Generic Java：多相的型付けによる安全かつ再利用性の高いオブジェクト指向プログラミング

京都大学大学院情報学研究科
五十嵐　淳
igarashi@kuis.kyoto-u.ac.jp

Java 1.5 では Java 言語の導入以来初めての大きな言語仕様の変更が行われる．そのうちの主要なものとして**総称クラス**（*generic class*）・**ワイルドカード型**の導入が挙げられる．総称クラスとは，C++ 言語のテンプレートのような，型パラメータにより抽象化されたクラス定義のことであり，ベクトル・木・リストなどの汎用データ構造のプログラミングにおいて有用である．総称クラスは ML，Haskell などの型付き関数型言語でみられる多相的型付けを応用した機構であるが，最近の筆者らによる研究において，（型付き）オブジェクト指向言語において伝統的である，部分型多相との新しい融合手法が考案されている．ワイルドカード型は，この手法に基づき導入された機構である．

本稿では，型システムの改良が，いかに言語の柔軟性を損なうことなく，プログラムの安全性の向上に貢献できるかという例として，これらの機構を概観する．

はじめに

プログラミング言語 Java は，今でこそサーバなどのソフトウェアに使われていたり，すっかり汎用言語として馴染んだ感があるが，その登場当初は「WWW ブラウザ上でアニメーションをするため」のアプレット用プログラミング言語として捉えられていたような気がする．アプレットは通常ネットワーク越しにダウンロードし実行するものであるが，これは，作者が誰ともつかず，どんな動作をするか分からないソフトウェアを使うという，ある意味とても危険が伴うことでもある．そのため，Java（言語だけでなく，実行時系も含めた全体）では，海のものとも山のものともつかないソフトウェアを「安心」して使えるように，さまざまな安全性向上のための技術が採用されている．その例としては，

- ガーベジコレクションによるメモリ安全性
- 型システムによるソース言語レベルでの型安全性
- バイトコード検証によるバイトコードレベルでの型安全性
- サンドボックス実行によるアクセスコントロール

などが挙げられるだろう．その技術の多くは当時としても決して新しいとはいえないものだったが，古典的な技術の集大成としての意義は十分にあったと思われる．

本稿では，このうちの「型システムによるソース言語レベルでの型安全性」について述べる．ちょうど，Java は次期バージョン 1.5 において大きな機能追加が行われ，型システムにも大きな変更がなされるということで，そこで導入される機能の一部である**総称クラス**（*generic class*）・**ワイルドカード型**の紹介を兼ねて，型システムの改良が，いかに言語の柔軟性を損なうことなくプログラムの安全性の向上に貢献できるかについても議論していきたい．

本稿の構成は以下の通りである．はじめに，簡単に型や型安全性といった用語のおさらいを行う．続いて，総称クラスの基本的な仕組み・ワイルドカード型と呼ばれる総称クラスの有効利用のための新しい型機構について順次述べ，まとめを行う．

型，型エラーと型安全性

プログラミング言語の文脈で「型」という時には，実行時のデータの種類（Java では，オブジェクトが生成されたクラスの名前と考えられる）を表す，いわゆる動

的な型と，コンパイル時に型付け規則に従って与えられる式の属性である，いわゆる静的な型のいずれか，もしくは両方を曖昧に指すことが多いと思われる．（静的な）型は，たいがい，式の計算結果のなんらかの意味 — たとえば int 型の式の計算結果はなんらかの整数であるなど — での近似であるが，特に Java のようにクラスに基づく言語では，クラスの名前がそのまま型の名前としても用いられるため，両者が混同されがちである．また「型エラー」という言葉も「型」の曖昧性に伴い曖昧になっている言葉である．Lisp などの静的型検査をしない言語では，文字列と整数の和をとるといった，ある操作がその対象としてふさわしくない種類のデータに適用されるという実行時エラーを「型エラー」と呼び，静的型付け言語では，コンパイル時に発生する，静的な型が与えられないというエラーを「型エラー」と呼ぶことが多い．

