ソフトウェア基礎論配布資料 (6)
オブジェクト計算: Featherweight Java

五十嵐 淳
京都大学 大学院情報学研究科知能情報学専攻
e-mail: igarashi@kuis.kyoto-u.ac.jp

平成17 年12 月6 日

1 オブジェクト計算—概要

λ 計算が関数の概念を中心に備えた計算体系であり，Lisp, Scheme, ML などの関数型プログラミング言語の中核部分のモデルであるのと同様に，オブジェクト計算は (Smalltalk, Java, C++ などオブジェクト指向言語における) オブジェクトの概念をモデル化した計算体系である．オブジェクト計算には Abadi と Cardelli による ς 計算などがあるが，ここでは，オブジェクトだけでなくクラスもモデル化しており，実際の言語により近い，Featherweight Java (FJ) をとりあげる．

1.1 オブジェクト

オブジェクトとは，(隠蔽された) 内部状態と，内部状態を使って計算をするメソッドと呼ばれる手続/関数 (群) から構成されるようなデータである．例えば，二次元点オブジェクトは，内部状態として座標値の組 x, y と，以下のメソッド

- x 座標値を知るためのメソッド *getx*
- y 座標値を知るためのメソッド *gety*
- x 座標値を変更するメソッド *putx*
- y 座標値を変更するメソッド *puty*
- 移動量 dx, dy を引数として受けとって点を $x + dx, y + dy$ に移動させるメソッド *move*

から構成することができる．また，究極的には数値などの基本的なデータでさえも，足し算やかけ算メソッドを持つオブジェクトと考えることもできる．内部状態は通常，名前のついたデータの組 (例えば x という名前でアクセスされる数値と y という名前でアクセスされる数値) の集まりで表わされ，その各々を **フィールド**(*field*) (もしくはインスタンス変数) と呼ぶ．フィールドやメソッドについた名前をラベルということもある．

計算は，オブジェクトのフィールドをアクセスしたりメソッドを呼び出すことで発生する．オブジェクトの重要な機能のひとつとして「自分自身のメソッドを呼び出せる」ことが挙げられる．例えば，*move* メソッドは，自分自身の *putx*,*puty* メソッドを呼び出して，実装することができる．これは，this (Java, C++) や self と呼ばれる機構として実際のプログラミング言語にも備 わっている．

1.2 クラス

多くのオブジェクト指向言語では，**クラス**(*class*) という機構によって，類似する構造を持つオブジェクトを簡潔に記述できるようになっている．例えば，点オブジェクトは，状態である座標値こそ各点毎に違うものの，メソッドに関しては同一であることが想定されるため，異なる初期座標値から異なる点オブジェクトを生成できる仕組みがあると便利である．クラスは，このような共通の構造を記述するためのものである．クラスには通常，

- フィールド (の名前) の宣言
- メソッド群の定義
- オブジェクトを生成する際の状態の初期化処理 (**コンストラクタ**(*constructor*)) の定義

が記述される．とりあえず，例の記述のために Java 風の言語を用いると，(一次元) 点オブジェクトのクラス `Point` は

```
class Point {
  field x;
  Point(x') { this.x = x'; }
  method getx() { return this.x; }
  method setx(x') { return new Point(x'); }
  method move(dx) { return this.setx(this.getx() + dx); }
}
```

と記述できる．ここでは `this` を自分自身を表すための特別な変数として用いている．また，`new Point(...)` のような式で，`Point` クラスの新しいオブジェクトが生成される．この時，コンストラクタ (`Point(x') {...}`) が呼ばれて，フィールドの初期化 `this.x = x';` が行なわれる．メソッド名の直後の括弧はメソッドの引数の宣言である．

このようなクラスを中心としてプログラムを記述するオブジェクト指向言語を**クラスに基づく言語**(*class-based (object-oriented) language*) ということがある．

1.3 サブクラスと継承

またクラスに基づく言語では，類似するクラスの記述，特に，あるクラスが別のクラスの機能を拡張して (より具体的には既存のクラスのフィールド宣言・メソッド定義を丸ごと複製して) 得られるような場合の記述を助ける仕組みが備わっている．これが**継承**(*inheritance*) と呼ばれる仕組みである．例えば，色付きの点オブジェクトのためのクラス CPoint は，継承を使って以下のように定義することができる．

```
class CPoint extends Point {
  field c;
  CPoint(x',c') { super(x'); this.c = c'; }
  method getc() { return this.c; }
  method setx(x') { return new CPoint(x',White); }
}
```

