平成30年度 修士論文

文脈指向プログラムの検証法の研究

東京工業大学 情報理工学院
数理・計算科学系 数理計算科学コース

学籍番号 17M30329

夏目 敦之

指導教員

増原 英彦

平成31年2月1日

概要

近年のソフトウェアは，求められる役割の増大や，センサーを使った自律的な動作の実現などに伴って，複雑化している．そのためメンテナンスの容易さや拡張性の高さが求められる中で，ソフトウェアの状態や，ソフトウェアを取り巻く環境など，状況によって変わるような振る舞いを分離することでモジュール性を高めたのが文脈指向プログラミングである．一方でソフトウェアには安全性も求められる．そのためプログラムが特定の性質を満たすことを数理的手法に基づいて証明するソフトウェア検証の技術が注目されている．一般にソフトウェアの検証にはプログラム全体の実装を参照し，その解析を行うことが必要である．しかしライブラリ内の関数など，プログラムの実装が見られないこともしばしばである．またプログラムが変更されるたびに改めて検証するとなるとコストもかかってしまう．したがって実装を見ずにインターフェースのみを参照して行う，モジュラーな検証が求められる．モジュラーな検証を行うアプローチとしては，オブジェクト指向プログラムの検証などに用いられる behavioural subtyping が存在する．この手法では子オブジェクトに対して，型として親クラスを継承するのみでなく，親クラスの振る舞いを変えないことも要求する．するとプログラム中の親クラスの出現を子クラスで書き換えても，プログラムの振る舞いが変わることがない．しかし文脈指向プログラムにおいては，この手法と同様にモジュールの振る舞いに制約を与えると，文脈の変化に伴い振る舞いを大きく変えることのできる文脈指向プログラミングの利便性が大きく損なわれることとなる．本研究では，Abstract predicate family を用いてメソッドの振る舞いを抽象化することで，自由度を残しつつ，文脈指向プログラムのモジュラーな検証の実現を目指した．また FVC#（Featherweight Verified C#） を拡張することで，言語の形式化を行った．また実際に簡単な文脈指向プログラムの検証を行うことで，その有効性の評価を行う．

謝辞

本研究を行うにあたり，ご指導いただいた増原英彦教授に心より感謝いたします．また日頃から研究テーマについてのアドバイスや，研究を進める上で欠かせない基礎知識などを教えてくださった青谷知幸助教に深く感謝いたします。また多くの時間を共にし，時には様々なアドバイスをくださった研究室メンバーの皆様にもこの場を借りてお礼申し上げます.

目 次

表 目 次

ソースコード目次

第1章　序論

近年のソフトウェアは，求められる役割の増大や，センサーを使った自律的な動作の実現などに伴って，複雑化している．そのためメンテナンスの容易さや拡張性の高さが求められる中で，ソフトウェアの状態や，ソフトウェアを取り巻く環境など，状況によって変わるような振る舞いを分離することでモジュール性を高めたのが文脈指向プログラミングである．一方でソフトウェアには安全性も求められる．そのためプログラムが特定の性質を満たすことを数理的手法に基づいて証明するソフトウェア検証の技術が注目されている．一般にソフトウェアの検証にはプログラム全体の実装を参照し，その解析を行うことが必要である．しかしライブラリ内の関数など，プログラムの実装が見られないこともしばしばである．またプログラムが変更されるたびに改めて検証するとなるとコストもかかってしまう．したがって実装を見ずにインターフェースのみを参照して行う，モジュラーな検証が求められる．

モジュラーな検証を行うアプローチとしては，オブジェクト指向プログラムの検証などに用いられる behavioural subtyping が存在する．この手法では子オブジェクトに対して，型として親クラスを継承するのみでなく，親クラスの振る舞いを変えないことも要求する．するとプログラム中の親クラスの出現を子クラスで書き換えても，プログラムの振る舞いが変わることがない．しかし文脈指向プログラムにおいては，この手法と同様にモジュールの振る舞いに制約を与えると，文脈の変化に伴い振る舞いを大きく変えることのできる文脈指向プログラミングの利便性が大きく損なわれることとなる．

本研究では，Abstract predicate family を用いてメソッドの振る舞いを抽象化することで，文脈指向プログラムのモジュラーな検証の実現を目指した．また FVC#（Featherweight Verified C#） を拡張することで，言語の形式化を行った．また実際に簡単な文脈指向プログラムの検証を行うことで，その有効性の評価を行う．

1.1　本論文の構成

第 2 章では，本論文を読む上で必要な背景知識として，文脈指向プログラミング，modular reasoning，behavioral subtyping，$FVC^{\#}$ について説明する．第 3 章では，本研究で解決した技術的な問題について例を交えて述べ，それを解決する手法を提案する．第 4 章では，本研究で提案する言語の形式化を行う．第 5 章では，第 3 章で挙げた問題が，提案手法により解決できることを示す．第 6 章では，本研究と関連の深い研究について述べる．第 7 章では，結論と今後の課題について述べる．

第2章　背景知識

この章では，本研究を理解する上で必要となる背景知識について説明する．

2.1　文脈指向プログラミング

文脈指向プログラミング (Context-Oriented Programming: COP) [8] は，プログラムの実行時の文脈に依存する振る舞いをモジュール化するためのプログラミング手法である．文脈指向プログラミング言語として，ContextJ [3]，ContexL [7]，JCop [4]，などが知られている．実行時の文脈とはプログラム実行時の計算機の状態（例：バッテリー残量，ネットワーク状態）や，ソフトウェアの状態などを指す．

COP 言語は次のような言語要素を持つ．

- pertial method（部分メソッド）
- layer（レイヤー）
- layer activation（層活性）

部分メソッドは，文脈に依存するメソッドの振る舞いを実装するメソッドである．同じ文脈で呼び出される部分メソッドはレイヤーというモジュールでまとめられる．層活性は，レイヤーの集合を動的に変化させることで文脈を指定する仕組みである．文脈指向プログラミングでは，プログラム実行時に指定されたレイヤーにおける部分メソッドの実装を参照することで，文脈に依存した振る舞いをディスパッチする．

ソースコード 2.1: COP プログラムの例 Banking system

```
1  class Account {
2    float balance;
3
4    void credit(float amount) {
5      balance = balance + amount;
6    }
7  }
8
9  class Encryption {
10   float key = 42.4711f;
11   float key2 = 45047028;
12
13   float encrypt(float val) {
14     return (val * key) / key2;
15   }
16   float decrypt(float val) {
17     return (val * key2) / key;
```

```
18    }
19  }
20
21  layer EncryptionLayer {
22    class Account {
23      void credit(float amount) {
24        proceed(Encryption.decrypt(amount));
25      }
26    }
27  }
```

ソースコード 2.1 は，文脈指向プログラムの例である．クラス Account にはメソッド credit が定義されており，またフィールドに貯金残高を表す balance を持つ．クラス Encryption は送金のやり取りの暗号化を目的としたメソッドをもち，平文を encrypt で暗号化し，暗号文を decrypt で復号する．

credit メソッドは，レイヤー Encryption でも部分メソッドとして定義されている．

このプログラムのメソッドは以下のように呼び出される．

ソースコード 2.2: Banking system の呼び出し

```
1   a = new Account()
2   with(Encryption) {
3     a.credit(300f)                    (1)
4   }
5
6   with(Encryption) {
7     without(Encryption){
8       a.credit(12000)                  (2)
9     }
10  }
```

ここで with(){...} は層活性を実現する言語要素であり，指定されたレイヤーをそのスコープ内で活性化させる．without(){...} は逆に指定されたレイヤーを非活性化させる．メソッド呼び出し式を評価する際は，そのスコープでのレイヤーの活性状態を参照し，活性されたレイヤーで宣言された部分メソッドがもとのメソッドの代わりに呼び出される．したがって (1) の呼び出しでは，Encryption の credit() が呼び出される．また Encryption の credit の定義に現れている proceed() 呼び出しは，Encryption 自身を非活性にした時の，同じ credit メソッドを呼び出す．ここではアクティブなレイヤーはただ 1 つであるので，proceed() では，Account クラスで定義された credit メソッドが呼び出される．

