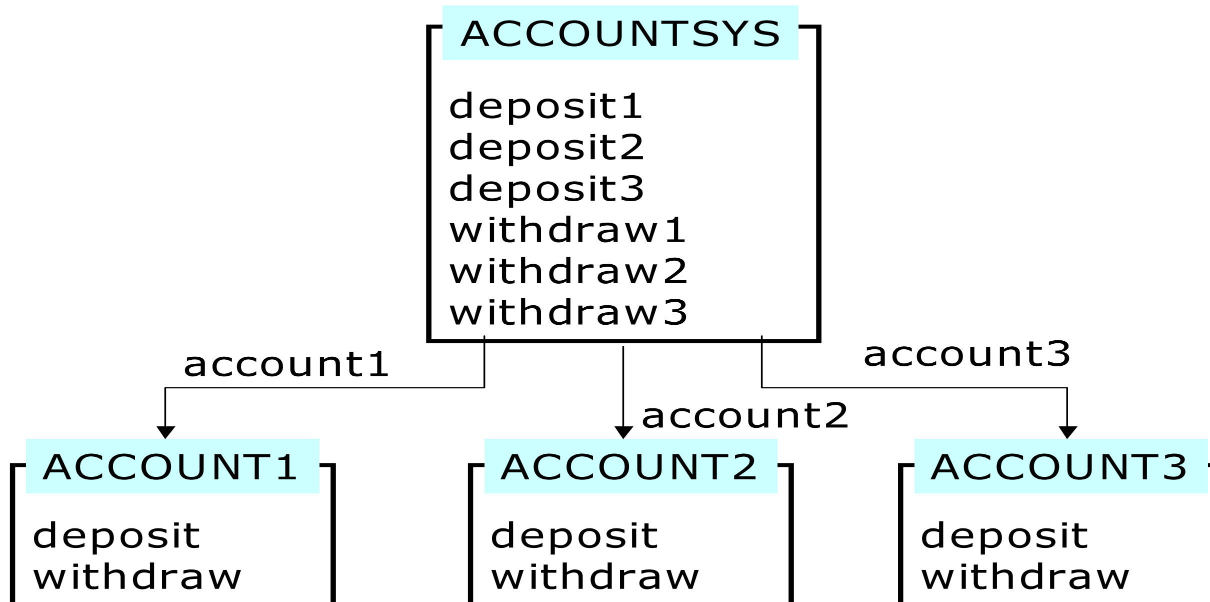

図 5.2: 3 つの銀行口座を合成したシステム

が定義されている．また，図 5.2 には無いが各銀行口座に関するなんらかの述語が複数定義されているとする．

この銀行システムはアクションが起きたときにそのサブコンポーネントとなる銀行口座の状態がどのように変化するかを等式で記述する．つまり，銀行システムに ACCOUNT1 の預金をおろすというアクションが起きたあとに，射影演算によりサブコンポーネントを取り出した状態は，ACCOUNT1 の口座の状態が変化することを記述する．等式は図 5.3 のようになる．

5.2 例題 1：複数の述語を定義する拡張構文

まずは，簡単な拡張構文を導入する．ここでは，複数の述語を 1 度に宣言する構文を定義する．例えば，銀行システムに対して複数の述語が定義されていた時に，1 つずつ並べて宣言する必要がある．そこで拡張構文を導入することで，1 度に宣言できるようにする．

CafeOBJ では，コアリティが Bool である関数をコアリティを省略して述語として定義することができる．実際には，pred 構文を用いて宣言する．この構文はシンタックスシュガーであり，基本モジュールからは取り除いている．そこで，まず述語を定義する構文を定義する．次に，複数の述語を定義するための拡張構文を定義する．ops 文では同じアリティとコアリティの関数を複数同時に定義する構文である．これを述語を定義する構文にも適用する．例えば，1 つの自然数を引数とする述語 f,g,h を定義する場合は，

```
var AS : AccountSys . var N : Nat .
-- 状態変化の記述
eq account1(deposit1(N, AS)) = deposit(N, account1(AS)) .
eq account2(deposit2(N, AS)) = deposit(N, account2(AS)) .
eq account3(deposit3(N, AS)) = deposit(N, account3(AS)) .
eq account1(withdraw1(N, AS)) = withdraw(N, account1(AS)) .
eq account2(withdraw2(N, AS)) = withdraw(N, account2(AS)) .
eq account3(withdraw3(N, AS)) = withdraw(N, account3(AS)) .
-- 状態が変化しないことを記述
eq account1(deposit2(N, AS)) = account1(AS) .
eq accoout1(deposit3(N, AS)) = account1(AS) .
eq account2(deposit1(N, AS)) = account1(AS) .
eq account2(deposit3(N, AS)) = account1(AS) .
eq account3(deposit1(N, AS)) = account1(AS) .
eq account3(deposit2(N, AS)) = account3(AS) .
eq account1(withdraw2(N, AS)) = account1(AS) .
eq accoout1(withdraw3(N, AS)) = account1(AS) .
eq account2(withdraw1(N, AS)) = account1(AS) .
eq account2(withdraw3(N, AS)) = account1(AS) .
eq account3(withdraw1(N, AS)) = account1(AS) .
eq account3(withdraw2(N, AS)) = account3(AS) .
```

図 5.3: 銀行口座システムの等式定義の一部

```
[ Predicate < OperationDecl ]
[ Predicates < OperationDecl ]
-- Bool 値を返す関数を定義するための構文
op (pred _ : _ .) : Identifier SortIdSet -> Predicate .
op (bpred _ : _ .) : Identifier SortIdSet -> Predicate .
--  複数の pred を 1 度に記述するための構文
op (preds _ : _ .) : IdSet SortIdSet -> Predicates .
op (bpreds _ : _ .) : IdSet SortIdSet -> Predicates .
```

