平成21年度 学士論文

暗黙的に型付けされる構造体のJava言語への導入

東京工業大学 理学部 情報科学科
学籍番号 06-0420-6
大久保 貴司

指導教員
千葉 滋 教授

平成22年2月24日

概要

本研究では、暗黙的に型付けされる構造体を局所的に用いることができるシステムを Java 言語に導入することを提案する。Java 言語では構造体はクラスを用いて定義される。しかし、プログラムにおいてデータをまとめるために局所的に構造体を利用したい場合がしばしば存在する。そのような場合、局所的にしか利用されない構造体のために、構造体を表すクラスの定義を分けて記述しなければならないというのは煩わしく感じられる。従って構造体を利用したい時に型の定義を記述する必要なく、暗黙的に型付けされる構造体を生成できると便利である。

既存の Java 言語では、局所的に新たなオブジェクトを簡単に生成するための言語機構として、内部クラス (inner class) や匿名クラス (anonymous class) が提供されている。しかし、これらの言語機構を用いて、目標とする構造体を実現することは難しい。なぜなら、内部クラスは一般的なクラスと同様に明示的にクラスを定義する必要があり、匿名クラスは、既存のクラスを拡張するという方法でフィールドを追加できるが、そのフィールドにオブジェクト外からアクセスすることはできなく、またローカル変数は final でないと匿名クラスの中からアクセスできないなどの制限が付いているためである。

本研究では、型推論を用いて構造体に型を暗黙的に付けるシステムを Java 言語に導入することを提案する。本システムでは、var 型とストラクトオブジェクトを用いることで、プログラマが型の定義を記述することなく構造体を表すオブジェクトを生成し、利用することを可能としている。var 型とは、本システムにおいて構造体を参照する変数の宣言時の型であり、ストラクトオブジェクトとは構造体を表すオブジェクトのことである。それぞれの型はコンパイル時に、OCaml における型推論のような方法で決定され、暗黙的に型付けされる。

本システムにおける構造体を生成・利用する構文は、JavaScript におけるオブジェクトの生成に類似している。つまり、構造体を生成するにはただ宣言するだけで済み、任意の名前のフィールドに任意のデータを格納することができる。これにより、プログラマは簡潔なコードの記述のみで構造体を利用することが可能となる。

また、本システムでは typeof 演算子を提供している。typeof 演算子は

暗黙的に型付けされた構造体の型を取得するための演算子である。本システムでは構造体の型を暗黙的に付けているので、逆に明示的に構造体の型の名前を指定することができないという問題がある。その問題を解決するために typeof 演算子を提供した。

本研究の実装は、JastAdd を用いて実装された Java コンパイラである JastAddJ を拡張することによって行った。抽象構文木から構造体であるオブジェクトとそれを参照する変数をサーチし、型を推論して自動的に型付けすることで、本システムを実現した。また、性能を調べるために実験を行い、実用上問題のない程度のオーバーヘッドで本システムが実現できていることを確かめた。

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いたします。また、本研究における仕様や実装方法などの点について指導して頂いた赤井駿平氏に心から感謝致します。最後に、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感謝致します。

目 次

第 1 章　はじめに　8

第 2 章　既存技術とその問題点　10
2.1　クラス定義による構造体の表現 10
2.2　JavaScript 10
2.2.1　Java で JavaScript の記述法を実現する上での問題点 12
2.3　Whiteoak 13
2.3.1　structural typing 13
2.3.2　Whiteoak 13
2.4　OCaml 14
2.4.1　OCaml の型推論 15
2.4.2　構造体の実現 15

第 3 章　暗黙的に型付けされる構造体　19
3.1　本システムの概要 19
3.1.1　本システムにおける var 型 19
3.1.2　ストラクトオブジェクト 20
3.2　仕様 20
3.2.1　フィールドの追加と代入 20
3.2.2　構造体のフィールドの型 21
3.2.3　var 型の変数への代入 23
3.2.4　structural typing の利用 25
3.3　typeof 26

第 4 章　実装　29
4.1　JastAdd 29
4.1.1　抽象構文 29
4.1.2　コンパイラの生成 29
4.1.3　インタータイプ宣言 30
4.1.4　属性 (attribute) 32
4.2　JasdAddJ 35
4.3　本システムの実装 35

4.3.1 var 型の変数の型 35
4.3.2 ストラクトオブジェクトの型 37
4.3.3 var 型の変数の型推論 39
4.3.4 typeof の実装 39
4.3.5 処理の順序 41

第 5 章 実験 **43**
5.1 実験概要 43
5.2 実験結果と考察 43

第 6 章 まとめと今後の課題 **47**
6.1 まとめ 47
6.2 今後の課題 47
6.2.1 言語の仕様 48
6.2.2 配列としての宣言 48
6.2.3 局所変数以外での構造体の利用 48

図 目 次

2.1 内部クラスを用いた構造体の生成例 11
2.2 匿名クラスを用いたフィールドの追加 11
2.3 JavaScript での構造体の生成 11
2.4 Whiteoak を用いたプログラム例 14
2.5 structural typing によるサブタイプ関係 17
2.6 OCaml における擬似的な構造体 18

3.1 var 型を用いた構造体 20
3.2 フィールドの追加と代入 21
3.3 異なる型のオブジェクトの代入 22
3.4 構造体のフィールドの型推論アルゴリズム 22
3.5 ストラクトオブジェクトの代入 23
3.6 複数のストラクトオブジェクトの代入 24
3.7 var 型の静的な型 25
3.8 structural type で表した図 3.6 の型 26
3.9 typeof を用いたプログラム例 28

4.1 型を表す AST クラス図 30
4.2 JastAdd におけるインタータイプ宣言の例 31
4.3 Synthesized attribute 32
4.4 Inherited attribute 33
4.5 Inherited attribute から生成されるコード 33
4.6 lazy attribute 34
4.7 lazy attribute から生成されるコード 34
4.8 構造体の型 38
4.9 型推論の結果による変換 40

5.1 実験に用いたプログラム 44

表 目 次

5.1 プログラムのコンパイル時間 (ミリ秒) 45
5.2 プログラムの実行時間 (ミリ秒) 45

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。構造体もその中の１つである。構造体は主に複数のデータをまとめて格納できる型である。代表的な例では、C 言語や C++言語などでサポートされており、これらの言語では struct というキーワードで構造体の型が定義される。配列とは異なり、各メンバがそれぞれ異なる型のデータを格納できることが特徴である。

しかしオブジェクト指向言語では、このような構造体は純粋なオブジェクト指向をめざしてサポートされない場合が多い。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、構造体の様なデータ型はクラスで代用することが可能であるためサポートされていない。本研究が対象とする Java 言語もその一例である。

構造体はデータをまとめるために、しばしば局所的に用いられる。しかし、局所的にしか利用されない構造体のために、型の定義を分けて書かなければならないというのは煩わしく感じられる。Java 言語では、このように局所的に新しいオブジェクトを生成する場合、一般的に内部クラスが用いられる。しかし、構造体を内部クラスを用いて定義しても、定義を分けて記述しなければいけないことに変わりはない。

このように型の定義を書かなければいけない原因は、Java 言語がプログラマに全ての変数の型を明示的に記述することを義務付けているからである。そのため、今まで用いられていない新しい型を用いるためには、その型の名前と構造 (メンバ) を定義しなければならないのである。

しかし逆を言えば、型が暗黙的に付けられるならば、新しい型であろうと、定義する必要がないのである。そこで、簡潔な記述で生成・利用できる構造体を実現するために、本研究では、局所的に利用できる暗黙的に型付けされる構造体を Java に導入するシステムを提案する。本システムを用いることで、構造体は JavaScript のオブジェクトのように、宣言するだけで生成でき、任意の名前のフィールドに任意のデータを格納することができる。

