平成23年度 修士論文

静的型付きオブジェクト指向言語のための暗黙的に型定義されるレコード

東京工業大学 情報理工学研究科
数理・計算科学専攻

学籍番号 10M37025

大久保 貴司

指導教員

千葉 滋 教授

平成24年1月31日

概要

主な静的型付きオブジェクト指向言語では、レコードを利用しようとすると、クラス定義等の型定義が必要となる。型を定義することで汎用的に型やその型のインスタンスを利用できるが、その一方、レコードはシンプルなデータ構造であるため、できる限り簡便に利用できることが望ましいと考えられる。そこで本研究では、そのような簡便に利用できるレコードを実現するための型システムを提案する。本システムではレコードの型を structural type として表現し、ソースコードから型推論をすることで、暗黙的に型を定義・付与している。本システムではレコードのメンバに対する代入等から、そのレコードの持っているメンバとその型を推論している。そうすることによって、プログラマは擬似的に任意の名前・型を持つメンバを持つレコードを型を定義することなく生成・利用できるようになる。一見すると、その振る舞いは動的型付き言語のようであるが、静的型付けは維持されているため、本システムのレコードについても、型検査をすることでコンパイル時にエラーを発見することが可能である。また本研究では、本システムを組み込んだ拡張 Java 言語を実装する。この言語では、レコードのメソッド間の受け渡しがサポートされており、これを利用すると、複数のオブジェクトをメソッドの返り値として返したい場合などに有用である。また、この言語では暗黙的に定義された型を取得する機能もサポートしており、暗黙的に定義された型の比較や、Generics の型引数に利用できる。また本研究では、本システムの有用性とオーバーヘッドを測る実験を行った。有用性を測る実験では、既存のプログラムに対し本システムを適用することで、どの程度のクラス定義の省略ができるかを測定している。オーバーヘッドを測る実験では、レコードをクラス定義して用いる Java 言語のプログラムと、それを本言語で記述したプログラムとのコンパイル時間と実行時間を計測して型推論等が与えるオーバーヘッドを計測している。最後に、本システムの核となる機能を持つ言語を用いて、システムの形式化を行った。

謝辞

本研究を進めるにあたり、研究の方針や構成など様々な点について指導して頂いた千葉滋教授に心より感謝致します。また、研究活動を共に行い、多くの助言を頂いた千葉研究室の皆様方にも感謝致します。

目 次

図 目 次

表 目 次

第1章　はじめに

今日、プログラミング言語では様々なデータや手続きなどをモジュール化するために、様々な複合データ型が提供されている、レコード型もその中の１つであり、その名の通りレコードを表す型である。レコードはC言語の構造体として代表されるデータ構造であり、配列やリストとは異なり、複数の異なる型のデータをまとめて格納できる点が特徴である。

オブジェクト指向言語では、レコード自体はサポートされていない場合が多い。例えばオブジェクトをクラスのインスタンスとして位置付けるクラスベースのオブジェクト指向言語の場合、レコードのようなデータ構造はクラスで代用することが可能だからである。本研究において主に対象としている Java 言語もその一例である。例えば Java 言語で、String 型の name, int 型の age というメンバをもつ人を表すレコードを生成したいとすると、図 1.1 のように各メンバをフィールドに持つような Person クラスを生成する。確かに Person がメソッドや、static なフィールド持っている場合、コードの再利用性や性質上の問題からクラスとして定義する意義は大きい。しかし上記の例のように、値の出し入れだけができれば良いというシンプルなレコードを生成したい場合、クラス定義はレコードのメンバの名前と型を宣言するという役割しかもたない。従って、このようなクラスを定義する意義は小さく、わざわざ定義するのは煩わしい。

以上の理由から、本研究では、クラスを定義をせずにレコードを利用できるシステムを導入する。上記の問題は、全てのオブジェクトがクラスを定義しないければ作れないということが大きな理由である。ある特定のクラスであれば、インポートするなどの方法があるが、レコードは要素であるメンバによって型が異なるため、それは困難である。従って解決法しては、レコードの型がその構造に応じて暗黙的に定義され、その型がさらに

```
1 class Person{
2  String name;
3  int age;
4 }
```

図 1.1: クラス定義によるレコードの定義

暗黙的に付けられれば良い。そのようなレコードを実現することができれば、簡潔なコードでレコードを利用することができるはずである。本研究では、そのようなレコードの実現法を提案する。

以下、第 2 章では、関連研究・既存の手法について、第 3 章では、本研究で提案する暗黙的に型定義されるレコードについて、第 4 章では、提案したシステムの形式化について、第 5 章では、提案したシステムを組み込んだ言語の実装について、第 6 章では、本システムの有意性とオーバーヘッドを計測するための実験について、そして最後に第 7 章では、本研究のまとめと future work について述べる。

第2章　関連研究と既存手法

この章では、本研究に関連のある既存の言語や関連研究について説明し、それらを用いたレコードの実現法やその問題点等について説明する。

2.1 WhiteOak

Whiteoak は Java に structural typing [1] を導入した言語である。以下では本システムでも利用されている structural typing と Whiteoak の型システムについて説明する。

2.1.1 structural typing

structural typing とは、Java 等に代表される型の等価性を型の名前で判断する nominal typing に対して、型の構造によって型の等価性を判断する型システムである。基本的には、型はメンバの名前と型のペアの集合として表され、この集合が等しければ型として等しいと判断される。

サブタイプ関係は、型 A,B が存在して、メンバの名前と型の集合として見た時、$A \subset B$ であるなら B は A のサブタイプであると判断される。より具体的には、以下で説明する Whiteoak を用いた図 2.1 の例では、型 XYZ は型 XY の要素であるメンバをすべて持っているので、XYZ は XY のサブタイプである。

2.1.2 Whiteoak

Whiteoak [5] とは、前述した structural typing を Java に導入した言語である。

この言語では、既存の noinal type のオブジェクト (クラスのインスタンス) に対し、プログラマが定義した structural type を型として付与することができる。例えば、図 2.1 のクラス定義された Point 型のオブジェクトは、フィールドとして int 型の x,y,z をもっているので、structural type である XYZ のサブタイプであり、そのため XYZ 型 の変数に代入

```
1  struct XY{int x, int y}
2  struct XYZ{int x, int y, int z}
3
4  int sumXY(XY a){
5    return a.x + a.y;
6  }
7  Point p1 = new Point(1, 2, 3);
8  XYZ xyz = p1;
9  System.out.println
10     (''sum of x and y is '' + sumXY(xyz));
11 //output: sum of x and y is 3
12
13 class Point{
14   int x,y,z;
15   String name;
16   Point(x,y,z){this.x = x; this.y = y; this.z = z;}
17 }
```

図 2.1: Whiteoak の structural type

することができる。また Whiteoak では、メソッドをレコードのメンバとすることも可能であり、柔軟なレコードを利用することが可能である。

しかし、レコードの型を structural type を用いて表現できるとしても、やはりクラス定義と同様に、明示的に型を定義しなければいけないということに関しては同じである。そのため本研究の目的を解決するには至らない。また、 Whiteoak は structural type を既存のオブジェクトに与えるだけであり、その型自身のインスタンスを生成することはできないという点もレコードを生成するという点において難点である。

2.2　Whiley

Whiley [8, 9] とは structural typing と flow-sensitive typing をもつ静的型付き言語である。flow-sensitive typing とは同じ変数が文脈によって異なる型を持つことができる型システムであり、whiley では変数への代入・操作から各文脈毎に変数の型が暗黙的に決定される。このような手法を用いることで静的な型を維持しつつ、動的な型のような柔軟性を得ることが可能となっている。

また Whiley ではレコードもサポートされており、上記のように暗黙的に型付けされることから、

```
b = {x: 1, y: 2}
```

と記述することで、{x: int, y: int} 型のレコードを生成することが可能である。これは目的とするレコードに近いものである。しかし、Whiley の flow-sensitive typing という特殊な型システムを採用しており、一般的な静的型付け言語の型付け方法と大きく異なる。従って、既存の言語に組み込んでこの言語機能を利用するというのは難しい。またメソッド間で受け渡しをする際には、やはり明示的に定義された型を仮引数の型として、宣言しなければならない。

