Union 型を導入したオブジェクト指向計算体系

Class-based Object-Oriented Caluculus with Union Types

柳楽 秀士 † 五十嵐 淳 †

Hideshi Nagira Atsushi Igarashi

† 京都大学 大学院情報学研究科

Graduate School of Informatics, Kyoto University

{nagira,igarashi}@kuis.kyoto-u.ac.jp

本稿では，union 型をクラスに基づくオブジェクト指向言語へ導入し，その基礎理論について報告する．union 型 A∨B は，型 A と型 B のそれぞれのオブジェクトの和集合を表し，A, B の共通の supertype のうち最小のものとして扱われる．union 型 A∨B に対する操作として，実行時型検査の一般化として導入する case 構文で場合わけ処理が行える他，A と B が同名のメンバを持っている時には，場合わけを行わずに，直接そのメンバにアクセスすることができる．これにより，union 型は異なる型のオブジェクトを一つの型に混ぜて使うためのより柔軟で安全な手段を提供する．このような言語機構のの安全性を示すために，Igarashi, Pierce, Wadler によって提案された計算体系 FJ に union 型を導入した体系 FJ∨ の構文，型付け規則，操作的意味の形式的定義を与え，型健全性を証明する．

1 はじめに

多くのプログラミング言語では，異なる種類のデータを混ぜて扱うための仕組みが用意されている．Pascal のヴァリアント・レコード，ML の datatype，C の共用体などがその例である．Java などの，クラスに基づくオブジェクト指向言語においては，継承・部分型の関係を利用することができる．クラス C をクラス D を継承して得た時，型 C は型 D の部分型であり，C のオブジェクトは型 C だけでなく型 D にも属するので，一つの親クラスから複数のクラスを派生させることで，親クラスの型に異なるオブジェクトを混ぜることができる．以下は，A, B のオブジェクトを要素とする List を Object 型を使って実現している Java 風プログラム例である．

```
class A extends Object { void m() { ... } }
class B extends Object { void n() { ... } }

class List{
  Object car; List cdr;
  List(Object car,List cdr){
    this.car=car; this.cdr=cdr;
} }

List l = new List(new A(),
                  new List(new B(),null));
```

上の例において A, B は Object を継承しているので，List の car フィールドの値として与えることができる．l.car の値の依存する処理を記述する場合，以下のように，まず型によって場合分けをして，その後，型変換 (typecast) をして固有の処理を行うことになる．

```
if (l.car instanceof A){
       ((A)(l.car)).m(); }
else { ((B)(l.car)).n(); }
```

この場合，プログラマにとって l が A または B を要素とするリストであることは明らかであり，必ず型 A のメソッド m または B のメソッド n が実行される．しかし，例えば，リスト l がメソッドパラメータである場合には，A，B 以外にも Object として扱える型が存在するため，型の情報だけでは l.car の値が A または B のみであることは保証できない．そのため，もし l.car が A，B と無関係な型であった場合，(B)(l.car) の計算が失敗してエラーとなってしまう (例外が発生する)．

そこで，A と B の共通のインターフェース AorB を用意することを考える．

```
class A implements AorB { void m(){ ... } }
class B implements AorB { void n(){ ... } }
class List{ AorB car; ... }

List l = new List(new A,
                  new List(new B,null));

if (l.car instanceof A) {
       ((A)(l.car)).m(); }
else { ((B)(l.car)).n(); }
```

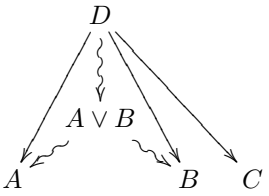

図 1: Union 型による部分型関係

このようなプログラムでは，AorB に A, B 以外に子クラスがなければ，l に関する型情報だけで，メソッド m または n が実行されることがわかる．しかし，一般的には，あるクラスの子クラスの数などを制限するのは難しく，また，この方法は A, B がプログラマが変更できないライブラリクラスである場合には適用できない．このように，クラス定義によって得られる型だけでは A と B のみをまとめて扱えるような型を実現するのは簡単ではない．