以下、第２章では、既存技術について、第３章では、var 型とストラク

トオブジェクトを用いた構造体とそれをサポートする typeof 演算子についていて、第 4 章では、本システムの実装について、第 5 章では性能を調べるための実験について、そして最後に第 6 章では、まとめと今後の課題について述べる。

第2章　既存技術とその問題点

この章では、本システムに関連のある既存の技術を用いて、暗黙的な型付けやその場での簡単な記述による構造体の生成を実現する方法、またはそうする上での問題点について説明する。

2.1　クラス定義による構造体の表現

Java 言語では局所的に新しいオブジェクトを生成する方法として、内部クラスや匿名クラスなどが提供されている。しかし共に、暗黙的に型付けされる構造体を生成することはできない。

内部クラスによる定義では、図 2.1 のように String 型の name と int 型の age をもつ人を表す Person という構造体は生成することはできるが、やはり一般のクラス定義同様に、明示的に型を定義することに変わりはない。

一方匿名クラスは、図 2.2 の 2 行目のように記述することで、インスタンスを生成する時に既存のクラスのフィールドを任意に追加できる。しかしこれは int 型の a,String 型の b をフィールドに持つ Object を継承したクラスのオブジェクトを生成しているが、変数 o に代入すると、o は単なる Object 型なので、a や b にアクセスすることはできない。また、匿名クラスの中からローカル変数アクセスするためには、ローカル変数が final でなくてはならないという制限が付いているため、3 行目はエラーとなる。従って既存のオブジェクトを構造体のメンバとすることは難しい。

2.2　JavaScript

明示的に型を宣言する必要のない構造体を実現する言語の例として JavaScript[3] が挙げられる。JavaScript を用いると、図 2.1 の Java のコードは図 2.3 のように記述できる。つまり、クラス定義を記述する必要がなくなり、またその分だけコードが省略できている。

しかしながら、このようなプログラムの記述を可能としているのは、JavaScript が動的型付け言語だからであり、暗黙的に型付けがなされて

```
1 class Parson{
2   String  name;
3   int age;
4 }
5
6 Parson p = new Parson();
7 p.name = " hoge ";
8 p.age = 20;
9       :
```

図 2.1: 内部クラスを用いた構造体の生成例

```
1 String s = "hoge";
2 Object o1 = new Object(){int a = 2; String b = "b";}
3 Object o2 = new Object(){int a = 2; String b = s;}
4 //error
```

図 2.2: 匿名クラスを用いたフィールドの追加

```
1 var p = {};
2 p.name = " hoge ";
3 p.age = 20;
4       :
```

図 2.3: JavaScript での構造体の生成

いるわけではない。つまりメンバを追加するにあたって、構造体の型が変化するのでこのような記述が可能となるのである。

動的型付け言語では当然ながら、コンパイル時の型検査ができないという欠点がある。コンパイル時の型検査はプログラムの実行前に、望ましくないであろうコードを検出し、プログラムの信頼性を高めたり、デバッグ作業の手助けとなる大きな利点である。特に Java 言語は強い静的型付け言語なので、安全性が高く、その利点は顕著である。なのでこれらの利点を失うことは好ましくない。

以上のことから、Java 言語において、強い静的型付けを維持しつつ、図 2.3 のようなコードを記述できることが望ましい。実際、本研究で提案する構造体を用いるための構文はは JavaScript の構文に類似している。

2.2.1　Java で JavaScript の記述法を実現する上での問題点

図 2.3 のようなコードを Java で実現するためにはいくつか問題点が存在する。言語が異なるので、構文が異なるということは当然であるが、言語の設計上、根本的な問題点について以下に示す。

構造体の生成

JavaScript ではというキーワードでオブジェクトを生成しており、自由にプロパティ (メンバ) を追加することが可能である。しかし、既存の Java ではクラス定義をせずにオブジェクトを生成することは不可能である。よってそれを可能にする必要がある。

構造体の型

前述した通り、JavaScript は動的に型付けされるため、図 2.3 のような記述が可能となる。構造体はその構造によって型が異なるはずであるから、メンバを追加した時に型が変化しなければならないが、既存の Java ではそのような機構は存在しない。

従って、このように動的に変化する型を導入、あるいは擬似的に実現しなければならない。

構造体を参照する変数の宣言時の型

上記の問題は生成した構造体を代入する変数についても同じである。Java 言語において、一般にオブジェクト (インスタンス) を生成したとき

に代入する変数は、そのオブジェクトの型もしくはそのスーパータイプである。つまり図 2.1 の例において、

```
Person p = new Person();
```

であるように、p の型は Person であると明示的に宣言するする必要がある。JavaScript では、var を用いて変数であることのみ表しているが、既存の Java にはこれに相当する型は存在しないため、宣言時に利用できる型を導入しなければならない。

2.3 Whiteoak

Whiteoak [4] は Java に structural typing を導入したシステムである。以下では本システムにも利用されている structural typing と、それを Java に導入した Whiteoak のシステムについて説明する。

2.3.1 structural typing

structural typing とは、型がその構造によって決定される型システムである。つまり、型はメンバの名前と型のペアの集合として表され、この構造が同じならば同じ型とみなされる。

サブタイプ関係は、B が A の構造を全て持っていれば (メンバの集合という意味なら A $\subset$ B であるなら) B は A のサブタイプである。具体的に Whiteoak における図 2.4 の例では、XYZ は XY のサブタイプである。

またそれに対応して、既存の Java に用いられているような、型の名前によってサブタイプ関係が決定される型システムを nominal typing という。

2.3.2 Whiteoak

Whiteoak は前述したような structural typing という型システムを Java に導入したシステムである。

Whiteoak では、structural type 間の合成が以下のように定義されている。

- T_1*T_2: T_1,T_2 に定義されているメンバの積集合のメンバをもつ型
- T_1+T_2: T_1,T_2 に定義されているメンバの和集合のメンバをもつ型
- T_1T_2: T_1 に定義されたメンバが T_2 に定義されたメンバによって override された型

```
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 2.4: Whiteoak を用いたプログラム例

これらの合成を用いることで、容易に新しい構造の structural type を定義でき、無駄な定義を減らすことができる。

構造体を structural type を用いて実現することを考えると、クラス定義と同様に明示的に定義しなければならないことに変わりはないため、そのまま用いることはできない。また、Whiteaok は構造的な型を与えるシステムではあるが、インスタンスを生成することはできない。つまり、既存のオブジェクトに新たに定義した型をあてはめるにすぎず、インタフェースに近いものである。

しかし、structural type は構造が異なれば別の型である等の、構造体の特徴と似た性質をもっている。なので本システムではこの structural type を利用して実装をしている。

2.4　OCaml

OCaml は強い静的型付けのある関数型言語である [6]。強い静的型付け言語である点は Java 言語と同じであり、コンパイル時の型検査によって、型エラーが検出される。

2.4.1　OCaml の型推論

OCaml において特徴的な点は、強い静的型付け言語でありながら、変数の型は明示的に宣言する必要がないという点である。OCaml における型は全てコンパイラによって型推論される。従って、プログラマが明示的に型を宣言しないにも関わらず、もし型エラーが起きていればそれは検出されるという利点がある。

例えば、