よって (1) の呼び出しは，引数 amount が暗号化されており，それを復号してから balance に足すという振る舞いをする．

(2) では Encryption を再び非活性にしているので．通常の Accout クラスの credit が呼び出される．

このプログラムでは，暗号化に関する記述を Account クラスから分離できているので，Account クラスの拡張の際に Encryption に関わる記述を織り込む必要がない．このように一般的にプログ

ラム中の様々なモジュールに横断的に存在してしまうような記述を，モジュール化し分離できるのが大きな特徴である．

また層活性を実現することで，文脈に応じて，対応する部分メソッドを持つ全てのモジュールの振る舞いを一律に，動的に変化させることができる．

2.2 modular programming

modular programming [18] とは，プログラムを独立したモジュールと呼ばれる部分に分離することで，プログラムの設計，検証，変更の難しさを削減する手法である．プログラムの振る舞いを，個々のモジュールの振る舞いから導くことができるために，プログラマは個々のモジュールについて独立して考えることができる．

例えば Java をはじめとするオブジェクト指向言語は，プログラムをオブジェクトと呼ばれるモジュールに分割する．またオブジェクトの振る舞いを定義するインターフェースを持ち，各モジュールは対応するインターフェースの定めるプロパティに従う．

2.3 modular reasoning

modular reasoning とは，それぞれのモジュールについて「モジュール自身の実装」と,「実装しているインターフェース」と,「実装やインターフェースで参照されているインターフェース（とそのインターフェースから推移的に参照される全てのインターフェース）」のみを参照して検証することである．一般にプログラムを検証したのちに，そのプログラムに修正や追加を施したならば，プログラム全体を再検証する必要がある．追加や修正された記述がプログラムのどこに影響を与えるかは実装を見なければわからないからである．modular reasoning が可能ならば，新しいモジュールが追加されたときにその他の既にあるコードの再検証をする必要がない．

例えば，モジュール A を含むプログラムがすでに検証されている時に，モジュール B を追加することを考える．ここでモジュール B がモジュール A と同じ振る舞いをすることさえ示せれば，モジュール A をモジュール B に置換してもよい，すなわちモジュール B の追加はプログラム全体の振る舞いを変えないことが示せる．

2.4 分離論理

分離論理 [13] [12] [17] はホーア論理の拡張で，コンピュータメモリへのアクセスやデータの書き換えを含むプログラムの検証に主に用いられる．排他的に分割された 2 つのメモリ領域でそれぞれ P および Q が成り立つことを主張する論理式 P * Q（Separating conjunction）を持ち，分離されていることによる様々な modular reasoning のための規則を提供する．以下は分離論理の規則の一部である．

$$\frac{\{P\}C\{Q\}}{\{P * R\}C\{Q * R\}} \text{ (Frame Rule)}$$

$$\frac{\{P\}C\{Q\}}{\{\exists x.P\}C\{\exists x.Q\}} \text{ (Auxiliary Variable Elimination)}$$

$$\frac{\{P\}C\{Q\}}{(\{P\}C\{Q\})[E_1/x_1, ..., E_k/x_k]} \text{ (Variable Substitution)}$$

$$\frac{P' \Rightarrow P \quad \{P\}C\{Q\} \quad Q \Rightarrow Q'}{\{P'\}C\{Q'\}} \text{ (Rule of Consequence)}$$

2.5 behavioural subtyping

Behavioral subtyping [9] [10] [11] は，オブジェクト指向プログラムにおける modular reasoning のための技術である．

オブジェクト指向プログラミングでは，子オブジェクトは親オブジェクトの代替として渡すことができる．Behavioral subtyping はこれら子クラスの振る舞いを，親クラスの振る舞いを詳細化（refinement）したもの，すなわち親クラスよりさらに制限された振る舞いであることを要求する．このとき，子クラスの振る舞いは，親クラスの振る舞いの behavioural subtype であるという．

仕様が behavioural subtyping 関係にあることは以下のように表現する．またその証明は以下のように記述される．

$$\vdash \{P_1\}\_\{Q_1\} \Rightarrow \{P_2\}\_\{Q_2\}$$

表 2.1: behavioural subtyping の証明

$$\frac{\vdash \{P_1\}\_\{Q_1\}}{\begin{array}{c}\vdots\end{array}}$$
$$\vdash \{P_2\}\_\{Q_2\}$$

親クラスのメソッドが $\{P_1\}\_\{Q_1\}$ なる仕様を満たし，子クラスが上書きするメソッドが $\{P_2\}\_\{Q_2\}$ なる仕様を満たすとき，2.1 が証明されたとする．このとき子オブジェクトの振る舞いは親オブジェクトの振る舞いを満足しているので，プログラム中の親オブジェクトの出現を，子オブジェクトで置き換えても，プログラム全体の振る舞いが変わることはない．

2.6　$FVC^{\#}$

$FVC^{\#}$ [14]（Featherweight Verified $C^{\#}$）は，Parkinson らによって形式化された，検証された，メソッドの継承を含むオブジェクト指向言語である．$FVC^{\#}$ では分離論理に基づき検証が行われる．

ソースコード 2.3: Cell と Recell [14]

```
1  class Cell {
2    int val;
3
4    void set(int x)
5    {this.val ↦ _}_{this.val ↦ x}
6    {
7      this.val=x;
8    }
9
10   int get()
11   {this.val ↦ x}_{this.val ↦ x ∗ ret = x}
12   {
13     return this.val
14   }
15 }
16
17 class Recell: Cell {
18   int bak;
19
20   void set(int x)
21   {this.val ↦ y ∗ this.bak ↦ _}_{this.val ↦ x ∗ this.bak ↦ y}
22   {
23     this.bak = base.get();
24     base.set(x);
25   }
26
27   int get()
28   {this.val ↦ x ∗ this.bak ↦ o}_{this.val ↦ x ∗ this.bak ↦ o ∗ ret = x}
29
30   void undo()
31   {this.val ↦ y ∗ this.bak ↦ o}_{this.val ↦ o ∗ this.bak ↦ _}
32   {
33     int tmp = this.bak;
34     base.set(tmp);
35   }
36 }
```

$FVC^{\#}$ による検証の具体例として，ソースコード 2.3 を考える．この例では継承関係にある 2 つのクラス Cell と Recell が定義されている．Cell クラスはフィールドに値を持ち，set，get メソッドによって値の参照と取得ができる．Recell クラスは Cell クラスの子クラスで．上書き定義された set メソッドは値を格納する際に前の値を捨てず，bak に格納する．undo メソッドを呼び出すことで bak に格納した以前の値を val に再び格納する．get メソッドは再定義されていないが，フィールド bak が変更されないことが仕様に書き加えられている．

各メソッドには青字で仕様が与えられている．ただし，$x.f \mapsto y$ は，オブジェクト x がフィールド f をもち，その値が y であることを表す．$x \mapsto \_$ は $\exists y.x \mapsto y$ を表す．ここで Cell と Recell のそれぞれの set メソッドに与えられた仕様に関して，これは behavioural subtyping 関係にはない．す

なわち

$$\vdash \{this.val \mapsto y * this.bak \mapsto \_\}\_\{this.val \mapsto x * this.bak \mapsto y\} \Rightarrow \{this.val \mapsto \_\}\_\{this.val \mapsto x\} \quad (2.1)$$

は成り立たない．しかし Recell クラスのオブジェクトは Cell オブジェクトと同じ振る舞いをし，常に Cell オブジェクトの替わりに与えることができる．このことは以下の議論からもわかる．クラス Recell ではメソッド undo を呼び出さない限り，フィールド bak が参照されることはない．メソッド get はフィールド bak の記述を除いて Cell のものと同じ仕様を持つ．またメソッド set は，フィールド val の値が y で具体化されている点で異なるが，これは Cell の仕様を洗練させたものと言える．

$FVC^{\#}$ では，Recell クラスのような異なる仕様が与えられているものの，同じ振る舞いをするような子クラスについて，behavioural subtyping が言えるようにするために，仕様を抽象化する abstract predicate family を導入する．