2.3　既存の言語機能

既存言語の言語機能を利用して、型定義を必要としないレコードを実現するため手法を考える。レコードのような複数の異なる型をもつデータ構造が組み込まれており、型を定義せずに済むことから、動的型付き言語や型推論を持つ言語が候補として挙げられる。

そのような言語機能をもつプログラミング言語について説明するとともに、本研究の意図とするレコードとして利用できるかどうか考察する。

```
1 class Person{
2   String name;
3   int age;
4 }
5
6 Person p = new Person();
7 p.name = "hoge";
8 p.age = 20;
```

図 2.2: Java 言語でのレコードの利用

2.3.1　動的型付き言語での実現

本研究の目的とする型を定義せずにレコードを実現する方法として、動的型付き言語を用いるということが考えられる。これは、動的型付き言語では明示的に型を宣言する必要がないためである。

図 2.2 はレコードをクラスとして定義して利用している、Java 言語のプログラムである。これを例えば、動的型付き言語である JavaScript [4] を用いて、同じ振る舞いをするコードに書き換えると

```
var p = {}
p.name = "hoge";
p.age = 20;
```

のように書くことができる。このコードは図 2.2 のクラス定義を省略して、同様なオブジェクト (レコード) を生成できている。これはメンバを追加するにあたって、レコードの型が変化するので、このようなことが可能となっている。

以上のことから、目的とする簡便さという点では優れていると考えられる。しかし、動的型付き言語では、コンパイル時の型検査ができないという欠点が存在する。コンパイル時の型検査はプログラムの実行前に、不適切なコードを検出し、プログラムの信頼性・安全性を向上させたり、デバッグ作業の手助けとなる大きな利点である。このような利点を重視し、静的型付き言語を使用したいと考えた時、前述の記述法を静的型付き言語に導入することは困難である。

2.3.2　Scala

Scala [7] は強い静的型を持つオブジェクト指向言語である。更に Scala は強力な型推論を持っており、型の宣言は必要最低限に留め、省略するこ

とが可能である。

また Scala には Tuple というデータ構造がサポートされており、これは任意の型のデータを要素とするこができるコレクションであり、型を定義することなく利用することができる。

例えば、

```
val p = (10,"hoge")
```

と記述することにより、Tuple のオブジェクトが生成される。そして変数 p は型推論され、 Tuple2[Int,String] 型であると暗黙的に型付けされる。(val は不変な変数を宣言するキーワードである。) 従って目的のレコードとしての機能を果たしているように見える。しかし、この Tuple を単純にレコードして扱うには以下のような問題点がある。

- 変更不可変なオブジェクトである
- メンバのアクセスがインデックスによって行われる
- Tuple 間にサブタイプ関係がない

1 つ目は、一度 Tuple オブジェクトが生成されると、その後メンバの代入・追加などが不可能であることを指す。従って値を変更したい場合、新しい Tuple を生成する必要がある。

2 つ目はメンバのアクセスがインデックスでされており、意味のある名前をメンバに付けられないということと、それにより順番が異なるだけで Tuple の型が変わってしまうという点が問題である。例えば ("hoge",10) という Tuple を生成したとすると、その型は Tuple2[String,Int] となり、前述の Tuple2[Int,String] とは異なるものとなる。

3 つ目に異なる Tuple 間にサブタイプ関係がないというのは、つまりはポリモルフィズムがないということである。従って汎用性の低さが問題として挙げられる。

以上のことから、Tuple を目的とするレコードとして扱うことは難しい。

Scala と同様に、型推論が用いられるプログラミング言語は多く存在するが、やはり同じように制約があり、型定義せずにレコードを柔軟に利用するのは難しいようである。

第3章　暗黙的に型定義されるレコード

3.1　本システムの概要

本研究では、前述したように、静的型付き言語において、レコードを簡便に利用できるようにすることが目的である。それを可能にするため、レコードの型が暗黙的に定義され、その型が付与される型システムを提案する。

本システムでは、レコードの型を structural type を用いて表現し、ソースコードからレコードの型を推論してレコードの型を決定する。このようにすることで、静的型付けでありながら、動的型付き言語のようなシンプルな記述法で、任意の名前・型のメンバを持つレコードを利用することが可能となる。また本システムのレコードでは擬似的にメンバの追加も行うことができる。つまり生成時ではなく、後にメンバを追加できる。これにより汎用性の高いレコードを実現している。

また本研究では、この型システムを Java に組み込んだ言語を実装した。これ以降はこの拡張 Java 言語を用いて説明を行う。

3.2　レコードの生成・利用

本システムでは、基本的に空のレコードを生成して、そのレコードにメンバを追加する。空のレコードは

```
var p = new();
```

```
1 var p = new();
2 p.name = "hoge";
3 p.age = 20;
4 int i = p.age;
```

図 3.1: 本システムにおけるレコードの生成

のように記述して生成できる。new() は空のレコードを生成するためのキーワードである。これによって生成されたレコードをレコードオブジェクトと呼ぶ。そしてレコードオブジェクトを代入する変数を宣言する際に var というキーワードを用いる。

var というキーワードは様々な他言語でも変数宣言に用いられているが、本システムではレコードオブジェクトを参照する変数であり、レコードに対応する型が型推論され暗黙的に付けられることを意味する。ちなみに C # では型推論される変数を表すために var が用いられていおり、意味としては本システムに近い用法である。

レコードオブジェクトを用いると、擬似的に任意の名前かつ任意の型のメンバを自由に追加できる。例えば、図 3.1 のコードでは、p は String 型の name というフィールドと int 型の age というフィールドを持つレコードであることになる。

また、初期値を与えている 1 行目から 3 行目を

```
var p = new{name:"hoge", age:20}
```

と略記することでより簡潔に記述できるが、本論文中では後の型推論等の説明のため略記は用いていない。

そして一般的なレコードと同様に、レコードのメンバへのアクセスは、(レコード).(メンバの名前) という構文で行われる。これにより、Java では、前章の図 2.2 のように記述しなければならなかったコードは図 3.1 のように記述することができ、動的型付き言語と同じような簡潔さでレコードを利用できるようになっている。

そしてレコードの型については以下で詳しく説明するが、静的な型付けは維持されている。従って、例えば図 3.1 のプログラムにおいて、

```
int i = p.num;
```

のように p の持っていないフィールドにアクセスしようとしたすると、同様のコードを JavaScript において記述した場合、実行時エラーになるのに対して、本システムではコンパイル時に型検査されるるため、コンパイルエラーとすることができる。

3.3　レコードの型推論

前述した通り、本システムではレコードオブジェクトを指す変数にはそのレコードに応じた型が暗黙的に付けられる。この節では付けられる型がどのような意味をもち、どのように定義されるのかを説明する。

3.3.1　structural type の利用

本システムではレコードの型は structural type を用いて表している。structural type は前章で説明したとおり、型を構造 (メンバの集合) として見た時に同じなら型が等しいと判断される型システムである。本システムにおける、レコードに対して与えられる structural type とは、レコードが持っている (アクセス可能な) メンバとその型の集合である。つまりアクセスできるフィールドが同じであるならば同じ型という事である。例えば図 3.1 の例では、変数 p が指すレコードオブジェクトは、 String 型の name と int 型の age がメンバに追加 (代入) されている。本システムでは、その代入式から p の型は {name: string, age: int} であると判断される。これが本システムでの一番基本的な型の決定法である。

3.3.2　レコードのメンバの型の決定

基本的な型の決定法は前小節での説明の通り、代入されたオブジェクトの型をメンバの型とするというシンプルな方法である。しかし、それだけでは不十分である。考慮しなくてはならないのは、レコードのメンバに対して再び代入がある場合である。

同一のメンバに対して、同じ型のオブジェクトが複数代入されている場合、前章と同様にその型自身がそのメンバの型となる。一方、異なる型のオブジェクトが同一のメンバに代入されている場合、本システムではそのメンバの型は代入されている各オブジェクトの型の least upper bound である型となる。least upper bound である型とは、それらの型の共通のスーパータイプの中で、最もサブタイプである型のことである。このように型付けされるのは、そのフィールドはそれら全てのオブジェクトが代入可能な型 (つまりそれらのスーパータイプ) であると判断され、かつできるだけ意味のある型を付けようとするからである。