そこで，本稿では union 型という概念をクラスに基づくオブジェクト指向言語に導入し，その型システムについて研究する．union 型は A∨B と記述され，直感的には型 A または型 B に属するオブジェクトの和集合を表現する．A∨B は，Java のインターフェースと同様にインスタンスを生成することはできない．部分型の関係としては，A と B を部分型として持つ型の中で最小となるような型として定義される．具体的には図 1 のような関係となる．ここで，$D \to A$ は A が D を継承したことで生じた部分型関係を示し，$D \rightsquigarrow A \vee B$ は union 型の導入によって生じる部分型関係を示す．以降，型 T が型 U の部分型 (subtype) であることを T<:U と書き，逆に U は T の supertype であるという．図より A<:A∨B，B<:A∨B，かつ A∨B<:D であり，A∨B は A, B 共通の supertype の中で (<: の順序に関して) 最小のものであることがわかる．しかし C<:A∨B や D<:A∨B は成り立たない．

union 型の式を操作するために，実行時型検査と型変換とを同時に行う構文として case 構文を導入する．例えば，上の例は以下のように書くことができる．

```
class List{ A∨B car; ... }

List l = new List(new A(),
                  new List(new B(),null));
case l.car of (A x)x.m();
            | (B y)y.n();
```

A<:A∨B , B<:A∨B より，要素として A と B を混ぜたようなリストは A∨B のリストとして定義できる．上の例の case 文においては l.car の計算結果が A 型だった場合 x にその結果が代入されて，x.m() が実行される．そうでない場合 y に代入されて y.m() が実行される．図 1 から，A∨B として扱える型としては A，B，また A∨B のどれかであるが，A∨B の値を生成するコンストラクタが無いため l.car の計算結果は A，B のどちらかのインスタンスであることが型情報のみから保証できる．

また，union で結合される二つのクラスが同名のメンバを持っている場合には，case 構文を使わずにそのメンバにアクセスすることを許す．例えば，

```
class A extends AorB { Integer m(){ ... } }
class B extends AorB { String m(){ ... } }

List l = ...;
```

のように，A, B が同名 (かつ引数の数が同じ) メソッドを持つ場合，

```
Integer∨String x = l.car.m();
```

と呼び出すことができる．それぞれの返り値の型が異なっているので，返り値の型として再び union 型が与えられる．

本論文では，このような union 型を持つ言語の安全性を示すために，既存の計算体系である FJ [6] に union 型と case 構文を導入した体系 FJ∨ を設計し，その形式的定義と型健全性の証明を与える．2 章で FJ∨ の形式的な定義を与え，3 章でその計算例を示す．4 章でその定義から得られる型健全性に関する定理について述べる．5 章で関連研究について議論した後，6 章において結論及び今後の課題について検討する．

2 FJ∨ の定義

FJ [6] は，Java などのクラスに基づくオブジェクト指向言語をモデル化した計算体系であり，クラスや継承，(virtual) メソッド呼び出し，フィールド，this といった最小限の機能のみをモデル化している． FJ [6] を union 型・case 構文によって拡張した体系となっている．(ただし，FJ の型変換機構は case でほぼ代用できるため除いている．)

2.1 文法

FJ∨ におけるクラスなどの文法を与える．以下で，L はクラス定義，M はメソッド定義，e は式，T は型

を表している．C，D，E はクラス名，T，U，V は型，x, y は変数，f，g はフィールド，m はメソッドの名前，を表すメタ変数とする．

```
L  ::=  class C extends C {T̄ f̄; M̄}
M  ::=  T m(T̄ x̄){ return e; }
e  ::=  x | e.f | e.m(ē) | new C(ē)
    |   (case e of (T x) e | (T x) e)
T  ::=  C | T∨T
```