図 5.4: 複数の述語を宣言する拡張構文の定義

```
pred f : Nat .
pred g : Nat .
pred h : Nat .
```

と定義する．また，f,g,h を 3 つ同時に定義するための拡張構文として，

```
preds f g h : Nat .
```

と定義する．

5.2.1 拡張モジュール PREDS の作成

複数の述語を宣言する拡張構文を定義する拡張モジュール PREDS の作成方法を説明する．CafeOBJ で述語を定義するために pred 構文がある．しかし，これは op 宣言で同じ事を宣言できるのでシンタックスシュガーである．そのため，基本モジュールで取り除いているので，pred，bpred 構文をまず作成する．次に複数の述語を一度に宣言する拡張構文 preds，bpreds を拡張する．また，前章の拡張モジュールを作成する例で定義した拡張モジュール OPS を拡張することで，インクリメンタルに拡張する．

拡張構文の定義

拡張構文を図 5.4 に示す．pred，bpred 構文は Predicate とし，preds，bpreds 構文は Predicates としそれぞれ OperationDecl のサブソートとして定義した．拡張構文の定義は，ops のように述語の名前の宣言部をシンボル列とすることで複数の宣言をするようにした．

```
op get-pred : ModuleElement -> ModuleElement { memo } .
op get-pred-in : ModuleElement -> ModuleElement { memo } .
eq get-pred(M:ModuleElement) = get-pred-in(get-op(M)) .
ceq get-pred-in(O:OperationDecl) = O if O :is Predicate .
ceq get-pred-in(O:OperationDecl) = no-elem if not(O :is Predicate) .
ceq get-pred-in(O:OperationDecl M:ModuleElement) = O get-pred-in(M)
  if O :is Predicate .
ceq get-pred-in(O:OperationDecl M:ModuleElement) = get-pred-in(M)
  if not(O :is Predicate) .
eq get-pred-in(no-elem) = no-elem .
```

図 5.5: 述語の宣言を得る get-pred

内部参照関数

内部参照関数は BASIC-SYNTAX で定義されている内部参照関数をいくつか利用して定義する．PREDS モジュールでは get-pred と get-preds 関数を定義した．pred 宣言を得るための内部参照関数 get-pred 関数では get-op を利用して記述する．内部参照関数は主に変換ルールを記述する際の意味解析を行うために定義するが，今回の例では内部参照関数を利用して変換ルールを記述しないので，述語を宣言したものを集める関数だけを定義した．図 5.5 に get-pred 関数の定義を示す．

変換ルール

まず，pred の変換ルールについて説明する．CafeOBJ で述語を表すのは，コアリティが Bool になる関数なので，pred が定義されたものを op に変換する．

```
pred f : Nat . -> op f : Nat -> Bool .
```

このように変換することで述語を定義する．複数の述語を定義するための変換ルールを定義する．この変換ルールは，前章の例で作成した ops 構文を利用して，変換ルールを記述する．ops 文を拡張構文とみなすことで，拡張構文を使ってさらに拡張構文を定義し，インクリメンタルな拡張を実現したといえる．このような拡張構文を使ってさらに拡張構文の変換ルールを記述する場合は，変換した構文に対しても変換関数を適用する必要がある．こうすることで，最終的には op の形に変換される．図 5.6 変換ルールを以下に示す．

pred を拡張構文とみなすと，preds は拡張構文を拡張している．これを記述するためには，等式の右辺に対しても translate-module-element 関数を適用することで記述できる．

```
var I : Identifier . var S : SortIdSet . var M : ModuleDecl .
-- pred の変換ルール
eq translate-module-element((pred I : S .), M)  = (op I : S -> Bool .) .
-- bpred の変換ルール
eq translate-module-element((bpred I : S .), M:ModuleDecl) =
                (bop I : S -> Bool .) .
-- preds の変換ルール
eq translate-module-element((preds IS:IdSet : S .),
      M:ModuleDecl) =
translate-module-element((ops IS : S -> Bool .), M) .
-- bpreds の変換ルール
eq translate-module-element((bpreds IS:IdSet : S .),
      M:ModuleDecl) = translate-module-element((bops IS : S -> Bool .), M) .
```

図 5.6: PREDS に定義した変換ルール

5.3 例題 2：射影演算の等式記述のための拡張構文

図 5.3 の仕様を見ると煩雑な部分がある．それは，射影演算を行った結果，状態が変化しないということも明示的に記述していることである．射影演算を用いた等式を記述する際に，状態が変化しないことを明示的に記述する必要がある．そのため，本来状態の変化が起こるところのみを記述するはずが，状態の変化が起こらないことも記述する必要あり，仕様が煩雑になる．状態が変化しない等式は，ある程度決まった形式で記述されるので，その部分は自動生成が可能である．そこで次に状態が変化する射影演算の等式だけを記述し，状態が変化しない等式を記述しないで済む拡張構文を定義する．そして，この拡張構文の変換ルールに状態が変化しない等式を生成するようにする．

5.3.1 拡張モジュール PROJ の作成

ここでは，拡張する構文について説明する．前節で説明したように例題として，状態が変化する射影演算を含む等式を定義するための構文を拡張し，その構文をもとに状態が変化しない等式を自動生成する．拡張モジュールの名前を PROJ とする．なお，ここでは次のような制約を設ける．

- 射影演算の最後の引数のソートは状態を表わす隠蔽ソート．
- 操作演算の最後の引数のソートは状態を表わす隠蔽ソート．

拡張構文	拡張構文の意味
act	射影演算に表われるアクションを定義する構文．op 文を拡張.
pro-eq pro-ceq	射影演算に関する等式を記述する構文．eq 文を拡張.

表 5.1: 射影演算を記述するための拡張構文

```
[ ProjectionOperation < OperationDecl ]
[ ProjectionEquation < EquationDecl ]
op (act _ : _ -> _ .) : Identifier SortIdSet SortId ->
                                ProjectionOperation .
op (pro-eq _ = _ .) : Term Term -> ProjectionEquation .
op (pro-ceq _ = _ if _ .) : Term Term Bool ->
                                ProjectionEquation { strat: (1 2) } .
```

図 5.7: 射影演算の拡張構文の定義

- 等式の左辺の状態を表わす引数は変数のみ．定数とパターンマッチによる記述は無い.
- 同じ名前の操作演算は定義してはならない.