```
let a = 3;
let b = a + 2;
```

というプログラムは、a は当然 int と推論され、+という演算子は int と int を受け取り int を返す演算子であるため、型環境に保存された a:int を参照し、型が合っているので、b の型は int となる。

2.4.2　構造体の実現

OCaml の型推論の機能とデータ構造の組を用いて、構造を明示的に宣言しないで構造体を生成するようなコードを書いてみると、図 2.6 のようなプログラムが考えられる。このプログラムではリスト構造を作り、各リストの要素を (メンバ名, 値) として、構造体を表現している。メンバ名を用いてアクセスをしたい場合はサーチする関数をさらに定義すれば良い。

このようにすると、p の let 定義により、p の型は 7,8,9 行目ではそれぞれ

```
 string*int
(string*int)*(string*string)
(((string*int)*(string*string))*(string*float))
```

と自動的に決定される。

しかしながら、これをそのまま本研究の目標とする構造体に適用させることは不可能である。このプログラムにおける変数 p と構造体の関係は、オブジェクトとそれを参照する変数という関係ではなく、単なる置き換えに過ぎない。つまり、1,2,3 行目の p は同じ名前ではあるが、スコープが異なる全く別の変数である。よって既存の Java では変数 p の型を変動させることはできないため、構造体の構造によって型が変わるような機構にするためには、構造体にメンバを加えるたびに異なる変数に代入されなければならないのである。

レコード

OCaml にはデータをまとめるためのデータ構造がいくつか提供されており、レコードが構造体の役割に近い機構である。

具体的にレコードを用いると、

```
# type person = {name : string; age : int}
```

と記述することができ、これは string 型の name と int 型の age をもつレコードの型を定義している。つまりこれは C 言語における struct と同じ機能である。しかし、フィールドの構造によって型を決定できるという点は共通だが、レコードのフィールドの名前として定義した名前は、他のレコードのフィールドの名前として再度用いることができないというデメリットがある。

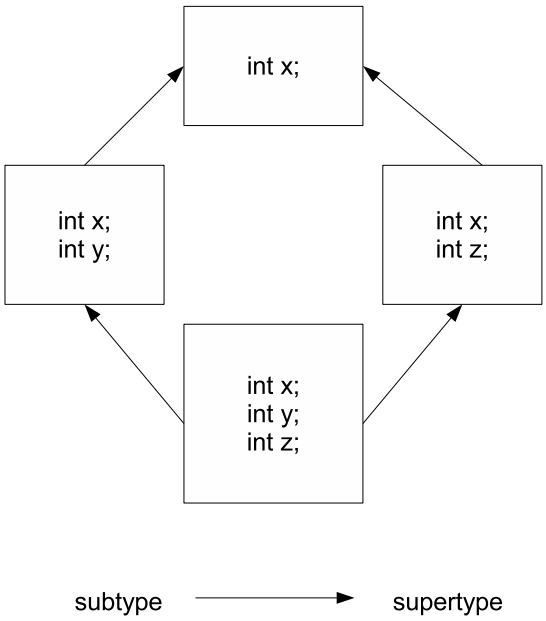

図 2.5: structural typing によるサブタイプ関係

```
1 let add a b = (a,b);;
2 let car (x,y) = x;;
3 let cdr (x,y) = y;;
4 let rec get (c,n) =
5   if n = 1 then cdr c else get(car c,n-1);;
6
7 let p = (''age'',10);;
8 let p = add p (''name'',''hoge'');;
9 let p = add p (''height'',170.5);;
```

図 2.6: OCaml における擬似的な構造体

第3章　暗黙的に型付けされる構造体

3.1　本システムの概要

前章で述べたように本研究では、暗黙的に型付けされる構造体を表すオブジェクトを生成し、前章で示した JavaScript のようなコードを Java でも書けるようにすることが目的である。従ってそのようなコードを Java でも書くことができるようにするため、本研究では var 型とストラクトオブジェクトを導入することを提案する。また本システムをサポートする機能として typeof 演算子を提供する。

var 型

var 型とは C#において提供されている暗黙的に型付けされるローカル変数の宣言時の型を表すものである。JavaScript における var 文は、変数の宣言を表しているだけであるため、少し異なる。

一般的に var 型である変数は、数値を入れれば数値型と認識し、文字列を入れれば文字列型といったように、後に型推論される。

C#のように var というキーワードで暗黙的に変数が型付けされる言語の他に、OCaml のように変数の型を書かないことで暗黙的に型付けされる言語もある。

3.1.1　本システムにおける var 型

本研究における var 型は、以下のような特徴をもつ。

- 構造体を参照する変数の宣言時の型として用いる。
- 本当の静的な型は後に型推論によって決定される。
- local 変数宣言時のみ使用可能。

つまり C#の var 型を構造体に適用したものである。構造体はフィールドの追加が任意に行えるようにするため、宣言時の型として var 型、つまり今の時点では分からない型としておく。そしてコンパイル時には静的な型

```
1 var p = new();
2 p.name = ''hoge'';
3 p.age = 20;
4       :
5 System.out.println(p.name);
6 //output: hoge
```

図 3.1: var 型を用いた構造体

を与えるため型推論され暗黙的に型付けがなされる。これについては次節以降で詳しく説明する。

3.1.2　ストラクトオブジェクト

ストラクトオブジェクトとは構造体を表すオブジェクトのことである。ストラクトオブジェクトは以下のような特徴を持つ。

- new() というキーワードによって生成される。
- var 型の変数にのみ代入可能。
- 生成した時に代入した変数によってアクセスされた場合、任意の名前のフィールドに任意の型のオブジェクトを代入可能。

既存の Java においてインスタントの生成は、new クラス名 (コンストラクタの引数) でなされるが、それに対してストラクトオブジェクトは new() というキーワードのみで生成される。また、ストラクトオブジェクトの型は暗黙的に生成されるので、var 型以外の型の変数に代入することはできない。正確には Object 型の変数に代入することはできるが、構造体として用いることはできない。

var 型とストラクトオブジェクトを用いることで図 2.1 のコードは図 3.1 のように記述できる。

3.2　仕様

3.2.1　フィールドの追加と代入

本システムにおいて、フィールドの追加とフィールドの代入は同様の構文で記述される。これは JavaScript と同じである。

```
1 var a = new();
2 a.name = ''hoge'';
3 a.age = 3;
4 a.age = 5;
```

図 3.2: フィールドの追加と代入

もし、構造体のフィールドの中に代入しようとしているフィールド名が存在していなければそのフィールドを追加し、値を代入する。そして、構造体の中に既にそのフィールドが存在すれば、そのフィールドに値を代入する。

図 3.2 の例では 2,3 行目では name というフィールドも age というフィールドも存在しないので、これらのフィールドを追加している。また 4 行目は、3 行目で既に age フィールドが作られているため、既存のフィールドに値を代入している。

3.2.2　構造体のフィールドの型

ここでは構造体のフィールド (メンバ) の型について説明する。前小節において、フィールドは初めて代入される時に生成されると説明した。よって、構造体の型として最も素直なのは、生成された時に代入されたオブジェクトの型である。しかし、それでは問題が生じる。

例えば、図 3.3 の例では、Circle クラスとそれを継承する RCircle クラスと BCircle クラスを生成し、それぞれのインスタンス rc と bc を構造体の circle というフィールドに代入している。つまり、構造体 c の circle フィールドには Circle クラスのサブクラスを代入したいという意図である。

もし構造体の型を初めに代入したときの型とするならば、初めて c.circle に代入するのは 13 行目の

```
c.circle = rc;
```

であるから、構造体 c の circle フィールドの型は RCircle ということになる。従って、16 行目の