$\overline{\mathtt{f}}$ のような上線付きのメタ変数は列を表す．例えば，$\overline{\mathtt{f}}$ は $\mathtt{f}_1,\mathtt{f}_2,\ldots,\mathtt{f}_n$ を，$\overline{\mathtt{T}}\ \overline{\mathtt{f}}$ は $\mathtt{T}_1\ \mathtt{f}_1,\ldots,\mathtt{T}_n\ \mathtt{f}_n$ を，$\overline{\mathtt{T}}\ \overline{\mathtt{f}}$; は $\mathtt{T}_1\ \mathtt{f}_1;\cdots\mathtt{T}_n\ \mathtt{f}_n;$ を表す．$n \geq 0$ であり $n = 0$ の場合は空列を表すものとする．また $\#$ を列の要素数を返す関数 $\#(\mathtt{X}_1,\ldots,\mathtt{X}_n) = n$ とする．

FJ にみられるコンストラクタの定義は Java の文法に合わせるための形式的なものなので省略しており，各クラスは，各フィールドの初期値を引数とし，それを各フィールドに代入する自明なコンストラクタをひとつ持つと仮定する．

FJ$\vee$ プログラムはクラスの集合 $\overline{\mathtt{L}}$ と (main メソッドに相当する) 式 e の組で与えられる．以下では，ある与えられたクラスの集合 $\overline{\mathtt{L}}$ を仮定する．さらに，$\overline{\mathtt{L}}$ の継承関係には循環がなく，$\overline{\mathtt{L}}$ に出現するクラス名は (Object を除き) すべて $\overline{\mathtt{L}}$ 内で定義されているものとする．

2.2 部分型，フィールド・メソッド型

部分型関係 S<: T は以下の規則で定義される．

$$\mathtt{T}\mathrel{<:}\mathtt{T} \qquad \frac{\mathtt{T}\mathrel{<:}\mathtt{U} \quad \mathtt{U}\mathrel{<:}\mathtt{V}}{\mathtt{T}\mathrel{<:}\mathtt{V}} \qquad \frac{\mathtt{class\ C\ extends\ D}\{\ldots\}}{\mathtt{C}\mathrel{<:}\mathtt{D}}$$

$$\mathtt{T}\mathrel{<:}\mathtt{T}\vee\mathtt{U} \qquad \mathtt{T}\mathrel{<:}\mathtt{U}\vee\mathtt{T} \qquad \frac{\mathtt{U}\mathrel{<:}\mathtt{T} \quad \mathtt{V}\mathrel{<:}\mathtt{T}}{\mathtt{U}\vee\mathtt{V}\mathrel{<:}\mathtt{T}}$$

最初の 3 つの規則は，部分型関係が継承関係の反射的推移的閉包を含むことを示す．また，下の 3 つの規則によって T$\vee$U は T と U の上限 (最小の共通部分型) となる．これにより部分型関係は反対称的ではなくなることに注意したい．例えば class C extends D {...} であるとき C$\vee$D <: D かつ D <: C$\vee$D である．このようにお互いに部分型関係が成り立つとき二つの型は互換であるという．以下，$\overline{\mathtt{S}}\mathrel{<:}\overline{\mathtt{T}}$ を $\mathtt{S}_1 \mathrel{<:} \mathtt{T}_1, \ldots, \mathtt{S}_n \mathrel{<:} \mathtt{T}_n$ と略す．

次に，型付規則で用いられる，フィールドやメソッドの型・定義情報を問い合わせるための関数群を定義する．*fields*(C) はクラス C およびその先祖のクラスで定義されるフィールドの型と名前の列であり，以下の規則で定義する．

$$\mathit{fields}(\mathtt{Object}) = \cdot$$

$$\frac{\mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\} \quad \mathit{fields}(\mathtt{D}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}}{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}},\overline{\mathtt{U}}\ \overline{\mathtt{g}}}$$

fields(Object) は空列であり，フィールドを持たないことを表す．*ftype*(f, T) は，型 T のフィールド f の型を返す関数として定義される．

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\mathit{ftype}(\mathtt{f}_i,\mathtt{C}) = \mathtt{T}_i} \qquad \frac{\mathit{ftype}(\mathtt{f},\mathtt{T}) = \mathtt{T}' \quad \mathit{ftype}(\mathtt{f},\mathtt{U}) = \mathtt{U}'}{\mathit{ftype}(\mathtt{f},\mathtt{T}\vee\mathtt{U}) = \mathtt{T}'\vee\mathtt{U}'}$$