2.6.1 Abstract Predicate Family

Abstract predicate family [15] は，$FVC^{\#}$ で用いられる，親クラスと，異なるフィールドを持つ子クラスとの間にも behavioural subtyping が言えるようにする仕組みである．2.5 節で述べた通り，継承を含むオブジェクト指向プログラムのモジュラーな検証においては，親クラスと子クラスそれぞれのメソッドの間に厳密な behavioural subtyping 関係が存在する必要がある．Abstract predicate family は仕様を抽象化することで，behavioural subtyping の関係を拡張する．

Cell と Recell の例では，メソッド set は abstract predicate family Val を用いて以下のように書き換えられる．

$$\{Val(\texttt{this}, \_)\}\ Cell.set(n)\ \{Val(\texttt{this}, n)\} \quad (2.2)$$

$$\{Val(\texttt{this}, X, \_)\}\ Recell.set(n)\ \{Val(\texttt{this}, n, X)\} \quad (2.3)$$

両者はどちらも同じ名前の述語 Val を使って書かれている．

述語 Val はプログラム中で以下のように定義される．

$$\texttt{define}\ \ Val_{Cell}(x, y)\ \texttt{as}\ \ x.val \mapsto y \quad (2.4)$$

$$\texttt{define}\ \ Val_{Recell}(x, y, z)\ \texttt{as}\ \ (x.val \mapsto y * x.bak \mapsto z) \quad (2.5)$$

(2.2) と (2.3) で用いられる添字なしの述語を abstract predicate family と呼ぶ．(2.4) と (2.5) で用いられる添字付きの述語は abstract predicate family の entry と呼ぶ．entry は，名前と定義とスコープをもつ．つまり (2.4) は，Val_{Cell} が abstract predicate family Val の entry であることと，添字で示

されたクラス Cell において，entry は $x.val \mapsto y$ と定義されることを示している．このとき Val は第 1 引数としてインスタンスパラメータを受け取っており，そのインスタンスの動的型によって，対応する entry が選択される．すなわち abstract predicate family Val は以下を満たす．

$$x : Cell \Rightarrow (Val(x, y) \ \Leftrightarrow x.val \mapsto y) \tag{2.6}$$

$$x : Recell \Rightarrow (Val(x, y, z) \ \Leftrightarrow x.val \mapsto y * x.bak \mapsto z) \tag{2.7}$$

またアリティの異なる述語を同一視するための規則を定義する．述語に定義より多い引数を与えた時，その引数は無視される．また少ない引数を与えた時，欠けている引数は存在量化子付きで与えられる．これは子クラスで新たに加えられたフィールドに関する仕様を無視できることに対応する．

$$Val(\overline{x}) \Leftrightarrow \exists v.Val(\overline{x}, v) \tag{2.8}$$

以上のように抽象化を施すと，(2.2) と (2.3) との間の behavioural subtyping は以下のように示せる．

$$\dfrac{\dfrac{\vdash \{Val(\texttt{this}; X, \_)\}\_\{Val(\texttt{this}; n, X)\}}{\vdash \{Val(\texttt{this}; \_, \_)\}\_\{Val(\texttt{this}; n, \_)\}}\ \text{Consequence}}{\vdash \{Val(\texttt{this}; \_)\}\_\{Val(\texttt{this}; n)\}}\ \text{Auxiliary Variable Elimination と (2.8)}$$

このようにメソッドの仕様を abstract predicate family によって抽象化することで，厳密な behavioural subtyping 関係にないメソッドも，置き換え可能であることを検証することができる．

2.6.2　静的仕様と動的仕様

FVC# では全てのメソッドには，静的仕様と動的仕様の 2 つの仕様が与えられる．静的仕様は，そのメソッドの振る舞いを正確に示したものである．動的仕様は，behavioural subtyping の議論するための抽象化された仕様である．前者はメソッド本体の実装の検証とメソッドが直接呼び出される場合の検証に用いられる．すなわち，動的にディスパッチされる場合でなく，対応するメソッドが静的に明らかな場合に静的仕様が利用される．後者は動的メソッド呼び出し式の検証に用いられる．

クラス Recell の場合，メソッド set の静的仕様と動的仕様は，それぞれ以下のように与えれられる．

$$\texttt{static}\ \{this.val \mapsto y * this.bak \mapsto \_\}\_\{this.val \mapsto x * this.bak \mapsto y\} \tag{2.9}$$

$$\texttt{dynamic}\ \{Val(\texttt{this}; y, \_)\}\_\{Val(\texttt{this}; x, y)\} \tag{2.10}$$

ここで静的仕様 (2.9) は，動的仕様 (2.10) 中の abstract predicate family を，this が Recell 型のオブジェクトであるとして開いたものになっている．このように実際の利用では多くの場合，静的仕様は abstract predicate family の定義に従って，動的仕様から自明に導くことができる．

また静的仕様のみが与えられた時，動的仕様は静的仕様と同じであるとみなしてよい．

したがって本論文でも自明な場合，各メソッドには一方の仕様のみを与える．

第3章　問題と提案

3.1　問題

本研究の目的は，文脈指向プログラムの modular reasoning である．この章ではその上で問題となることとを述べる．

文脈指向プログラミングにおいて，部分メソッドにはクラスでもともと定義された基底メソッドと異なる振る舞いをすることが許されている．例えば以下のソースコード 2.1 ではメソッド credit は，基底レイヤーと Encryption レイヤーとで明らかに違う振る舞いを見せる．

ソースコード 3.1: Bank Account の例

```
1  class Account {
2    float balance;
3
4    void credit(float amount)
5    {this.balance -> b}_{this.balance -> b + amount}
6    {
7      balance = balance + amount;
8    }
9  }
10
11 class Encryption {
12   float key = 42.4711f;
13   float key2 = 45047028;
14
15   static float encrypt(float val) {
16     return (val * key) / key2;
17   }
18   static float decrypt(float val) {
19     return (val * key2) / key;
20   }
21 }
22
23 layer EncryptionLayer {
24   class Account {
25     void credit(float amount)
26     {this.balance -> b}_{this.balance -> b + Encryption::decrypt(amount)}
27     {
28       without(EncryptionLayer){
29         credit(decrypt(amount));
30       }
31     }
32   }
33 }
```

これらのメソッドに仕様を与えたい．ただし，encrypt decrypt メソッドは static メソッドとして定義されており，Encryption::encrypt Encryption::decrypt でそれぞれを static メソッドとして呼び出している．

まずはソースコード 2.1 のように各々の部分メソッドに対してそれぞれ独自の仕様を与える．このとき 2 つの振る舞いは明らかに behavioural subtyping の関係にないことがわかる．ここで文脈指向プログラムにおけるメソッド呼び出しは， オブジェクト指向プログラムにおけるメソッド呼び出しと同様に，動的にディスパッチされるものである．すなわちレシーバーの型によってメソッド呼び出し式の振る舞いが異なるように，呼び出し時の文脈によって呼び出し式の振る舞いが異なる．このことから，すでに定義された同名メソッドの behavioural subtype でないような部分メソッドの追加を許すと，modular reasoning が実現できないことがわかる．例えば新たなレイヤーの追加に伴って，部分メソッド credit が新たに定義された時，全ての credit メソッドの呼び出し式は振る舞いを変えうるため，credit メソッドの呼び出し式が出現しうる全てのモジュールを再検証する必要がある．

しかし，すでに定義された同名メソッドの behavioural subtype でないような部分メソッドの追加を制限することは，文脈に応じて振る舞いを変えられることを大きな特徴とする文脈指向プログラムの特徴に反するものである．

3.2　提案

前節の問題を解決し，文脈指向プログラムにおける modular reasoning を実現した方法を示す．

3.2.1　提案の概要

本研究では $FVC^{\#}$ を拡張し，メソッドの振る舞いを抽象化する言語要素 abstract function symbol を追加することで，異なる振る舞いを持つ部分メソッドの追加を許しつつ，modular reasoning を実現した．

$$\begin{aligned} &\{this.balance \mapsto b\}\_\{this.balance \mapsto b + amount\} \\ &\{this.balance \mapsto b\}\_\{this.balance \mapsto b + Encryption.decrypt(amount)\} \end{aligned} \tag{3.1}$$