拡張構文の定義

上記の制約を踏まえた上で射影演算を含む等式を記述するための拡張構文を定義する．拡張構文の定義は表 5.1 のように定義する．また，拡張モジュールでの定義は図 5.7 のようになる．

act 構文で射影演算の等式に現われるアクションを宣言する．act の構文を宣言しているところに，strat という属性がある．これは，条件式のところを書換えてはならないので，条件式の部分を書換えないことを明示的に記述している．pro-eq 構文は，射影演算の状態が変化する等式を記述するための構文である．pro-ceq 構文は，条件付きの射影演算の状態が変化する等式を記述するための構文である．この拡張構文を使った例を図 5.8 に示す．この例は先程示した銀行口座の例 (図 5.3) である．

内部参照関数

拡張構文の情報を参照するために，いくつかの内部参照関数を定義する．内部参照関数は主に変換ルールを記述する際の意味解析を行うために定義する．内部参照関数は PREDS で定義した get-pred と同じように BASIC-SYNTAX に定義してある内部参照関数を利用

```
bpreds predicate1 predicate2 predicate3 : Nat Account .
act deposit1 : Nat AccountSys -> AccountSys .
act withdraw1 : Nat AccountSys -> AccountSys .
...
var N : Nat . var AS : AccountSys .
pro-eq account1(deposit1(N,AS)) = deposit(N, account1(AS)) .  -- (1)
pro-eq account2(deposit2(N,AS)) = deposit(N, account2(AS)) .  -- (2)
pro-eq account3(deposit3(N,AS)) = deposit(N, account3(AS)) .  -- (3)
pro-eq account1(withdraw1(N,AS)) = withdraw(N,account1(AS)) . -- (4)
pro-eq account2(withdraw2(N,AS)) = withdraw(N,account2(AS)) . -- (5)
pro-eq account3(withdraw3(N,AS)) = withdraw(N,account3(AS)) . -- (6)
```

図 5.8: 拡張構文を使った銀行口座の仕様の一部

関数名	機能
get-act	act で定義した全てのアクションを得る．
get-pro-eq	pro-eq，pro-ceq で定義した全ての等式を得る．
get-action-name	act で定義した全てのアクション名を得る．
action-name	act で定義したアクション名を得る．
get-act-arity	act で定義したアリティを得る．

表 5.2: PROJ モジュールの主な内部参照関数

する．例えば，仕様に定義されている act 宣言を得るための内部参照関数を get-act 関数として定義する場合は，図 5.9 のように get-op を利用して記述する．

ProjectionOperation は OperationDecl のサブソートであるので，BASIC-SYNTAX に定義されている OperationDecl のサブソートを全て取り出す get-op 関数を利用する．そして，ProjectionOperation ソートの項を:is 関数で判定して，act 宣言されているものを全て得る．その他の PROJ モジュールに定義している主な内部参照関数を表 5.2 に示す．

変換ルール

実際に状態の変化がある等式から状態が変化しない等式を自動生成するための変換ルールについて説明する．変換ルールは，act，pro-eq，pro-ceq の 3 つを定義している．

act の変換ルール　本章ですでに説明したように，アクションは振舞操作により定義される．そこで，act の場合 bop に変換するルールを定義する．図 5.8 の deposit1 の場合は，

```
-- act 宣言をすべて得るための関数
op get-act : ModuleElement -> ModuleElement { memo } .
eq get-act(M:ModuleElement) = get-in(get-op(M)) .
-- get-act の補助関数
op get-in : ModuleElement -> ModuleElement { memo } .
ceq get-in(O:OperationDecl) = O if O :is ProjectionOperation .
ceq get-in(O:OperationDecl) = no-elem if not(O :is ProjectionOperation) .
ceq get-in(O:OperationDecl M:ModuleElement) = O get-in(M)
          if O :is ProjectionOperation  .
ceq get-in(O:OperationDecl M:ModuleElement) = get-in(M)
          if not (O :is ProjectionOperation) .
   eq get-in(no-elem) = no-elem .
```

図 5.9: get-act 内部参照関数の定義

```
eq translate-module-element((act I:Identifier : SS:SortIdSet ->
 S:SortId .), M:ModuleDecl) = (bop I : SS -> S .) .
eq translate-module-element((act I:Identifier : -> S:SortId .),
      M:ModuleDecl) = (op I : -> S .) .
```

図 5.10: act の変換ルール

```
bop deposit1 : Nat AccountSys -> AccountSys .
```

と変換される．変換ルールは図 5.10 のように定義している．

pro-eq の変換ルール　pro-eq の場合は若干複雑なので，図 5.8 をもとに変換ルールを説明していく．この例では，射影演算を記述する等式では，account1，account2，account3 の 3 つの射影演算がある．変換ルールでは account1 から始まる等式を変換する場合は，以下の手順で行う．

1. account1 で始まる等式を集める．
2. account1 の最後の引数のアクション名を集める．
3. act で宣言しているアクション名と 2 で集めたアクション名の差分を計算する．そうすることで，等式に明示的に現われないアクション名を集める．
4. 3 で得られた明示的に現われないアクション名に対して，状態が変化しないという等式を生成する．

```
var AS : AccountSys .
var N : Nat .
eq account1(deposit1(N, AS)) = deposit(N, account1(AS)) . -- (1)
eq account1(deposit2(G16:Nat, AS)) = account1(AS) .       -- (2)
eq account1(deposit3(G18:Nat, AS)) = account1(AS) .       -- (3)
eq account1(withdraw2(G20:Nat, AS)) = account1(AS) .      -- (4)
eq account1(withdraw3(G22:Nat, AS) =  account1(AS) .      -- (5)
...
```

図 5.11: 変換された銀行口座の仕様の一部

1では，pro-eq account1で始まる等式を集めてくる．例の場合だと，図5.8の(1)と(3)の等式である．2では，(1)と(3)のaccount1の最後の引数のアクション名を集める．つまり，withdraw1とdeposit1を集める．3では，actで宣言された差分を集めるので，{deposit1, deposit2，deposit3, withdraw1, withdraw2, withdraw3}から{deposit1, withdraw1}を取り除くと{deposit2，deposit3，withdraw2，withdraw3}が得られる．最後に，{deposit2, deposit3, withdraw2, withdraw3}に対しては，状態が変化しないという等式を生成する．このルールで変換された仕様を図5.11に示す.