この規則によって，union 型 T$\vee$U に対しては，T, U が同名のフィールドを持つ場合，そのフィールドにアクセスでき，型はそれぞれの union となる．

mtype(m, T) は型 T のメソッド m の引数と返り値の型を表す．

$$\frac{\mathtt{class\ C\ extends\ D}\{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\} \quad \mathtt{U}_0\ \mathtt{m}(\overline{\mathtt{U}}\ \overline{\mathtt{x}})\{\mathtt{return\ e;}\} \in \overline{\mathtt{M}}}{\mathit{mtype}(\mathtt{m},\mathtt{C}) = \overline{\mathtt{U}}\to\mathtt{U}_0}$$

$$\frac{\mathtt{class\ C\ extends\ D}\{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\} \quad \mathtt{m} \notin \overline{\mathtt{M}} \quad \mathit{mtype}(\mathtt{m},\mathtt{D}) = \overline{\mathtt{U}}\to\mathtt{U}_0}{\mathit{mtype}(\mathtt{m},\mathtt{C}) = \overline{\mathtt{U}}\to\mathtt{U}_0}$$

$$\frac{\mathit{mtype}(\mathtt{m},\mathtt{T}') = \overline{\mathtt{T}}\to\mathtt{T}_0 \quad \mathit{mtype}(\mathtt{m},\mathtt{U}') = \overline{\mathtt{U}}\to\mathtt{U}_0 \quad \overline{\mathtt{T}}\mathrel{<:}\overline{\mathtt{U}} \quad \overline{\mathtt{U}}\mathrel{<:}\overline{\mathtt{T}}}{\mathit{mtype}(\mathtt{m},\mathtt{T}'\vee\mathtt{U}') = \overline{\mathtt{T}}\to\mathtt{T}_0\vee\mathtt{U}_0}$$

3 番目の規則により，union 型 T$\vee$U に対するメソッド起動は，同じ名前のメソッドが T, U に存在し，引数の数と型が互換であるときに許され，返り値の型はそれぞれの union となる．

mbody(m, C) はクラス C におけるメソッド m の仮引数と実行されるプログラムを表す．

$$\frac{\mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\} \quad \mathtt{U}_0\ \mathtt{m}(\overline{\mathtt{U}}\ \overline{\mathtt{x}})\{\mathtt{return\ e;}\} \in \overline{\mathtt{M}}}{\mathit{mbody}(\mathtt{m},\mathtt{C}) = \overline{\mathtt{x}}.\mathtt{e}}$$

$$\frac{\mathtt{class\ C\ extends\ D}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\} \quad \mathtt{m} \notin \overline{\mathtt{M}} \quad \mathit{mbody}(\mathtt{m},\mathtt{D}) = \overline{\mathtt{x}}.\mathtt{e}}{\mathit{mbody}(\mathtt{m},\mathtt{C}) = \overline{\mathtt{x}}.\mathtt{e}}$$

2.3 型付け規則

式に対する型付け判断は $\Gamma \vdash \mathtt{e} : \mathtt{T}$ と書き，Γ の下で e が型 T を持つことを意味する．ここで Γ は変数から型への関数であり，$\overline{\mathtt{x}}{:}\overline{\mathtt{T}}$ の形で表される有限集合である．以下で，$\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}}$ は $\Gamma \vdash \mathtt{e}_1 : \mathtt{T}_1, \ldots, \Gamma \vdash \mathtt{e}_n : \mathtt{T}_n$ の略記である．型付け規則は以下の通りである．

$$\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x})$$

$$\frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \quad \mathit{ftype}(\mathtt{f},\mathtt{T}_0) = \mathtt{T}}{\Gamma \vdash \mathtt{e}_0.\mathtt{f} : \mathtt{T}}$$

$$\frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \quad \mathit{mtype}(\mathtt{m}, \mathtt{T}_0) = \overline{\mathtt{U}} \to \mathtt{U}_0 \quad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}} \quad \overline{\mathtt{T}} <: \overline{\mathtt{U}}}{\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) : \mathtt{U}_0}$$