例えば上の基底レイヤー Encryption レイヤーそれぞれでの credit メソッドの仕様について，以下のように抽象化する．

$$\{this.balance \mapsto b\}\_\{this.balance \mapsto b + q(amount)\} \tag{3.2}$$

ここで q は，メソッド呼び出し式がどの部分メソッド定義にディスパッチされるかによって異なる定義を持つ．すなわち基底レイヤーでの credit メソッドにディスパッチされるなら q は恒等写像に対応する．Encryption レイヤーでの credit メソッドにディスパッチされるなら q は decrypt に対応する．

本言語では，プログラマはこのように抽象化した仕様を各メソッドに与える．そして部分メソッドを定義する際には，その抽象の範囲内での振る舞いの変更のみを認めている．

3.2.2　$FVC^{\#}$ を COP について拡張

$FVC^{\#}$ を文脈指向プログラミングについて拡張した．

文法をレイヤー定義，部分メソッド定義，層活性について拡張し定義した．また評価規則と検証規則についても文脈を扱うように拡張した．

なお本言語では，レイヤーでのフィールドの追加を認めていない，したがって abstract predicate family については，$FVC^{\#}$ と同様に，基底クラスそれぞれに対して entry が定義され，レイヤー追加時には entry は加えられない．そしていかなる文脈でも，abstract predicate family は基底クラスでの定義に従う．

3.2.3　Abstract function symbol

提案手法では全ての部分メソッドが，基底メソッドの behavioural subtype であることを要求する．この時にその仕様を抽象化して記述することで，その抽象度の範囲での振る舞いの変更のみを許すことができる．ここでメソッドの振る舞いを抽象化するために，abstract function symbol を提案する．この手法は $FVC^{\#}$ における abstract predicate family に習ったものである．

本研究では仕様記述言語に abstract function symbol を追加した．この abstract function symbol レイヤーごとに異なる定義を与えることができ，検証時には検証する部分メソッドの属するレイヤーにおける定義によって展開される．

ただし abstract function symbol に与えることのできる定義について，プログラマの指定する代数的関係を満たすもののみ定義可能であるものとする．プログラマは abstract function symbol が満たすべき代数的関係を axiom として記述し，全ての abstract function symbol 定義はそれに従うように定義されていなければならない．これは encrypt と decrypt のような互いに依存するようなメソッドを抽象化するために必要な制約である．

第4章　言語の形式化

4.1　構文定義

表 4.1 に言語の構文定義を示す．この定義は Matthew J. Parkinson らの $FVC^{\#}$ を layer，部分メソッド，抽象関数で拡張したものである．ただし C,D はクラス名を表し，m はメソッド名を表す．x,y,z はプログラム中の変数を表す．またプログラム変数は常に `this` を含むものとする．l はレイヤー名を表す．P,Q は論理式を表す．具体的な定義は後述する．α と β はそれぞれ abstract predicate family と abstract function symbol を表す．これも詳細は後述する．

Abstract predicate family の定義 A は基底クラスにのみ存在する．

レイヤー定義 L はレイヤー名 l とそのレイヤーにおける abstract function symbol の定義と，同じクラスの部分メソッド定義をまとめた N からなる．

4.2　意味論

この節では意味論を定める．意味論の構成は次の４つ組間の遷移関係として表現される．４つ組は S, H, L, $\overline{s}$ からなる．S はスタックで識別子からヒープアドレスへのマップである．H はヒープでヒープアドレスからオブジェクト表現へのマップである．またオブジェクト表現は，そのオブジェクトの型と，フィールド名からアドレスへのマップを持つ．$\overline{s}$ は文のシーケンスである．環境 L は以下のように定義される．

- $L := l \circ L \mid \texttt{Base} \circ []$
- $l \oplus L = l \circ L$
- $l \ominus L = \texttt{remove}(l, L)$
- $$\texttt{remove}(l, l' \circ L) = \begin{cases} l' \circ \texttt{remove}(l, L) & if\ l \neq l' \\ L & otherwise \end{cases}$$

L の実体はレイヤー名を格納したスタックであり，文脈，すなわちレイヤーの活性，非活性を管理する．１つ目の式から L は常に末尾要素として Base をもつ．これは基底レイヤー，すなわち元々のクラスにおけるメソッド定義群を指す．これは非活性になることはない．２つ目の式は活性化，３つ目の式は非活性化にそれぞれ対応する．また，既に活性なレイヤーを活性化するときもスタッ

Program		
$\mathbb{P}$	::=	$\overline{X}\ \overline{F}\ \overline{\mathbb{C}}\ \overline{L}\ ;\overline{s}$
Axiom definition		
X	::=	`axiom` P
Function symbol entry		
F	::=	`define` $\beta_l(\overline{x})$ `as` $f(\overline{x})$
Class definition		
$\mathbb{C}$	::=	`class` $C\{\overline{D}\ \overline{f};\overline{A}\ K\ \overline{M_s}\ \overline{M}\}$
Predicate family entry		
A	::=	`define` $\alpha_C(\overline{x})$ `as` P
Constructor		
K	::=	$C()\ S_d\ S_s\{\overline{s}\}$
Static Method definition		
M_s	::=	`static` $C\ m(\overline{D}\ \overline{x})\ S_s\ B$
Method definition		
M	::=	$C\ m(\overline{D}\ \overline{x})\ S_d\ S_s\ B$
Dynamic specification		
S_d	::=	`dynamic` S
Static specification		
S_s	::=	`static` S
Method specification		
S	::=	$\{P\}\_\{Q\}$
		S `also` $\{P\}\_\{Q\}$
Method body		
B	::=	$\{\overline{C}\overline{x};\overline{s}$ `return` $y;\}$
Layer definition		
L	::=	`layer` $l\ \{\overline{F}\ \overline{N}\}$
Partial methods in layer		
N	::=	`class` $C\ \{\overline{M}\}$
Statement		
s	::=	$x = y;$
		$x =$ `null`$;$
		$x = y.f;$ *// Field access*
		$x.f = y;$ *// Field assignment*
		`if`$(x == y)\{\overline{s}\}$ `else` $\{\overline{t}\}$
		$x =$ `new` $C();$
		$x = y.m(\overline{z});$ *// Dynamic method invocation*
		$x = C :: m(\overline{z});$ *// Static method invocation*
		`with`$(l)\{\overline{s}\}$ *// Activate layer*
		`without`$(l)\{\overline{s}\}$ *// Deactivate layer*

表 4.1: 構文定義

クからは取り除かれずそのまま積まれる．したがって 2 度活性化されたレイヤーを文脈から取り除くには 2 度非活性化する必要がある．以下では Base ∘ [] を単に Base と書くこととする．

次に lookup 関数を定義する．

Method look up in Class

$$\frac{\texttt{class}\ C\{\overline{T}\ \overline{f}; \overline{A}\ K\ \overline{M}\} \quad C_0\ m(\overline{D}\ \overline{x})\ S_d\ S_s\ B}{\texttt{lmbody}(\texttt{Base} \circ [], m, C) = (\overline{x}, B)}$$

Method look up in layer

$$\frac{\texttt{layer}\ l\ \{\overline{N}\} \quad \texttt{class}\ C\ \{\overline{A}\ \overline{M}\} \in \overline{N} \quad C_0\ m\ (\overline{D}\ \overline{x})\ S_d\ S_s\ B \in \overline{M}}{\texttt{lmbody}(l \circ L, m, C) = (\overline{x}, B)}$$

$$\frac{\texttt{layer}\ l\ \{\overline{N}\} \quad \texttt{class}\ C\ \{\overline{A}\ \overline{M}\} \notin \overline{N} \quad \texttt{lmbody}(L, m, C) = (\overline{x}, B)}{\texttt{lmbody}(l \circ L, m, C) = (\overline{x}, B)}$$

$$\frac{\texttt{layer}\ l\ \{\overline{N}\} \quad \texttt{class}\ C\ \{\overline{A}\ \overline{M}\} \in \overline{N} \quad C_0\ m\ (\overline{D}\ \overline{x})\ S_d\ S_s\ B \notin \overline{M} \quad \texttt{lmbody}(L, m, C) = (\overline{x}, B)}{\texttt{lmbody}(l \circ L, m, C) = (\overline{x}, B)}$$