図5.8の(1)の等式を変換した結果は，図5.11の(1)，(2)，(3)，(4)，(5)の5つの等式になる．3で得られた明示的に現われないアクション{deposit2, deposit3, withdraw2, withdraw3}のアクションを適用した射影演算は状態が変化していないという等式を生成している．状態が変わらないということは，あるアクションを適用したあとに射影演算をしてサブコンポーネントの状態を取り出したものは，アクションを適用する前の状態に対して射影演算によりサブコンポーネントの状態を取り出したものと同じことを意味する．変換ルールを記述したものを図5.12に示す．図5.12で重要なところはceqによる変換ルールを記述している等式である．ceqの条件部には，すべてのアクションが明示的に現れている場合とそうでない場合を場合分けしている．もし，すべてのアクションが現れている場合はpro-eqをeqに変換するだけである．そうでない場合は，上で述べた手順で状態が変化していない等式を生成する．これを行っているのがno-trans-state-eq関数である．この関数は1から4までの手順を行っている．この関数を定義したコードについては省略する.

pro-ceqの変換ルール　pro-ceqを変換する場合は，pro-eqで定義した変換ルールの他に，条件にnotを付けたものは状態が変わらないという変換ルールを追加する． 例えば,

```
pro-ceq account(U, add-account(U’, N, AS)) = deposit(N, init-account)
                                             if U == U’ .
```

```
-- 左辺が定数
eq translate-module-element((pro-eq I:Identifier = T:Term .),
       M:ModuleDecl)  =   (eq I = T .) .
-- 左辺の関数の引数が定数 or 変数
eq translate-module-element((pro-eq I:Identifier I':Identifier = T:Term .),
       M:ModuleDecl)  = (eq I(I') = T .) .
-- 左辺が n 引数関数
ceq translate-module-element((pro-eq I:Identifier IS:IdSet = T:Term .),
       M:ModuleDecl) = (eq I IS = T .)
  if diff-id(module-element(M), I) == null-id .
ceq translate-module-element((pro-eq I:Identifier IS:IdSet = T:Term .),
       M:ModuleDecl) = (eq I IS = T .)
                       no-trans-state-eq(I, IS, module-element(M))
  if not (diff-id(module-element(M), I) == null-id) .

ceq translate-module-element((pro-eq I:Identifier(A:Arity) = T:Term .),
       M:ModuleDecl) = (eq I A = T .)
  if diff-id(module-element(M), I) == null-id .
ceq translate-module-element((pro-eq I:Identifier(A:Arity) = T:Term .),
       M:ModuleDecl)  = (eq I A = T .)
                       no-trans-state-eq(I, A, module-element(M))
  if not (diff-id(module-element(M), I) == null-id) .
```

図 5.12: pro-eq の変換ルール

このような等式が定義されていた場合，次のように変換される．

```
ceq account(U, add-account(U', N, AS)) = deposit(N, init-account)
                                          if U == U' .
ceq account(U, add-account(U', N, AS)) = account(U, AS)
                                          if not ( U == U' ) .
...
```

変換された1つめの等式はpro-ceqで定義した等式をceqに変換しただけのものである．2つめの等式は条件が成り立たない場合は状態が変化しないという等式を生成している．それは，add-accountアクションのあとに射影演算によって得られた状態は，アクションをする前の状態に射影演算によって得られた状態と変わらない，つまり，状態が変化していないことを示している．その後は，pro-eqと同じように状態が変化していない等式を生成している．この変換ルールは図5.13に示す．

pro-ceqで定義された左辺の項が定数の場合，pro-ceqを単にceqに変換する．定数の場合は，何も状態変化を起こすためのアクションがないので状態が変化しないという項を作る必要がない．そのため，pro-ceqをceqに変換するだけとした．

pro-ceqの左辺が1引数の場合，さらに，2つの場合を考慮する必要がある．まず，射影演算の引数が定数もしくは変数の場合は，pro-ceqをceqに変更するだけである．これは定数の場合と同様に，状態変化を起こすアクションがないので，状態が変化しないという項を作る必要がない．射影演算の引数が定数もしくは変数でない場合は，状態を表す変数を取り出し，条件を否定した場合は状態が変化しないという等式を生成する．

pro-ceqの左辺が2引数以上の関数の場合，状態が変化しない項を生成する必要がある．PROJモジュールでは，状態が変化しない項を作り出すために，図5.13にあるstate-variable関数とmake-arity-ceq関数を変換するための補助関数として定義した．state-variable関数は状態変数を取り出すための関数であり，make-arity-pro-ceqは状態が変わらないという項を作り出すための関数である．状態が変わらないという項を作り出すには，最後の引数だけ状態を表わす変数もしくは定数にアクションをかけているものをとりはずした引数に変更したものを等式の右辺とする．上記の例では，accountは2引数の関数で，ASが状態を表わす変数である．条件を否定した場合は，add-accountアクションを取り除いて最後の引数をASだけにしたものを等式の右辺にしている．その他の引数についてはそのままである．

