論文紹介: A First-Class Approach to Genericity

久野　靖*

2004.5.27

1　はじめに

□ Genericity --- 「型パラメタ」を持つような型のこと

- 最初は CLU あたりから…
- 現在は多くの言語に「普通の機能」として採り入れられつつある
- しかしまだ「Ad hoc」「特別扱い」「ちょっと違う」が多い

2　紹介する論文

□ Eric Allen, Jonathan Bannet, Robert Cartwright, A First-Class Approach to Genericity, OOPSLA 2003, pp. 96-114.

- mixin が作れるような 1st-class genericity。
- いろいろと「そうだったのか!」と思わせるところがある。
- genericity とはどうあるべきか考えさせてくれるペーパー。

3　INTRODUCTION

□ ここ 10 年来、主流の OOPL は「static type systems with nominal subtyping」なもの☆

□ 言語例← C++、Eiffel、Java、C#

- 単純かつ直観にかなう型検査規則
- 明示的な型階層
- 相互再帰的データ構造を透明に扱える
- 実行時エラーになったときその原因が判りやすい

□ 現在の nominal（名前づけ）type system → 厳密な型検査には非力

- 実行時キャストが避けられない→実行中にそこで止まってしまう可能性
- 最大の弱点→ generic (parameterized) types → これを追加することで多くのプログラムの構造が改善されコンパイル時検査がより多く適用可能に

□ Java(JDK 1.5) と C#で導入

- Java の場合→ generic type は 2nd class(実行時にはその情報がない)
- C#の場合→実行時に型情報はあるが、パラメタは親クラスになれない (mixin クラスが作れない)

□ 以下では上述の問題のない 1st-class generic type の作り方を提案

- さらにそのように拡張された型体系が堅固であることを証明

4　BACKGROUND

4.1　Nominal versus Structural Subtyping

□ structural subtyping(構造型) な OOPL →オブジェクトはタグ (名前) なしレコード型。

- t1 ⊂ t2 とは、t1 が t2 の持つフィールドをすべて持ち、各フィールドの型が⊂であること。
- subtype 関係とコード継承とは無関係

□ nominal subtyping(名前型) な OOPL →オブジェクトはタグつきレコード型。

- タグ (通常クラス名) について subtype 関係を宣言→フィールドを継承
- たまたまフィールドが含まれても宣言がなければ subtype でない

□ 例:

*筑波大学大学院経営システム科学専攻

```
abstract class MultiSet {
  abstract public void insert(Object o);
  abstract public void remove(Object o);
  abstract public boolean isMember(Objet o); }

abstract class Set {
  abstract public void insert(Object o);
  abstract public void remove(Object o);
  abstract public boolean isMember(Objet o); }
```

□ Java では Set と MultiSet は無関係だが構造型な言語であれば subtype 関係になるはず

4.2 Generic Types

□ 名前型な OOPL に generic type を入れる場合、クラス定義に型変数がつけられ、クラス内ではその型変数を型として自由に使える

- Java の場合の構文:

  ```
  class 名前<型パラメタ, ...> extends ...
  ```

- 型パラメタには bounds が指定できる。「extends クラス」または「implements インタフェース」。指定しないと Object。
- 実体化 (パラメタあてはめ) の際にこの指定に適合する必要

□ genreric type(汎用型) →型変数、またはパラメタつきのクラス (パラメタが再び汎用型でもよい)

□ 30 年にわたり研究されてきたが、名前型な言語における汎用型についてはあまり研究されていない

- 名前型の OOPL の場合、クラス情報が実行時に参照できそれに基づいて分岐などもできる→動的な型の言語に近くなる

□ 構造型な OOPL の場合、実行時には型情報を消去→同じ構造なら区別がつかない

- タグによる判別をしない→コンパイル時の検査により安全性を保証

4.3 Nominal Type Systems Supporting Genericity

□ Java への genericity の付加→ Odersky, Wadler らによる Generic Java (GJ)