Method look up in Static

$$\frac{\texttt{class}\ C\{\overline{T}\ \overline{f}; \overline{A}\ K\ \overline{M}\} \quad \texttt{static}\ C_0\ m\ (\overline{D}\ \overline{x})\ S_s\ B \in \overline{M}}{\texttt{smbody}(m, C) = (\overline{x}, B)}$$

lmbody は文脈とメソッド名とクラス名を受け取り，そのクラスのメソッド定義を含む最初の活性なレイヤーにおけるメソッド定義から引数名とメソッドボディを得る．そのメソッドの定義が全ての活性なレイヤーに存在しない時は，基底レイヤーから得る．なお本言語では基底レイヤーに存在しないメソッドをレイヤーで宣言することはできない．

意味論は以下の通り．

Assignment

$$\frac{S' = S[x \mapsto S(y)]}{S, H, L, x = y; \overline{s}\ \rightarrow\ S', H, L, \overline{s}}$$

Initialization

$$\frac{S' = S[x \mapsto \texttt{null}]}{S, H, L, x = \texttt{null}; \overline{s} \;\rightarrow\; S', H, L, \overline{s}}$$

Field access

$$\frac{S(y) \in \texttt{dom}(H) \qquad \texttt{field}(H(S(y))) = \mathbb{F} \qquad \mathbb{F}(f) = v}{S, H, L, x = y.f; \overline{s} \;\rightarrow\; S, H, L, x = y`; \overline{s}}$$

Field update

$$\frac{H(S(x)) = (C, \mathbb{F}) \qquad f \in \texttt{dom}(\mathbb{F}) \qquad \mathbb{F}' = \mathbb{F}[f \mapsto y] \qquad H' = H[S(x) \mapsto (C, \mathbb{F}')]}{S, H, L, x.f = y; \overline{s} \;\rightarrow\; S, H', L, \overline{s}}$$

Condition then

$$\frac{S(x) = S(y)}{S, H, \texttt{if}(x == y)\{\overline{s}\}\ \texttt{else}\ \{\overline{t}\} \;\rightarrow\; S, H, \{\overline{s}\}}$$

Condition else

$$\frac{S(x) \neq S(y)}{S, H, \texttt{if}(x == y)\{\overline{s}\}\ \texttt{else}\ \{\overline{t}\} \;\rightarrow\; S, H, \{\overline{t}\}}$$

Object creation

$$\frac{\texttt{class}\ C\{\overline{T}\ \overline{f}; \overline{A}\ K\ \overline{M}\} \qquad C()\ S_d\ S_s\{\overline{t}\} \qquad \theta = [x/\texttt{this}] \qquad \mathbb{F} = \{f \mapsto \texttt{null}\} \quad \forall f \in \overline{f} \qquad H' = H[S(x) \mapsto (C, \mathbb{F})]}{S, H, L, x = \texttt{new}\ C(); \overline{s} \;\rightarrow\; S, H', L, (\theta\overline{t}); \overline{s}}$$

Method invocation

$$\frac{\texttt{type}(H(S(y))) = C \qquad \texttt{lmbody}(L, m, C) = (\overline{z}'', B) \qquad B \equiv \overline{C}\overline{x}; \overline{s}'\ \texttt{return}\ x'; \qquad \theta = [y, \overline{z}', \overline{x}'/\texttt{this}, \overline{z}'', \overline{x}] \qquad \overline{z}', \overline{x}' \text{fresh} \qquad S' = S[\overline{z}' \mapsto S(\overline{z})]}{S, H, L, x = y.m(\overline{z}); \overline{s} \;\rightarrow\; S', H, L, (\theta\overline{s}')x = (\theta x'); \overline{s}}$$

Static Method invocation

$$\frac{\begin{array}{c}\texttt{smbody}(m, C) = (\overline{z}'', B) \\ B \equiv \overline{C}\overline{x}; \overline{s}'\ \texttt{return}\ \ x'; \qquad \theta = [y, \overline{z}', \overline{x}'/\texttt{this}, \overline{z}'', \overline{x}] \\ \overline{z}', \overline{x}' \text{fresh} \qquad S' = S[\overline{z}' \mapsto S(\overline{z})]\end{array}}{S, H, L, x = C :: m(\overline{z}); \overline{s} \ \rightarrow\ S', H, L, (\theta\overline{s}')x = (\theta x'); \overline{s}}$$

Activate layer

$$\frac{\begin{array}{c}L' = l \oplus L \\ S, H, L', \overline{s} \rightarrow S', H', L', \texttt{skip}\end{array}}{S, H, L, \texttt{with}(l)\{\overline{s}\}; \overline{t} \ \rightarrow\ S', H', L, \overline{t}}$$

Deactivate layer

$$\frac{\begin{array}{c}L' = l \ominus L \\ S, H, L', \overline{s} \rightarrow S', H', L', \texttt{skip}\end{array}}{S, H, L, \texttt{without}(l)\{\overline{s}\}; \overline{t} \ \rightarrow\ S', H', L, \overline{t}}$$

4.3　検証規則

4.3.1　Logic syntax

この節では，提案言語を検証するための論理式を定義する．
論理式 P は，$FVC^{\#}$ に Abstract function symbol を加えた以下になる．

$$\begin{array}{rcl} P, Q & ::= & \forall x.P \mid P \Rightarrow Q \mid false \mid \alpha(\overline{x}) \mid e = e' \mid x : C \\ & & x.f \mapsto e \mid P * Q \mid P -\!\!* Q \mid \beta(\overline{x}) \\ e & ::= & x \mid \texttt{null} \end{array}$$

環境 Γ を定義する．環境 Γ には，静的・動的仕様を含む．ここで基底レイヤーのクラス C で定義された $\overline{x}$ を引数とするメソッド m の動的仕様 $\{P\}\_\{Q\}$ を環境が含む時，$C.m(\overline{x})\{P\}\_\{Q\} \in \Gamma$ と表す．同様に静的仕様を持つことを $C :: m(\overline{x})\{P\}\_\{Q\} \in \Gamma$ とする．
環境 Δ は abstract predicate family と abstract function symbol を含む．$\Delta = \Delta_p \wedge \Delta_f$ と分割できて，Δ_p に abstract predicate family，Δ_f に abstract function symbol の定義を持つものとする．それぞれについての詳細は対応する節で定義する．
環境 Δ，Γ と，文脈 L において，$\overline{s}$ が仕様 $\{P\}\_\{Q\}$ を満たすことを

$$L; \Delta; \Gamma \vdash \{P\}\ \overline{s}\{Q\} \tag{4.1}$$

とかく．

4.3.2　Statement verification

プログラム中のステートメントの検証規則は以下のようになる．

Assignment

$$\frac{}{L; \Delta; \Gamma \vdash \{P[y/x]\}x = y; \{P\}}$$

Initialization

$$\frac{}{L; \Delta; \Gamma \vdash \{P[\texttt{null}/x]\}x = \texttt{null}; \{P\}}$$

Field access

$$\frac{\begin{array}{c} y \text{ has static type } C \\ C \text{ has field } f \end{array}}{L; \Delta; \Gamma \vdash \{P[y.f/x]\}x = y.f; \{P\}}$$

Field update

$$\frac{}{L;\Delta;\Gamma \vdash \{P[y/x.f]\}x.f = y;\{P\}}$$

Condition

$$\frac{L;\Delta;\Gamma \vdash \ \{P \wedge x = y\}\overline{s}\{Q\} \qquad L;\Delta;\Gamma \vdash \ \{P \wedge x \neq y\}\overline{t}\{Q\}}{L;\Delta;\Gamma \vdash \{P\}\texttt{if}(x == y)\{\overline{s}\}\ \texttt{else}\ \ \{\overline{t}\}\{Q\}}$$

Object creation

$$\frac{C..ctor() : \{P\}\_\{Q\} \in \Gamma}{L;\Delta;\Gamma \vdash \{P\}x = \texttt{new}\ C()\{Q[x/\texttt{this}]\}}$$