このクラス定義のポイントは以下の通りである．

- extends Point と書くことで，Point クラスのフィールド/メソッド/コンストラクタ定義を継承し，CPoint オブジェクトは x フィールド, getx, move メソッドを持つことになる．CPoint は Point の**サブクラス・派生クラス**(*subclass, derived class*) である，といったりする．逆に Point は CPoint の**親クラス**(*superclass*) である，ということがある．

- c, getc は追加されたフィールド/メソッドである．

- 追加されたフィールドの初期化コードがコンストラクタに記述されている．super(x') は，継承した x フィールドの初期化のために親クラスのコンストラクタを呼び出すための，特別な記法である．

- setx は親クラスの定義を使わずに独自のものをここで定義している．これをメソッド定義の**上書き**(*override*) と呼ぶ．CPoint クラスのオブジェクトに対して setx を呼んだ時には新しい定義が呼び出されるのは勿論，継承された move メソッドを呼んだ場合にも，その内部から呼び出される setx は新しい定義になる．

このようにして，クラスを拡張して新しい機能を追加 (場合によっては上書きによる変更) をして，新しいクラスを定義することができる．

1.4 動的なクラス検査

Java, C# といったクラスに基づく言語では，オブジェクトが生成されたクラスを検査するための機構が備わっている．代表的なものは，以下のふたつである．

- ⟨式⟩ `instanceOf C`: ⟨式⟩の評価結果であるオブジェクトのクラスが `C` もしくはそのサブクラスであれば `true` を，そうでなければ `false` を返す．
- `(C)`⟨式⟩: ⟨式⟩の評価結果である オブジェクトのクラスが `C` もしくはそのサブクラスであるかを検査する．成功すれば，そのオブジェクト自身が式全体の評価結果であり，失敗すればプログラム実行を中断する．(実際には例外を発生する．)

2 形無し Featherweight Java

Featherweight Java [1, 参考図書 1 の 19 章] (以下 FJ) は，上のようなクラス機構のエッセンスを Java 風の文法で形式化した計算体系である．元々の体系は最初から型システムが備わっているが，ここではまず型無しの体系 (Untyped FJ, UFJ) から始めて，型を加えていくことにする．

2.1 文法

B, C, D はクラス名を表すメタ変数，f, m, x はそれぞれフィールド・メソッド・メソッドパラメータの名前を表すメタ変数，L, K, M, e はそれぞれクラス，コンストラクタ，メソッド，式を表すメタ変数，v, w は値を表すメタ変数である．上付き線は (長さ 0 以上の) 列を表す．列に現れる名前には重複を認めない．(例えば $\overline{f}$; は，$f_1; \cdots f_n;$ の略であり，フィールド名 f_i に重複を認めない．)

$$
\begin{array}{lcl}
L & ::= & \texttt{class } C \texttt{ extends } C \texttt{ \{} \overline{f}\texttt{; } K \texttt{ } \overline{M}\texttt{\}} \\
K & ::= & C\texttt{(}\overline{f}\texttt{)\{super(}\overline{f}\texttt{); this.}\overline{f}\texttt{=}\overline{f}\texttt{;\}} \\
M & ::= & m\texttt{(}\overline{x}\texttt{)\{ return } e\texttt{; \}} \\
e & ::= & x \mid \texttt{this} \mid e.f \mid e.m\texttt{(}\overline{e}\texttt{)} \mid \texttt{new } C\texttt{(}\overline{e}\texttt{)} \mid \texttt{(}C\texttt{)}e \\
v & ::= & \texttt{new } C\texttt{(}\overline{v}\texttt{)}
\end{array}
$$