本稿では，これらの区別を明確にするために，いわゆる動的な型は「オブジェクトのクラス」と呼び，静的な型を単に「型」と呼ぶことにする．また「型エラー」とは単に式に静的な型が与えられない場合に発生する，コンパイル時のエラーのことであるとする．そして，型安全性とは，「型エラーのないプログラムは，実行時にある種の実行時イベント（エラー）が発生しない」という性質である．「ある種の実行時イベント」というのは言語・型システムに依存するものであり，Java の場合は，たとえばオブジェクトにないフィールド／メソッドがアクセスされることである．そうではないイベントとしては，0 での除算や配列アクセスのインデックスが配列の長さを超える，または型キャストに失敗する，といったことが含まれる．

総称クラス

本章では，総称クラス機構を概観する．まず最初に，総称クラスの動機付けとなる，リスト・木などの，いわゆる汎用データ構造を Java で記述することを考え，安全性の観点からその記述の不十分な点を指摘する．

動機付け

ここでは例として，リスト構造（のセル）を表現するクラスを考える．リストに格納されるデータがたとえば文字列 String であれば，

```
class StrList {
  String head;  StrList tail;
  public StrList(String h, StrList t){ head = h; tail = t; }

  public int length() {
    if (tail == null) return 1;
    else return tail.length() + 1;
  }
  ...
}
```

などと記述できる．しかし，これでは格納される要素ごとに別のクラスを書かなくてはならず，クラスの再利用性が損なわれる．このような場合，Java のプログラミング・イディオムとして，部分型関係を利用することが一般的である．具体的には Object 型が任意の（参照）型を部分型とする性質を利用して，データが格納されるフィールドには Object 型を与え，

```
class List {
  Object head;  List tail;
  public List(Object h, List t){ head = h; tail = t; }

  public int length() {
    if (tail == null) return 1;
    else return tail.length() + 1;
  }
  ...
}
```

と定義する．このような定義を用いると，

```
List l1 = new List("foo", new List("bar", null));
List l2 = new List(new Integer(10), null);
```

などと，要素が何であろうとリスト構造を作ることができる．これは，String や Integer が Object の部分型であることで，Object 型の引数を要求するコンストラクタに，文字列・整数型の式が渡せているのである．

しかしながら，l1 や l2 から要素を取り出すには，素朴に head フィールドを読むだけではだめである．たとえば，

```
String s = l1.head;
```

とすると，以下のような型エラーが発生する．

```
List.java:14: 互換性のない型
検出値 : java.lang.Object
期待値 : java.lang.String
       String s = l1.head;
                  ^
エラー 1 個
```

これは，head の型が Object として宣言されているために，l1.head の計算結果は文字列であるにもかかわらず，型システムは Object 型しか与えてくれず，String 型の変数には代入できないのである．つまり，いったん，部分型関係を使って Object としてリストに格納すると，本来の型情報が失われてしまい，読み出し時には粗い型情報しか得られないのである．

この問題を解消するために，Java では型キャストという機構を利用する．型キャストは（〈型〉）〈式〉という形の式で，実行時には〈式〉の計算結果のオブジェクトのクラスが〈型〉を継承（インタフェースなら implements）しているかを検査する．検査に通れば，型キャスト式全体の計算結果は〈式〉の計算結果であり，検査に通らなければ ClassCastException 例外が発生する．また，型付けとしては，〈型〉と〈式〉の型の一方が他方の部分型である時に，型キャスト式全体の型は〈型〉になる．これを使うことで，型情報を回復することが可能である．つまり，先程の head を読む例は

```
String s = (String)l1.head;
```

と書くことで，型エラーを出さずに s に head の内容を代入することができる．

しかし，プログラムの安全性の観点からすると，この解決策は

- どんな型キャストを挿入するかは，プログラマの「このリストには文字列が入っている（はず）」という知識・確信に基づくものであり，あるキャストが必ず成功することは Java の型システムでは保証されない（つまり，型キャストの失敗は前章で述べた「実行時に起きないことが保証されるイベント」ではない）．
- 一方，必ず成功する（はずの）キャストであっても，原則的には実行時検査が行われるので，プログラムの実行効率が落ちる☆1．