```
c.circle = bc;
```

は、RCircle と BCircle の間にサブタイプ関係がないので型エラーとなってしまう。

上記のような代入を認めずエラーとし、意図するプログラムを記述するためには、プログラマが明示的にキャストしてから代入しなくてはならな

```
1  class Circle{
2    …;
3  }
4  class RCircle extends Circle{
5    …;
6  }
7  class BCircle extends Circle{
8    …;
9  }
10
11 var c = new();
12 rc = new RCircle();
13 c.circle = rc;
14
15 bc = new BCircle();
16 c.circle = bc;
```

図 3.3: 異なる型のオブジェクトの代入

いとするのも仕様の 1 つとして考えられる。しかし、本システムではできるだけ明示的な記述をなくすために、c.circle の型が暗黙的に Circle となるようなアルゴリズムを採用した。それは、フィールドの型をそのフィールドに代入されるオブジェクト全てが代入可能となるように、決定するという方法である。そのアルゴリズムは以下の通りである。

T をフィールドの型、 $T_1, T_2, \cdots, T_n$ をフィールドに代入されるオブジェクトの型とする。
T = S s.t. i=1,2,…,n に対して S:> T_i
　　かつ i=1,2,…,n に対して S':> T_i ⇒ S':>S
(ただし、「A:>B ⇔ B は A のサブタイプ」 である)

図 3.4: 構造体のフィールドの型推論アルゴリズム

簡潔に言えば、代入されたオブジェクトの型全ての共通のスーパータイプである型の中で、最もサブタイプな型をフィールドの型とするのである。このアルゴリズムを用いれば、図 3.3 のコードを書き換えることなく、c.circle の型は Circle となり、rc, bc 共に代入可能である。

またこのアルゴリズムを用いれば、同じ型のオブジェクトのみ代入される場合や代入が 1 つだけの場合でも、望まれる型が結果として出力される

ことが容易に分かる。

3.2.3　var 型の変数への代入

ストラクトオブジェクトは生成されると var 型の変数に代入される。しかし、本システムの構造体は参照型であるから、他のクラスのオブジェクトと同様に生成後に他の変数に代入することも可能である。

```
1 var a = new();
2 a.name = ''hoge'';
3 var b = a;
4 System.out.println(b.name);
5 //output: hoge
```

図 3.5: ストラクトオブジェクトの代入

この場合、a と b は同じストラクト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ストラクトオブジェクトのフィールドを変更すれば、もう一方のストラクトオブジェクトも変更される。ただし、現在の仕様では、b の変数からアクセスし、フィールドを追加することはできない。

また、１つの変数に 2 回以上のストラクトオブジェクトの代入がある時は、少し複雑な挙動をする。

複数回ストラクトオブジェクトが 1 つの変数に代入される場合、代入される変数は、代入するストラクトオブジェクトのフィールドの中で、互いに共有するフィールドのみアクセスが許される。

例えば図 3.6 の例は、circle という構造体と square という構造体を作り、それらを shape に代入している。circle, square が参照するストラクトオブジェクトは String 型の color というフィールドのみ共有している。従って、10 行目の代入によって shape は circle と同じストラクトオブジェクトを参照するが、shape は color フィールドにしかアクセスできない。よって 11 行目の radius フィールドへのアクセスはエラーとなる。同様にして 15 行目の size フィールドへのアクセスもエラーである。

このような仕様でるため、異なるフィールドを持つ構造体をむやみに同じ変数へ代入することは望ましくない。しかし、共有するフィールドがあり、それだけを用いるならば、様々な構造体を 1 つの変数に代入し処理ができるので便利である。

もしこのような仕様でないとするならば、代入される毎にアクセスできるフィールドを変化させる方法と、代入された構造体がアクセスできるフィールド全てを持つ方法が考えられる。しかし前者である場合、shape

```
1  var circle = new();
2  circle.color = ''red'';
3  circle.radius = 1;
4
5  var square = new();
6  square.color = ''white'';
7  square.size = 3;
8
9  var shape;
10 shape = circle;
11 System.out.println(shape.radius);
12 //error
13
14 shape = square;
15 System.out.println(shape.size);
16 //error
17
18 System.out.println(shape.color);
19 //output: white
```

図 3.6: 複数のストラクトオブジェクトの代入

という変数がアクセスできるフィールドがプログラムの位置で変化するということは、shape の型つまり var 型が後に一意に定まらないということである。このような挙動を Java ですることは不可能である。また後者である場合、 3.6 の例を用いれば、shape はフィールドとして color, radius, size を持つことになるが、この shape に square を代入したとすると shape は radius を持つはずなのに、代入された square は raius を持たないというおかしな構造になってしまうため不適切である。

3.2.4　structural typing の利用

前小節における仕様は、クラスの継承関係と同じである。図 3.6 の例の circle, square, shape を Circle, Square, Shape 型の変数だと思い、Circle, Square は Shape を継承しているクラスと考える。そして、共有するフィールドは親クラスである Shape クラスで宣言されていると考えれば、図 3.6 の例はそのままクラス関係に帰着できる。

そして、これは structural typing による型システムと同じ様なシステムである。つまり図 3.6 における例では、アクセスできるフィールドを structural typing における型だと考えれば、図 3.8 のように表すことができる。2 章で述べたように、structural typing におけるサブタイプ関係の付け方から、Circle と Square はまさに Shape のサブタイプである。従って shape には circle も square も代入できるが、color フィールドにしかアクセスはできない。

以上から代入する変数の型は Whiteoak による表現を用いれば、以下のように表すことができる。

代入するストラクトオブジェクトを指す var 型の変数の structural typing で表現される型 (アクセス可能なフィールドの集合) を $T_1, T_2, \cdots, T_n$、代入される変数の型を T とすると、
$\mathrm{T} = T_1 * T_2 * \cdots * T_n$

図 3.7: var 型の静的な型

詳しくは次章で説明するが、実際に var 型として宣言された変数の型は、structural type のようなインタフェースとして実装され暗黙的に型付けされる。そして、ストラクトオブジェクトはそのインタフェースを実装するクラスのインスタンスとして実装されている。

```
1  struct Circle{
2    String color;
3    int radius;
4  }
5  struct Square{
6    String color;
7    int size;
8  }
9  struct Shape{
10   String color;
11 }
```

図 3.8: structural type で表した図 3.6 の型

3.3 typeof

本システムでは、前述したように暗黙的に var 型の変数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作られたクラスの名前を認識できないという欠点も存在する。クラス名を認識できないということは、明示的にクラス名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、変数宣言時の型は var 型として記述され、インスタンスの生成は new() というキーワードで行われている。なので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。

- 生成した構造体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。exp は型をもつ任意の表現である。本システムの typeof は gcc の拡張文法や D 言語における typeof と同様の機能であり、JavaScript における型の名前を表す String を返す typeof や、c#における型を表すオブジェクトを返す typeof とは異なるものである。

typeof を導入する目的は、暗黙的に定義された型の取得ではあるが、既存の Java における表現を取得することも可能である。

typeof 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。図 3.9 の例では、上のプログラムでは本システムに対する typeof の適用を、下のプログラムでは既存の Java のプログラムに対して typeof を適用している。

上のプログラムでは p という構造体を生成し、それを HashMap に格納している。ジェネリクスを用いて HashMap には String をキーとして、p と同じ型のオブジェクトのみ格納できる。しかし p の宣言時の型は var 型なので、p の実際の型は暗黙的に付けられる構造体の型である。従って、HashMap に格納できるオブジェクトは p と同じ構造を持つ構造体、つまり String 型の name と int 型の age をフィールドに持つ構造体のみである。

また typeof は変数宣言時にも用いることができるので、14 行目のように変数を取り出す場合にも用いることができる。ただし、格納されているオブジェクトは構造体であるから、今までと同様に、15 行目のように var 型を用いて宣言した変数に代入することも可能である。