例えば図 3.2 の例では，shape の型は Circle と Square の least upper bound の型である Shape となり、a の型は {shape: Shape} となる。ただ単に代入可能であるならば，shape を Object 型にすれば良いが、それでは shape は Object 型としてしか利用できない。一方、shape を Shape 型とすることで Shape クラス内の情報は保持され、またポリモルフィズムも失われない。

異なる例として、以下のようなプログラムを考える。

```
var a = new();
a.name = "hoge";
a.name = 1;
```

```
1  class Circle{
2    …;
3  }
4  class RedCircle extends Circle{
5    …;
6  }
7  class BlueCircle extends Circle{
8    …;
9  }
10
11 var c = new();
12
13 c.circle = new RedCircle();
14 c.circle = new BlueCircle();
```

図 3.2: メンバへの異なる型のオブジェクトの代入

このプログラムは、String と int というまったく関係のない型のオブジェクトを同一のフィールドに代入している。従って、静的型付け言語では直観的にはコンパイルエラーとなるプログラムである。しかし int 型はオートボクシングにより Integer 型となり、フィールド name の型は、String 型と Integer 型のアッパーバウンドの型である Object 型となり、問題なく実行できてしまう。つまり、どんなオブジェクトも代入できる代わりに、意図しない代入があってもそれを許可してしまうということである。それによって意図しない問題が実行時に発生するように感じられる。しかし実際は、意図しない振る舞いが実行時に起こることはない。なぜなら name の型は Object 型であるため、Object 型以外として利用とした時に、その時点で型エラーが発生するからである。もし name を Object 型としてのみ利用しているならば、型エラーは発生しないが、そもそもそのプログラムは問題の無いプログラムである。

3.4　レコードオブジェクトの代入

var によって宣言された変数はレコードオブジェクトを指す変数である。従って、var 型の変数に既に生成したレコードオブジェクトを代入することも可能である。例えば、図 3.3 のようなプログラムを書くことができる。この場合、a と b は同じレコードオブジェクトを参照する変数となり、当然一方の変数が指すレコードオブジェクトのメンバの値を変更すれば、

```
1  var a = new();
2  a.name = "hoge";
3  var b = a;
```

図 3.3: レコードオブジェクトの代入

もう一方の変数からアクセスした場合のメンバの値も変更されている。そして変数 a の型は {name: String} であり、変数 b の型は変数 a の型と同様に {name: String} となる。

ただし、このようにレコードの代入があった場合、後に代入された変数 b からアクセスして、メンバを追加することはできないという制限が付いている。逆に言えば、メンバを追加できるのは、レコードオブジェクトを生成したとき、つまり new() を代入した変数を用いた場合のみである。ただしその変数 (例えば a) に他のレコードが代入されている場合も、その代入以降ではメンバは追加されない。

また通常の変数と同様に、1 つの var 型の変数に対し複数回レコードオブジェクトを代入することも可能である。例えば図 3.4 のようなプログラムが書くことができる。前節の型付け規則より、a の型は {name: String, age: int}、b の型は {name: String, height: int} である。そしてこのプログラムにおいて、変数 c の型は {name: String} と決定される。これは前節のメンバの型の決定と同じ考えで、変数 c には a,b の両方が代入されているが、実際に付けられる型は 1 つであり、変数 c が指すレコードオブジェクトが a,b のどちらを指しているのかが分からない状態で、c がアクセス可能なフィールドは a,b の共通するフィールドのみだからである。

また structural typing を考えれば、c に a と b が代入可能であるということは、c の型は a の型と b の型の共通のスーパータイプでなければならない。従って、structural typing のサブタイプ関係より、c の型は φ (アクセスできるフィールドがない) もしくは {name: String} のいずれかであり、より意味のある後者を選ぶべきである。つまりこれは、前節におけるメンバの型が代入されるオブジェクトの型の least upper bound である型になるのと同様に、structural type における least upper bound である型が変数 c の型となるのである。以上から、変数 c は最終的に b が指すレコードオブジェクト (name と height を持ったレコード) が代入されているが、 c からアクセス可能なメンバは name のみとなる。

```
1  var a = new();
2  a.name = "hoge";
3  a.age = 1;
4
5  var b = new();
6  b.name = "foo";
7  b.height = 170;
8
9  var c;
10 c = a;
11 c = b;
```

図 3.4: 複数のレコードオブジェクトの代入

3.5　レコードオブジェクトのメソッド間の受け渡し

Java 言語に本システムを組み込んだ本言語では、より汎用的にレコードを利用できるようにするため、レコードオブジェクトのメソッド間の受け渡しをサポートしている。具体的には、メソッドの仮引数に既存の型でなく var を用いることで、メソッドにレコードオブジェクトを渡すことができ、またメソッドの返り値の型に var を用いることで、返り値としてレコードトオブジェクトを返すことができる。これまでに説明したレコードオブジェクトと var の用法は、メソッド内のローカルな利用だけであったが、これによってよりグローバルな利用が可能となる。

メソッドの仮引数として var を用いた場合、その仮引数が表すレコードオブジェクトが、任意の型かつ任意の名前のメンバをもっているようかのようにメソッド内で利用することができる。ただし利用法としては、var を用いて宣言した仮引数を x とすると、以下の 2 通りに限られる。それは、

```
String s = x.name;
```

という形のメンバが指すオブジェクトを他の変数へ代入する式と、

```
x.age = 20;
```

という形のメンバへの代入である。メンバへの代入式は引数のレコードがその名前のメンバを持っていなければそのメンバは追加され、既にある場合は代入される。つまりプログラマは、var で宣言された仮引数 x が、既に String 型の name というメンバを持ち、かつ age というメンバに int 型のオブジェクトが代入できるということを意識せずにメソッド内で自由に記述することができる。

```
1 void f(var x){
2   String s = x.name;
3   x.age = 20;
4 }
5
6 var r = new();
7 r.name = "foo";
8 f(r);
```

図 3.5: メソッドの仮引数に var 型を用いた例

var で宣言された仮引数にも暗黙的に型が付けられる。型の意味としてはこれまでと同様に、アクセス可能なフィールドの集合であるが、前述の２種類のメンバに対応して２種類の structural type で分けられ、それらを列挙することで表す。

一方は図 3.5 の name のように、実引数が最低限持っているべきメンバの集合であり、これまでと同様に {name: String} で表わされる。これはメソッド呼び出し時の型検査で用いられ、実引数 r が {name: String} を持っていなければ (この型のサブタイプでなければ)、タイプエラーとされる。

他方は図 3.5 の age のように、メソッド内で代入され得るメンバの集合である。これは、メソッド内でメンバが代入されることを表し、() 付きで表わされ (age: int) と表現される。この集合はメソッド呼び出し側における実引数の型推論時に利用される。つまり、メソッドの実引数である r はメソッド内で int 型のオブジェクトが代入されるので r の型推論時に {age: int} も追加される (age 対して int 型のオブジェクトの代入があると判断される)。よって、 r の型は結果として {name: String, age:int} となる。

また r のメンバとして既に age が存在している場合、つまりメソッド呼び出し側で

```
r.age = exp;
```

というコードがある場合も同様に、メソッド内で age というメンバに int 型のオブジェクトが代入されるので、r.age の型は (他に代入がなければ) int 型と exp の型の least upper bound である型になる。

ただし注意すべき点は {} と () の区別は、あくまでメソッド呼び出し時のタイプチェックや実引数の型推論に用いられるためのものであり、前述したとおり、どちらも x がアクセスできるメンバということに違いはな

く、メソッド内のみを考えれば、これまでと同様に {name: String, age: int} と同じ意味である。

またより複雑な例としては、仮引数のレコードの同一のメンバに対して、メソッド内で複数の代入・参照がある場合が考えられる。例えば、図 3.6 のような例である。このような場合、各メンバの型は以下のアルゴリズムに則って決定される。