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{U}}\ \overline{\mathtt{f}} \quad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}} \quad \overline{\mathtt{T}} <: \overline{\mathtt{U}}}{\Gamma \vdash \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}}) : \mathtt{C}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \quad \mathtt{T}_0 <: \mathtt{T'}_1 \vee \mathtt{T'}_2 \\ \Gamma, \mathtt{x}{:}\mathtt{T'}_1 \vdash \mathtt{e}_1 : \mathtt{T}_1 \quad \Gamma, \mathtt{y}{:}\mathtt{T'}_2 \vdash \mathtt{e}_2 : \mathtt{T}_2\end{array}}{\Gamma \vdash \mathtt{case}\ \mathtt{e}_0\ \mathtt{of}\ (\mathtt{T}_1'\ \mathtt{x})\ \mathtt{e}_1\ \mid\ (\mathtt{T}_2'\ \mathtt{y})\ \mathtt{e}_2 : \mathtt{T}_1 \vee \mathtt{T}_2}$$

フィールドアクセス・メソッド起動・インスタンス生成は補助関数に定義情報を問い合わせ，必要な場合実引数との型の整合性を検査する．また，上述のように，union 型からインスタンスを生成することはできない．case 式の規則における $\mathtt{T}_1'$ と $\mathtt{T}_2'$ は，場合わけでマッチするオブジェクトの型を示しているため，対象となる $\mathtt{e}_0$ の型 $\mathtt{T}_0$ はそれらの和の部分型であることが必要となる．

次の規則は，メソッドの型付け規則である．

$$\frac{\begin{array}{c}\overline{\mathtt{x}}{:}\overline{\mathtt{T}}, \mathtt{this}{:}\mathtt{C} \vdash \mathtt{e}{:}\mathtt{V} \quad \mathtt{V} <: \mathtt{T} \\ \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\{\ldots\} \\ \text{if } \mathit{mtype}(\mathtt{m}, \mathtt{D}) = \overline{\mathtt{U}} \to \mathtt{U}, \text{ then } \overline{\mathtt{U}} = \overline{\mathtt{T}} \text{ and } \mathtt{U} = \mathtt{T}\end{array}}{\mathtt{T}\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{\ \mathtt{return}\ \mathtt{e};\ \}\ \mathtt{OK\ IN}\ \mathtt{C}}$$

クラス L 内の全てのメソッドが型付けできるとき，クラスは型付けできるという．

2.4 簡約規則

簡約関係は $\mathtt{e} \longrightarrow \mathtt{e}'$ と表され，e が $\mathtt{e}'$ に 1 ステップで簡約されることを意味する．以下は式の基本的な計算規則である．$[\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{e}'/\mathtt{this}]\mathtt{e}$ は e 中の $\overline{\mathtt{x}}$, this に $\overline{\mathtt{d}}$, $\mathtt{e}'$ を代入した式を示す．代入の定義は省略するが標準的なものである．

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{(\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})).\mathtt{f}_i \longrightarrow \mathtt{e}_i}$$

$$\frac{\mathit{mbody}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{x}}.\mathtt{e}_0}{(\mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})).\mathtt{m}(\overline{\mathtt{d}}) \longrightarrow [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new}\ \mathtt{C}(\overline{\mathtt{e}})/\mathtt{this}]\mathtt{e}_0}$$

$$\frac{\mathtt{C} <: \mathtt{T}}{\begin{array}{c}\mathtt{case\ new\ C}(\overline{\mathtt{d}})\ \mathtt{of}\ (\mathtt{T}\ \mathtt{x})\mathtt{e}_1 \mid (\mathtt{U}\ \mathtt{y})\mathtt{e}_2 \\ \longrightarrow [\mathtt{new\ C}(\overline{\mathtt{d}})/\mathtt{x}]\mathtt{e}_1\end{array}}$$

$$\frac{\mathtt{C} \not<: \mathtt{T} \quad \mathtt{C} <: \mathtt{U}}{\begin{array}{c}\mathtt{case\ new\ C}(\overline{\mathtt{d}})\ \mathtt{of}\ (\mathtt{T}\ \mathtt{x})\mathtt{e}_1 \mid (\mathtt{U}\ \mathtt{y})\mathtt{e}_2 \\ \longrightarrow [\mathtt{new\ C}(\overline{\mathtt{d}})/\mathtt{y}]\mathtt{e}_2\end{array}}$$