プログラムは，クラスの集合と式の組 $(\overline{L}, e)$ である．この式は初めに実行される，いわゆる `main` メソッドに相当する．

式 e_0 中の変数 $x_1, \ldots, x_n$ をそれぞれ $e_1, \ldots, e_n$ に同時に置き換えた式を $[\overline{x} \mapsto \overline{e}]e_0$ と表記する．厳密には e_0 の構成に関して帰納的に定義するが，λ 項とは違い，変数を宣言するような項がないため定義は非常に単純なものになるため，詳細は省略する．

Java では，全てのクラスの先祖のクラスとして `Object` というクラスが仮定されている．FJ では，`Object` をフィールドもメソッドも何も持たず，クラスの集合にも含まれない特別なクラスとして扱う．

以下，プログラムとして，同一名のクラスは複数存在しない，`Object` は明示的には定義されていない，`extends` 関係は循環していない，プログラムに現われるクラス名は `Object` であるかクラス集合で定義されている，という条件を満たすものだけを考える．

以下で定義される規則は，厳密にいうと，先ず，何らかのクラス集合を固定することで定義されるものだが，表記を単純にするために省略している．

2.2 簡約規則

簡約は，式の二項関係 $e \longrightarrow e'$ として定義するが，そのために必要な補助関数として，クラス名からそのフィールド名を問い合わせるための $\mathit{fields}(C)$ とクラス名とメソッド名からその定義を問い合わせるための $\mathit{mbody}(m, C)$ を以下の規則で定義する．これらの関数がフィールド・メソッドの継承を表現している．

$$\mathit{fields}(\texttt{Object}) = \bullet$$

$$\frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{\{}\overline{f}\texttt{;}\ K\ \overline{M}\texttt{\}} \qquad \mathit{fields}(D) = \overline{g}}{\mathit{fields}(C) = \overline{g}, \overline{f}}$$

$$\frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{\{}\overline{f}\texttt{;}\ K\ \overline{M}\texttt{\}} \qquad m\texttt{(}\overline{x}\texttt{)\{ return}\ e\texttt{; \}} \in \overline{M}}{\mathit{mbody}(m, C) = \lambda\overline{x}.e}$$

$$\frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{\{}\overline{f}\texttt{;}\ K\ \overline{M}\texttt{\}} \qquad m \notin \overline{M} \qquad \mathit{mbody}(m, D) = \lambda\overline{x}.e}{\mathit{mbody}(m, C) = \lambda\overline{x}.e}$$

$m \notin \overline{M}$ は m という名前を持つメソッドが $\overline{M}$ の中にない，ということを示す記法である．

また，クラス C が D を継承している，という関係を $C \leq D$ と表記し，以下の規則で定義する．

$$C \leq C \qquad \frac{C \leq D \qquad D \leq E}{C \leq E} \qquad \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{\{...\}}}{C \leq D}$$

2.2.1 定義: 簡約関係 $e \longrightarrow e'$ を以下の規則で定義する．

$$\frac{\mathit{fields}(C) = \overline{f}}{\texttt{new}\ C\texttt{(}\overline{e}\texttt{)}\,.\,f_i \longrightarrow e_i} \qquad \text{(R-Field)}$$

$$\frac{\mathit{mbody}(m, C) = \lambda\overline{x}.e_0}{\texttt{new}\ C\texttt{(}\overline{d}\texttt{)}\,.\,m\texttt{(}\overline{e}\texttt{)} \longrightarrow [\overline{x} \mapsto \overline{e}, \texttt{this} \mapsto \texttt{new}\ C\texttt{(}\overline{d}\texttt{)}]e_0} \qquad \text{(R-Invk)}$$

$$\frac{C \leq D}{\texttt{(}D\texttt{)new}\ C\texttt{(}\overline{e}\texttt{)} \longrightarrow \texttt{new}\ C\texttt{(}\overline{e}\texttt{)}} \qquad \text{(R-Cast)}$$

$$\frac{e_0 \longrightarrow e_0'}{e_0.f \longrightarrow e_0'.f} \qquad \text{(RC-Field)}$$

$$\frac{e_0 \longrightarrow e_0'}{e_0.m(\overline{e}) \longrightarrow e_0'.m(\overline{e})} \quad \text{(RC-Invk-Recv)}$$

$$\frac{e_i \longrightarrow e_i'}{e_0.m(\ldots,e_i,\ldots) \longrightarrow e_0.m(\ldots,e_i',\ldots)} \quad \text{(RC-Invk-Arg)}$$

$$\frac{e_i \longrightarrow e_i'}{\texttt{new}\ C(\ldots,e_i,\ldots) \longrightarrow \texttt{new}\ C(\ldots,e_i',\ldots)} \quad \text{(RC-New-Arg)}$$

$$\frac{e_0 \longrightarrow e_0'}{(C)e_0 \longrightarrow (C)e_0'} \quad \text{(RC-Cast)}$$