1. メソッド内で初めて現れるのが、代入式の右辺なら {} 付きの型に含まれる。つまり実引数に必要なメンバ
2. {} 付きの型に含まれるメンバの型は、そのメンバが必要とされる型 (代入式の左辺の変数の型) 全ての中でサブタイプ関係があれば、その中の最もサブタイプな型。サブタイプ関係の無いものがあれば型付け不可能 (タイプエラー)
3. メソッド内で初めて現れるのが、代入式の左辺なら逆に () 付きの型、つまり実引数に代入されうるメンバ
4. () 付きの型に含まれるメンバの型は、そのメンバに代入されるオブジェクトの各型の least upper bound である型

このアルゴリズムから、仮引数 x の型は {a: Object, c: String}(b: Object) と推論される。

1 と 3 については、{} 付き () 付きの意味から考えて自然な分け方である。そして 2 と 4 のメンバの型の決定法については、それぞれメンバが代入される変数の型・メンバに代入するオブジェクトの型のみを考慮に入れている。つまり、図 3.6 の a,b といったメンバのように、代入式の左辺と右辺の両方で用いられている場合は、他方は型推論には用いられない。これは一見すると、不都合なことが起こりそうであるが、付けられた型によって既存の型検査を行うと問題がないことが分かる。図 3.6 の例の b というメンバについては、代入式の関係から

```
String <: x.b の型 <: Object
```

とならなければいけないが、 実際は

```
Object <: String
```

であるから明らかに間違ったコードであるが、型推論時には x.b の型は Object であると問題無く決定されてしまう。しかし、型が決定された後に 5 行目の型検査において、

```
String <: x.b の型＝ Object
```

```
1 void f(var x){
2   Object o = x.a;
3   x.a = "hoge";
4   x.b = new Object();
5   String s2 = x.b;
6   String s3 = x.c;
7   Object o = x.c;
8 }
```

図 3.6: メソッドの仮引数にレコードを用いた例

```
1   var foo(){
2     var x = new();
3     x.name = "hoge";
4     x.age = 20;
5     return x;
6   }
```

図 3.7: メソッドの返り値にレコードを用いた例

が正しいかという型検査が行われるので問題なく、タイプエラーを検出できる。

メソッドの引数の型推論は、実際はいろいろなアルゴリズムが考えられるかもしれないが、本アルゴリズムでも問題無いことがわかる。

メソッドの返り値の型を var 型とした場合、メソッドの返り値をレコードオブジェクトとすることができる。こちらの型付けは単純で、メソッドの返り値として返されるレコードオブジェクトに付けられる型がそのままメソッドの型となる。

例えば、図 3.7 のようなメソッドがあった場合、x の型は {String: name, int: age} であるから、同様に foo の返り値の型は {String: name, int:age} である。そして、

```
var y = foo();
```

と foo メソッドを呼びだすと，y の型は foo の返り値の型と同様に {String: name, int;age} となる。

以上がレコードをメソッドで受け渡す方法であるが、これを用いる上で制限がもう 1 つ存在する。それは、これらのレコードを受け渡すメソッドの中から、同じようにレコードを受け渡すメソッドを呼ぶことができない

ということである。これは、仮引数や返り値の型を推論するとき、他のメソッドの推論結果を必要とすると、その推論のループが起こってしまう可能性があるからである。推論のループがなければ、理論上は型を決定することが可能であるが、そもそもたくさんのメソッドを渡った推論結果を把握しつつプログラミングするのは難しいため、本来の目的に合うものではない。従ってそのような制限を設けている。

3.6　レコードのメンバの代入に他のレコードが用いられる例

本システムでは、任意の型のオブジェクトをレコードのメンバとすることができる。従って、他のレコードのメンバやレコード自身もレコードのメンバにすることができる。

例えば、図 3.8 の例では、

```
r2.rec = r1;
```

の代入式によって、r2 の rec というメンバには r1 の指すレコードオブジェクトが代入されている。そしてこれまでと同様に、 r2.rec の型は代入されているのが r1 だけなので、r1 の型である {a:int} が付けられる。

これだけをみると、これまでの規則と同様に型を決定できそうであるが、必ずしも型を決定できないということが注意すべき点である。例えば、図 3.8 の r3.self には r3 自身が代入されているため、r3.self の型は r3 の型そのものである。しかし、r3.self の型を推論している時点では r3 の型は決定していないため、r3.self の型も決定することができない。従って、型付けが失敗となりエラーとなる。相互参照しているような場合も、このようなエラーとなる。

具体的に型推論は以下のようなアルゴリズムで行われる。

1. レコードに関する全ての代入式を走査し、型についての制約式の集合を生成
2. 代入されるオブジェクトの型が決定されるものから制約式を解く。
3. 制約式が全て解ければ成功、そうでなければエラー。

更なる詳しいアルゴリズムは次章の形式化で説明している。

アイデアとしては、再帰型を利用すれば、上記のような型を表現することも可能である。つまり、 r2 の型は {rec: {a:int}, self:(r2 自身の型)} であるから、μX.{rec: {a: int}, self: X} と表せられる。しかし、再帰型を用いると、そのサブタイプ関係等の扱いが複雑となること、またその複雑

```
1  void f(){
2   var r1 = new();
3   var r2 = new();
4   var r3 = new();
5   r1.a = 1;
6   r2.rec = r1;
7   r3.rec = r2.rec;
8   r3.self = r3;
9      :
10 }
```

図 3.8: レコードのメンバにレコードを追加する例

な再帰型まで暗黙的に付けた場合、プログラマが把握するのが困難であることから本システムではサポートしないことにした。

3.7　typeof

本システムでは、前述したように暗黙的にレコードの型付けを行っている。暗黙的に型付けがなされることで、プログラマが書くコードを省略することが可能となるが、一方ではプログラマがその暗黙的に定義された型の名前を認識できないという欠点も存在する。型の名前を認識できないということは、明示的に型の名前を指定できないということである。型の名前を明示的に指定する最も一般的な例は、変数宣言・インスタンスの生成などである。本システムでは、変数宣言時は var というキーワードが用いられ、レコードオブジェクトの生成は new() というキーワードで行われている。従って、一見すると必要がないように感じられるが、他にも以下のような利用法が存在する。

- レコードの型をジェネリクスの型変数に渡す
- instanceof による型の比較

従って、明示的に型を指定する機能はやはり提供する必要がある。

そこで本システムでは、 typeof 演算子を提供する。typeof 演算子は上記のように、型を明示的に記述できる部分でのみ用いられ、typeof(exp) と記述することで、 exp の静的な型を記述することと同様の意味とすることができる。基本的に exp はレコードであることを目的としているが、既存の型をもつ任意の表現を引数として取ることが可能である。

```
1  var p = new():
2  p.name = "hoge";
3  p.age = 5;
4
5  var q = new();
6  q.name = "foo";
7  q.age = 10;
8
9  Map<String, typeof(p)> map =
10         new HashMap<String, typeof(p)>();
11 map.put("p",p);
12 map.put("q",q);
13
14 var q2 = map.get("q");
```

図 3.9: typeof を用いたプログラム例

typeof 演算子を用いることで図 3.9 のようなコードが記述することが可能となる。このプログラムでは p というレコードを生成し、それを Map に格納している。ジェネリクスを用いて Map には String をキーとして、p と同じ型 (もしくはサブタイプ) のオブジェクトのみ格納できるようになっている。しかし p はレコードを指す変数なので、p の型は暗黙的に付けられる structural type である。従って、Map に格納できるオブジェクトは p と同じ構造を持つレコード、つまり String 型の name と int 型の age をメンバに持つレコードのみとなる。また、格納されているオブジェクトはレコードであるから、今までと同様に、取り出したレコードを var を用いて宣言した変数に代入することも可能である。

第4章　形式化

本システムでは、型付け不可能なものも含めて、様々な状況が考えられる。本システムでの型推論・評価規則がより詳細に分かりやすくするため、本言語の核となる機能を取り出したサブセットの言語を用いてシステムの形式化を行った。

形式化には Java 言語の核となる機能のみをもつ Featherweight Java [6, 10](以下 FJ) をベースにして、本研究の暗黙的に型が付けられるレコードを導入した言語を用いている。ここではこの言語を Featherweight Record Java (以下 FRJ) と呼ぶ。