- GJ の実装→型消去 (type erasure) による
- Featherweight GJ(FGJ) → GJ の型モデル。堅固さの証明

□ .NET でも Kennedy, Syme らのモデルによる genericity の付加

- .NET では型情報を実行時まで保持

4.4 Type Erasure

□ 型消去→コンパイル時にチェックし終わったら、実行時には型パラメタの情報なしで実行

- たとえば List<T>を型検査し終わったら、実行時はただの List として実行
- 型消去を用いた場合、実行時には汎用型の情報はない→ 2nd class な型ということになる
- 実行時に型情報がないので、型情報に依存する実行時の操作を安全に用いることはできない

5 FIRST-CLASS GENERIC TYPES

□ 2nd class であることの問題いろいろ

- 型パラメタに対して Cloneable インタフェースを使うことができない (clone() した結果は Object なので元の型にキャストしたいが元の型は実行時には存在していない)
- 型パラメタ T について「new T[]」と書くと「new E[]」になる (E は bound 型)

□ NextGen --- GJ の拡張版で実行時にも型情報を残す。C#の genericity もこれに相当

□ パラメタ型が実体化されるごとに (特別なクラスローダによって) インタフェース、クラスを生成してパラメタなしクラス階層の中に情報を埋め込む (図 1)

- これにより実行時に型情報が使え、上述の問題も解消され、実行オーバヘッドもさほどない

□ ただし型パラメタを親クラスにはできない

```
class C<T> extends T { ... }
```

- これにより mixin が作れない。これができるのは mainstream 言語では C++だけ
- たとえば図 2 のような用途

```
class TimeStamped<T> extends T {
  public long time;
  TimeStamped() {
    super();
    time = new java.util.Date().getTime();
  }
}
```

- これができない場合、毎回コピーして作るか、コンポジションするか。どちらも嬉しくない。
- しかしこれをできるようにすると色々考えるべきことが…

6 MIXINS: HISTORY AND MOTIVATION

□ mixin という言葉のはじまり --- Lisp 系言語の Flavors

- あるクラスを「修飾」するためのクラス
- 意味論的には「クラスをクラスにマッピングする関数」

□ mixin の重要な用途

- uniform なクラスの拡張機能を提供 (TimeStamped のように)
- 多重継承の代替 (簡潔、きちんとした)
- アプリケーションをきれいに分割する上で重要な機能

□ mixin を generic クラスにすることの利点←可変部分をパラメタとして指定することできちんと型検査可能

7 THE DESIGN OF MIXIN

□ NextGen の拡張として MixGen 言語を提案した

□ その説明の準備として NextGen の簡単な拡張、GJ の弱点の説明から

7.1 Preliminaries

□ GJ にはさまざまな制約があるがその 1 つとして…

□ パラメタつきクラス内で new T() を使う場合、T としては具体的なクラスを渡さないといけないし、コンストラクタのパラメタがちゃんと一致していないといけない。

□ C#ではコンストラクタの引数は 0 個でないといけない。

□ NextGen の次期バージョンでは with 句でコンストラクタのシグニチャを規定するようになる

```
class Container<T with {init(); init(Integer);}> ...
```

□ また、もっと読みやすくするためいくつかの構文塘

- init の代わりに型名を書いてもよい
- 中かっこやセミコロンをはぶいてよい
- その場に書くかわりにインタフェースを定義してもよい

7.2 MixGen Extensions

□ NextGen と上位互換な拡張。次の 2 点が拡張されている。

- 親クラスとして型パラメタが書ける --- mixin が作れる
- with の中に abstract とか final が指定できる --- メソッドの同定に必要

7.3 Cyclic and Infinite Class Hierarchies

□ Java でも GJ でも型階層は DAG(サイクルがない)