部分式に対して計算を行うための規則は省略している．また $\longrightarrow$ 関係の反射的推移的閉包を $\longrightarrow^*$ と表す．

FJ $\vee$ は以上の規則によって定義される．以下では，与えられた全てのクラスが型付けできると仮定する．

3 計算例

本節では FJ$\vee$ の簡単な例を示す．以下の例では，Integer や String などのクラス，整数・文字列定数，任意の型として扱える定数 null を用いる．

```
class C extends Object{}
class A extends C {
  Integer m(){ return new Integer(10); }
}
class B extends C {
  String m(){ return "foo"; }
}
class List extends Object{
  A∨B car; List cdr;
}
```

以下，l = new List(new A(), new List(new B(),null)) とする．また，式の型を e:T として示す．

```
case l.car of (A x) x.m() | (B y) y.m()
  : Integer∨String
→ case new A() of (A x) x.m() | (B y) y.m()
→ new A().m()
→ new Integer(10)
```

この例において e.car は A∨B に型付けされる式であるが，計算した結果は new A() になり，正しく A のメソッド m を呼び出している．そして，最初の式の静的な型 Integer∨String に対し，Integer のオブジェクトが結果として得られている．

```
case l.cdr.car of (A x) x.m() | (B y) y.m()
→→ case new B() of (A x) x.m() | (B y) y.m()
→→ "foo"
```

一方，上の例ではリストの 2 番目の要素は new B() であり，二つ目の節が実行され，正しく B のメソッド m が呼び出されている．

この例は，どちらも同じ名前のメソッドを呼び出しているので，case を使わずに記述することもできる．

```
l.car.m() : Integer∨String
→ new A().m()
→ new Integer(10)
```

4 FJ$\vee$ の性質

本節では，FJ$\vee$ の型健全性について述べる．主定理は簡約前後で型付けが保存されるという Subject Reduction，および，型付けされる式は存在しないメンバにアクセスしていないという Progress であり，型健全性はそれらから導かれる．

定理 1 (Subject Reduction) $\Gamma \vdash \mathtt{e}{:}\mathtt{T}$ かつ $\mathtt{e} \longrightarrow \mathtt{e}'$ ならば，ある $\mathtt{T}'{<:}\mathtt{T}$ について $\Gamma \vdash \mathtt{e}'{:}\mathtt{T}'$ である．

定理 2 (Progress) $\vdash \mathtt{e} : \mathtt{T}$ とするとき，以下の *3* つが成り立つ．

1. e の任意の部分式 $\mathtt{new\ C}(\overline{\mathtt{e}}).\mathtt{f}_i$ に対し，$\mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$ かつ $\mathtt{f}_i \in \overline{\mathtt{f}}$.
2. e の任意の部分式 $\mathtt{new\ C}_0(\overline{\mathtt{e}}).\mathtt{m}(\overline{\mathtt{d}})$ に対し，$\mathit{mbody}(\mathtt{m}, \mathtt{C}_0) = \overline{\mathtt{x}}.\mathtt{e}_0$ かつ $\#(\overline{\mathtt{d}}) = \#(\overline{\mathtt{x}})$.
3. e の任意の部分式 $\mathtt{case\ new\ C}(\overline{\mathtt{e}})\ \mathtt{of\ (T}_1\ \mathtt{x)d}_1\ \mathtt{|\ (T}_2\ \mathtt{y)d}_2$ に対し，$\mathtt{C} <: \mathtt{T}_1$ または $\mathtt{C} <: \mathtt{T}_2$.