という点で不十分である．

この問題点を解決するのが，総称クラスの機構である．

総称クラスの基本

総称クラスは，ML などの関数型言語の型システムに見られる，パラメトリック多相的型付けを応用した仕組みである．大雑把にいえば，手続き・関数がプログラム中の値の一部を変数としてパラメータ抽象したものであるとすると，総称クラスはクラス定義☆2 の型情報の一部を**型変数**としてパラメータ抽象したものである，といえる．実際の例を見てみよう．先程の List クラスを総称クラスで書き直すと，

```
class List<X> {
  X head;  List<X> tail;
  public List(X h, List<X> t) { head = h; tail = t; }

  public int length() {
    if (tail == null) return 1;
    else return tail.length() + 1;
  }
  ...
}
```

となる．ここでクラス名 List の後の <> で挟まれた X が型変数パラメータであり，このクラス定義中では，通常の型として使用することができる．また，クラス名 List は List<〈型〉> という形式で型として使用することができる☆3．このような形式の型表現をしばしば**パラメトリック型**（*parametric type*）と呼ぶ．このような形式を使うことによって，あたかも List クラスの型変数 X を〈型〉で具体化したような（総称でない）クラス定義が存在するかのごとくプログラムをすることができる．つまり，List<String> は String を要素とするリストとして，List<Integer> は Integer を要素とするリストとして

```
List<String> l1 = new List<String>("foo",
                    new List<String>("bar", null));
List<Integer> l2 = new List<Integer>(new Integer(10),
                    null);

String s = l1.head;
```

のようにプログラムを書くことができる．l1 や l2 の型，new に与えられる型としてパラメトリック型が使われているのもさることながら，ここで注目すべきは，l1 に与えた型のおかげで，l1.head 型が String（クラス定義から分かる head の型 X を String で具体化して得られる）であることが分かり，型キャストが不要になっている点である．このように総称クラスを用いることにより，（型だけ違う）同様な定義を繰り返すことなく，（型キャストに頼らない）安全なプログラミングが可能になる．

パラメトリック型の型引数（<> で囲まれた部分）は概念的にはどんな型でもよいが，Java（1.5）においては，型引数として用いることができるのは Object，String，List<String> などの（総称）クラスに由来する，いわゆる参照型のみであり，int などのプリミティブ型は使用できない，という制限がある．

総称クラスは複数の型変数で抽象化することも可能である．

```
class Pair<X,Y> {
  X fst; Y snd;

  Pair(X f, Y s) { fst = f; snd = s; }
  void setfst(X newfst) { fst = newfst; }
}
```

☆1 もちろん JIT コンパイラの最適化で無駄な型キャストが除去できる可能性はある．

☆2 本稿では詳しく扱わないが，総称インタフェース，総称抽象クラスを定義することも可能である．

☆3 上の例のフィールド tail やコンストラクタの引数の型として使われている List<X> も型である．

このクラスに対応するパラメトリック型はPair<〈型$_1$〉,〈型$_2$〉>という形式で記述される．また，List<List<Integer>>（整数リストのリスト）やList<Pair<String, Integer>>（文字列と整数の組のリスト）のような入れ子になったパラメトリック型を使用することも可能である．このように，既存のクラスを組み合わせて新しいデータ型を定義できることは総称化することによる利点の大きな利点の1つであろう．

その他にも，総称クラスと継承・部分型，型変数の動く範囲の上限指定，総称メソッド（型引数をとるメソッド）と型推論など表面言語上の機能として重要なものがあるが，詳しくはチュートリアルなどの文章を参照してもらいたい．重要なことは，総称クラス・パラメトリック型を用いることで，型キャストに頼ることなく型付けできるプログラムが増え，安全性が向上する，という点である．