5.3.2 拡張構文の衝突

銀行口座の仕様を記述するために，PREDSに定義されているbpreds構文とPROJのact構文，pro-eq構文，pro-ceq構文を利用して記述した．そこで，ユーザは，拡張モジュールの利用ポリシーを以下のように記述した．

```
-- state variable を得る
op state-variable : Arity -> Identifier .
eq state-variable(I:Identifier) = I .
eq state-variable(IS:IdSet) = last(IS) .
eq state-variable(I:Identifier A:Arity) = state-variable(A) .
eq state-variable(I:Identifier, A:Arity) = state-variable(A) .
-- arity をつくる.
op make-arity-pro-ceq : Arity -> Arity .
eq make-arity-pro-ceq(T:Term) = state-variable(T) .
eq make-arity-pro-ceq(T:Term, A:Arity) = T, make-arity-pro-ceq(A) .
-- pro-ceq の変換ルール
-- 左辺が定数の場合
eq translate-module-element((pro-ceq I:Identifier = T:Term if C:Bool .),
      M:ModuleDecl) = (ceq I = T if C .) .
-- 左辺の関数の引数が定数 or 変数
ceq translate-module-element((pro-ceq I:Identifier IS:IdSet = T':Term
             if C:Bool .), M:ModuleDecl) = (ceq I IS = T' if C .)
                 (ceq I IS = I (last(IS)) if not C .)
  if diff-id(module-element(M), I) == null-id .
ceq translate-module-element((pro-ceq I:Identifier IS:IdSet = T':Term
             if C:Bool .), M:ModuleDecl) = (ceq I IS = T' if C .)
                 (ceq I IS = I (last(IS)) if not C .)
                  no-trans-state-eq(I, IS, module-element(M))
  if not (diff-id(module-element(M), I) == null-id) .
-- 左辺の引数が n 引数
ceq translate-module-element((pro-ceq I:Identifier(A:Arity) = T':Term
             if C:Bool .), M:ModuleDecl) = (ceq I(A) = T' if C .)
                  (ceq I(A) = I make-arity-pro-ceq(A) if not C .)
  if diff-id(module-element(M), I) == null-id .
ceq translate-module-element((pro-ceq I:Identifier(A:Arity) = T':Term
             if C:Bool .), M:ModuleDecl) = (ceq I(A) = T' if C .)
                  (ceq I(A) = I make-arity-pro-ceq(A) if not C .)
                              no-trans-state-eq(I, A, module-element(M))
  if not (diff-id(module-element(M), I) == null-id) .
```

図 5.13: pro-ceq の変換ルールの記述

```
import System$*
import Projection$PROJ
```

図 5.11 は等式を変換した結果である．act 構文を利用した部分の変換結果は以下のようになった.

```
op deposit1 : Nat AccountSys -> AccountSys .
op withdraw1 : Nat AccountSys -> AccountSys .
...
```

deposit1 と withdraw1 はアクションであり，CafeOBJ では振舞操作として宣言される必要がある．しかし，変換結果は op 文が出力された．これは，PROJ と PREDS モジュールにおいて，拡張構文 act が衝突しており，PROJ に定義されている act 構文の変換ルールが適用されず，PREDS に定義されている変換ルールが適用されている．実際に Espresso を実行すると，次のように衝突を検出し，警告を出力する.

```
warning!! conflict op ( act _ : _ -> _ . ) : at Projection$PROJ, System$PREDS,
```

つまり，拡張構文 act が 2 つの拡張モジュールにおいて衝突が起こり，意図しない変換ルールが適用された．そこで上で示したポリシーに次のことを記述することで衝突を回避する.

```
context * :: * :: (act _ : _ -> _ .) -> Projection$PROJ
```

この衝突の回避に関するポリシーを記述することで，ユーザの意図した通りに正しく変換された.

5.4 考察

ここでは，作成した 2 つの拡張モジュールに関する考察を行う．また，変換の処理時間についても言及する.

5.4.1 PREDS モジュール

複数の述語を定義するための拡張構文を定義した PREDS モジュールのサイズは以下のようになった.

- 全体のコード量：80 行
- 拡張構文を定義した部分：約 10 行
- 内部参照関数の定義した部分：約 30 行

- 変換ルールの記述：約 40 行

複数の述語を定義するための拡張構文は，前章で作成した OPS モジュールを利用して定義した．つまり，拡張モジュールを拡張することでインクリメンタルな拡張を実現した．今回は内部参照関数を定義したが，実際には変換ルールを記述する際には利用しなかった．そのため，述語を宣言した項を返すための内部参照関数のみを定義した．変換ルールについては，ops 構文で定義した方法とほぼ同じである．ただし，preds と bpreds 構文は等式の右辺に拡張構文を使った項を記述しているので，右辺に対しても変換関数を適用する必要がある．

5.4.2 PROJ モジュール

射影演算の等式を記述する拡張構文を定義した PROJ モジュールのサイズは以下のようになった．

- 全体のコード量：257 行
- 拡張構文を定義した部分：約 10 行
- 内部参照関数の定義した部分：約 100 行
- 変換ルールの記述：約 150 行

拡張構文の追加については図 5.7 非終端記号を表わすソート名と拡張構文を表わす op 宣言をするだけであり，容易に記述できる．これは，BNF を記述するのと変わらないので，CafeOBJ についてあまり詳しくない人でもすぐに拡張できる．本論文で作成した ops 文と射影演算のための拡張構文はモジュールのシグニチャや等式を宣言するための構文を拡張したが，モジュールを定義するための拡張構文を作成する際も ModuleDecl のサブソートとしてあるソートを宣言して，op 宣言で拡張構文を定義することにより，拡張構文を作成することができる．

内部参照関数は，BASIC-SYNTAX の内部参照関数を利用することで，少ないコード量で定義できる．BASIC-SYNTAX ではある非終端記号に属する項を集めるための関数が用意されているので，あとは，拡張構文が属する非終端記号の場合はその項を返すような等式を記述するだけで内部参照関数を定義できる．

変換ルールの記述は，act については意味解析をする必要もなく，項を変形させるだけなので単純な記述となった．pro-eq と pro-ceq の変換ルールを記述するために，仕様上で定義されている等式から使われていないアクションを検索する必要があるので，多少コード量が増えた．しかし，必要な操作は QIDSET と BASIC-SYNTAX にある程度定義してあるので，これらのモジュールに定義したライブラリ関数を利用することで，250 行程度の拡張モジュールになった．

射影演算を含む等式を記述するための拡張構文の例は，ops のように単なるマクロ的な処理とは異なり，意味解析も含めた変換であると言える．等式を自動生成するために，入力となる仕様を意味解析した．そして，状態が変化しないところを推測して状態が変化しないことを表現する等式を自動生成した．つまり，単なるマクロを処理するだけの前処理系ではなく，意味解析して拡張構文の変換ルールを記述できることを示したといえる．また，今回の例では幾つか制限を加えたが，その制限をなくした拡張モジュール定義可能であると考えている．

