近似評価パターン

小室　睦(komuro@ori.hitachi-sk.co.jp)

v1.0:　2000/6/16,　v1.1:　2000/11/28,　v1.2:　2000/12/1,　v1.3　2001/4/25

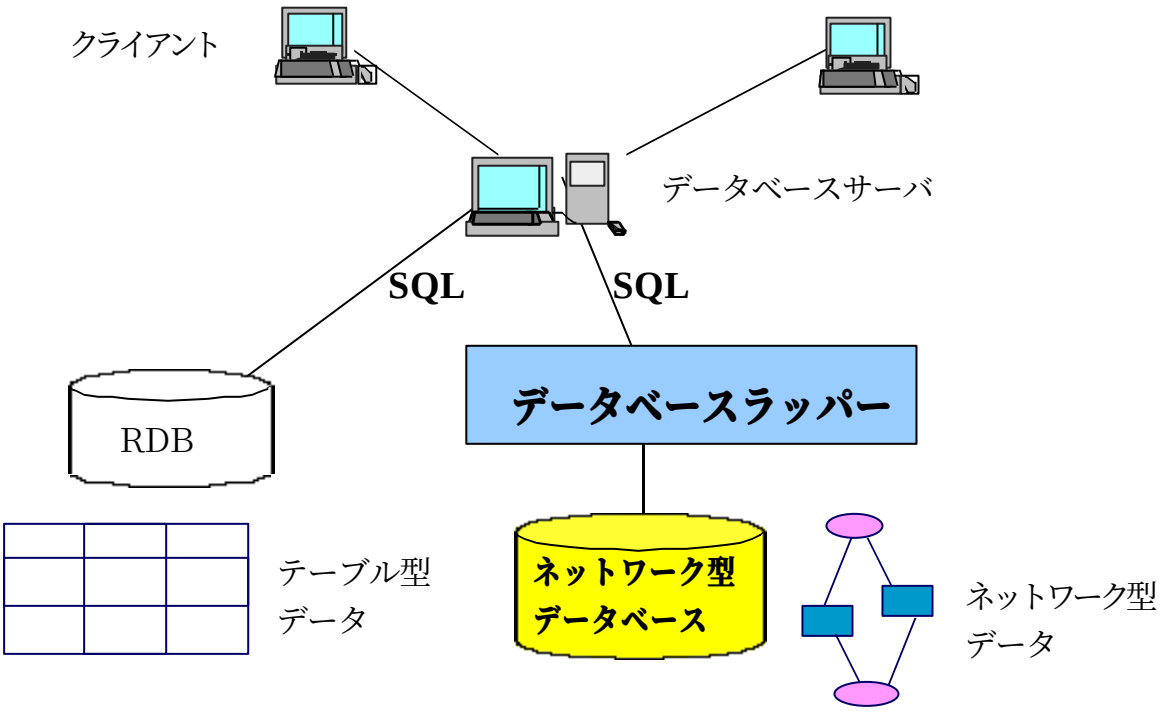
近似評価デザインパターンは、与えられた複雑な評価式に対し、それをある特定の観点から見ることで必要な情報を抜き出し、本評価のまえに見通しをつけるために用いられる。

別名 (Also Known As)

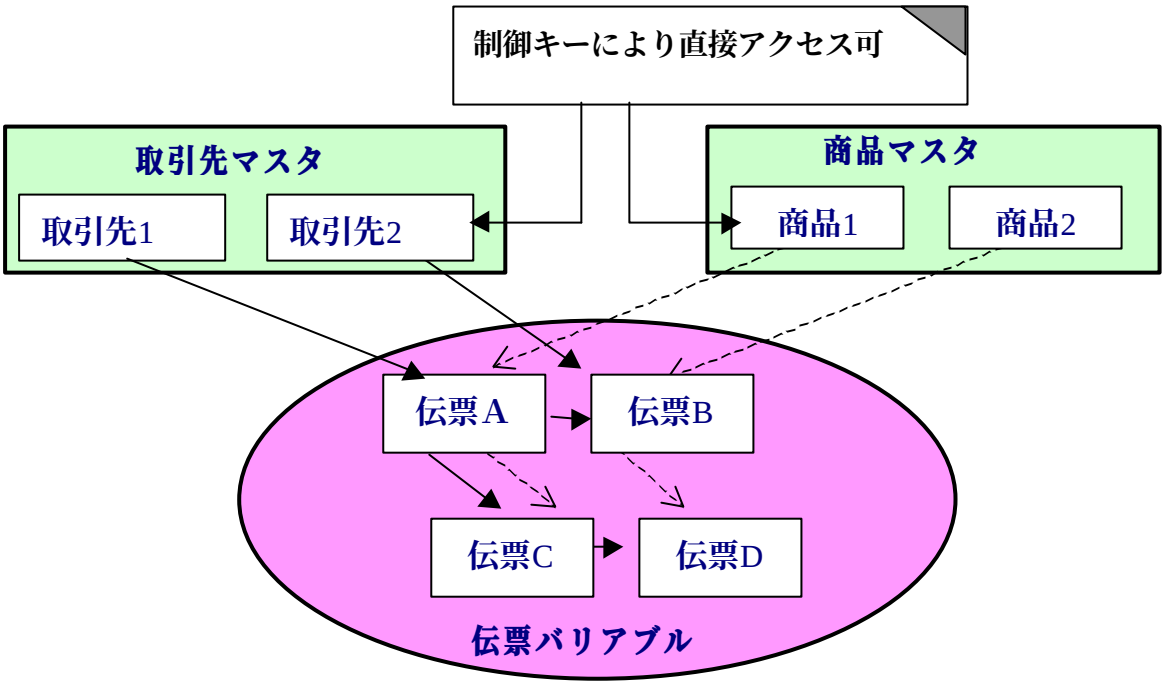
抽象解釈(Abstract Interpretation)