□ 1st-class genericity でも同様だが、そのチェックは面倒

- Ex. 1

```
class C<X with ...> extends X ...
class D extends C<D> ...
```

- Ex. 2

```
class C<X with ...> extends X ...
class D<X> extends C<X> ...
class E extends D<E> ...
```

- Ex. 3

```
class C<X with ...> extends X ...
class D1 extends C<D0> ...
...
class Dk extends C<Dk-2> ...
class D0 extends C<Dk-1> ...
```

- Ex. 4

```
class C<X with ...> extends D<C<C<X>>> ...
class D<X with ...> extends X
```

(これは D<C<Object>> → C<Object> → D<C<C<Object>>> → … となる)

□ DAG を保証する 1 つの方法

- generic class が mixin class をパラメタつき拡張することを禁止
- 厳しすぎて不便→ mixgen では別の方法で検査

□ 用語: secondary mixin --- C<T_1, ..., T_n>が T_i を親クラスとして持つこと (Ex.2 が相当)

- すべての secondary mixin を網羅するには→まず普通の mixin をすべて同定し、「class D<X ...> extends C_1<C_2<...<>>>」においてすべての C_i が普通の/secondary mixin であるものを集める

□ DAG を保証する mixgen のアルゴリズム

- (1) extends に現れるすべての mixin を引数 (親クラス) に置換← mixin の介在を削除してもループの有無は変化しない
- (2) クラスヘッダ中の mixin 定義を削除← mixin ヘッダはそれ単独ではループにならない
- (3) すべての型引数を削除←パラメタを削除してもループの有無は変化しない
- あとは通常のアルゴリズムで DAG 性をチェックすればよい

□ 先の例に適用すると…

- Ex.1 「D extends D」
- Ex.2 「E extends E」
- Ex.3 循環する extends 関係に
- Ex.4 「C extends C」

7.4 Accidental Overriding

□ 「M<T extends B with ...> extends T」にさまざまな T を与えた場合、T に含まれ B に含まれないメソッドを M で定義してしまうかも→「事故的上書き」(上書き事故?)

- mixin の型検査は難しい←何がパラメタに入って来るか判らない
- Figure 3.

```
interface I {
  Object f(); }
class C<T with T()> extends T implements I {
  C() { ... }
  Object f() { ... }
  Integer m() { ... } }
class D<T implements I with T()> extends T {
  D() { ... }
  Object f() { ... }
  String m() { ... } }
class DFactory<T implements I with T()> {
  T create() { return new D<T>(); }
class E<T with T()> extends T {
  Integer typeBreaker(C<Object> x) { return x.m(); } }
...
Object d = new DFactory<C<Object>>().create();
Integer e = new E<Object>().typeBreaker(d);
```

- 最後の typeBreaker() の x.m() で Integer のはずが String が返される→コンパイル時にチェックできない型エラー (?)
- このような上書き事故は Java や C#のような分割コンパイルでは発見不可能→全プログラム検査が必要になってしまう
- エラーが発見できてもパッケージ内側の問題なので利用側には理解しがたいかも。

7.5 Hyginenic Mixin

□ 上書き事故をなくさない限り型検査は困難→mixin の意味を変更するしかない→実はそういうものがあった [16]

□ 動的分配の静的型検査…型 T でチェックし、実行時にはそれに適合するシグニチャS のメソッド (存在は保証されている) を探索する

- モデル: T から実行時の型に向かって下向きに S を探しレシーバに最も近いところで見つかったものを採用
- モデル: 実行時の型から T に向かって上向きに S を探し最初に見つかったものを採用
- これまでの場合、どちらでも同じだったわけだが…

□ MixGen ではクラス階層が実行時まで判らないのでもっと工夫が必要

□ T から下向きに S を探すが、mixin クラスがあってその bounds に S が含まれていなければそこで止まる。

- すなわち、mixin クラスは S を隠す (パラメタ型について利用可能なものは bounds に書かれているものだけだから)。
- 先の typeBreaker の例でも m() の探索は D<>で止まる→外側の m() しか見つからないので OK。

