日本ソフトウェア科学会第 27 回大会 (2010 年度) 講演論文集

Javaのための暗黙的に型定義される構造体

大久保 貴司　千葉 滋

本研究では，暗黙的に型が定義される構造体と，その構造体を表す型 var を Java 言語に導入することを提案する．プログラミングにおいては，データをまとめるために構造体を利用したい場合がしばしば存在する．しかし構造体のような，複数のデータを保持するだけの単純なデータ構造を利用したい場合にすら，その都度その構造体を表すクラスを明示的に定義するのは煩わしい．本言語を用いると，型の定義をせずに，任意の名前のフィールドに任意の型のデータを格納することができる構造体を簡潔なコードで利用できるようになる．また，本言語で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 typeof 演算子も提供する．これはジェネリクスの型引数などに用いることができる．

1　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つである．レコード型は 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．以下，本論文ではレコード型を構造体と呼ぶ．

オブジェクト指向言語では，構造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，構造体のようなデータ型はクラスで代用することが可能だからである．本研究が対象とする Java 言語もその一例である．しかし構造体を利用するためには，本当にクラス定義をする必要があるのだろうか．構造体は，中にメンバ (フィールド) を持っていさえすれば良い簡単なデータ構造である．それにもかかわらず，データの要素が異なるというだけで，それぞれその構造体に応じた型を定義しなければいけないというのは幾分か冗長である．

以上の理由から，本研究では，Java 言語においてクラスを定義せずに構造体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，構造体は要素によって型が異なるため，それは難しい．従って解決法しては，構造体の型がその構造体に応じて暗黙的に定義されて，その型が付けられれば良い．本言語では，それを var 型とストラクトオブジェクトを導入することで実現した．ストラクトオブジェクトはクラスを定義せずに生成できる構造体を表すオブジェクトであり，var 型は暗黙的に定義される型が付く変数であることを表す型である．これらを用いることで，Java において簡潔なコードで構造体を利用することができる．

以下，2 章では構造体を生成する方法とその問題点について述べ，3 章では本言語の提案・説明をし，4 章では実装方法と評価について述べ，5 章では関連研究との比較をし，6 章ではまとめと今後の課題を述べる．

Implicitly type-defined structures for Java

Takashi Ookubo, Shigeru Chiba, 東京工業大学情報理工学研究科数理・計算科学専攻, Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology.

2 Java における構造体の生成における煩雑さ

2.1 クラス定義による構造体

例えば Java において，人に関する複数のデータを１つにまとめたデータ構造 (構造体) Person 型を作り，HashMap 等のコレクションなどに入れて管理することを考える．この場合，図 1 のように，各データをフィールドに持つ Person クラスを定義することが自然な方法である．確かに Person がメソッドを持っていたり，static なフィールドや初期化されるフィールドを持っている場合，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シンプルな構造体を生成したい場合，クラス定義は構造体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．よってこのようなクラス定義は省くことができることが望ましいが，Java ではそのような省略は不可能である．なぜなら Java がクラスベースなオブジェクト指向言語だからである．つまり，どんなオブジェクトを生成するためにも，標準ライブラリ等を除けば，クラス定義が必要なのである．

クラス定義をせずに．新しい (既存のクラスのインスタンスではない) オブジェクトを生成する方法として，匿名クラスの利用が考えられる．匿名クラスを利用すると，図 1 のコードは図 2 のように記述でき，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 行目のように final ではい変数にアクセスすることはできない．以上のことから，匿名クラスを利用して構造体を実装することは難しい．

```
1  class Person{
2    String name;
3    int age;
4  }
5
6  Person p = new Person();
7  p.name = "hoge";
8  p.age = 20;
9  Map<String,Person> map =
10       new HashMap<String,Person>();
11 map.put("p",p);
```

図 1　Java における構造体の生成

```
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　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;
```

図 3　JavaScript における構造体の生成

2.2 JavaScript での実現法

明示的に型を定義する必要のない構造体を実現できる言語の例として JavaScript [3] が挙げられる．JavaScript を用いると，図 1 の Java のコードは図 3 のように記述できる．これは前節において省略したかったクラス定義を省略できている．