銀行口座の仕様を拡張構文で記述することで，コード量が約半分になった．これは，状態が変化しないという等式を記述するのが減ったためである．これにより，状態の変化に関する部分だけを等式に記述するので，仕様の可読性と保守性が向上した．また，今回の例では，意図的に拡張構文の衝突を実際に示し，Espresso がその衝突を適切に回避することについても確かめた．

5.4.3 パフォーマンス測定

図 5.8 の拡張構文を用いた銀行口座の仕様を変換する速度を計測した．その結果は以下のようなになった．

CPU	OS	メモリ容量	処理時間
Pentium Celeron 725MHz	Linux 2.4.18	384MB	4.479 秒
Ultra SPARC 400MHz	Solaris 8	256MB	7.55 秒

さらに細かく見ていくと，これは CafeOBJ システムを起動する時間が大半を占めている．仕様を構文解析する時間が 0.14 秒，仕様の変換する時間は 0.14 秒であり，モジュール管理機構が処理している時間は 1,2 秒程度であるので，起動する以外は対した時間がかからなかった．しかし毎回，拡張モジュールだけでなく基本モジュールも構文解析しているので，これについてはもう少し検討する余地がある． また，[3] に記載されているカウンターの例を拡張構文を用いて書き直した結果，同じ位の時間で変換処理が完了した．

第6章　関連研究

本章では，本研究と関連する研究について述べる．関連研究は代数仕様言語と前処理系の観点から言及する．また，本研究との比較について言及する．なお，ここでは研究の優劣を述べるのではなく，相補的な関係としての立場から述べる．

6.1　代数仕様言語

6.1.1　Maude

Maude[2] は SRI で Jose Meseguer 氏を中心に開発された代数仕様言語である．Maude は，並行書換え論理にもとづく意味論を導入することで，状態変化を代数仕様技術の流れの中で表現することを可能とした．また，Maude の特徴としては自己反映計算 (リフレクション) により Maude 自身の拡張をサポートしている点にある．自己反映計算を行なうことにより，書換え戦略等のメタレベルを記述することができる．実際に自己反映計算を実現するために，メタな書換えというものを導入し，メタレベルの計算を行なう．

メタな操作を行なうために，meta-reduce，meta-parse，meta-apply という操作が組込まれている．また，up 関数で 1 つ上のメタレベルへ移動し，down 関数でベースレベルへコンテキストを移動できる．例えば，メタレベルで 0 + s 0 を簡約する場合は，

```
Maude> (red meta-reduce(NAT, '_+_[{'0}'Nat, 's_[{'0}'Nat]]) .)
```

とすることで，メタレベルによる項の書換えが行われる．

Maude は Core Maude と Full Maude とがあり，Core Maude は自己反映計算を実現するための機構や書換えシステムになり，Full Maude は自己反映計算を用いて，パラメータ付きモジュールやモジュールの表現の定義などを行えるように構文を拡張している．また，オブジェクト指向モジュールを定義するための拡張構文を定義し，並行システムの仕様記述を行っている．

6.1.2　Larch

Larch[6] は OBJ2 等の代数仕様言語と同様な性質規範型の形式仕様言語である．Larch では 2 つのスタイルの仕様がある．これを 2 層 (two-tiered) と呼ぶ．2 層とは LSL(Larch Shared Language) と LIL(Larch Interface Language) である．LIL とは特定のプログラミ

ング言語の構文に従って仕様を記述する．LIL にはC，C++，ML などのためのインターフェース言語がある．LSL とはどのプログラミング言語に独立な代数仕様言語である．つまり，Larch のベースとなる部分である．この LSL によって LIL で記述された仕様は意味を与えられる．つまり，仕様の検証は最終的には，LSL の意味論に従って行なわれる．

6.2 前処理系

6.2.1 EPP

EPP[12] は産業技術総合研究所の一杉氏により開発された，Java 言語に拡張機能を追加できるようにする前処理系である．EPP ではプラグインと呼ばれるモジュールを前処理系に組込むことで，前処理系の機能を拡張できるという特徴をもつ．プラグインは mixin を採用することで差分的な拡張を実現している．また EPP は，Java 言語を拡張するための前処理系としてだけでなく，言語研究者による新しい言語機能実験の道具，拡張 Java の実装のためのフレームワーク，Java 言語のソースコード解析・変換ツールのためのフレームワークとして用いることもできる．

EPP では，構文の拡張だけでなく抽象構文木，型システム，名前空間などを扱う重要なクラスがすべて拡張可能な構造になっている．実際に，並列言語，スレッドマイグレーション，パラメータ付きクラスなどの幅広い範囲の言語拡張を行っている．

6.2.2 Mix Juice

Mix Juice [7] は mixin をベースとした差分的なモジュールシステムを Java 言語に適用したプログラミング言語である．差分ベースモジュールは，クラスとモジュールの機構を完全に分離し，その代わりにモジュールと差分プログラミングの機構を統一したモジュール機構である．個々のモジュールは mixin のように組み合わせることができるが，このとき，名前の衝突と実装欠損と呼ぶ現象が生じる．名前の衝突については，FQN をフィールド・メソッド名にも適用することで，名前の衝突の問題を解決している．

プログラムを実行する際には，プログラムのユーザが使用するモジュールを指定する．指定された複数のモジュールはトポロジカルソートによって順序付けられる．そして，その順番に従って差分を追加していく．

6.2.3 AspectJ

AspectJ[8] とはアスペクト指向プログラミング (AOP)[9] の考え方を取り込んだ Java 言語のための前処理系である．AOP とはプログラムの複数のモジュールに横断して現われる側面 (**aspect**) を別々に記述する．アスペクトはオブジェクトに起こるイベントを監視し，それを Aspect Weaver と呼ばれるプログラムで機械的に合成することにより最終的