3 オブジェクト指向計算のための型システム

上のような計算体系では,

- 存在しないフィールドの値の読み出し
- 存在しないメソッドの呼び出し，定義とは異なる個数の引数での呼び出し
- 動的クラス検査 $(C)e$ の失敗

などが実行時エラーとして考えられる．ここでは，Java と同様に，このうち最初の二種類のエラーを検知するための型システムを設計する．

3.1 型情報=インターフェース情報=クラス名

λ 計算の項でも述べたように，型は式をその評価結果に関して分類する基準である．ここでは上のようなエラーに着目しているので，ある式の評価結果 (のオブジェクト) が，どんなフィールド，どんなメソッドを持っているかが重要な情報となる．また，λ 計算における関数型と同様に，メソッドが引数としてどのようなものを受け取り，どのようなものを返すのかも重要である．

このような，オブジェクトの持つ，フィールド名とそこに格納される値の型・メソッド名と引数の個数，型，返り値の型情報を**インターフェース**(*interface*) という[1].

Java では，同一クラスから生成されたオブジェクトが同じフィールド・メソッドを持つことに着目し,

クラス名 = 型名

[1] ここでは一般的な用語として導入している．Java 言語などは，これを実現したインターフェースという機能が備わっている．

として使うという設計にしている．ただし，今までのクラス記述だけでは，フィールドに格納される値やメソッド引数・返値に関する型情報がないので，適宜型情報 (つまりクラス名) をフィールド宣言・メソッド定義に付加することになる．

例えば，上記の Point, CPoint クラスは以下のように記述される．(int, Color 型は既に存在していると仮定する．)

```
class Point {
  int x;
  Point(int x') { this.x = x'; }
  int getx() { return this.x; }
  Point setx(int x') { return new Point(x'); }
  Point move(int dx) { return this.setx(this.getx() + dx); }
}
class CPoint extends Point {
  Color c;
  CPoint(int x', Color c') { super(x'); this.c = c'; }
  Color getc() { return this.c; }
  Point setx(int x') { return new CPoint(x',White); }
}
```

3.2 部分型と継承

メソッドの型検査は，λ 計算における λ 抽象のように，パラメータが宣言された型を持つという仮定の下で，return 以下の式を型検査することで行なう．ここで this の型としては，そのメソッドが定義されているクラスが仮定される．しかし，継承機構により，あるクラスのメソッド内の this は，そのクラスのオブジェクトではなく，そのサブクラスのオブジェクトを指している可能性が発生するため，型安全性を保証するためには，CPoint オブジェクトは Point オブジェクトを**代用可能**($substitutable$)— あるオブジェクトが，別のオブジェクトの代わりに使われても「安全」(余計なエラーを発生させない)—ということを考慮する必要がある．代用可能であれば，例えば，Point 型を引数とするメソッドには，CPoint 型の式を実引数として渡すプログラムを記述することができる．

この代用可能性を捉えたのが**部分型**($subtype$) の概念である．部分型はふたつの型の間に成立する関係で,「型 A が型 B の部分型」ならば「型 A に属するオブジェクトは型 B に属するオブジェクトを代用可能」という性質が期待される．では，ふたつの型はどのような場合に部分型関係にあるのだろうか？

直感的には，例えば，ふたつの型が表すインターフェースが包含関係にあれば部分型としてみなしてよさそうである．これを単純化して，Java では，継承関係をそのまま部分型関係として読みかえる，という設計にしている．この読みかえがうまく働くために，メソッド定義の上書き時には，引数の数はもちろん，引数・返り値の型が一致する定義で上書きしなくてはならない，という規則を設けている．