```
var p = new():
p.name = ''hoge'';
p.age = 5;

var q = new();
q.name = ''foo'';
q.age = 10;

HashMap<String, typeof(p)> hm =
        new HashMap<String, typof(p)>();
hm.put(''p'',p);
hm.put(''q'',q);

typeof(p) p2 = hm.get(''p'');
var q2 = hm.get(''q'');
```

```
int i = 10;
typeof(i) j = 11; //ok(⇔ int j = 11;)
typeof(i) k = ''hoge'';
//type error(⇔ int k = ''hoge'';)
```

図 3.9: typeof を用いたプログラム例

第4章　実装

本システムは、前章で述べた var 型、ストラクトオブジェクト、そして typeof を、JastAdd というコンパイラ実装フレームワークを用いた Java コンパイラである JastAddJ を拡張することで実現した。

4.1　JastAdd

JastAdd [1, 2, 5] は Java-based なコンパイラを実装するシステムである。非常に柔軟な拡張性をもったコンパイラの実装を可能とするのが、JastAdd の利点である。

4.1.1　抽象構文

JastAdd における各抽象構文は Java のクラスを用いて表現されている。これらのクラスは抽象構文木を構成することから AST クラスと呼ばれ、各クラスのインスタンスは AST ノードと呼ばれる。各 AST クラスは言語における表現をより抽象的な表現をスーパークラスとして、より具体的な表現をサブクラスとして表現している。詳しくは次節で説明するが、例えば Java における型を表す AST クラスの概要は図 4.1 のようになっている。

4.1.2　コンパイラの生成

JastAdd におけるコンパイラは以下のファイルによって生成される。

- ast ファイル
- parser ファイル
- jrag,jadd ファイル

ast ファイルは AST クラスの構造を定義するファイルである。どの AST クラスがどの AST クラスのサブクラスであるか、またその AST クラスはどのクラスの子ノードを持つかなどが ast ファイル内で指定できる。

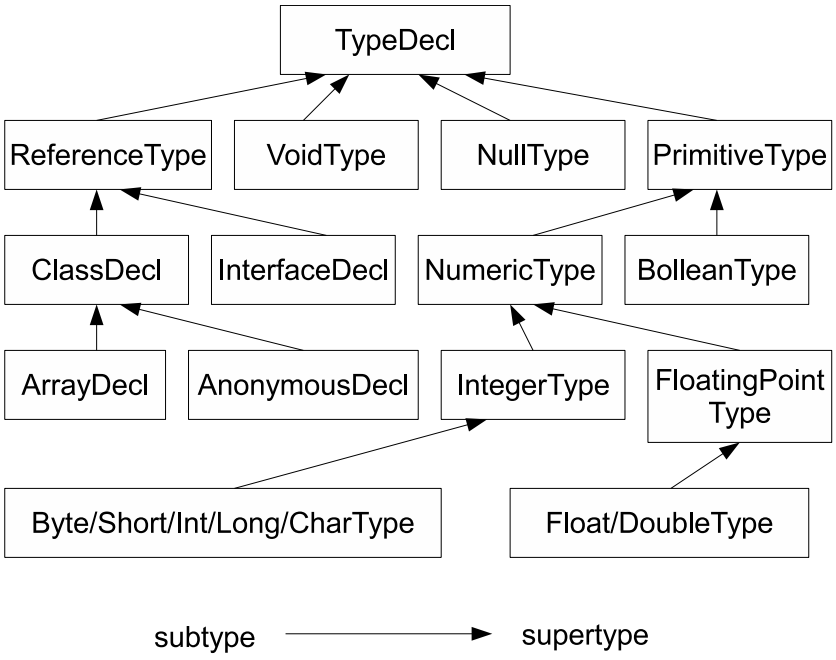

図 4.1: 型を表す AST クラス図

parser ファイルは構文解析のアルゴリズムを定義するファイルである。parser ファイルの内容に従ってプログラムのソースをもとに、AST ノードで表わされる抽象構文木が生成される。

jrag ファイルと jadd ファイルは AST クラスの実装を定義するクラスである。jrag ファイルと jadd ファイルにはファイルの種類としての差はなく、AST クラスの実装の大部分、またコンパイラを拡張するための新しい AST クラスの実装は主に jrag ファイルに記述される。jrag ファイルでの実装は、アスペクトを用いたインタータイプ宣言で記述される。インタータイプ宣言については、次小節において詳しく説明する。

4.1.3　インタータイプ宣言

JastAdd の大きな利点は柔軟な拡張性である。それを実現しているのが、アスペクト指向プログラミングのインタータイプ宣言を用いた実装である。

```
1 aspect AddName{
2   private String A.name;
3   private String B.name = ''hoge''; //nameの初期化
4   public String A.name(){return name;}
5   public String B.name(){return name;}
6 }
```

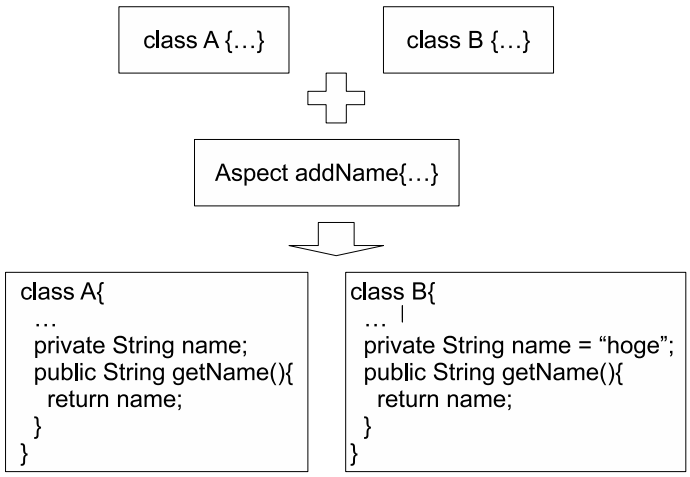

図 4.2: JastAdd におけるインタータイプ宣言の例

アスペクト指向プログラミング

アスペクト指向プログラミングとは、オブジェクト指向プログラミングではモジュール化できないソフトウェアの関心事 (横断的関心事) をアスペクトとして分離するプログラミング技法である。

JastAdd ではこのアスペクト指向プログラミングによる実装がサポートされており、これにより既存のコンパイラを拡張しやすい構文を提供している。

インタータイプ宣言

インタータイプ宣言とは、他のクラスやインタフェースのメンバ (フィールドやメソッド) を宣言することである。例えば図 4.2 では、A, B というクラスに String 型の name というフィールドと name の値を返す getName() メソッドを追加している。