実装について

Javaのプラットフォームでは互換性が重視されているため，総称クラスを実装するにあたって，仮想機械（の仕様）の変更は望ましくない．このため，今までと同じ仮想機械言語上のバイトコードにコンパイルされるよう実装がなされている．ここでは議論を簡単にするために，仮想機械言語へのコンパイルではなく，総称クラスを含むコードから（総称クラスを含まない）Javaコードにどのように変換されるかを述べる．

基本的には，1つの総称クラスは同名のJavaクラス（もしくはクラスファイル）1つへ変換される．この際，型変数宣言・型変数・パラメトリック型など（古い）Javaにないものは以下のように変換される．

- class C<X> {...}はclass C {...}というように型変数の宣言は消去
- クラスの定義内で型として使われている型変数Xは Objectに変換
- パラメトリック型C<...>はCに変換
- 変換後のプログラムで型が合わないところには，型キャストを挿入

具体的な例を見てみよう．先程のList<X>クラス

```
class List<X> {
  X head;  List<X> tail;
  public List(X h, List<X> t) { head = h; tail = t; }

  public int length() {
    if (tail == null) return 1;
    else return tail.length() + 1;
  }
  ...
}
```

は，上の規則に従って，

```
class List  {
  Object head;  List  tail;
  public List(Object h, List  t) { head = h; tail = t; }

  public int length() {
    if (tail == null) return 1;
    else return tail.length() + 1;
  }
  ...
}
```

と変換され，

```
List<String> l = new List<String>("foo",null);
String s = l.hd;
```

は

```
List         l = new List        ("foo",null);
String s = (String)l.hd;
```

となる．ここで，hdフィールドで型キャストが挿入されたのは，変換後のプログラムではl.hdの型がObjectであるためである．型キャストは，変換前のプログラムで，型変数を具体化することで型が得られているような式（l.hdの型はListの型変数Xを Stringで具体化することで得られていた）に対して挿入されることになる．

と，こうして変換後のプログラムをよくよく眺めてみると，実は，なんのことはない最初に示した（総称クラスのない）Javaで汎用的なデータ構造を表現する際に使ったクラスと同じである．では，総称クラスを使ってプログラムを組むことの意義は一体なんだったのだろうか．それは，

> パラメトリック型という，より情報量の多い型を使って機械的に型検査をしてから，機械的に変換することにより，（変換の過程で導入された）**型キャスト**に関しては**必ず成功する**ことが保証できる

ということである．これにより，先に述べた，問題点の1つが解決できたわけである（この「キャストが失敗しない」ことの証明は形式化されたJavaの部分言語Featherweight Java[4]に対して，形式的に行われている）．もう1つの型キャストの実行時のコストの問題に関しても，パラメトリック型を理解する（，かつ古いバイトコードも受け付けるような）バイトコード検証器により可能であると思われる．

互換性という設計思想

Javaに総称クラス機構を導入するにあたって，最も重要視されたことの1つが「互換性」である．ここでいう互換性には，

- 拡張言語は古い言語をサブセットとして含む（表面言語の上位互換性）だけではなく，
- 拡張言語のプログラムは古い仮想機械上でも走るバイトコードにコンパイルされる（バイトコード互換性）
- 古い Java プログラムは新しいプラットフォーム上でもコンパイル可能（レガシープログラム互換性）

といったプラットフォーム全体を考慮したさまざまな互換性が含まれている．上で見たように，総称クラスはコンパイル時に同等な Java クラスにコンパイルされるため，バイトコード互換性が満たされている．

最後のレガシープログラム互換性は一見自明に思えるかもしれないが，これは正確にいうと「『古いライブラリを念頭に置いて書かれた』古い Java プログラムは『総称化された』新しいプラットフォーム（ライブラリ）上でも引き続きコンパイル可能」という性質である．つまり，総称化された `List<X>` クラスに対しては，最早 `List` という名前は型として意味をなさないため，`List` がプログラム中に現れるような古いプログラムは，普通に考えるとコンパイル不可能である．このようなレガシープログラム互換性を保つために Java 1.5 では，`List<X>` に対して，型引数のない `List` も型として認め，`new` でインスタンスを作ることも認めている．このような型引数を欠いた型を *raw type* と呼んでいる．raw type の詳しい仕組みについては他の文書[2]に譲るが，基本的なアイディアとしては，raw type `List` は仮想的に変換後の `List<X>` クラスのようなものとして使える．つまり，たとえば raw type `List` の `hd` フィールドは `Object` 型として使用することができる．