最後に，型健全性を述べるために，値の概念を導入する．値 v は $\mathtt{v} ::= \mathtt{new\ C}(\overline{\mathtt{v}})$ と定義する．($\overline{\mathtt{v}}$ は空列である場合もあることに注意．)

定理 3 (FJ∨ Type Soundness) $\vdash \mathtt{e} : \mathtt{T}$ かつ $\mathtt{e} \longrightarrow^* \mathtt{e}'$ かつ $\mathtt{e}'$ が *normal form* であるならば，ある $\mathtt{T}'$，値 v について $\mathtt{e}' = \mathtt{v}$ かつ $\vdash \mathtt{v} : \mathtt{T}'$ かつ $\mathtt{T}' <: \mathtt{T}$.

5 関連研究

ML [7] の datatype は，様々なデータ構造を定義するための言語機構だが，ここでは値の集合 (すなわち型) の和を構成する機能について比較する．datatype 宣言された型の値を構成するには，タグ付け (コンストラクタの適用) を明示的に行う必要があるため，しばしば直和 (disjoint union) 型であると言われる．また，同じ型同士の直和型を構成することができ，同じデータに違ったタグ付けをして case 式で区別することができる．FJ∨ では，元々オブジェクトに付加されているクラス情報をタグと見なすことで，明示的なタグ付けなしに union 型の値を構成することができるが，同じ型同士の和 (T∨T) はそれ自身 (T) と互換になり，同じクラスからのオブジェクトを case 構文で区別することはできない．

タグ付けを伴わない union 型は型付き λ 計算で研究 [1] されてきている．また，近年 XML のような半構造データ処理のための言語機構として union 型が採り入れられ始めている [3, 5]．FJ∨ での，C, D が持つ同名のメンバに C∨D から直接アクセスできる機能は，それらの型システムにおける，union 型のレコード型に対する分配則にヒントを得たものである．Xtatic 言語 [4] は C# に XML 処理のための言語機構を導入した拡張で，union 型を含む正則表現型 [5] が備わっている．ただし FJ∨ とは違い，XML を表す型とオブジェクト型は区別されているため，任意のオブジェクト型の和を考えられるわけではない．

6 おわりに

本稿では，クラスに基づくオブジェクト指向言語の union 型による拡張を議論した．動的型検査と型変換を統合した case 構文や，union 型で組み合わされたクラスの同名メンバへの直接アクセスにより，より柔軟で安全なプログラム記述が可能になった．また，union 型を導入した体系 FJ∨ を形式化し，その型システムが健全であることを示した．

今後の課題としては，union 型の実装が挙げられる．基本的な実装は，case 構文などを動的型検査と型変換で表現することで union 型のない言語に変換できると考えているが，FJ∨ で扱っていない言語機構との組み合わせについてはさらに検討を要する．また，union 型により，型情報をより詳細に扱えるようになったが，本稿で扱った List の例のように，要素型ひとつひとつにつき別のクラスを定義するのはクラス再利用性の観点からも現実的ではない．このため，C++ のテンプレートや GJ [2] に採り入れられている，クラス定義を型情報でパラメータ化するための，パラメトリック多相型との統合が望ましい．

参考文献

[1] Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Intersection and union types: Syntax and semantics. *Information and Computation*, 119(2):202–230, June 1995.

[2] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proc. of OOPSLA'98*, pages 183–200, October 1998.

[3] Peter Buneman and Benjamin Pierce. Union types for semistructured data. In *Proc. of the International Database Programming Languages Workshop*, September 1999.

[4] Vladimir Gapeyev and Benjamin Pierce. Regular object types. In *Proc. of ECOOP2003*, volume 2743 of *Springer LNCS*, pages 151–175, July 2003.

[5] Haruo Hosoya, Jérôme Voullion, and Benjamin Pierce. Regular expression types for XML. In *Proc. of ACM ICFP*, pages 11–22, September 2000.

[6] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, May 2001.

[7] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML*. The MIT Press, revised edition, 1997.