3.3 動的クラス検査と型変換

Java の動的クラス検査式 $(C)e$ は，e の型が何であろうとも[2]，$(C)e$ の型を C とみなせるため，型変換 (型キャスト) とも呼ばれる．このような規則でよい理由は，$(C)e$ の実行が成功するのは e の評価結果が C もしくはサブクラスのオブジェクトの場合だけだからである．

型変換は，部分型によって失われた型情報を取り戻すために使われることが多い．

4 (型付き) Featherweight Java

4.1 文法

以下で $\overline{C}\ \overline{f}$; は，$C_1\ f_1; \cdots C_n\ f_n$; の略であり，フィールド名 f_i にのみ重複を認めない．

$$
\begin{array}{lcl}
L & ::= & \texttt{class}\ C\ \texttt{extends}\ C\ \texttt{\{}\overline{C}\ \overline{f}\texttt{;}\ K\ \overline{M}\texttt{\}} \\
K & ::= & C\texttt{(}\overline{C}\ \overline{f}\texttt{)\{super(}\overline{f}\texttt{);}\ \texttt{this.}\overline{f}\texttt{=}\overline{f}\texttt{;\}} \\
M & ::= & C\ m\texttt{(}\overline{C}\ \overline{x}\texttt{)\{}\ \texttt{return}\ e\texttt{;}\ \texttt{\}} \\
e & ::= & x \mid \texttt{this} \mid e.f \mid e.m\texttt{(}\overline{e}\texttt{)} \mid \texttt{new}\ C\texttt{(}\overline{e}\texttt{)} \\
v & ::= & \texttt{new}\ C\texttt{(}\overline{v}\texttt{)}
\end{array}
$$

プログラムを構成するクラスの集合に関する制約は形無しの場合と同様である．型付き FJ のクラスは，そのまま Java のクラス定義としても有効である．(つまり，FJ は Java のサブセット言語である．)

4.2 簡約規則

簡約は，形無しの場合と全く同様に定義できるため，詳細な定義は $\mathit{fields}(C)$ を除き省略する．$\mathit{fields}(C)$ は，ここではクラス C のオブジェクトが持つフィールド名をその型とともに列挙する．

$$\mathit{fields}(\texttt{Object}) = \bullet$$

$$\frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{\{}\overline{C}\ \overline{f}\texttt{;}\ K\ \overline{M}\texttt{\}} \qquad \mathit{fields}(D) = \overline{D}\ \overline{g}}{\mathit{fields}(C) = \overline{D}\ \overline{g}, \overline{C}\ \overline{f}}$$

[2]実際の Java では e の型は C と部分型関係 (どちらがどちらの部分型でもよい) にあることを要請している．

4.3 型システム

既に述べたように，型としてクラス名を用い，継承関係を部分型関係 $C \leq D$ として用いる．また，型から，それが属するオブジェクトが持つ，メソッドの型を問い合わせる関数 $mtype(m, C)$ を定義する．(フィールドの型情報に関しては既に定義した $fields(C)$ で得られる．)

$$\frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{\{}\overline{C}\ \overline{f}\texttt{;}\ K\ \overline{M}\texttt{\}} \qquad B_0\ m\texttt{(}\overline{B}\ \overline{x}\texttt{)\{ return}\ e\texttt{; \}} \in \overline{M}}{mtype(m, C) = \overline{B} \to B_0}$$

$$\frac{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{\{}\overline{C}\ \overline{f}\texttt{;}\ K\ \overline{M}\texttt{\}} \qquad m \notin \overline{M} \qquad mtype(m, D) = \overline{B} \to B_0}{mtype(m, C) = \overline{B} \to B_0}$$