このような類いの互換性を考慮に入れた言語設計は珍しいと思われるが，raw type は，あくまでその場しのぎのための措置である．raw type に対する一部の操作は，安全であること（コンパイラが挿入した型キャストが失敗しないこと）が保証ができない．コンパイラはそれを型検査の失敗とせず警告を出力するだけである．そのため，raw type は「とりあえず昔のプログラムでも変更なしに新しいコンパイラでコンパイルできる」といった程度のものと考えた方がよいだろう．

また，実装として C++ のように型引数ごとに特化したクラスへコンパイルする手法をとらなかった理由としては，しばしば指摘されるコード爆発の可能性以外にも，（総称）クラス =1 クラスファイルという性質を保つ，ということがあったと考えられる．また，現在の実装方法では総称クラスをプリミティブ型で具体化することができない，など実装方法を知らない限り不自然とも思える制限がいくつか設けられている．このあたりのトレードオフは言語設計の観点から興味深いものである．

ワイルドカード型

本章では，パラメトリック型に対してより柔軟な部分型関係を許すための機構であるワイルドカード型に関して説明する．

動機付け

Java の配列型は，構文は違うが，一種のパラメトリック型である．つまり `[]` に型引数 `Object`，`String` などを前置して，`Object[]`，`String[]` などの型が構成できる．また，Java の配列型には「要素型同士に部分型関係があれば，それらの配列型にも部分型関係がある」という規則がある．つまり，`String` は `Object` の部分型であるため，`String[]` は `Object[]` の部分型として扱える．このように，ある総称クラスに対し，型引数の部分型関係がパラメトリック型に保存されることを，（そのパラメトリック型に）**共変部分型**（*covariant subtyping*）が許されている，といったりする．これは，あくまで，パラメトリック型同士の部分型関係を導出するための規則として共変部分型を認めるか認めないか，（文字列の配列オブジェクトを `Object[]` 型の変数に代入することを認めるかどうか）という話であり，共変部分型を認めることが安全かそうでないかは別に議論しなければいけない．実際，配列型に対して共変部分型を認めるのは，あまり安全ではない．

たとえば，以下のようなプログラム

```
String[] ss = ...;
Object[] os = ss;  // covariant subtyping
os[0] = new Integer(100);
int i = ss[0].length();
```

は，共変部分型が認められているおかげで，型検査に通る．しかし，これを素朴に実行しようとすると，`ss` の指す配列の第 1 要素には `Integer` オブジェクトが格納されてしまい，`length` メソッドの起動に失敗してしまう（つまり，型安全ではなくなってしまう！）．Java では，この問題を回避するために，配列への代入が本当に許されるものか（ここでは代入される `Integer` が `os` が指す配列の要素クラスの部分型であるかどうか）を実行時に検査し，この例では，`os` が文字列配列を指しているので，`os[0] = ...` の代入に対して例外が発生し，`length` メソッドの起動は起こらない．このため，ありもしないメソッド起動が起きないという意味では型安全性が保たれている．しかし，

- 代入される要素が正しいものであるかは，プログラマの知識に頼っており，コンパイラは保証してくれない[☆4]．
- 代入のたびに実行時検査が行われるので，（JIT コンパイラの最適化で除去できる可能性があるとはいえ）プログラムの実行効率が落ちる．

という意味で広い意味では安全性が落ちているといえる．このように一般的に共変部分型は認めるべきではない規則である．実際，Java 1.5 では（配列型を除く）パラメトリック型に対する共変部分型は許されていない．

ワイルドカード型はこの問題を回避するために導入された型機構である．

ワイルドカード型の基本アイディア

配列型の例から推測できるかもしれないが，実は，配列への要素の代入をしない限り，共変部分型を許しても安全なのである．そこで，代入と共変部分型をトレードオフの関係にあると考え，配列型として