なお、FRJ は FJ をベースとしているため、評価規則や型付け規則の多くが重複している。従って本章では、本システムに関連した部分のみを説明し、それ以外の FJ に由来する部分は本論文末尾の付録部分に掲載している。

また説明のため、FJ の特徴を以下に示す。

- 項 (expression) として、オブジェクト生成・メソッド呼び出し・フィールドアクセス・キャスト・変数のみを持つ。
- メソッドノディは返り値のみであり、副作用もない

4.1　構文とサブタイプ関係

FRJ の構文・サブタイプ関係は図 4.1 の通りである。FRJ は FJ をもとに作られているため、構文も FJ を拡張したものになっている。FJ では簡潔にモデル化できるように、上記の通り副作用つまり代入も存在しない。

しかし本システムでは、レコードのメンバの追加・型推論ともに代入によって行われているため、代入無しの構文では難しい。従って FRJ では、メソッドボディは既存の Java のように statement にしており、その statement ではレコードに関する代入と return 式のみ許可しており、r は var で宣言されるローカル変数を表している。

サブタイプ関係では既存のクラスのサブタイプ関係に加え、図 4.2 で示している本システムで用いられるレコードを表す structural type におけ

$$
\begin{array}{l}
CD ::= class\ C\ extends\ C\ \{\overline{C}\ \overline{f}; K\ \overline{M}\} \\
K ::=\ C(\overline{C\ f})\{super(\overline{f});\ this.\overline{f} = \overline{f}\} \\
M ::=\ C\ m(\overline{C}\ \overline{x})\{S\ return\ e\} \\
S ::=\ var\ r = new();\ |\ var\ r = e;\ |\ r = e;\ |\ r.n = e;\ |\ SS \\
e ::=\ x\ |\ r\ |\ e.f\ |\ e.n\ |\ e.m(\overline{e})\ |\ new\ C(\overline{e}) \\
v ::=\ new\ C(\overline{v})\ |\ \{\overline{n : v}\}
\end{array}
$$

図 4.1: Syntax in FRJ

$$
\begin{array}{l}
\quad i <= j \\
T_a = \{T_1 n_1, \cdots T_j n_j\} \\
T_b = \{T'_1 n_1, \cdots T'_i n_i\} \\
\dfrac{\forall k, k = 1 \cdots i, T_k <: T'_k}{T_a <: T_b}(ST - RECORD)
\end{array}
$$

図 4.2: sutructural type の subtyping

るサブタイプ関係を追加する。

4.2　reduction(評価) 規則

FRJ における、項の評価規則とメソッドの statement の評価規則はそれぞれ、 図 4.3, 4.4 の通りである。ここでは Δ は値環境を表しており、代入された変数とその変数が指すオブジェクト (FRJ ではレコードのみ) の情報を保持している。また $< \Delta, S >$ は環境 Δ で statement S を評価することを表す。そして、(R-DECL-OR-ASSIGN) における var* の*はオプションを表しており、var がある場合 (変数宣言時) とない場合のどちらでもこの規則が使えるという意味である。

メソッド呼び出しにおける mbody(m,C) はクラス C で定義されているメソッド m のメソッドボディを表しており、(仮引数,statement,return 式) の三つ組が返される。FRJ ではメソッドの statement でのみレコードに関する代入が行われる。従ってメソッド呼び出しを評価する際は、メソッドの statement を評価して、その結果の値環境を用いて return 式を評価すれば良い。

$$\frac{e \to e'}{e.n \to e'.n}(R-MEMBER-OR-FIELD)$$

$$\frac{n = n_i}{\{\overline{n:v}\}.n \to v_i}(R-MACCESS)$$

$$\frac{\begin{array}{c} mbody(m,C) = (\overline{x}, S, e) \\ <\phi, S[\overline{x} \mapsto \overline{u}] > \to <\Delta, \phi > \\ \Delta \vdash e[\overline{x} \mapsto \overline{u}] \to v \end{array}}{new\ C(\overline{v}).m(\overline{u}) \to v}(R-INVOKE)$$

図 4.3: expression reduction in FRJ

FRJ の statement は大きく分けると、レコードの初期化・レコードの代入・メンバの代入の 3 つからなる。statement の評価は 型推論が終わった後に行われ、レコードを生成する場合には、その型の情報を用いている。つまり、推論結果はそのレコードが持っているはずのメンバの集合であるから、そのメンバをもった (ただしメンバには何も代入されていない) レコードが作られ、値環境に保存される。

4.3　型推論・型付け規則

本システムでは型推論を用いて型付けを行っており、代入式を走査して、その型を決定している。FRJ では代入は全て statement の中で行われるので、statement から各レコードを指す変数の型を推論する。

FRJ の型推論は、制約式の収集と収集された制約式を解いてレコードの型を決定するという 2 段階に分けて行われる。

4.3.1　制約式の収集

制約式を収集する規則は図 4.5 で示される。型推論を行うための制約式は、

$$c = (name, isfixed, assignedMember, assignedRecord)$$

という形で表される。name はレコードを指す var で宣言された変数の名前、isfixed は true ならば、もうメンバが追加されないこと表し false であれば、まだ追加される可能性があることを表す。assignedMember は

$$\frac{r : \{\overline{T\ n}\}}{< \Delta, var\ r = new(); S > \rightarrow < \Delta[r \mapsto \{\overline{n : null}\}], S >}(R - DECL - INIT)$$

$$\frac{\Delta \vdash\ e \rightarrow v}{< \Delta, var^*\ r = e; S > \rightarrow < \Delta[r \mapsto v], S >}(R - DECL - OR - ASSIGN)$$

$$\frac{\begin{array}{c}\Delta \vdash\ e \rightarrow v \\ \Delta \vdash\ r \rightarrow \{\overline{n : v}\} \\ v' = \{\overline{n : v}[n_i \mapsto v]\}\end{array}}{< \Delta, r.n_i = e; S > \rightarrow < \Delta[r \mapsto v'], S >}(R - MEMBER - ASSIGN)$$

図 4.4: statement reduction in FRJ

そのレコードのメンバとそのメンバに代入されたオブジェクトの組であり、assignedRecord はその変数自身に代入された他のレコードを表す。また、C は制約式 c の集合を表し、$< C, S >$ の C は現在ある制約式の集合、S はこれから制約式を収集する statement を表す。収集規則の中にある $\{\overline{n : e}\}.put(n_0, e_0)$ は $\overline{n}$ の中に n が存在すれば $\overline{e}$ に e_0 を加え、なければ $n_0 : e_0$ を加える。制約式を集めるアルゴリズムはシンプルで、主に以下のことをまとめたのが収集規則となっている。

- 変数宣言時に名前がかぶっていないかのチェック
- メンバ・レコード自身の代入を保存
- レコードが代入された後はメンバが追加されない

例えば、statement が

```
var r = new();
r.a = new T();
r.a = new U();
var r2 = r1;
var r3 = new();
r3.b = r.a;
```

$$\frac{(r,,,) \notin C}{< C, var\ r = new(); S >\ \rightarrow < C \cup (r, false, \phi, \phi), S >}(C - DECL - INIT)$$

$$\frac{(r,,,) \notin C}{< C, var\ r = e; S >\ \rightarrow < C \cup (r, true, \phi, \{e\}), S >}(C - DECL - ASSIGN)$$

$$\frac{\begin{array}{c} c = (r, isfixed, \{\overline{n : e}\}, \{\overline{e'}\}) \\ c \in C \\ c' = (r, true, \{\overline{n : e}\}, \{\overline{e'}\ e_0\}) \end{array}}{< C, r = e_0; S > \rightarrow < C[c \mapsto c'], S >}(C - ASSIGN)$$

$$\frac{\begin{array}{c} c = (r, false, \{\overline{n : e}\}, \{\overline{e'}\}) \\ c \in C \\ c' = (r, false, \{\overline{n : e}\}.put(n_0, e_0), \{\overline{e'}\}) \end{array}}{< C, r.n_0 = e_0; S > \rightarrow < C[c \mapsto c']; S >}(C - MEM - ASSIGN1)$$

$$\frac{c = (r, true, \{\overline{n : e}\}, \{\overline{e'}\}) \quad c \in C}{< C, r.m_0 = e_0; S > \rightarrow < C; S >}(C - MEM - ASSIGN2)$$