Method invocation

$$\frac{x \text{ has static type } C \qquad C.m(\overline{x}) : \{P\}\_\{Q\} \in \Gamma}{\begin{array}{c}L;\Delta;\Gamma \vdash\{P[x,\overline{y}/\texttt{this},\overline{x}] \wedge \texttt{this} \neq \texttt{null}\} \\ z = x.m(\overline{y}) \\ \{Q[z,x,\overline{y}/ret,\texttt{this},\overline{x}]\}\end{array}}$$

Static method invocation

$$\frac{C :: m(\overline{x}) : \{S\}\_\{T\} \in \Gamma}{\begin{array}{c}L;\Delta;\Gamma \vdash\{S[x,\overline{y}/\texttt{this},\overline{x}] \wedge \texttt{this} \neq \texttt{null}\} \\ z = C :: m(\overline{y}) \\ \{T[z,x,\overline{y}/ret,\texttt{this},\overline{x}]\}\end{array}}$$

Activate layer

$$\frac{l \oplus L;\Delta;\Gamma \vdash \{P\}\overline{s}\{Q\}}{L;\Delta;\Gamma \vdash \{P\}\ \texttt{with}(l)\{\overline{s}\}\ \{Q\}}$$

Deactivate layer

$$\frac{l \ominus L;\Delta;\Gamma \vdash \{P\}\overline{s}\{Q\}}{L;\Delta;\Gamma \vdash \{P\}\ \texttt{without}(l)\{\overline{s}\}\ \{Q\}}$$

4.3.3 Abstract predicate family

次に Abstract predicate family を定義する．

$$\Delta ::= P \mid \Delta_1 \wedge \Delta_2 \mid \exists \alpha.\, \Delta \tag{4.2}$$

本研究では，layer で新たなフィールドを定義することを許していないので，Abstract predicate family の entry は，$FVC^{\#}$ と変わらずクラスごとに定義される．$A(\alpha, n)$ は，述語 α に n のアリティを与えることができることを表す．

$$FtoE(\alpha, C) \stackrel{def}{=} \forall x, \overline{x}.x : C \Rightarrow (\alpha(x, \overline{x}) \Leftrightarrow \alpha_C(x, \overline{x})) \tag{4.3}$$

$$\begin{aligned} &EtoD(\texttt{define}\ \ \alpha_C(x, \overline{x})\ \texttt{as}\ \ P) \\ &\qquad \stackrel{def}{=} \forall x, \overline{x}.\alpha_C(x, \overline{x}) \Leftrightarrow P \end{aligned} \tag{4.4}$$

$$\begin{aligned} &apf_C(\texttt{define}\ \ \alpha_C(x, \overline{x})\ \texttt{as}\ \ P) \\ &\quad \stackrel{def}{=} FtoE(\alpha, C) \wedge EtoD(\texttt{define}\ \ \alpha_C(x, \overline{x})\ \texttt{as}\ \ P) \wedge A(\alpha; \mid \overline{x} \mid) \end{aligned} \tag{4.5}$$

$$\begin{aligned} &apf(\texttt{class}\ \ C\{...; A_1, ..., A_n\ ...\}) \\ &\quad \stackrel{def}{=} apf_C(A_1) \wedge ... \wedge apf_C(A_n) \end{aligned} \tag{4.6}$$

アリティの異なる述語を同一視するための規則を定義する．述語に定義より多い引数を与えた時，その引数は無視される．また少ない引数を与えた時，欠けている引数は存在量化子付きで与えられる．

$$\begin{aligned} A(\alpha; n+1) \stackrel{def}{=} &A(\alpha; n) \wedge \\ &\forall y_1, ..., y_n.\alpha(y_1, ..., y_n) \Leftrightarrow \exists z.\alpha(y_1, ..., y_n, z) \end{aligned} \tag{4.7}$$

$$A(\alpha; 0) \stackrel{def}{=} true \tag{4.8}$$

以上の定義に従ってプログラム中から Abstract predicate family の定義が得られる．Abstract predicate family は環境 Δ に含まれる．

4.3.4 Abstract function sylmbol

Abstract function symbol の定義中の f は以下のように定める．

$$f ::= static\ method \in \Gamma \mid f_1 \circ f_2 \mid id \mid \beta_{L'} \tag{4.9}$$

Abstract function symbol は，仕様記述言語の一部として用いられ，レイヤーごとに定義される．ただし axiom を満たすように定義されている必要がある．

$$\frac{\overline{F} \text{ is satisfy } \overline{X} \text{ in } \Gamma}{afs(\overline{F},\ \overline{X},\ l,\ \Gamma) \stackrel{def}{=} afs_{F1}(F_1, l) \wedge ...afs_{Fn}(F_n, l)}$$

$$\frac{\begin{array}{c} F_i = \texttt{define}\ \beta_l\ \texttt{as}\ f_i \\ X_j = \texttt{axiom}\ P_j \\ P_j[f_1/\beta_1]...[f_i/\beta_i]... \end{array}}{\overline{F} \text{ is satisfy } \overline{X} \text{ in } \Gamma}$$

$$afs_{Fi}(\texttt{define}\ \beta_l\ \texttt{as}\ f,\ l) \stackrel{def}{=} \beta_l \Leftrightarrow f \tag{4.10}$$

Abstract function symbol は β_l という形の時にのみ開くことができる．それ以外の形の時は，アトミックに扱うことしかできない．プログラマはメソッドの仕様を記述する際は，abstract function symbol は添字なしで記述する．`layer` l で定義されたメソッドの abstract function symbol β を含む静的仕様が要求された時，β_l が定義されているなら，$\beta \Leftrightarrow \beta_l$ としてよい．

4.3.5 Method verification

メソッド定義の検証規則は基底メソッドと部分メソッドにそれぞれ与えられる．ただし $\Gamma \vdash M\ in\ (\texttt{class}\ E, \texttt{layer}\ l)$ は，環境 Δ と Γ が与えられた時に，クラス E，レイヤー l でのメソッド定義 M がその仕様を満たすことを検証できることを表す．l が `Base` のときは，クラス E の基底メソッド定義 M の仕様を検証できることを表す．

behavioural subtyping の証明において，以下のルールを追加する．

$$\begin{array}{l} \Delta \vdash \{P_1\}\_\{Q_1\} \stackrel{this;\ C}{\Rightarrow} \{P_2\}\_\{Q_2\} \\ \stackrel{\text{def}}{=} \Delta \vdash \{P_1\}\_\{Q_1\} \Rightarrow \{P_2 * this : C\}\_\{Q_2\} \end{array} \tag{4.11}$$

また 2 つの仕様が `also` で結ばれた仕様は，どちらの仕様も満たしていて初めて満たすものとする．以下のように定義される．

$$\begin{array}{l} \{P_1\}\_\{Q_1\}\texttt{also}\ \{P_2\}\_\{Q_2\}\ is\ defined\ as \\ \{(P_1 \wedge X = 1) \vee (P_2 \wedge X \neq 1)\}\_\{(Q_1 \wedge X = 1) \vee (Q_2 \wedge X \neq 1)\} \end{array} \tag{4.12}$$

基底レイヤーにおけるメソッド定義の検証規則は以下の通り

$$\frac{\begin{array}{l} B = \{\overline{G}\overline{y}; \overline{s}\ \texttt{return}\ z;\} \\ S_d = \texttt{dynamic}\ \{P_E\}\_\{Q_E\} \\ S_s = \texttt{static}\ \{S_E\}\_\{T_E\} \\ \Delta \vdash \{S_E\}\_\{T_E\} \stackrel{\texttt{this}:E}{\Rightarrow} \{P_E\}\_\{Q_E\} \qquad (Dynamic\ dispatch) \\ \Delta; \Gamma \vdash \{S_E\}\overline{s}\{T_E[z/ret]\} \qquad (Body\ Verification) \end{array}}{\Delta; \Gamma \vdash C\ m(\overline{D}\ \overline{x})\ S_d\ S_s\ B\ in\ (\texttt{class}\ E, \texttt{layer Base})}$$