- 要素の読み出し・要素の代入・new を含むすべての操作を許すが，共変部分型は許さないもの（仮に T[] と書く）
- 共変部分型や要素読み出しを許すが，代入や new を許さないもの（仮に T[+] と書く）

の 2 種類を用意するという解決策が考えられる．そして，2 種類の型の間に T[] は T[+] の部分型であるという規則を設ける（これは，読み書き可能な配列を読み出し専用のものとみなすことに相当する）．こうすると，共変部分型が許されない T[] 型の変数は T[] を new したオブジェクトしか指さないことが予想されるので，T 型の要素を代入することは安全そうである．また，T[+] 型の変数は T の部分型を要素とする配列を指しており，正確な要素型が分からないので，代入が安全であるとは限らない（一方，読み出した要素を T 型とみなすことは安全である）．実際，上で見た例をそのまま，新しい規則に沿って型検査すると，Object[] が共変部分型を許さないので，os = ss; の部分で型エラーになる．一方，共変部分型を用いるように

```
String[] ss = ...;
Object[+] os = ss;  // covariant subtyping
os[0] = new Integer(100);  // type error !
int i = ss[0].length();
```

と os の型を Object[] ではなく，Object[+] に変更したとしても，3 行目の要素代入で型エラーが起こることになり，危険なプログラムとしてコンパイル時にはじくことができる．

ワイルドカード型はこれを一般化した機構で，共変部分型を許す代わりに，ある種の危険なメソッド呼び出し・フィールドアクセスを禁止するような型を導入する．具体的には，総称クラス C<X> に対し，今までと同様の C<〈型〉> というパラメトリック型に加え，C<? extends〈型〉> という形式のワイルドカード型表現を導入する．たとえば，List<? extends Object> や List<? extends String> などはワイルドカード型である．

この型は直感的には「Object（String）の部分型のクラスを要素とする何かのリスト」もしくは，型を集合と思うと，「Object の部分型の任意のクラス C に対する List<C> のインスタンスを要素として含む集合」と理解することが可能である．この直感（プラス，部分型 = 包含関係）に従うと，ワイルドカード型 C<? extends〈型〉> は，〈型〉の部分型を型引数とするパラメトリック型を部分型として持ち，かつ，C<? extends〈型〉> には（extends の後の型に関して）共変部分型が許される．たとえば，

```
List<Integer> l0 = ...;
List<? extends Integer> l1 = l0;
  // List<Integer> is a subtype
  //  of List<? extends Integer>
List<? extends Number> l2 = l1;
  // List<? extends Integer> is a subtype
  //  of List<? extends Number>
```

という代入を行うことができる．一方，ワイルドカード型の式に対しては，そのクラスの一部のメソッド・フィールドにしかアクセスできない．正確な規則はここでは述べないが，メソッドの引数の型に型変数が含まれているようなメソッドの呼び出しやフィールドへの書きこみは（大抵）できない．たとえば，引数のない length メソッドを呼び出したり，head，tail フィールドを読み出すことはできるが，head，tail フィールドへの書きこみをすることはできない．これは配列の例と同じで，List<? extends Number> という型は，Number の部分型の「なにか」を要素とするリストを指しているので，その「なにか」の正体が分からない以上，head への代入などはできないのである．

☆4 配列アクセスの場合，配列添字が範囲内にあるかどうかも静的には保証されないが，これはまた別の問題である．

応用例

配列の例からすると，何だか，また新しい形式の型が加わって，型付けできないプログラムが増えただけに見えるかもしれない．ここでは，ワイルドカード型を使うことでよりメソッドの適用範囲を広げることができる，（型付けできるプログラムが増える）という例をみる．ワイルドカード型はパラメトリック型に比べ，より多くの部分型を許すため，メソッド引数の使われ方が限定的である場合には，ワイルドカード型を使った方が，よりいろいろな引数が許されるのである．