```
1 syn T A.x();
2
3 eq A.x() = Java-expression;
4 //もしくは、eq A.x(){Java}
```

図 4.3: Synthesized attribute

JastAdd ではこのようなアスペクトを jrag ファイルと jadd ファイル内で記述でき、AST クラスの定義を容易に拡張できるようになっている。またこのような記述法ができるおかげで、複数のクラスにまたがった機能の拡張をする際に、本来ならば各クラスを定義している部分にコードを追加しなければならないが、1 箇所にまとめて記述することができるようになっている。これはアスペクト指向プログラミングの利点によるものである。

4.1.4　属性 (attribute)

JastAdd において拡張性の高さの理由の 1 つにインタータイプ宣言時に用いることができる豊富な属性定義のための構文が挙げられる。属性とはどの AST クラスにどのようなメソッドを定義するかを決定するために記述されるコードである。基本的には、インタータイプ宣言によるメソッドの定義・実装のことを指すが、ここで示す属性構文と共に用いることで、メソッドに特別な意味を持たせることができる。

以下では、JastAdd で提供されている属性定義のための構文の中で、主に用いられる構文について説明する。

Synthesized attributes

synthesized attributes は以下のようにして定義される。

1 行目は x がクラス A における、型 T を持った synthesized attribute であることを示している。これは、A が抽象クラスであれば、A の抽象クラスでない全てのサブクラスにおいて A の実装が eq によって定義されなければならなく、また A が抽象クラスでないならば、A において実装されなければならないことを意味している。つまり、端的に言えば、クラス A に abstract メソッドを定義することに等しい。

eq キーワードは上で説明したようにメソッドの実装を定義するための属性構文である。= の右辺に既存の Java 表現を記述でき、それがそのま

```
1  inh T A.y();
2
3  eq C.getA().y() = Java-expression;
4  eq D.getA().y() = Java-expression2;
```

図 4.4: Inherited attribute

```
1  class A{
2    public T y(){
3      return getParent().Define_T_y(this,null);
4    }
5  }
6  class C{
7    public T Define_T_y(ASTNode caller, ASTNode child){
8      Java-expression;
9    }
10 }
11 class D{
12   public T Define_T_y(ASTNode caller, ASTNode child){
13     Java-expression2;
14   }
15 }
```

図 4.5: Inherited attribute から生成されるコード

ま左辺の属性の表すメソッドの返り値となる。また既存の Java のメソッド定義のように具体的に実装を記述することもできる。

Inherited attributes

inherited attribute は以下のように定義される。

inherited attribute は AST における親ノードのクラスによって実装が変化するメソッドを表す属性である。図 4.4 の例では、クラス C, D はクラス A のノードを子ノードとして持っており (これより前でそう定義されており)、A の親ノードのクラスが C であるとき、y() の実装は 3 行目であり、A の親ノードが D であるとき、y() の実装は 4 行目となる。具体的に図 4.4 からは図 4.5 のようなコードが生成される。

```
1 syn lazy T A.x();
2 eq A.x() = Java-expression;
```

図 4.6: lazy attribute

```
1  class A{
2    protected boolean x_computed = false;
3    protected T x_value;
4    public T x(){
5      if(x_computed)
6        return x_value;
7      x_value = x_compute();
8      x_computed = true;
9      return x_value;
10   }
11   public T x_compute(){
12     Java-expression;
13   }
14 }
```

図 4.7: lazy attribute から生成されるコード

Lazy attributes

lazy attribute はその属性が表すメソッドが、多く呼ばれ、かつ各インスタンスにおいて返り値が変化しない場合に用いられる。初めてメソッドが呼ばれた時は返り値を計算し、その値を保存する。そして 2 度目以降にそのメソッドが呼ばれた時は、保存しておいた値をそのまま返すという、キャッシュを用いたメソッドの実装を表す属性である。

具体的には、図 4.6 のコードから図 4.7 のようなコードが生成される。

4.2　JasdAddJ

JastAddJ は JastAdd を用いて実装された Java コンパイラである。JastAddJ におけるコンパイル処理は、一般の Java コンパイラと同様に以下のようになっている。

- 字句・構文解析
- 名前検査・型検査などのエラー検知
- バイトコード生成

JastAddJ では字句解析は Jflex を、構文解析は Beaver を用いて行われる。構文解析における抽象構文木は、CompilationUnit ノード (コンパイル単位を表す AST ノード) をルートとする木構造になっており、型検査・バイトコード生成はこの抽象構文木を辿って行われる。

4.3　本システムの実装

前述した通り、Java では全てのオブジェクトはクラスのインスタンスである。従ってストラクトオブジェクトも、クラス定義を書くことを省略しようとしても、クラス定義なくオブジェクトを生成することは不可能である。しかし逆を言えば、このようなことをするためには、クラス定義が暗黙の内になされるようにすれば良い。そして、本システムでストラクトオブジェクトの型も var 型の変数の型も自動的に決定され、暗黙的に型付けがなされる。

この節では、これらの暗黙的に決定・生成されるストラクトオブジェクトの型 (クラス) と var 型の変数の型 (インタフェース) について説明し、最後に typeof の実装についても説明する。

4.3.1　var 型の変数の型

ストラクトオブジェクトが代入される変数の型である var 型は、変数の宣言時の一時的な、また見かけ上の型である。Java 言語は静的型付け言語であるから、C#のおける var 型の変数と同様に、コンパイル時には特定の型を決定しなければならない。

本システムでは、var 型によって宣言された変数の型は Object 型である。それはいかなる構造体も代入できるようにするためである。しかしそれではその変数に代入はできるが、その変数を用いることはできない。従って、var 型によって宣言された各変数に対し、バイトコード上の型と

は別に、その変数を用いるときの型を定義した。つまり var 型として宣言された変数は、その変数へ代入される場合を除き、その型に暗黙的にキャストされて用いられるということである。

変数使用時の型はインタフェースとして実装した。前章でも簡単に説明したが、このインタフェースは structural type と似た構造となっており、アクセス可能なフィールドの集合を表している。

例えば図 3.1 の例では p のアクセス可能なフィールドは String name と int age である。これを structural type によって表現すると、

```
struct P{
  String name;
  int age;
}
```

となる。そしてこれを本システムでは、

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

というインタフェースで表現する。つまりインタフェースはあるフィールドを持っているかどうかで、そのインタフェースを実装しているしているかどうか判定できないため、そのフィールドへのアクセスをするメソッド (get,set) を実装しているかどうかで、そのインタフェースを実装しているかを表すようにした。つまり、IntAge_StringName というインタフェースは int age, String name というフィールドをアクセスできる構造体を参照する変数を表している。従って、§4.3.2 で詳しく説明するがストラクトオブジェクトの型はこのようなインタフェースを実装するクラスとなっている。また、インタフェース名はフィールドの型+フィールド名をアルファベッド順にアンダーバーで繋いだ名前になっている。

しかしながら、このままでは var 型の変数の利用時の型がインタフェースであるため、その変数から直接ストラクトオブジェクトのフィールドにアクセスできないという問題が存在する。だが、それを可能とするためにインタフェースの定義において、フィールドをアクセスできるかどうかを、そのフィールドに対応する get メソッド・set メソッドを実装しているかどうかで表現した。つまり本システムでは、全てのフィールドアクセスは get メソッドと set メソッドで書き換えられる。具体的には、p を var

型として宣言された変数であり、推論された結果のキャストされる型であるインタフェースを StinrgName であるとすると、

```
p.name = "hoge";
System.out.println(p.name);
```

というソースコードはそれぞれ、

```
((StringName) p).setName("hoge");
System.out.println(((StringName) p).getName());
```

というソースコードに書き換えられる。

具体的にどのようにして、このインタフェースが決定・生成されるかは §4.3.3 において説明する。

4.3.2　ストラクトオブジェクトの型

オブジェクトの型 (クラス) は大まかにはフィールド、メソッド、ストラクタでできている。ストラクトオブジェクトの型についても暗黙的に作成するものの、その例外ではない。

ストラクトオブジェクトの型の構造として素直な実装は、構造体のメンバをフィールドに持たせただけのクラスである。構造体はフィールドアクセスさえ可能ならば良いので、このシンプルな構造で十分である。ただし、前小節で述べたように、このストラクトオブジェクトが代入される変数が用いられる時の型はインタフェースであるからこれらを実装するように、get メソッド・set メソッドを加え、実装するインタフェースを指定する必要がある。

以上のことからストラクトオブジェクトの型は以下のようにして決定される。

1. ストラクトオブジェクトが生成されるコードをサーチする (var p = new(); であるとする)。
2. p のフィールドへの代入文を全てサーチし、フィールド名と代入されたオブジェクトの型のペアの集合として保存する。
3. 取得したペアの中で、同一名のフィールドが複数存在するならば、対応する型の集合から §3.2.2 によって決定される型をそのフィールドの型として、推論に用いた型は全てペアの集合から取り除く。
4. フィールド名と型のペアをそのままフィールドとして定義し、各フィールドに対して get メソッドと set メソッドを定義