図 4.5: 型推論のための制約式の収集

のようになっている例を考える。すると各収集規則より、

```
C = {(r,true,{a: new T(),new U()},{}),
     (r2,false,{},{r}),
     (r3,true,{b: r.a},{})}
```

という制約式の集合が作られる。

4.3.2　制約式の解

制約式から型を推論するアルゴリズムは図 4.6 で示している。また本システムでは型推論の過程で、既に確定した他のレコードやメンバの型を用いるため、そのための型付け規則もまとめて示している。

まず各レコード (制約式 1 つ) に対して、その結果を保存する R_{name} が作られる。この R_r は

$R_r = (assignedMembern, assignedRecord, memberType)$

という形で表わされる。assignedMenber,assignedRecord は制約式と同じであり、memberType は暫定の推論結果を表す。従って各 R_r は

$for\ c\ in\ C, R_{c.name} = (c.assignedMember, c.assignedRecord, \{\})$

となる。またレコード全体として、型が決定したら R にその情報が保存される。

FRJ の型推論は初めに new() によって初期化されたレコードとそうでないものに分けられる。もし初期化されていなければ、メンバは追加されないので、代入されたレコードのみを考慮に入れ、それらのレコードの型、つまり structural type としての least upper bound の型を推論結果としている。これは、(S-NOT-INIT-REC) の R_r が $(\phi, \bar{e}, \phi)$ である時である。推論規則中 LUB は least upper bound　を表している。

逆に new() によって初期化されている場合、assignedMember から型が決定可能なメンバがあれば、そのメンバを assignedMembar から取り除き、型を membarType に登録している。型はそのメンバに代入されているオブジェクトの型の least upperbound の型である。初期化されたレコードはメンバが追加されるため、assignedMember か memberType のどちらかが存在する。従って (S-NOT-INIT-REC) が適用されることはない。

推論は以上の流れで行われるが、これらの推論は代入されるレコード・メンバに代入されるオブジェクトの型が分かっているのが前提である。従って、他のレコードやレコードのメンバを参照していて、まだその型が分からない場合、その推論は後回しにされる。その型が推論済みかどうかは R_r や R から判断している。そして逆に、他の型が未決定であることなどにより、制約式が残ったまま推論規則が適用できない場合、型付け不可能となりエラーとなる。前章でも説明したが、これは互いにレコードを参照しあっている場合等で発生する。

以上から前節での例の

```
C = {(r,true,{a: new T(),new U()},{}),
     (r2,false,{},{r}),
     (r3,true,{b: r.a},{})}
```

という制約式をこの推論規則を用いて解く場合、T と U が T:>U というサブタイプ関係があるとすると、

```
R = {r:{a:T}, r2:{a: T}, r3:{b:T}}
```

が推論結果となる。

$$\frac{\begin{array}{c} n : \overline{e} \in R_r.assignedMember \\ \Gamma \vdash \overline{e : T} \end{array}}{\begin{array}{l} R_r.assignedMember.remove(n : \overline{e}) \\ R_r.type.add(n : LUB(\overline{T})) \end{array}}(S - INIT - MEMBER)$$

$$\frac{R_r = (\phi, \{\overline{e}\}, T)}{R = R \cup \{r : T\}}(S - INIT - REC)$$

$$\frac{R_r = (\phi, \{\overline{e}\}, \phi) \quad \Gamma \vdash \overline{e : T}}{R = R \cup r : LUB(\overline{T})}(S - NOT - INIT - REC)$$

$$\frac{n : T \in R_r.membarType}{\Gamma \vdash r.n : T}(T - MEMBER)$$

$$\frac{r : T \in R}{\Gamma \vdash r : T}(T - REC)$$

図 4.6: 型推論規則とその過程での型付け規則

$$\frac{r : T \in R}{\Gamma \vdash r : T}(T - REC)$$

$$\frac{\Gamma \vdash r : T \quad T\ is\ recordType \quad n : T' \in T}{\Gamma \vdash r.n : T'}(T - REC - MEMBER)$$

図 4.7: 型検査時の型付け規則

4.3.3　Typing rule

型推論ではなく、型検査のときに用いられる型付け規則を図 4.7 に示す。この時点で、型推論によってレコードの型は決定しているので、そのレコードとそのメンバの型はその推論結果を用いられる。前節の型付け規則とほとんど同じであるが、ここでは推論結果である R だけを用いている。

第5章　実装

本研究では、提案した暗黙的に型定義するレコードを組み込んだ拡張 Java 言語を実装した。本言語の実装では、JastAdd [2, 3, 11] を利用した。JastAdd は Java-based なコンパイラを実装するシステムであり、この JastAdd で実装された Java コンパイラである、 JastAddJ を拡張することで本言語を実現している。

本言語ではレコードオブジェクトや structural type といった、既存の Java 言語にはない言語機構を導入しているが、実際は標準の Java 言語への変換と型検査によってこれらを実現している。この章では、本言語のプログラムがどのように標準 Java 言語のコードに変換され、本システムが実現されているかを説明する。

5.1　ソースコード変換

本言語では、暗黙的にレコードの型が付けられた後、型検査が行われる。この型検査の実装については単純で、以下の点を既存の型検査等のルールに加えている。

- structural type についてのサブタイプ関係を追加
- レコードのメンバアクセスが正しいか (レコードの型にそのメンバがあるか)

```
1 void hoge(){
2   var x = new();
3   x.name = "hoge";
4   x.age = 22;
5   System.out.print(x.name);
6 }
```

図 5.1: 本言語を用いたサンプルコード

```
1   class record_name$$01
2         implements StringName,IntAge{
3
4     private String name;
5     private int age;
6
7     public void getName(){return name;}
8     public void setName(String name)
9                         {this.name = name;}
10    public int getAge(){return age;}
11    public void setAge(int age)
12                        {this.age = age;}
13  }
```

図 5.2: レコードを表すクラス

この型検査によって、型に関してはプログラムが正しいと判断された後、本言語ではレコードを用いた場合と同様な振る舞いをする標準の Java 言語のプログラムに変換され、実行可能なプログラムを生成する。本言語において、行われるコード変換を図 5.1 のプログラムを用いて説明する。

まずはじめに、レコードオブジェクトはそのレコードに対応するクラスが作られ、そのインスタンスに変換される。この対応するクラスとは、レコードの各メンバに対して、そのメンバを表すフィールド・そのフィールドに対応する getter・setter を持ったクラスである。これは、もともとのアイデアが、クラス定義せずにレコードを利用したいというものであったので、暗黙的にそのクラスを作り、レコードをそのインスタンスにするというのは自然な方法である。例えば、図 5.1 の例では、変数 x の型は {name: String, age: int} であると推論されるため、それに対応するクラスとして、図 5.2 のクラスが作られる。ここでのクラス名は、new() で作られたレコードが初めに代入される変数を表す名前である。

一方、レコードオブジェクトが代入される var で宣言される変数の型は、暗黙的に付けられるためソースコード上では存在せず、この型はコンパイル時に推論され、その型が付けられると説明した。しかし、本言語のバイトコード上では、var によって宣言された変数は、全て Object 型であるとされる。それはレコードがそのレコードを表すクラスのインスタンスに変換され、いかなるレコードも代入できるようにするためである。一見すると、レコードをクラスのインスタンスで表すのだから、変数の型もそのクラスの型を付ければ良いように思える。しかし本システムではレ

コードの型を structural type で表しており、structural type のサブタイプ関係を nominal type を用いて表すには、多重継承を用いらなければならないためこのような実装にしている。従って、図 5.1 の例の x の型も Object に変換される。

しかし、変数の型を Object 型とすると、その変数に代入はできるが、その変数をレコードとして用いることはできない。従って、var によって宣言された変数は、そのメンバをアクセスするとき、そのメンバに対応するインタフェースにキャストされるようにした。そして、そのインタフェースに定義されたメソッドを用いて、レコードのメンバにアクセスする。そして、メンバに対応するインタフェースは以下のように定義される。

```
interface StringName {
  public String getName();
  public void setName(String name);
}
interface IntAge {
  public int getAge();
  public void setAge(int age);
}
```