具体例として，リストを結合する以下のメソッドを考えてみる．

```
class List<X> {
  X head;  List<X> tail;
  public List(X h, List<X> t) { head = h; tail = t; }
  ...
  public List<X> prepend(List<X> l) {
    if (l == null) return this;
    else return new List<X>(l.head, prepend(l.tail));
  }
}
```

このprependメソッドはl1.prepend(l2)と呼び出されると，l1の前にl2の要素を結合したリストを返すメソッドである．しかし，引数の型がList<X>となっていることから分かるように，要素型が同じリスト同士の結合しか行うことができない．たとえば，List<Number>にList<Integer>を結合しようと

```
List<Number> ln = ...;
List<Integer> li = ...;
List<Number> l = ln.prepend(li);
```

しても，引数の型List<Number>とliの型List<Integer>が部分型関係になく，型エラーになってしまう．しかし，メソッドの動作から考えると，これは別に問題なさそうである．

ワイルドカード型はこの状況を改善させることができる．実は，よく見ると，メソッド定義中ではパラメータlに対し，head，tailの読み出ししか行っていないので，lに対する型としては，List<X>ではなく，List<? extends X>としても，メソッド本体を型付けすることができる．すると，lnに対するprependの引数型はList<? extends Number>ということになり，共変部分型によりList<Number>のインスタンスだけではなく，List<Integer>やList<Float>のインスタンスも渡せるようになる．

おわりに

本稿では，Java 1.5で導入された総称クラスとワイルドカード型の機構（のごく基本的な部分）について簡単な紹介を行った．

総称クラスはクラス定義の型情報を型変数を使って抽象化するための機構で，汎用的なデータ構造などを（型キャストに頼ることなく）型安全に記述しつつ，コードの再利用性を高めることができる．総称クラスはC++のテンプレートも同じ目的の機構であるが，C++とは違い，型変数を具体化することなく総称クラスを型検査の対象とできることは大きな長所である．総称クラスはML，Haskellなどの関数型言語における多相型関数にヒントを得たもので Eiffel などでは古くから導入されている[7]が，Javaにおける設計に関しては，古くからのコード・仮想機械との互換性をいかに保つかということに力点が置かれている☆5．

ワイルドカード型は，以前から指摘されてきていた共変部分型をいかに総称クラスと組み合わせるか，という問題に対する1つの解答であろう．Eiffelでは，Javaの配列型のように，すべてのパラメトリック型を共変であると見なしていたため，型安全性が失われていた[3]（Meyerは，複雑な回避策を提案している[8]が，今のところそれが正しい方法なのかは確かめられておらず，また実装もされていないようである）．また，別のアプローチとして，共変部分型を許したい場合には，総称クラス定義の中にそこから派生するパラメトリック型が共変かそうでないかを宣言させ，クラス定義内に危険なメソッドが定義されていないかを検査するという方法が提案されていた．ワイルドカード型は，クラスを定義する時ではなく，使う時に共変かそうでないかを選択し，必要なアクセス制限を課すことにしている．そのため，クラス定義には制限がなく，使う側がより柔軟な選択をすることができる．このアイディアは，BETA言語のvirtual class機構[6]を応用したvirtual type機構[9),10)]や，モジュールの型理論によく使われる**特称型**（*existential type*）にヒントを得て，筆者らが提案したものである[5]．

本稿では共変部分型に注目して紹介したが，Java 1.5には**反変部分型**（*contravariant subtyping*）を許す，List<? super Number>といったワイルドカード型表現もある．これは，Numberを部分型とする「なにか」を要素とするリストを指し，

☆5 その分，複雑な点もある．

```
List<Number> l3 = ...
List<? super Number> l4 = l3;
List<? super Integer> l5 = l4;
```

というように，共変とは逆の部分型関係が成立するものである．また，共変とはアクセス制限も逆で，書きこみは自由だが，要素の読み出しは（禁止はされていないが）Object を返すものとして扱われる．