Java の静的型付けを維持しつつ，図 3 のようなコードの記述ができることが望ましい．図 3 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る．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．従ってこれらの利点を失うことは好ましくない．

2.3 Java において JavaScript の記述法を実現する上での問題点

図 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．

1つ目はオブジェクトの生成法である．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．しかし前述したように，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．

2つ目は，構造体の型をどのように定義するか，つまり図 3 における p の型をどうするかである．Java は強い静的型付け言語であり，全ての変数には明示的に型を付けなければならない．一般的に Java のプログラムでは，定義したクラスを型として利用するが，クラス定義を省略してしまうと明示的な型付けができなくなってしまう．従って，クラス定義をしないということは，暗黙的に何かしらの型が定義され，その型が構造体に付けられなければならないということである．

3 ストラクトオブジェクトと var 型

本研究では，Java にストラクトオブジェクトと var 型を導入し，構造体のための暗黙的に型が定義され，またその型が暗黙的に付けられるシステムを提案する．これにより，構造体をクラス定義をせずに，簡潔に生成・利用できるようになる．

3.1 ストラクトオブジェクトと var 型の利用

本言語では構造体を生成するために，以下のように宣言する．

```
var st = new();
```

このように宣言すると，new() というキーワードによって，構造体を表すオブジェクトを生成できる．これをストラクトオブジェクトと呼ぶ．またストラクトオブジェクトを代入する変数である st の型として var 型が利用できる．これはその構造体に応じた型が推論され，暗黙的に付けられることを意味している．詳しくは次節以降で説明するが，暗黙的に付けられる構造体の型はどの様なフィールドをその構造体が持っているかで決定される．従って表記法としては JavaScript と同じであるが，意味としては JavaScript とは異なり，どちらかといえば，型推論される変数であることを意味する C#における var に近いものである．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．そしてストラクトオブジェクトはメソッド内でのみ生成でき，var 型はクラスのフィールドの型以外でしか宣言できないという制限が付く．

ストラクト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フィールドを実行中に自由に追加できる．例えば，

```
st.name = "hoge";
st.num = 3;
```

と記述すると，s は String 型の name というフィールドと int 型の num というフィールドを持つ構造体であることになる．つまり，代入したオブジェクトの型がフィールドの型となる (必ずそうなるとは限らない．3.2 節を参照)．追加されたフィールドは代入された行以降の任意の場所でアクセス可能である．

本言語では，var 型として宣言された変数の型は暗黙的に型付けされる．しかしそれによって，プログラマがその暗黙的に付けられた型の名前を認識できないという問題が存在する．型の名前を認識できないということは，明示的にその型を指定できないということである．変数宣言には var 型という特殊な型を付けるので問題はないが，生成した構造体の型をジェネリクスの型変数に渡したり，instanceof による構造体の型の比較をしたりすることができなくなってしまう．そこで本言語では typeof 演算子を提供する．typeof 演

```
1 var p = new ();
2 p.name = "hoge";
3 p.age = 20;
4 Map<String,typeof(p)> map =
5    new HashMap<String,typeof(p)>();
6 map.put("p",p);
```

図 4　本言語における図 1 のコード

算子は型を明示的に記述できる部分でのみ用いることが可能であり，typeof(exp) と記述することで，exp の型を取得することができる．exp は型をもつ任意の表現である．従って，typeof(st) とすることで st に暗黙的に付けられた型を取得することができる．

以上より，本言語を用いると図 1 のコードは図 4 のように記述できる．構造体を生成する部分に関しては，図 3 における JavaScript のコードとほぼ同じである．ただし，コードの意味としては異なる部分が存在する．大きな違いは p に静的な型が付いている点である．前述したように，どのようなフィールドを構造体が持っているかという情報をもった型が暗黙的に付けられる．従って，例えば図 4 のプログラムにおいて，

```
int i = p.height;
```

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ができるため，コンパイルエラーとすることができる．

3.2　構造体の型

前述した通り，本言語では var 型の変数には構造体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．

本言語の暗黙的な型付けは structural typing を利用して行われる．structural typing とは，型がその構造によって決定される型付け法である．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 ⊂ B であるなら)B は A のサブタイプである．

p の型 T は，初期値を T = φとして，p を宣言したメソッド内の全ての

```
p.foo = exp;
```

という代入式に対し，

```
if(T ∋{foo: t})
 then  T = (T-{foo: t})
        ∪{foo: Upper(t,type(exp))}
 else T = T ∪{foo: type(exp)}
```