なプログラムを得るという考え方である．

AspectJ では Java のプログラムにさまざまなタイミング (**join point**) を定義し，join point の前や後などといったタイミングを指定して他のアスペクトの処理を割り込ませることによって，プログラムを合成する．タイミングには名前，型，メソッドの起動時/終了時などを指定できる．また，実行時にオブジェクト単位でアスペクトの有無を制御できる．

6.3 関連研究との比較

本研究では，拡張モジュールの利用ポリシーを導入することで衝突の回避を実現した．具体的には，拡張構文の名前換えとコンテキストに応じた変換ルールの指定をポリシーに記述することで実現した．これは，他の研究にはない手法である．また，ポリシーの導入により，拡張構文を使った仕様とモジュールの利用に関する記述を分離することができる．これにより，仕様を変えずにポリシーを変えることで柔軟な変更が可能となる．

Maude は自己反映計算による拡張を実現している．これにより，構文の拡張だけではなく，書換え戦略等を拡張することができる．一方，本研究では，前処理系による拡張を実現している．ポリシーを導入することで衝突を回避している．また，Core Maude を本研究の基本モジュール，Full Maude を幾つかの拡張モジュールを組み合わせたものと考えることもできる．また，本研究ではポリシーの記述により拡張の衝突を回避することができ，さらに，コンテキストに応じた変換ルールの指定することができる．一方，Maude の場合，自己反映計算にもとづく拡張を行っており，一般に自己反映計算による拡張の衝突を検出し回避するのは困難である．しかし，Maude では様々な拡張を行なっているので，Maude による拡張を参考にして，Espresso を改良していきたい．Larch は Espresso を用いることで，LSL を基本モジュールで定義されている構文情報，LIL を拡張モジュールとして拡張構文と変換ルールを定義することで，CafeOBJ の上でも Larch とある程度同じことができる．EPP の場合，拡張の衝突を検出した場合は，安全に停止するか，差分を取り除くことができる．本研究では，拡張構文を取り除くのではなくコンテキストを導入し，コンテキストに応じて衝突した構文の変換ルールを指定できる．また，拡張構文の名前を変更することで，新しい構文を作ることもできる．また，本研究ではモジュールの名前をパッケージ機構の導入によりモジュール名の一意性を確保したが，Mix Juice で提案されているモジュール機構を参考にしたい．AspectJ は，本研究で提案したコンテキストを join point とみなすこともできる．AspectJ の場合，多様な場所を join point とすることができる．本研究では，コンテキストとは入力ファイルのある範囲というだけである．AspectJ を参考にして，コンキストに関するところをフィードバックしていきたい．

第7章　結論

本章では，本研究の結論を述べ，最後に今後の課題について述べる．

7.1　まとめ

本論文では，CafeOBJ の構文を容易かつインクリメンタルに拡張するための機構を提案し，その機構を拡張可能な前処理系 Espresso として設計・実装した．Espresso は拡張したい構文を独立したモジュール単位で定義し，モジュールを組合せることでインクリメンタルに構文の拡張を行った．Espresso では拡張モジュールを記述するのに CafeOBJ を用いた．それにより，CafeOBJ で CafeOBJ の構文を擬似的に拡張でき，また，CafeOBJ の強力なモジュールシステムを利用することもできた．また，拡張モジュールの利用ポリシーを導入することで，拡張構文間の衝突回避を実現した．そのために，コンテキストの概念を導入し，ユーザが拡張構の変換処理を柔軟に制御可能とした．

Espresso を実現するために，CafeOBJ の文法情報や拡張モジュールを作成するためのライブラリを定義した基本モジュールを作成した．これにより，拡張モジュールの作成を支援することができた．また，モジュール管理機構の導入により，拡張モジュールの衝突を検出し，ポリシーに従って衝突の回避を実現した．

さらに，本研究では Espresso を使っていくつか構文拡張の例題を作成した．例題は，CafeOBJ による銀行システムの仕様をもとに行い，拡張構文を利用して銀行システムの仕様を改訂した．その結果，仕様のサイズが半分程度になり，Espresso による構文拡張の有用性を示した．

7.2　今後の課題

本研究の今後の課題として，以下の点が挙げられる．

- より複雑な拡張の例題作成
  本論文では，プロジェクションを含む等式の自動生成するための構文を拡張した．この拡張構文を用いてさらなる拡張を行いたい．例えば，並行プログラムの仕様の記述のための拡張構文などより複雑な拡張を行うことで Espresso を改良していきたい．また，本論文では拡張モジュールを多くは示せなかったので，Espresso を用いて多くの例題を記述し，詳細な実験と考察を行う必要がある．特に，拡張モジュー

ルの数に関する実験を行い，本研究のアプローチの限界について考察する．さらに，これらの実験結果から得られた知見を Espresso の設計に反映させ，Espresso の改良を行う．

- パラメータ付きモジュールの利用
パラメータ付きモジュールはある種のテンプレートとみなすことができる．本研究で作成した拡張モジュールの例題ではパラメータ付きモジュールを利用していないが，これを用いてさらに柔軟な拡張が出来るのではないかと考えている．

- 本研究で提案したアーキテクチャの応用
Espresso は CafeOBJ の構文を拡張するための前処理系であるが，他の言語のプログラムの検証のためのツールへの応用や前処理系としての応用が可能ではないかと考えている．例えば，Java 用の拡張モジュールや OCL 用の拡張モジュールを用意し，Espresso で CafeOBJ に変換して検証を行うなど可能である．本論文で示した射影演算を含む等式を記述するための拡張構文の例題は意味解析も行っており，CafeOBJ やその他の言語のコード解析としてのツールにも応用できる．

  また，基本モジュールをあるターゲットの仕様記述言語もしくはプログラミング言語用のものを定義することで，Espresso を CafeOBJ 以外の言語の前処理系としても利用することが可能である．このように，本研究で提案した Espresso は CafeOBJ のための前処理系としてではなく，様々なところで応用できるのではないかと期待している．

謝辞

本研究を進めるにあたり終始御指導頂いた二木厚吉教授に深く感謝致します．天野憲樹助手，中村正樹助手には本研究について御指導と御助言をして頂きました．深く感謝致します．また，ゼミを通して有益な助言をしてくださった，緒方和博客員研究員をはじめ言語設計学講座の皆様に感謝致します.