型環境 Γ は変数と型の組の (変数名に重複のない) 列である．式・メソッド・クラスに関して型付け関係はそれぞれ $\Gamma \vdash e \in C$, M ok in C, L ok と表記する．型付け規則は以下の通り．

$$\Gamma, x \in C \vdash x \in C \tag{T-Var}$$

$$\frac{\Gamma \vdash x \in C \qquad x \neq y}{\Gamma, y \in D \vdash x \in C} \tag{T-Weak}$$

$$\frac{\Gamma \vdash e_0 \in C_0 \qquad fields(C_0) = \overline{C}\ \overline{f}}{\Gamma \vdash e_0.f_i \in C_i} \tag{T-Field}$$

$$\frac{\Gamma \vdash e_0 \in C_0 \qquad mtype(m, C_0) = \overline{D} \to C \qquad \Gamma \vdash \overline{e} \in \overline{C} \qquad \overline{C} \leq \overline{D}}{\Gamma \vdash e_0.m\texttt{(}\overline{e}\texttt{)} \in C} \tag{T-Invk}$$

$$\frac{fields(C) = \overline{D}\ \overline{f} \qquad \Gamma \vdash \overline{e} \in \overline{C} \qquad \overline{C} \leq \overline{D}}{\Gamma \vdash \texttt{new}\ C\texttt{(}\overline{e}\texttt{)} \in C} \tag{T-New}$$

$$\frac{\Gamma \vdash e \in C}{\Gamma \vdash \texttt{(}D\texttt{)}e \in D} \tag{T-Cast}$$

$$\frac{\begin{array}{c}\overline{x} \in \overline{C}, \texttt{this} \in C \vdash e_0 \in E_0 \qquad E_0 \leq C_0 \\ \texttt{class}\ C\ \texttt{extends}\ D\ \texttt{\{...\}} \\ \text{if } mtype(m, D) = \overline{D} \to D_0, \text{ then } \overline{C} = \overline{D} \text{ and } C_0 = D_0\end{array}}{C_0\ m\texttt{(}\overline{C}\ \overline{x}\texttt{)\{ return}\ e_0\texttt{; \}} \text{ ok in } C} \tag{T-Method}$$

$$\frac{K = C\texttt{(}\overline{D}\ \overline{g}\texttt{, }\overline{C}\ \overline{f}\texttt{)\{super(}\overline{g}\texttt{); this.}\overline{f}\texttt{=}\overline{f}\texttt{;\}} \qquad fields(D) = \overline{D}\ \overline{g} \qquad \overline{M} \text{ ok in } C}{\texttt{class}\ C\ \texttt{extends}\ D\ \texttt{\{}\overline{C}\ \overline{f}\texttt{;}\ K\ \overline{M}\texttt{\}} \text{ ok}} \tag{T-Class}$$

メソッド呼び出し・オブジェクト生成時に，部分型関係が用いられることに着目したい．規則 T-Method の最後の前提条件は，上書きメソッドの型が親クラスのものと同一であることを示している．T-Class のコンストラクタに関する条件で，`new(...)` の引数は，各フィールドの値を示すようにしている．

4.4 型システムの性質

FJ の型システムの安全性は，単純型付き λ 計算と同様に Type Preservation, Progress の両定理によって述べられる．しかし，前述のように，動的クラス検査 (型変換) が成功することについては保証されない．(Progress の文言)

4.4.1 定理 [**Type Preservation**]: プログラム $(\overline{L}, e)$ に対し，$\overline{L}$ ok かつ $\Gamma \vdash e \in C$ かつ $e \longrightarrow e'$ ならば，ある D が存在して $\Gamma \vdash e' \in D$ かつ $D \leq C$ である．

4.4.2 定理 [**Progress**]: プログラム $(\overline{L}, e)$ に対し，$\overline{L}$ ok かつ $\bullet \vdash e \in C$ かつ e が値でなければ，(D)`new` $E(\overline{e})$ かつ $E \not\leq D$ なる e の部分項が存在するか，ある項 e' が存在して $e \longrightarrow e'$ である．

5 プログラミング in FJ

5.1 オブジェクト指向自然数

```
class Nat {
  /* abstract methods */
  Nat add(Nat x) { return this.add(x); }
  Nat mult(Nat x) { return this.mult(x); }
}
class Zero extends Nat {
  Nat add(Nat x) { return x; }
  Nat mul(Nat x) { return this; }
}
class Succ extends Nat {
  Nat pred;
  Nat add(Nat x) { return new Succ(this.pred.add(x)); }
  Nat mult(Nat x) { return x.add(this.pred.mult(x)); }
}

new Succ(new Succ(new Zero())).add(new Succ(new Zero()))
```

参考文献

[1] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, Vol. 23, No. 3, pp. 396–450, May 2001.