を適用したものである．
ただし，type(exp) は exp の型，Upper(a,b) は型 a と型 b のアッパーバウンドの型とする．

図 5　var 型の変数に付けられる型を決定するアルゴリズム

本言語の structural typing における構造とは，その構造体がアクセスできるフィールドの名前とその型のペアの集合である．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．クラス定義を用いて構造体を表した図 1 の場合，構造体の型であるクラスはフィールドを宣言しただけのものであった．従って，これは直感的に正しい型の定義である．

構造体の型，つまりアクセスできるフィールドは次のようにして決定される．

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

というプログラムがあった場合，p の型は次のアルゴリズムから導出されるフィールドの集合である．

- 構造体のフィールドに代入する式があれば，そのフィールドの名前と代入されたオブジェクトの型のペアを追加する．
- もしも同一の名前に複数の代入があった場合，そのフィールドの名前と代入された全てのオブジェクトの型に対するアッパーバウンドの型とのペアを追加し，その他の型とのペアは除く．

これを形式化すると図 5 のようになる．

重要な点は同じフィールドに複数の代入があった場合に，アッパーバウンドの型をとることである．なぜ

なら，複数の代入式があった場合，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であると認識し，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，

```
class Shape{…}
class Circle extends Shape{…}
class Square extends Shape{…}
```

というクラス構造において，

```
var a = new();
a.shape = new Circle();
a.shape = new Square();
```

というプログラムがあったとする．この場合，shape の型は Circle と Square のアッパーバウンドの型である 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;
```

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．従って直観的には誤りのあ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 型となるので，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．

3.3 var 型の変数への代入

var 型はストラクトオブジェクトを指す変数の型である．従って，var 型の変数に既に生成したストラクトオブジェクトを代入することも可能である．例えば，

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

というプログラムを書くことができる．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の値を変更すれば，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，変数 b からアクセスし，フィールド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，フィールドを追加できるのは，ストラクトオブジェクトを生成したときに代入した変数を用いた場合のみである．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．

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ストラクトオブジェクトが代入することも可能である．例えば，図 6 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} となる．なぜなら変数 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 が指すストラクトオブジェクトが，a,b のどちらが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，どちらでも良いなら意味のある後者を選ぶべきである．つまりこれは，前節におけるフィールドの型が代入されるオブジェクトの型のアッパーバウンドの型になるのと同様に，structural type におけるアッパーバウンドの型

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

図 6　複数のストラクトオブジェクトの代入

初期化の右辺式が new() でない var 型の変数 a に付けられる型 T は，a を宣言したメソッド内で a に代入されるストラクトオブジェクトを表す各表現を exp_i とすると，

$\mathrm{T} = \bigcap_i \mathrm{type}(exp_i)$

である．ただし，type(exp) は exp の型とする．

図 7　var 型の変数に付けられる型を決定するアルゴリズム 2

が変数 c の型となるのである．一般的に，初期化の右辺式が new() でない var 型の変数に付けられる型は，図 7 のアルゴリズムで決定される．

3.4　メソッドにおけるストラクトオブジェクトの受け渡し

本言語では，メソッドの仮引数の型に var 型を用いることで，メソッドにストラクトオブジェクトを渡すことができ，また返り値の型に var 型を用いることで，返り値としてストラク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ストラクトオブジェクトと var 型の用法は，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 型を用いた場合，その仮引数が表すストラクトオブジェクトが，任意の型かつ任意の名前のフィールド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 型の仮引数を x とすると，以下の 2 通りに限られる．それは，

```
v = x.hoge;
```

という形のフィールドが指すオブジェクトを他の変数へ代入する式と，

```
x.foo = exp;
```

という形のフィールドへの代入である．フィールドへの代入式はフィールドを追加しているのではなく，実引数に既にフィールド foo があり，そこに exp を代入することを意味している．