参考文献

[1] A. V. Aho, R. Sethi, and J. D.Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] M. Clavel, F. Duran, S. Eker, P. Liincoln, N. Marti-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and programming in rewriting logic. Technical report, Computer Science Laboratory, SRI International, 1999.

[3] Razvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report*, Vol. 6 of *AMAST Series in Computing*. World Scientific, 1998.

[4] Joseph Goguen and Grant Malcolm. A hidden agenda. Technical Report CS97-538, Computer Science Laboratory, SRI International, 1999.

[5] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 2000.

[6] John V. GuTTag and James J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[7] Yuuji Ichisugi and Akira Tanaka. Difference-based modules: A class-independent module mechanism. In *Proc. of the ECOOP 2002*, LNCS 2374, pp. 62–88. Springer-Verlag, 2002.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of aspectj. In *Proc. of ECOOP 2001*, LNCS 2072, pp. 327–329. Springer-Verlag, 2001.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. of ECOOP 1997*, LNCS 1241, pp. 220–242. Springer-Verlag, 1997.

[10] Ataru T. Nakagawa, Toshimi Sawada, and Kokichi Futatsugi. *CafeOBJ Manual User's Manual – ver.1.4 –*. http://www.ldl.jaist.ac.jp/cafeobj/manual/manual.ps.gz.

[11] まつもとゆきひろ, 石塚圭樹. オブジェクト指向スクリプト言語 Ruby. アスキー出版局, 1999.

[12] 一杉裕志. 拡張可能 java プリプロセッサ epp の型チェック機構フレームワーク. In *Systems for Programming and Application '99*, 1999.

[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] 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. 4–22, 1996.

[15] 二木厚吉, 外山芳人. 項書換え型計算モデルとその応用. 情報処理, Vol. 24, No. 2, pp. 133–146, 1983.

付 録A　Espressoの使用方法

ここでは，本研究で作成したEspressoのインストール方法と使用方法について説明する．Espressoは以下のURLから入手できる．

http://www.jaist.ac.jp/~y-asaba/espresso

A.1　ファイル構成

Espressoは，表A.1に示したファイル構成となっている．Espressoのソースコードだけでなく，本研究で作成した拡張モジュールと拡張構文を利用した仕様の例もセットにしておく．拡張モジュールを作成する際には，基本モジュールや本研究で作成した拡張モジュールのコードを参考にしてもらいたい．

A.2　動作環境

Espressoを実行するために以下の環境が必要である．それぞれのシステムのインストールについては，他の資料を参照されたい．

- Ruby
  バージョン1.6.x以降のもの
- Racc
  Ruby用のパーサジェネレータ
  http://www.ruby-lang.org/raa/list.rhtml?name=racc
- CafeOBJ
  http://www.ldl.jaist.ac.jp/cafeobj/index.html

OSについてはSolarisとLinuxでの動作確認した．Windows系については，確認していないがEspressoに必要なシステムはWindows上にも移植されているので動くのではないかと思われる．

dependchecker.rb	モジュールの依存関係の解決など.
module.rb	モジュール情報のクラス.
policy.rb	ポリシーファイルのパーザ.
espresso	Espresso のメイン.
reduce.rb	CafeOBJ システムとパイプで通信.
tsort.rb	トポロジカルソートモジュール.
result-parser.rb	CafeOBJ の結果から変換したモジュールを取り出す.
connect-cafe.rb	変換した仕様を CafeOBJ に渡す関数.
moduleparser.y	拡張モジュールのパーザ.
policyparser.y	ポリシーファイル用のパーザ.
resultparser.y	CafeOBJ からの結果を解析.
Makefile	インストールするための Makefile.
README	利用規約など様々な情報を記したファイル.
MEMO	Espresso の開発メモ.
example/*	本研究で作成した拡張構文を利用した仕様の例題.
module/*	基本モジュールと本研究で作成した拡張モジュール.

表 A.1: ファイル一覧

A.3 インストール

あらかじめ，Ruby と CafeOBJ へパスを通しておく必要がある．上の URL から espresso.tar.gz をダウンロードし，適当なディレクトリにファイルを展開する．あとは以下のコマンドを入力すれば完了である．

```
% tar zxvf espresso.tar.gz
% make
% make install
```

A.4 使用方法

変換された仕様は標準出力に出力されるので，オートロードするなりリダイレクトでファイルに書き込む必要がある．

A.4.1 コマンドオプション

本システムはコマンドラインをベースとしたインターフェースである．起動するには次のようにする．

コマンドオプション	説明
-p, –policy	ポリシーファイルの指定.
-m, –modpath	モジュールを検索するパスの指定.
-d, –dump	変換過程をファイルに出力．出力ファイルは espresso.output.
-a, –autoload	変換した仕様を自動で CafeOBJ にロード.
-s, –showsyntax	利用可能な構文の表示.
-c, –checksyntax	構文の衝突のチェック.
-v, –version	Espresso のバージョンの表示.
-h, –help	ヘルプの表示.

表 A.2: コマンドオプション

espresso [コマンドオプション] 変換するファイル 1 変換するファイル 2 ...

起動するときにコマンドオプションを指定することができる．コマンドオプションは表 A.2 にある．

A.4.2 環境変数

Espresso では拡張モジュールを検索する際に，MODLIBPATH という環境変数を参照する．デファルトではカレントディレクトリだけが検索パスになる．bash の場合では，以下のように環境変数を定義する．

```
% export MODLIBPATH=/usr/local/lib/epsresso:~/lib/espresso
```

A.5 サンプルの実行

5 章で説明した銀行システムを例にサンプルの実行方法を説明する．銀行システムの仕様は example/account3 にある．また，ポリシーファイルは，policy/projection.policy に置いてある．これを実行するには,

```
% espresso -p policy/projection.policy account3
```

とする．オプション'p' はポリシーファイルを指定している．espresso はこの結果を標準出力に出すので，もし，ファイルに書き込みたい場合は,

```
% espresso -p policy/projection.policy account3 > account3.mod
```

と，リダイレクトをする．