また，本稿では紹介しきれなかった総称クラス・ワイルドカード型に関する付帯的な機能についてのより詳しい仕様・解説に関しては，Java 1.5 の総称クラスの元となった GJ に関する文献 2），1）や文献 11）を参考にしてもらいたい．また，C# などでも総称クラスの導入が検討されているが，違った言語上の制約から細かいところで異なる設計になっている．

型や型システムというのは複雑なわりに，うまく動くはずのプログラムに対し型エラーを出したり，型キャストの失敗など防いでほしい誤りを検出できなかったり，と柔軟性があまりないものと思われがち☆6 であるが，実は，ある種簡単なプログラムの仕様・値の近似を表しているものであるし（たとえ静的型のない言語でプログラムしていたとしても），プログラマが意識せざるを得ないものであるはずである．問題は，型表現の情報量の多さと型システムの複雑さのトレードオフである．もし，ワイルドカード型のようにそれなりに複雑な型機構が，Java のように広く使われている言語に採用されることにより，広く受け入れられるとしたら，プログラミング言語の型システムを研究している筆者のような者にとっては嬉しい限りである．

謝辞　本稿の執筆にあたり有用なコメントをくださった，東京大学大学院総合文化研究科玉井研究室を中心とする組木プロジェクトの皆様に感謝いたします．

参考文献

1）Bracha, G., Odersky, M., Stoutamire, D. and Wadler, P.: GJ: Extending the Java Programming Language with Type Parameters, Manuscript（1998）. Available through http://homepages. inf.ed.ac.uk/wadler/gj/Documents/
2）Bracha, G., Odersky, M., Stoutamire, D. and Wadler, P.: Making the Future Safe for the Past: Adding Genericity to the Java Programming Language, Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications（OOPSLA '98）, Vancouver, BC, pp.183-200（1998）.
3）Cook, W.: A Proposal for Making Eiffel Type-safe, Proceedings of the 3rd European Conference on Object-Oriented Programming（ECOOP '89）, Nottingham, England, Cambridge University Press, pp.57-70（1989）.
4）Igarashi, A., Pierce, B. C. and Wadler, P.: Featherweight Java: A Minimal Core Calculus for Java and GJ, ACM Transactions on Programming Languages and Systems, Vol.23, No.3, pp.396-450（2001）. A Preliminary Summary Appeared in Proc. of OOPSLA '99, pp.132-146, Denver, CO（Oct. 1999）.
5）Igarashi, A. and Viroli, M.: On Variance-Based Subtyping For Parametric Types, Proceedings of the 16th European Conference on Object-Oriented Programming（ECOOP2002）（Magnusson, B.（ed.））, Lecture Notes on Computer Science, Vol.2374, Málaga, Spain, Springer-Verlag, pp.441-469（2002）.
6）Madsen, O. L. and Møller-Pedersen, B.: Virtual Classes: A Powerful Mechanism in Object-Oriented Programming, Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications（OOPSLA '89）, New Orleans, LA, pp.397-406（1989）.
7）Meyer, B.: Genericity versus Inheritance, Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications（OOPSLA '86）, pp.391-405（1986）.
8）Meyer, B.: Static Typing, Addendum to Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp.20-29（1995）.
9）Thorup, K. K.: Genericity in Java with Virtual Types, Proceedings of the 11th European Conference on Object-Oriented Programming（ECOOP '97）, LNCS, Vol.1241, Jyväskylä, Finland, Springer-Verlag, pp.444-471（1997）.
10）Thorup, K. K. and Torgersen, M.: Unifying Genericity: Combining the Benefits of Virtual Types and Parameterized Classes, Proceedings of the 13th European Conference on Object-Oriented Programming（ECOOP '99）, LNCS, Vol.1628, Lisbon, Portugal, Springer-Verlag, pp.186-204（1999）.
11）Torgersen, M., Hansen, C. P., Ernst, E., von der Ahé, P., Bracha, G. and Gafter, N.: Adding Wildcards to the Java Programming Language, Proceedings of the ACM Symposium on Applied Computing（SAC '04）, pp.1289-1296（2004）.

（平成 16 年 5 月 14 日受付）

☆6 という気がするのは筆者の気のせい・僻みだろうか….