```
1 void f(var x){
2   String s = x.name;
3   int i = x.age;
4   Object o = x.age;
5 }
```

図 8　メソッドの仮引数に var 型を用いた例

var 型として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．また仮引数の型であることから，この型がどのような構造体ならば実引数としてメソッドに渡すことができるかを表している．型付けの例を以下に挙げる．例えば，図 8 のようなメソッドがあった場合，仮引数 x の型は {name: String, age: int} である．これは仮引数 x が最低限どんなフィールドを持っていれば，このメソッドが正しいプログラムになるのかを考えれば明らかである．つまり，代入の右辺にフィールドアクセスがあれば，左辺の型と同じ型となり，更に複数の式の右辺に同じフィールドアクセスがあれば，それらのロウアーバウンドな型となる．またロウアーバウンドな型が存在しない場合，例えば上の例で変数 o の型が String であった場合では，型付けにおいてエラーが発生する．

図 8 のメソッドにフィールドに対する代入式を加えた場合，型には影響を与えないが，代入式の右辺の型がフィールドの型のサブタイプでなければ，型エラーが発生する．具体的に，

```
x.name = exp;
```

という代入式を図 8 のメソッドに加えた場合，exp の型が String のサブタイプであれば，型に変化はなく，サブタイプでなければ型エラーである．もしメソッド

内の代入式で，左辺にしかそのフィールドが現れない場合，つまり上の式があり，かつ図 8 の中の

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

という式がなかった場合，x.name の型は exp の型となる．これは x をローカル変数として宣言した場合と同様で，代入される各オブジェクトの型のアッパーバウンドの型がそのフィールドの型となるからである．

実際にメソッドを呼び出す場合，仮引数に付けられた型の structural typing におけるサブタイプであるストラクトオブジェクトであれば，メソッドに渡すことが可能である．つまり var 型の仮引数に暗黙的に付けられた型を T とすると，フィールドの集合として T ⊆ S となる任意の S 型のストラクトオブジェクトならば，そのメソッドの引数として渡せるということである．ストラクトオブジェクト以外も含め，そうでないオブジェクトをメソッドに渡した場合は型エラーが発生する．

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ストラクト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ストラクト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，

```
var foo(){
  var x = new();
  x.name = "hoge";
  return x;
}
```

というメソッドがあった場合，x の型は {String: name} であるから，同様に foo の返り値の型は {String: name} 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} となる．

4 実装と実験

本言語は，プログラム内のコードを標準の Java 言語に変換するすることと型検査によって実現されている．この章では，本言語のプログラムがどのような Java のコードに変換されているかを説明し，最後に本言語のプログラムと既存の Java のプログラムとを比較・評価する．

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

図 9 本言語を用いたサンプルコード

4.1 ソースコード変換と型検査

本言語で行われるコード変換を図 9 のプログラムを用いて説明する．

まず，var 型として宣言される変数 x の型は Object 型に変換される．つまり，var と記述されている部分は全て Object となる．言語の上では，変数 x に暗黙的に付けられる型は {name: String, age: int} であるがそれとは異なっている．これは，本言語では暗黙的に付けられる型を structural type を用いて表現しているが，これをクラスやインタフェースに単純に置き換えて表すためには多重継承を用いない限り不可能だからである．

次に，new() というキーワードで生成されるストラクトオブジェクトは，それぞれ各構造体に応じたクラスが生成され，そのクラスのインスタンス生成式に変換される．このクラスは暗黙的に生成され，名前も暗黙的に付けられる．図 9 の場合，次の様なクラスが内部クラスとして定義される．

```
class method_name$$01
      implements StringName,IntAge{
  private String name;
  private int age;
  public String getName(){return name;}
  public void setName(String name)
                {this.name = name;}
  public int getAge(){return age;}
  public void setAge(int age)
                {this.age = age;}
}
```

そして StringAge と IntAge は以下のようなインタフェースであり，これらも暗黙的に定義される．

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

つまり，構造体の各フィールドはインタフェースに対応し，構造体そのものはクラスに対応しているのである．

最後に，構造体のフィールドアクセスは対応する getter・setter メソッドに変換され，かつその構造体を指す変数は Object 型であるから，対応するインタフェースにキャストされる．従って，図 9 において