```
1  class Var$$0
2     implements StringName, IntAge, IntAge_StringName{
3   String name;
4   int age;
5
6   public String getName(){
7     return name;
8   }
9   public void setName(String name){
10    this.name = name;
11  }
12  public int getAge(){
13    return age;
14  }
15  public void setAge(int age){
16    this.age = age;
17  }
18 }
```

図 4.8: 構造体の型

5. 各フィールドと各フィールドの組み合わせに対応する §4.3.1 の interface を実装するクラスとする。

6. 同様の構造のクラスが存在したら既存のクラスを用いり、存在しないならばそのクラスを新しいクラス (内部クラス) として定義する。

7. ストラクトオブジェクトの生成式 (new()) を 6 のクラスのインスタンス生成式に書き換える。

8. 他のストラクトオブジェクトに対して 1〜7 を繰り返す。

この処理は、コンパイルの処理における抽象構文木の生成と型検査の間に行われる。なぜなら型検査時には全ての変数の型は決定されなければならないからである。

具体的に図 3.1 の例では、図 4.8 のような推論とクラス定義がなされる。

このようにストラクトオブジェクトの型を後に推論して定義することで、任意のフィールドに、任意のオブジェクトを代入できるオブジェクトを擬似的に作っている。

なお、クラス名は Var$$+ID 番号となっている。このようにして定義されたクラスはストラクトオブジェクトの型であるから、そのストラクトオブジェクトの生成式 (new()) は、図 4.9 のように、定義したクラスのインスタンス生成として書き換えられる。

4.3.3　var 型の変数の型推論

§4.3.1 において述べたように、var 型によって宣言された変数の使用時の型は後に推論され、インタフェースがその型として用いられる。

var 型によって宣言された変数の使用時の型であるインタフェースは以下のようにして推論される。

1. var 型によって宣言された変数 (p とする) をサーチする。
2. p に代入された変数とストラクトオブジェクトをサーチする。
3. p に代入された変数が推論されていなければ推論する。(p が var 型であるならば、代入された変数も構造体を参照する変数であるから、var 型のはずである)
4. 全ての変数・ストラクトオブジェクトからアクセス可能である (get,set メソッドが宣言されている) フィールドを集める。
5. 4 のフィールドの集合をアクセス可能なフィールドとして、§4.3.1 におけるインタフェースを決定。
6. まだ推論されていない他の var 型の変数に対し 1〜5 を繰り返す。

このようにして決定されたインタフェースは、図 4.9 のように、その変数が用いられる時の型として、その変数をキャストするために用いられる。またこの時、フィールドアクセスからメソッド呼び出しの書き換えも行われる。

4.3.4　typeof の実装

typeof は typeof(exp) で記述された exp の表現の型を取得する演算子である。typeof(exp) で 1 つの型を表現することから、JastAdd における型を表す AST クラスである TypeDecl クラスのサブクラスに Typeof という独自のクラスを作り、構文解析時は typeof(exp) という表現がある部分はただ単に何の情報もない typeof という型であるとして表現している。そして、exp という表現を表すノードを子ノードとして持たせている。

```
1  //記述されるプログラム
2  var p = new();
3  p.name = ''hoge'';
4  p.age = 20;
5  System.out.println(p.name);
```

```
1  //本システムで変換されたプログラム
2  Object p = new Var$$0();
3  ((IntAge_StringName) p).setName(''hoge'');
4  ((IntAge_StringName) p).setAge(20);
5  System.out.println(((IntAge_StringName) p).getName());
6
7  class Var$$0
8     implements StringName, IntAge, IntAge_StringName{
9   String name;
10  int age;
11  public String getName(){return name;}
12  public void setName(String name){this.name = name;}
13  public int getAge(){return age; }
14  public void setAge(int age){this.age = age;}
15 }
16
17 //interfaceはtopレベルで定義
18 interface IntAgeStringName{
19  public String getName();
20  public void setName(String name);
21  public int getAge()
22  public void setAge(int age)}
23 }
24
25 interface IntAge{…}
26 interface NameString{…}
```

図 4.9: 型推論の結果による変換

exp が var 型の変数以外である場合、exp の型を取得して typeof(exp) をその取得した型で書き換える。exp の型は JastAddJ における型を表す AST クラスを返すメソッド type() を用いることで取得できる。

もし exp が var 型の変数である場合、exp の利用時の型、つまりキャストされる型に置き換わる。次節における処理の順序から、既に var 型の変数の推論は終わっているので、この時点では、exp の利用時の型は分かっている。

4.3.5　処理の順序

本システムを実現するために、これまで述べたような処理がなされる。それらが実際にどのような順序で実行され、どの時点で §4.3.2 や §4.3.3 におけるクラスやインタフェースが生成されるかを以下に示す。

1. 構文解析により AST が生成される。
2. ストラクトオブジェクトの型推論をする。
3. ストラクトオブジェクトの型であるクラスを定義すると同時に、var 型である変数の使用時の型となりうるインタフェースを定義し、new() のコードを書き換える。
4. var 型である変数の使用時の型を推論し、それに合ったインタフェースを既に定義してあるインタフェースの中から探し、変数を使用している部分をキャストし、フィールドアクセスをメソッド呼び出しに書き換える。
5. typeof の処理
6. エラー検知へと続く。

JastAddJ の既存部は 1 までと 6 以降であり、2〜5 が本システムにおける主な処理である。

var 型の型推論はストラクトオブジェクトの型をから決定されるので、ストラクトオブジェクトの型推論の後である。また、typeof は引数である表現が var 型の変数が含まれる可能性があるので、typeof の処理は var 型の型推論の後である。従って処理の順序は上記のような順番となる。

ただし、必ずしもこの順序で処理が行われるわけではない。なぜなら、例えば var 型の使用時の型を推論するために、typeof が表す型を用いりたい場合があるからである。このような場合には、適宜他の処理が部分的に行われる。

またストラクトオブジェクトの型であるクラスの定義時には、まだ実装すべきインタフェースが生成されていないため、§4.3.2 によるクラス定義のように、実装しうる全てのインタフェースをこの時点で定義し、それらのインタフェースを実装するクラスを生成している。

逆に var 型の型推論では、インタフェースを生成するのではなく、見合ったインタフェースを既に定義したものの中から探し出している。アクセス可能なフィールドとインタフェース名は一対一に対応しており、推論によって、インタフェースは用いられる部分ではキャストができることが保障されている。したがって、推論結果のインタフェースは必ず、何れかのストラクトオブジェクトの型によって実装されている。つまり既に定義されているのである。

第5章　実験

本システムにおける性能を判定するため、実験を行った。

5.1　実験概要

本実験では、既存の Java による内部クラスを用いて構造体を表現するプログラムと、本システムを用いたプログラムにおけるコンパイル時間と実行時間を比較する。前者については既存の JastAddJ コンパイラと本システムのコンパイラの両方について、後者は本システムのコンパイラを用いてそれぞれ計測する。

実験をするプログラムは図 5.1 のように構造体を表すオブジェクトを生成し、HashMap に格納し、そして取り出して出力するという一連の流れを行う run メソッドを 100 個定義し、それらを実行するというプログラムである。また比較する内部クラスを用いた構造体のプログラムでは、それぞれのメソッド内で