□ この方式では「どこから探すか」を (アップキャストによって) 制御可能→プログラマがより多くの制御を行える。

□ 現在の JVM や CLR で効率よく実装可能 (次節)

8 IMPLEMENTING MIXGEN

□ NextGen →できるだけ「均一な表現」(homogeneous representation)

- パラメタ型のインスタンスに共通なコード→共通の抽象親クラスに
- 特定のインスタンス専用のコード→そのサブクラス(個別のクラス)に
- すべての generic なメソッド呼び出しは型消去された形で実行
- 個別型固有の部分→ snippet(断片) メソッド呼び出し→各型で上書き

□ NextGen コンパイラ→ 2 つのクラス+インタフェースを生成

- 型消去した親クラス
- 個別クラスのテンプレートクラス (snippet の集まり)
- 対応するテンプレートインタフェース (親子関係を表現)
- テンプレート中では List<T>は List$$L{0}$$R といった加工された形で存在→クラスローダが展開

8.1 Basic Implementation Strategy

□ NextGen 方式では駄目 (親クラスが変わるから共通の親クラスは作れない)

□ 「非均質な表現」(heterogeneous representation) を採用

- 個別のクラスがそれぞれの実装コードを持つ

8.2 Enforcing Hygienic Method Invocation

□ MixGen の下向き探索を通常の分岐テーブルにコンパイルしたい→クラスローダにより隠されるメソッドを改名

- mixin クラスで導入されるすべての新しいメソッドは改名が必要 (mixin のインスタンスのサブクラスでその名前の定義をするかも)
- 改名されたメソッドへの参照も対応して書き換え

□ インタフェース呼び出しでは統一的な書き換えは難しい (呼び出し側に応じて書き換えが違う) →転送メソッドを生成

□ インタフェースの適用にかかわる問題

```
interface I<T> { T m(); }
class C<S with S()> extends S implements I<S> {
  S m() { ... } }
class D implements I<String> {
  String m() { ... } }
... C<D> ...
```

- 1 つのメソッドが複数の generic インスタンス化で何回も定義されることもある→各回ごとに違う書き換えを生成
- 結局、メソッド名の書き換え方式とは、クラス (インタフェース)C で新たに導入されたメソッドに対して、そのフルクラス名を前置することに。
- 上の場合クラス D での m は「I<String>m」。クラス C での m は「I<S>m」で、ロード時にパラメタがあてはめられる。ただの m() は転送メソッドでパラメタ S の m() を呼び出す

□ いろいろ大変だけどこれで普通の JVM 上で下向き探索のようにふるまうコードが作れる

9 CORE MIXGEN

□ 型システムの堅固さ…(略)

10 RELATED WORK

□ Agesen, Freund, Mitchell[1] --- 構文拡張による型パラメタ→ hygienic でないので型検査は困難なはず

□ JAM[6] --- 型パラメタではなくトップレベルの構文としての mixin → これも hygienic でない

□ Jiazzi[20] --- コンポーネントシステム、ローダによる組み合わせで mixin ができる。これも言語ではないので…

□ MixedJava[16] --- 最初に hygienic mixin を提案したが generic type はない。

- すべての型を mixin で表現する。すべての値は型 (mixin の連鎖) とオブジェクト参照のペア (ポインタがすべて倍の大きさになるという問題あり)
- ダイナミックに型を持つという点で興味深いが Java や C#の代案にはならない

11 FURTHER RESEARCH

☐ nominal type OOPL + 1st-class genericity → 強力な言語

- 今後広まると考える
- JVM 上に実装→いろいろオーバヘッドあるが重大ではないと思う (forwarding method は JIT がインライン化できるし)

☐ CLR はオーバライドされない new をサポート→これを利用して名前書き換えでない実装ができるかも