```
x.name = "hoge";
x.age = 22;
System.out.print(x.name);
```

はそれぞれ，

```
((StringName)x).setName("hoge");
((IntAge)x).setAge(22);
System.out.print(((StringName)x).getName());
```

に変換される．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り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．しかしそれだけでは不十分である．それはストラクトオブジェクトをメソッドに渡す場合である．var 型の変数を Object 型に変換するのと同様に，メソッドの仮引数の型も同様に Object 型となっており，任意のオブジェクトをメソッドの引数として渡すことができてしまう．従って，メソッドに渡すオブジェクトが正しいか型検査をする必要がある．これは単純に，メソッドに渡すストラクトオブジェクトの型が，仮引数の型の struct type としてのサブタイプであるかどうかを検査する．もしサブタイプ関係が成り立てば，メソッド内で要求されるフィールドを全て持っているということなので，メソッドに渡すことが可能である．

表 1　プログラムのコンパイル時間と実行時間 (ミリ秒)

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

4.2　評価

本言語でのコンパイラでは前節の通り，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．それによってどれほど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，構造体をクラスとして表した Java のプログラムと，その構造体を var 型とストラクト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比較する．

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 1 のようになった．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%ほどの時間がかかった．これは，ソースコードの書き換え，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 型の変数を用いている部分を探し，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，各フィールド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構造体の生成・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，現実的には構造体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．

5 関連研究

この章では，本言語と暗黙的に型が付けられる他の言語，関連する研究との比較をする．

型を定義することなく構造体を生成でき，静的に型が付けられる言語の例として scala[4] が挙げられる．scala のタプルは，型を定義することなく生成でき，任意のデータを要素とすることができるコレクションである．例えば，

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

と記述することにより，変数 p は Tuple2[Int,String] 型であると暗黙的に付けられる．しかしタプルは変更不可能なオブジェクトであり，要素の追加や代入はできない．また要素へのアクセスは要素のインデックスによって行われ，名前によってアクセスすることができなく，また順番が異なっても異なる型とみなされる．そして，異なるタプル間にサブタイプ関係が存在しないので，メソッドに渡す場合，完全に同じ構造のタプルを渡さなければならない．一方，本言語ではストラクトオブジェクトを生成後に代入可能なメンバを追加できる．また型を structural type として表し，サブタイプ関係も付けることが可能である．

また，OCaml[1] も強力な型推論を持つ静的型付け言語である．OCaml も scala と同様に，暗黙的に型が付けられるタプルを作ることができるが，やはり上記の問題は解決できない．OCaml の場合，フィールドと値をもつレコード型を利用することはできるが，これに関しては型をプログラマ自身が定義しなければならない．

本言語に関連のある研究として Whiteoak[2] がある．これは Java に structural type を導入した言語である．Whiteoak では，フィールドだけでなくメソッドも structural type の要素として指定できる．しかし，本研究の目的としている暗黙的な型付けはできなく，またインタフェースのような利用法であり，インスタンスを生成することはできない．

6 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が定義され，その型が付けられる構造体を実現する方法を提案・実装した．本言語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ことが特徴である．

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，後にそ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という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 を提供した．

6.2 今後の課題

本言語にはいくつかの課題点が残されている．まず，構造体のフィールドとして更に他の構造体を代入することができない．大きな理由としては，2 つの構造体が，それぞれのフィールドに互いの構造体が代入されている場合，型を決定することができないからである．つまり，2 つの構造体を a,b とすると，a の型を決めるためには b の型が，b の型を決めるためには a の型が必要なのである．

また現時点では，メソッドの仮引数として var 型を宣言した場合，仮引数の利用法をフィールドアクセスが代入式の右辺又は左辺であるように制限を付けている．なぜなら，例えば x を仮引数として，x.name.getSize() のように直接フィールドにメソッド呼び出しがあるとすると，name フィールドの型として，getSize を持つものという情報が必要となるからである．現在は，まだその点については実現できてい

ない.

参 考 文 献

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

[2] Joseph, G. and Itay, M.：Whiteoak: introducing structural typing into java, Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications(OOPSLA '08), 2008, pp.73–90.

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

[4] Odersky, O., Spoon, L. and Venners, B：Programmig in Scala, Artima Press, 1st edition, 2008.