つまり、 x.name にアクセスしようとした場合、x は StringName にキャストされ、代入式ならば setter メソッドに、参照ならば getter メソッドに変換される。従って、図 5.1 において

```
x.name = "hoge";
```

は

```
((StringName)x).setName("hoge");
```

に変換される．図 5.1 のコードを全て変換すると図 5.3 のようになる。

これらの変換は型検査が成功した後に行われるため、暗黙的に付けられた型がフィールドにアクセスできることを保証しているので、キャストは必ず正しく行われる。もしも正しくないプログラムである場合は、型付けの時に型付け不可能となるか型検査時に型エラーが発生するので問題はない。

以上の変換によって、このプログラムは暗黙的に付けられた型の意味に合い、Java で実行可能なプログラムにすることができる。

```
1  class Class_method_x$$01
2   implements StringName, IntAge{…}
3  interface StringAge{…}
4  interface IntAge{…}
5
6  void hoge(){
7     Object x = new Class_method_x$$01();
8     ((StringName)x).setName("hoge");
9     ((IntAge)x).setAge(22);
10    System.out.print(((StringName)x).getName());
11 }
```

図 5.3: 図 5.1 の変換結果

第6章　実験

本システムにおける有用性を測るために２種類の実験を行った。１つ目は、本システムがプログラミングを行う上で、どれほど有用に働く可能性があるかを計測する実験であり、もう１つは、本システムのオーバーヘッドを計測する実験である。

本章では、これらの実験についての説明とその考察を述べる。

6.1　本システムで削減可能なクラスの計測

本研究の目的は簡便なレコードの利用であるが、その背景としてあるのは、シンプルなレコードであってもクラス定義等の型定義をしなければならないという事である。つまり本システムを用いることによって、そのような余計なクラス定義等をしなくても済むということである。

そこで、本システムによって省略できる (できる可能性のある) クラスが既存のプログラムにどの程度あるのかを計測する実験を行った。

6.1.1　実験概要

本実験では、Java 言語で実装されている統合開発環境の Eclipse のプログラムを対象に、本システムにより各クラスが省略可能かどうかをランク・種類別に計測する。これは、本システムを用いて省略可能か否かを単純に分離できないためである。従って、一定の基準を設定して、それに当てはまるクラスの数を計測している。

本システムによって省略可能となるクラスは、単純に言えばレコードを表すクラスである。レコードを表すクラスとは厳密にはフィールドのみを持つクラスであるが、一般的に Java のプログラムではカプセル化のため、フィールドを private として getter,setter メソッドを定義する。従って、ここでは図 6.1 のような、フィールドと getter,setter のみを持つクラスを最も基本の省略可能なクラスとしている。ただし getter,setter といっても、単純にフィールドの値の参照・代入をしているだけ (この場合純粋な getter,setter と呼ぶ) でなく複雑なことをしている場合は除いている。また、getter,setter 以外のメソッドがあったとしても、それが簡単なメソッ

```
1  class Person{
2    private String name;
3    private int age;
4
5    Person(String name,int age)
6      {this.name = name; this.age = age;}
7    public String getName(){return name;}
8    public void setName(String n){name = n;}
9    public int getAge(){return age;}
10   public void setAge(int a){age = a;}
11 }
```

図 6.1: 純粋な getter,setter のみをもつクラスの例

ドであれば本システムで表すことができる可能性はある。従って、計測するクラスを以下のように分けた。

- 純粋な getter, setter のみ
- メソッド宣言無し
- 純粋な getter,setter 以外のメソッドが１つ・２つ・３つの各場合
- interface
- abstract class
- 上記以外

ただし、他のクラスを継承しているクラスは継承元のクラスも考慮する。

計測方法としては、本研究で作成した言語の実装でも利用した、Java コンパイラである JastAddJ を用いて、プログラムの抽象構文木を解析することで計測した。

6.1.2　実験結果と考察

実験結果は表 6.1 の通りである。まず初めに純粋な getter,setter のみを持つクラスは、全体 (interface を含む) に対しては 0.99% 実際にインスタンス化できない interface と abstract class を除くと 1.30%であった。当然、クラス全体からすればわずかではあるが、省略可能なクラスという意味ならば有意な数であると思われる。しかもこれらは、本システムを用い

純粋な getter, setter のみ	211
メソッド無し	228
純粋な getter,setter 以外のメソッドが 1 つ	798
純粋な getter,setter 以外のメソッドが 2 つ	677
純粋な getter,setter 以外のメソッドが 3 つ	2912
interface	3390
abstract class	1824
上記以外	13211
計	21401

表 6.1: Eclipse の省略可能なクラスの分類

ることで、単純かつ無駄なコストなく省略できるクラスを表しているため、本システムの有用性は少なくてもあるということは実証できている。

また純粋な getter,setter 以外に多少のメソッドがあるようなクラスも少なくない数存在していることが分かる。これは前小節でも説明したが、そのメソッドの複雑さによって、単純に省略できるかできないかを判断するのは難しい。また仮に、クラス定義を省略できたとしても、そのレコードを利用する側で多くのソースコードを書かなくてはならないというケースも存在する。しかし逆に、純粋な getter,setter に加え簡単な toString メソッドがあるだけというクラスも存在し、このようなクラスは簡単に省略可能である。

メソッドの無いクラスとは、主にある特定の (値の決まっている) データ群を保管するために作られたクラスであり、これはどこからでも参照されるべきものであり、フィールドの値も変化しないため、本システムを適用するクラスではない。

以上結果から、一概にどれだけのクラス定義を省略可能かを判断はできないが、有用性は少なからずあるということを示すことはできている。

また本実験では、既存の Java プログラムに対して本システムを適用して省略でき得るクラスを計測したが、プログラムの設計段階から、本システムがある前提でプログラムを組んだ場合を考えると、既存の Java 言語等での設計とは異なるものとなり、本実験で行ったように、後にシステムを適用したプログラム (実際に書き換えは行っていないが) より、更に簡潔なプログラムを記述することが可能であると考えられる。その点を踏まえると、本システムの利便性は少なくないと思われる。

計測 / プログラム	コンパイル時間	実行時間
Java	1452	23.3
本言語	2048	23.8

表 6.2: プログラムのコンパイル時間と実行時間 (ミリ秒)

6.2　オーバーヘッドの計測

6.2.1　実験概要

本言語でのコンパイラでは前章の通り新たなクラスやインタフェースの作成を含んだ様々なソースコード変換が行われている。それによってどの程度のオーバーヘッドがかかっているか実験・評価する。実験では、マイクロベンチマークを作成し、レコードをクラスとして表した Java のプログラムと、そのレコードをレコードオブジェクトを用いて表した本言語のプログラムのコンパイル時間と実行時間を計測・比較する。

本実験で用いたマイクロベンチマークは、レコードの生成・メンバの代入・メンバの参照を繰り返し行うものである。また実験環境は以下の通りである。

- OS: Windows Vista
- CPU: Intel Core2 Quad 2.66GHz
- Memory: 4GB

6.2.2　実験結果と考察

それぞれのプログラムのコンパイル時間と実行時間はそれぞれ表 6.2 の通りである。

コンパイル時間については、本言語を用いたプログラムは既存の Java のプログラムと比べて約 40%ほどのオーバーヘッドがかかった。これはソースコードの変換・型の推論、クラス・インタフェースの定義等が主なオーバーヘッドの原因として考えられる。本言語の実装では、暗黙的に型を付けるために、ソースコードを全て調べ、var で宣言された変数を用いている部分を全て探し、それを基に型を推論をしている。従ってその走査・推論に多く時間がかかっていると考えられる。また、レコードの各メンバごとに対応するインタフェースを定義し、構造の異なるストラクトオブジェクト毎にクラスを定義しているので、その点もオーバーヘッドにな

ると考えられる。しかし、今回実験に用いたプログラムはレコードの生成・操作を繰り返すプログラムであったため 40%のオーバーヘッドがあったが、現実的にはレコードを利用するのはプログラムの一部分のみと考えられるため、それほど大きな問題ではないと考えられる。