部分メソッドは，基底メソッドの behavioural Subtype であることを要求している．全ての部分メソッドの呼び出しは，基底メソッドの呼び出しと置換可能である．部分メソッドの検証規則は以下の通り．

$$\frac{\begin{array}{rl} & E.m(\overline{x}) : \{P_{\texttt{Base}}\}\_\{Q_{\texttt{Base}}\} \in \Gamma \\ B = & \{\overline{G}\overline{y}; \overline{s}\ \texttt{return}\ z;\} \\ S_d = & \texttt{dynamic}\ \{P_l\}\_\{Q_l\} \\ S_s = & \texttt{static}\ \{S_l\}\_\{T_l\} \\ \Delta \vdash & \{P_l\}\_\{Q_l\} \Rightarrow \{P_{\texttt{Base}}\}\_\{Q_{\texttt{Base}}\} \qquad (\textit{Behavioural Subtyping}) \\ \Delta \vdash & \{S_l\}\_\{T_l\} \overset{\texttt{this}:E}{\Rightarrow} \{P_l\}\_\{Q_l\} \qquad (\textit{Dynamic dispatch}) \\ \Delta;\Gamma \vdash & \{S_l\}\overline{s}\{T_l[z/ret]\} \qquad (\textit{Body Verification}) \end{array}}{\Delta;\Gamma \vdash C\ m(\overline{D}\ \overline{x})\ S_d\ S_s\ B\ in\ (\texttt{class}\ E,\ \texttt{layer}\ l)}$$

4.3.6 Program verification

レイヤー定義の検証は以下の規則によって行われる．

$$\frac{\forall M_i \in \overline{M}.\ \Delta;\Gamma \vdash M_i\ in\ (\texttt{class}\ C,\ \texttt{layer}\ l)}{L;\Delta;\Gamma \vdash \texttt{class}\ C\{\overline{A},\overline{M}\}}$$

$$\frac{\forall i.\ N_i = \texttt{class}\ C_i\ \{\overline{M_i}\}}{\Delta;\Gamma \vdash \texttt{layer}\ l\ \{\overline{F}\ N_1 ... N_n\}}$$

プログラム全体を検証するためのルールは次の通り．

$$\frac{\begin{array}{c} \Gamma = spec(\mathbb{C}_1 ... \mathbb{C}_n) \\ \forall i \in \{1,...,n\} \\ \Delta_i = apf(\mathbb{C}_i) \wedge afs(\overline{F},\overline{X},\texttt{Base},\Gamma) \\ \Delta_i;\Gamma \vdash \mathbb{C}_i \\ \forall j \in \{1,...,m\} \\ L_j = \texttt{layer}\ l\ \{\overline{F_j}\ \overline{N_j}\} \\ \Delta_j = apf(\mathbb{C}_1) \wedge ... \wedge apf(\mathbb{C}_n) \wedge afs(\overline{F_j},\overline{X},l,\Gamma) \\ \Delta_j;\Gamma \vdash L_j \\ true;\Gamma \vdash \{true\}\overline{s}\{true\} \end{array}}{\vdash \overline{X}\ \overline{F}\ \mathbb{C}_1 ... \mathbb{C}_n\ L_1 ... L_m\ \overline{s}}$$

ここで環境 Γ には基底レイヤーでのクラス定義中のメソッドの仕様のみを持たせればよい．

第5章　事例研究

この章では，第 3 章で与えた例プログラムが定義した言語によって検証できることを示す.

ソースコード 5.1: Banking system の検証

```
1
2  axiom q*p = id;
3
4  define p_{Base,L}(x) as id(x);
5  define q_{Base,L}(x) as id(x);
6
7  class Encryption {
8    float key = 42.4711f;
9    float key2 = 45047028;
10
11   static float encrypt(float val)
12   static {true}_{ret=(val*key)/key2}
13   {
14     return (val * key) / key2;
15   }
16
17   static float decrypt(float val)
18   static {true}_{ret=(val*key2)/key}
19   {
20     return (val * key2) / key;
21   }
22 }
23
24 class Account {
25   float balance;
26
27   define Balance_{Account}(x,v) as x.balance ↦ v
28
29   Account()
30   dynamic {true}_{Balance(this,_)}
31   {}
32
33   void credit(float amount)
34   dynamic {Balance(this,v)}_{Balance(this, v + q(amount))}
35   {
36     balance = balance + amount;
37   }
38
39   void debit(float amount)
40   dynamic {Balance(this,v)}_{Balance(this, v - q(amount))}
41   {
42     if(balance < amount)
43       System.err.println("Total␣balance␣not␣sufficient:␣"+ amount + "," + balance);
44     else {
45       balance = balance - amount;
46     }
47   }
48 }
49
```

```
50  class TransferSystem {
51    define Balance_Account(x,v) as x.balance ↦ v
52
53    void transfer(Account from, Account to, float amount)
54    dynamic {Balance(to,v1) * Balance(from,v2)}
55        _{Balance(to,v1 - amount) * Balance(from,v2 + amount)}
56    {
57      amount_ = f(amount)
58      to.credit(amount_);
59      from.debit(amount_);
60    }
61
62    float f(float amount)
63    {true}_{ret=p(amount)}
64    {
65      return amount;
66    }
67  }
68
69
70  layer Encryption{
71
72    define p_Encryption,L(x) as encrypt(p_Encryption,L(x));
73    define q_Encryption,L(x) as q_Encryption,L(decrypt(x));
74
75    class Account {
76
77      void credit(float amount)
78      dynamic {Balance(this,v)}_{Balance(this, v + q(amount))}
79      {
80        without(Encryption){
81          tmp = Encryption::decrypt(amount)
82          credit(tmp);
83        }
84      }
85      void debit(float amount)
86      dynamic {Balance(this,v)}_{Balance(this, v - q(amount))}
87      {
88        without(Encryption){
89          tmp = Encryption::decrypt(amount)
90          debit(tmp);
91        }
92      }
93    }
94    class TransferSystem {
95      float f(float amount)
96      {true}_{ret=p(amount)}
97      {
98        without(Encryption){
99          tmp = f(amount)
100       }
101       return Encryption::encrypt(tmp);
102     }
103   }
104 }
```

以下では添字において $_{Account}$ を $_{A}$，$_{Base}$ を $_{B}$，$_{Encryption}$ を $_{E}$，$_{Transfer}$ を $_{T}$ のように略記する．
抽象述語は，以下のように定義している．これらは axiom を満たすように定義しなければなら

ない．

$$\begin{aligned} p_{B,L} &= id \\ q_{B,L} &= id \\ p_{E,L} &= encrypt \circ p_{E \ominus L} \\ q_{E,L} &= q_{E \ominus L} \circ decrypt \end{aligned} \tag{5.1}$$

$$q_{B,L} \circ p_{B,L} = id \tag{5.2}$$

$$q_{E,L} \circ p_{E,L} = q_{E \ominus L} \circ decrypt \circ encrypt \circ p_{E \ominus L} \tag{5.3}$$

Encryption レイヤーにおける credit メソッドについて考察する．credit メソッドを検証するには，(Body Verification) と (Behavioural subtyping) と (Dynamic dispatch) についてそれぞれ証明する必要がる．

Body Verification は以下のように示される．

ソースコード 5.2: credit() Body Verification

```
1    L ⊢ {Balance_A(this,v)}
2    without(Encryption) {
3      Encryption⊖L ⊢ {Balance_A(this,v)}
4        tmp = Encryption::decrypt(amount)
5      Encryption⊖L ⊢ {Balance_A(this,v) * tmp = decrypt(amount)}
6        this.credit(tmp)
7      Encryption⊖L ⊢ {Balance_A(this,v + q_{E⊖L}(tmp)) * tmp = decrypt(amount)}
8      Encryption⊖L ⊢ {Balance_A(this,v + q_{E⊖L}(decrypt(amount)))}
9    }
10   L ⊢ {Balance_A(this,v + q_{E,L}(amount))}
```

この証明では以下のことに注意が必要である．this.credit(tmp) について，文脈 L にかかわらず基底メソッドの動的仕様が満たされている．そしてその仕様に含まれる abstract function symbol q は，レイヤー Encryption における定義に従う．

Behavioural subtyping は以下に対応し，自明である．

$$\frac{\vdash \{Balance(this, v)\}\_\{Balance(this, v + q(amount))\}}{\vdash \{Balance(this, v)\}\_\{Balance(this, v + q(amount))\}}$$