```
class Person{
  String name;
  int age;
}
```

という内部クラスを定義して、図 5.1 における p,p2 ジェネリクスの型引数の型を Person としたプログラムを用いて実験をした。

実験環境は以下の通りである。

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

5.2　実験結果と考察

実験結果は表 5.1、表 5.2 のようになった。

```
1  void run0{
2    var p = new();
3    p.name = ''hoge'';
4    p.age = 10;
5    HashMap<String,typeof(p)>hm =
6         new HashMap<String,typeof(p)>();
7    hm.put(''p'',p);
8    var p2 = hm.get(''p'');
9    Systen.out.println(p2.name);
10 }
11 void run1{…}
12      :
13
14 public static void main(){
15   run0();
16   run1();
17      :
18 }
```

図 5.1: 実験に用いたプログラム

表 5.1: プログラムのコンパイル時間 (ミリ秒)

用いたコンパイラ / プログラム	既存の JastAddJ	本システムのコンパイラ
構造体をクラスで表現したプログラム	1452	1474
暗黙的に型付けされる構造体のプログラム		2048

表 5.2: プログラムの実行時間 (ミリ秒)

用いたコンパイラ / プログラム	既存の JastAddJ	本システムのコンパイラ
構造体をクラスで表現したプログラム	23.3	23.6
var 型を用いた構造体のプログラム		23.8

まず、既存の Java におけるクラス定義を用いた構造体のプログラムのコンパイル時間は、既存の JastAddJ と本システムにおけるコンパイラで差はなかった。これは、Java を拡張した本システムが、既存の Java のプログラムをコンパイルするにあたって影響を与えないことを示している。従って、本システムのコンパイラを Java のコンパイラとしてそのまま用いたとしても、問題がないことが分かる。実際に本システムの実装では、抽象構文木から var 型とストラクトオブジェクトを探すために、まず一度全ての木構造を探索するが、もしそれらがなかったとすると、後は既存の JastAddJ の振る舞いと同じである。従って、オーバーヘッドはその点のみである。逆にこの実験によって、抽象構文木の探索・走査はオーバーヘッドとならないことが分かった。

本システムを用いた暗黙的に型付けされる構造体のプログラムを、コンパイルすると既存のプログラムにおけるコンパイル時間と比べて、約 40%ほどの時間がかかった。これは、構造体へのアクセスのための書き換え、型の推論、コード生成の 3 つがオーバーヘッドの原因として考えられる。本システムの実装では、構造体へのフィールドアクセスの書き換え、キャストなど多くのソースコード変換がおこなわれている。また本システムにおける型推論においても構造体のフィールド、キャストされるインタフェース、ストラクトオブジェクトなどの様々な処理を行っているため、

オーバーヘッドがかかってしまっている。さらに生成している Person を表す構造体のクラスでは、get メソッドや set メソッドがもとのクラス定義に追加してメンバとなっているため、クラスを生成するために余分なオーバーヘッドがかかるであろう。

しかし、今回実験対象としたプログラムは、構造体の生成・アクセスを繰り返すプログラムであり、また局所的に用いる構造体であるのに 100 箇所の異なる場所で定義されているといった偏ったプログラムであるためオーバーヘッドが顕著に現れたと考えられる。その一方一般的なプログラムは、構造体を用いるのはプログラムの一部分のみである。従って、現実的には、本システムを用いてプログラムを記述したとしても、それほど大きなオーバーヘッドとはならないと考えられる。

また実行時間については、何れにも差はなかった。これにより、本システムはプログラムの実行時間に影響を与えないことが分かる。本システムは、主にコンパイル時に型付けを行い、暗黙的に既存の 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がなかったと考えられる。

第6章　まとめと今後の課題

6.1　まとめ

本研究では、Java 言語において暗黙的に型付けされる構造体を実現する方法を提案・実装した。本システムでにおける構造体の生成は、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、構造体を生成したい時にその場所で、かつ簡潔なコードのみを使って記述できるという特徴がある。

Java 言語では、全てのオブジェクトはクラス定義が必要である。従って本システムでは、構造体を表すストラクトオブジェクトを生成するために、暗黙的にクラス定義を行い、そのクラスのインスタンスとすることで、クラス定義を必要としない構造体を実現した。そして、var 型とストラクトオブジェクトを用いて構造体を生成し、後にこれらの型を推論することで、擬似的に任意のフィールドを追加できる構造体を実現した。また、暗黙的に作られた構造体の型がプログラマから見えないため、構造体の型の名前を明示的に指定できない。その問題点を解決するために、推論された構造体の型等を取得する演算子である typeof を実装した。

本システムは、JastAdd によって実装された Java コンパイラである JastAddJ を拡張することで実装した。var 型などの新しい機能を定義し、ストラクトオブジェクトの型と var 型の変数の使用時の型を推論して決定し、プログラムの書き換えを行った。そして実験を行い、内部クラスを用いた構造体と比較して本システムにおける構造体が、コンパイル時間については現実的には問題ない程度のオーバーヘッドで、実行時間については影響なく実現できていることを確かめた。

6.2　今後の課題

本システムには、まだ課題がいくつか残されている。それらについて以下に説明する。

6.2.1　言語の仕様

本研究では、構造体の実現を研究の中軸として行ってきた。従って、仕様として曖昧であったり、安全性の保証・エラー検出について不十分である部分がある。また現在の仕様についても、本システムにおける言語のようなプログラムを記述するにあたって、それが本当に最も良い仕様であるかどうか疑問に残る。

以上のことから、仕様をしっかりと固めることが今後の第一の課題であると考えている。

6.2.2　配列としての宣言

構造体のようなオブジェクトは、同じようなオブジェクトを多く生成することが多々考えられる。そのような場合、それらのオブジェクトをまとめて管理する必要がある。1 つ目の方法として Collection クラスのオブジェクトに格納することが考えられ、実際に本システムでは、そのために typeof を導入した。しかし複数の同じオブジェクトを管理するという点では、配列も考えられる。例えば、

```
var[] n;
for(int i = 0; i<10; i++){
  n[i] = new();
  n[i].name = "n"+i;
  n[i].number = i:
}
```

のような使い方も考えられる。

今の時点で本システムでは、構造体の配列を宣言することは不可能である。従って、var 型の配列としての宣言など、いろいろな仕様を模索したいと考えている。

6.2.3　局所変数以外での構造体の利用

本システムにおける構造体は、局所的な利用を目的としたものである。しかし、その他では利用できないかという発想が自然に出てくる。例えばクラスのフィールドとして、構造体つまり var 型の変数を宣言したとする。この場合以下のような問題点が考えられる。

- フィールドのアクセス可能範囲はどうするか。
- 複数のクラスにまたがって利用されていたらどのように推論するか。

　またその他にもメソッドの引数としての利用も考えられる。例えば、

```
public setName(var n, String s){
 n.name = s;
}
```

のようなコードが書けると便利である。今は当然このようなプログラムは書くことはできないが、これらのような応用がいろいろ考えられる。

参考文献

[1] 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).

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

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

[4] 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).

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

[6] 五十嵐淳: プログラミング in OCaml, 技術評論社 (2007).