一方、実行時間にはほとんど差はなかった。本言語はコンパイル時に型付けを行い、暗黙的に Java における構造体を表すクラスを生成している。従って、バイトコードとして主に異なるのはフィールドアクセスがメソッド呼び出しになっているという点のみである。よって実行時間に影響を与えることはほとんどなかったと考えられる。

第7章　まとめと future work

7.1　まとめ

本研究では、Java 言語を初めとする静的型付けオブジェクト指向言語において暗黙的に型が定義され、その型が付けられるレコードを実現するシステムを提案した。本システムではソースコードを読み取り、各レコードの型を推論することで暗黙的な型付けをすることで、擬似的に任意のメンバを追加可能なレコードを実現している。そして本システムを用いることで、レコードを利用したいときに、クラスを定義してインスタンスを生成するという一般的なオブジェクトの生成とは異なり、その場所で、かつ簡潔なコードのみを使うことで、そのレコードを利用できるようになる。

また本研究では、このシステムを組み込んだ拡張 Java 言語を実装した。本言語では Java の既存の型システム等を変更することなく、本システムの機能のみを加えることができている。これにより既存の言語に対して、本システムを組み込むことが可能であることも示した。

そして本システムを形式化し、具体的な型推論アルゴリズム等を示した。また実験を行うことで、本システムの有意性・オーバーヘッドについて考察した。

7.2　future work

本研究における改善点・不十分な点について以下にまとめる。

形式化についての議論

本研究では、本システムをモデル化するために本システムのベースとなる機能を組み込んだ言語を用いて形式化を行った。しかし型の健全性や完全性についての議論・証明が不十分であるため、本型システムの詳細を示す為にそれらについて、より明確にする必要がある。

更なる実験

本研究では既存のプログラムを対象に、その程度のクラスが本システムを適用することで省略可能であるかを計測実験を行った。Eclipse のみを対象に実験を行ったが、実験結果が Eclipse 独特なものであるか、一般的な結果であるのかを判断することができないため、更に様々なプログラムに対して、実験を行うべきである。そうすることのによって、どのようなプログラムの場合、より有意性が高いかを知ることも可能になるはずである。

参考文献

[1] Cardelli, L.: Structural subtyping and the notion of power type, *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, New York, NY, USA, ACM, pp. 70–79 (1988).

[2] Ekman, T. and Hedin, G.: The jastadd extensible java compiler, *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, New York, NY, USA, ACM, pp. 1–18 (2007).

[3] Ekman, T. and Hedin, G.: The JastAdd system - modular extensible compiler construction, *Science of Computer Programing*, Vol. 69, pp. 14–26 (2007).

[4] Flanagan, D.: JavaScript 第５版, オライリー・ジャパン (2007).

[5] Gil, J. and Maman, I.: Whiteoak: introducing structural typing into java, *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, New York, NY, USA, ACM, pp. 73–90 (2008).

[6] Igarashi, A., Pierce, B. C. and Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ, *ACM Trans. Program. Lang. Syst.*, Vol. 23, pp. 396–450 (2001).

[7] Martin Odersky, Lex Spoon, B. V.: *Programming in Scala*, artima Press (2008).

[8] Pearce, D. J. and Noble, J.: Implementing a Language with Flow-Sensitive and Structural Typing on the JVM, *Technical Report ECSTR10-23*.

[9] Pearce, D. J. and Noble, J.: Structural and Flow-Sensitive Types for Whiley, *the Workshop on Bytecode Semantics, Verification, Analysis and Transformation*.

[10] Pierce, B. C.: *Types and Programming Laguages*, The MIT Press (2002).

[11] Team, T. J.: JastAdd, `http://jastadd.org`.

付 録A　Featherweight Java の規則

ここでは本研究のシステムの形式化で用いられる、各規則の中で本システムの核ではなく、ベースとしたFeatherweight Java に由来する規則 [6, 10] を引用して示す。多くはFJ の形式化に用いられる規則と同じであるが、本システムの構文等に合わせて変更している部分もいくつか存在する。

$$C <: C$$

$$\frac{C <: D \quad D <: E}{C <: E}$$

$$\frac{CT(C) = class C extends D\{\cdots\}}{C <: D}$$

図 A.1: subtyping

$$fields(Object) = \cdot$$

$$\frac{CT(C) = class C extends D\{\overline{C\ f};\ K\ \overline{M}\} \quad fields(D) = \overline{Dg}}{fields(C) = \overline{D\ g}, \overline{C\ f}}$$

図 A.2: Field lookup

$$\frac{CT(C) = class\ C\ extends\ D\{\overline{C\ f};\ K\ \overline{M}\} \quad B\ m(\overline{B\ x}\{S; return\ e;\}) \in\ \overline{M}}{mtype(m, C) = \overline{B} \to B}$$

$$\frac{CT(C) = class\ C\ extends\ D\{\overline{C\ f};\ K\ \overline{M}\} \quad m\ is\ not\ defined\ in\ \overline{M}}{mtype(m, C) = mtype(m, D)}$$

図 A.3: Method type lookup

$$\frac{CT(C) = class\ C\ extends\ D\{\overline{C\ f};\ K\ \overline{M}\} \quad B\ m(\overline{B\ x}\{S; return\ e;\}) \in\ \overline{M}}{mbody(m, C) = (\overline{x}, t)}$$

$$\frac{CT(C) = class\ C\ extends\ D\{\overline{C\ f};\ K\ \overline{M}\} \quad m\ is\ not\ defined\ in\ \overline{M}}{mbody(m, C) = mbody(m, D)}$$

図 A.4: Method body lookup

$$\frac{mtype(m, D) = \overline{D} \to D_0 \ implies \ \overline{C} = \overline{D} \ and \ C_0 = D_0}{override(m, D, \overline{C} \to C_0)}$$

図 A.5: Valid method overriding

$$\frac{fields(C) = \overline{C\ f}}{(new\ C(\overline{v})).f_i \to v_i} E - PROJNEW$$

$$\frac{e_0 \to e_0'}{e_0.m(\overline{e}) \to e_0'.m(\overline{(e)})} (E - INVK - RECV)$$

$$\frac{e_i \to e_i'}{v_0.m(\overline{v}, e_i, \overline{e}) \to v_0.m(\overline{v}, e_i', \overline{e})} (E - INVK - ARG)$$

$$\frac{e_i \to e_i'}{new\ C(\overline{v}, e_i, \overline{e}) \to new\ C(\overline{v}, e_i', \overline{e})} (E - INVK - ARG)$$

図 A.6: expression reduction

$$\frac{x : C \in \Gamma}{\Gamma \vdash x : C}(T-VAR)$$

$$\frac{\Gamma \vdash e_0 : C_0 \quad fields(C_0) = \overline{C\ f}}{\Gamma \vdash e_0.f_i : C_i}(T-FIELD)$$

$$\frac{\Gamma \vdash e_0 : C_0 \quad mtype(m, C_0) = \overline{D} \to C \quad \Gamma \vdash \overline{e} : \overline{C} \quad \overline{C} <: \overline{D}}{\Gamma \vdash e_0.m(\overline{e}) : C}(T-INVK)$$

$$\frac{fields(C) = \overline{D\ f} \quad \Gamma \vdash \overline{e} : \overline{C} \quad \overline{C} <: \overline{D}}{\Gamma \vdash new\ C(\overline{e}) : C}(T-NEW)$$

図 A.7: expression typing

$$\frac{\overline{x} : \overline{C}, this : C \vdash e_0 : E_0 \quad E_0 <: C_0 \quad CT(C) = class\ C\ extends\ D\{\cdots\} \quad override(m, D, \overline{C} \to C_0)}{C_0\ m(\overline{C\ x})\{S;\ return\ e_0;\}\ OK\ in\ C}(T-METHOD)$$

$$\frac{K = C(\overline{D\ g}, \overline{C\ f})\{super(\overline{g});\ this.\overline{f} = \overline{f}\} \quad fields(D) = \overline{D\ g} \quad \overline{M}\ OK\ in\ C}{class\ C\ extends\ D\{\overline{C\ f};\ K\ \overline{M}\}\ OK}(T-CLASS)$$

図 A.8: Method typing と Class typing