Dynamic dispatch は節 2.6.2 で議論した通り，即座に示せる．

次に transfer を検証する．transfer は基底クラスで定義されるメソッドである．よって (Dynamic dispatch) と (Body Verification) を示せば良い．後者の証明を以下に示す．

ソースコード 5.3: transfer() Body Verification

```
1    L ⊢ {Balance_T(from,v1) * Balance_T(to,v2)}
2      amount_ = f(amount)
3    L ⊢ {Balance_T(from,v1) * Balance_T(to,v2) * amount_=p(amount)}
4      to.credit(amount_)
5    L ⊢ {Balance_T(from,v1) * Balance_T(to,v2+q(amount_)) * amount_=p(amount)}
6      from.debit(amount_)
7    L ⊢ {Balance_T(from,v1-q(amount_)) * Balance_T(to,v2+q(amount_)) * amount_=p(amount)}
8    L ⊢ {Balance_T(from,v1-q(p(amount))) * Balance_T(to,v2+q(p(amount))) * amount_=p(amount)}
9    L ⊢ {Balance_T(from,v1-amount) * Balance_T(to,v2+amount) * amount_=p(amount)}
10   L ⊢ {Balance_T(from,v1-amount) * Balance_T(to,v2+amount)}
```

transfer メソッドは基底レイヤーでのみ定義されており，文脈にかかわらず同じ仕様を与えられている．しかしその中で呼び出されている credit debit f の 3 つのメソッドはそれぞれ Encryption レイヤーでも定義されているメソッドである．文脈に応じて送金着金の際の暗号化の有無が異なるが，transfer はどの文脈でも同じように amount として受け取った金額をそのままそれぞれの口座から加減している．

ここでは q(p(amount)) が axiom から amount となっていることに注意したい．

第6章　関連研究

この章では本研究と関連性の深い研究について述べる．

6.1 Layer Interface

Layer Interface [2] は，本研究と同様に，文脈指向プログラムのモジュラーな検証へアプローチするものである．部分メソッドの仕様を記述する方法として，Behavioral subtyping を適用することを提案している．具体的にはレイヤーインターフェースというレイヤーとクラスを仲介するインターフェースを定義し，ここで部分メソッドの仕様を宣言する．レイヤーインターフェースを実装する全てのレイヤーの全ての部分メソッドは，レイヤーインターフェースで定義された仕様を満たさねばならない，

またオブジェクト指向言語の delegation パターンを用いて文脈指向プログラムをエミュレートする手法について触れている．

本研究では，Behavioral subtyping を適用する手法が共通し，abstract predicate families によって部分メソッドの仕様を宣言する点で異なる．

6.2 A Semantics for Context-oriented Programming with Layers

Dave Clarke ら [6] による．この研究では文脈指向言語である ContextFJ に対して動的意味論と型付けルールを与えている．我々の研究では，単純化のためにレイヤーの並び順は固定として，レイヤーの活性化の順番を無視したディスパッチを行なったのに対して，この研究ではレイヤーの活性化，非活性化の順番も反映している．これは部分メソッド参照できないレイヤーの集合を excluded layers set としてメソッドの束縛情報とともに受け渡し，非活性化されたレイヤーや，proceed() 呼び出しの過程で既に訪れたレイヤーを順次加え入れることで実現させている，

この研究と同様に proceed 呼び出しの評価規則，検証規則を定められるか否かは，今後の課題のひとつである．

またこの研究では，もともとクラスで定義されていないメソッドをレイヤーで新たに定義することも認めている．これは layer introduced (base) method と呼ばれ，議論されている．[1] クラスで定義されていないメソッドの呼び出しでは，呼び出し時の文脈で，そのメソッドが定義されているレイヤーが活性でなければならない．

この研究ではメソッド内で呼び出されるクラスで定義されていないメソッドの名前と，そのメソッドが宣言されているクラスの組を受け渡し，ディスパッチできないような呼び出しが存在しないことを検査している．

6.3 Translucid Contract

Mehdi Bagherzadeh らによってアスペクト指向言語である Ptoremy [16] の仕様記述とそのモジュラーな検証が研究されている．

この研究では，アスペクトの仕様を AO interface（aspect-oriented interface） に記述する際に，Translucid Contract [5] なる手法でアプローチしている．この手法は仕様を曖昧性を持たせた形で

与える gray box アプローチであり，副作用やメソッド呼び出しの制限のみを記述することで．実際の実装に抽象度を残しつつ制限を与える．

　仕様をインターフェースに記述し，その仕様に抽象度を持たせることで，モジュラーな検証を実現している点で，本研究と共通する．

第7章　結論と今後の課題

7.1　結論

本研究では，文脈指向プログラムのモジュラーな検証のための言語の形式化を行った．この言語はメソッドに記述する仕様を abstract function symbol により抽象化したことで，レイヤーや振る舞いの異なる部分メソッドの追加に関してモジュラーに検証可能である．

また 5 章では簡単な文脈指向プログラムを実際に検証し，本言語の有効性を示した．

7.2　今後の課題

証明系の健全性の証明は今後の課題である．また proceed() 呼び出しなど，文脈指向プログラミングに望ましい言語機能を簡略しているため，この実装についても今後の課題である．

またレイヤー同士の相互関係を定義可能にすると検証の幅が広がることも考えられる．例えば「別の銀行とのやり取り」という文脈ならば，必ず暗号化レイヤーが活性でなければならない，など他のレイヤーに依存する振る舞いを検証することが可能になる．

参考文献

[1] Tomoyuki Aotani, Tetsuo Kamina, and Hidehiko Masuhara. Type-safe layer-introduced base functions with imperative layer activation. In *Proceedings of the 7th International Workshop on Context-Oriented Programming*, p. 8. ACM, 2015.

[2] Tomoyuki Aotani and Gary T Leavens. Towards modular reasoning for context-oriented programs. In *Proceedings of the 18th Workshop on Formal Techniques for Java-like Programs*, p. 8. ACM, 2016.

[3] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. Contextj: Context-oriented programming with java. *Information and Media Technologies*, Vol. 6, No. 2, pp. 399–419, 2011.

[4] Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. Event-specific software composition in context-oriented programming. In *International Conference on Software Composition*, pp. 50–65. Springer, 2010.

[5] Mehdi Bagherzadeh, Hridesh Rajan, Gary T Leavens, and Sean Mooney. Translucid contracts: Expressive specification and modular verification for aspect-oriented interfaces. In *Proceedings of the tenth international conference on Aspect-oriented software development*, pp. 141–152. ACM, 2011.

[6] Dave Clarke and Ilya Sergey. A semantics for context-oriented programming with layers. In *International Workshop on Context-Oriented Programming*, p. 10. ACM, 2009.

[7] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: an overview of contextl. In *Proceedings of the 2005 symposium on Dynamic languages*, pp. 1–10. ACM, 2005.

[8] Robert Hirschfeld, Pascal Costanza, and Oscar Marius Nierstrasz. Context-oriented programming. *Journal of Object technology*, Vol. 7, No. 3, pp. 125–151, 2008.

[9] Gary T Leavens and David A Naumann. Behavioral subtyping is equivalent to modular reasoning for object-oriented programs. 2006.

[10] Gary T Leavens and David A Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 37, No. 4, p. 13, 2015.

[11] Barbara H Liskov and Jeannette M Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 16, No. 6, pp. 1811–1841, 1994.

[12] Peter W O'Hearn. A primer on separation logic (and automatic program verification and analysis)., 2012.

[13] PeterO ' Hearn, John Reynolds, Hongseok Yang. Local reasoning about programs that alter data structures. In *International Workshop on Computer Science Logic*, pp. 1–19. Springer, 2001.

[14] Matthew J Parkinson and Gavin M Bierman. Separation logic, abstraction and inheritance. In *ACM SIGPLAN Notices*, Vol. 43, pp. 75–86. ACM, 2008.

[15] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *ACM SIGPLAN Notices*, Vol. 40, pp. 247–258. ACM, 2005.

[16] Hridesh Rajan and Gary T Leavens. Ptolemy: A language with quantified, typed events. In *European Conference on Object-Oriented Programming*, pp. 155–179. Springer, 2008.

[17] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pp. 55–74. IEEE, 2002.

[18] Oliver Schoett. Data abstraction and the correctness of modular programming. 1986.