例 (Example)

以下のようなネットワーク型データベースに対するラッピングシステム（データベースラッパー）を考える。このシステムはメインフレーム上にある既存のネットワーク型データベースに対して関係データベースと同様のインターフェイス(SQL)を与え、クライアントサーバシステムからアクセス可能とするものである [1] 。このようなデータベースラッパーを用いるとユーザはネットワーク型データベースの内部構造等を意識することなしに、通常の SQL 命令でデータベースアクセスを行うことができる。



ネットワーク型データベースではデータが相互にリンクされたレコードの構造体（ネットワーク）として格納されている。特に、ここで扱ったネットワーク型データベースの場合には、ネットワークのノードにマスタおよびバリアブルと呼ばれる２つの種類がある。マスタはデータアクセスの基点として用いられ、データベースに対するビューを与えるノードであり、バリアブルは実際のデータを格納するノードである。マスタノードに対しては制御キーと呼ばれるキーが付与されており、マスタノード内のレコードへの直接アクセスが高速に行えるよう設計されている。



ラッパーは複数のレコードを関係データベースのテーブルとしてユーザに見せる。上に述べた制御キーもこのテーブルの一つのフィールドとして呈示される。ネットワーク型データベースについて知識を持つ既存ユーザは検索の際、このフィールドの値を指定すればアクセスが高速に行われることを期待すると考えられる。一方、ここでのデータベースラッパーはネットワーク型データベースを関係データベースとしてのインターフェイス、すなわち、SQL を通してアクセス可能とするものであり、特に扱うテーブルの各フィールドは基本的に同等に処理される。したがって、既存ユーザが期待するような高速アクセスを実現するためには、与えられた SQL の検索条件式から、制御キーの情報を抽出する技術が求められる。検索条件式から制御キーが関与している部分を抽出することで、その存在範囲を（完全に正確ではなくてもよいから）絞り込むことができれば、制御キーによる直接アクセスを検索に利用できる。ただし、範囲限定は完全に正確とは限らないから、検索条件式が本当に満たされているかどう

かアクセス後に再チェックする必要がある。

データベースラッパーに渡される SQL の条件式は、GUI で入力された検索要求からクライアント側プログラムが機械的に生成している場合があり、非常に複雑な式になっている可能性がある。

文脈 (Context)

複雑で計算量の大きい式の評価を効率よく行いたい。式の性質を予め知り、評価に対する見通しをつけたい。

問題 (Problem)

与えられた複雑な式から必要な情報のみを抽出するため、近似的な評価を（事前に）行う。

以下のような力(forces)のバランスをとる必要がある。

- 与えられた式を事前にわかる情報のみで評価し、見通しをつけたい。
  - 例えば、データベースラッパの場合、ラッパーに渡される検索条件式には、データベースに格納されているデータに関する条件も含まれている。このような条件は実際にデータベースにアクセスしないと、正否がわからないから、近似評価を行う時点では値を決定できない。
- 見通しをつけるための計算量は少なくおさえたい。
- できるだけ正確な見通しをつけたい。

  これら２つのフォースは相反するところがあり、トレードオフが問題となる。例えば、評価すべき式の範囲が非常に一般的で、チューリングマシンをエミュレートできるのであれば、評価手続きは決定不能となってしまう。評価対象を限定して決定不能となるのを避けたとしても、現実に生じる重要な問題では、その評価手続きは NP 完全問題となることが多い。したがって、完全に正確な評価はあきらめて、近似的な評価で代用する必要が生じる。
- 評価の正当性を保障したい。

  ここでの評価結果として得られた見通しが後の本格的処理の基礎となる。見通しに誤りがあると、全体の処理が破綻する危険がある。このため、評価の正当性を保障することが重要である。
- 評価の方法を拡張可能としたい。

  評価される式の形式は問題領域によってほぼ決まっているが、その評価方法は構築するシステム要件により変わって来る可能性がある。したがって、評価方法に依存する部分を独立に拡張可能とすることで再利用性が高まる。

解 (Solution)

抽出したい情報をモデル化した評価領域を用意し、この領域上で評価を実行する。取り出したいもの以外の情報や事前にわからない情報は評価領域の「未定義の値」に写すことで、評価結果に影響しないようにし、評価自体の計算量も押さえる。評価領域の構成に束などの代数構造を利用することで評価の正当性証明を与える。

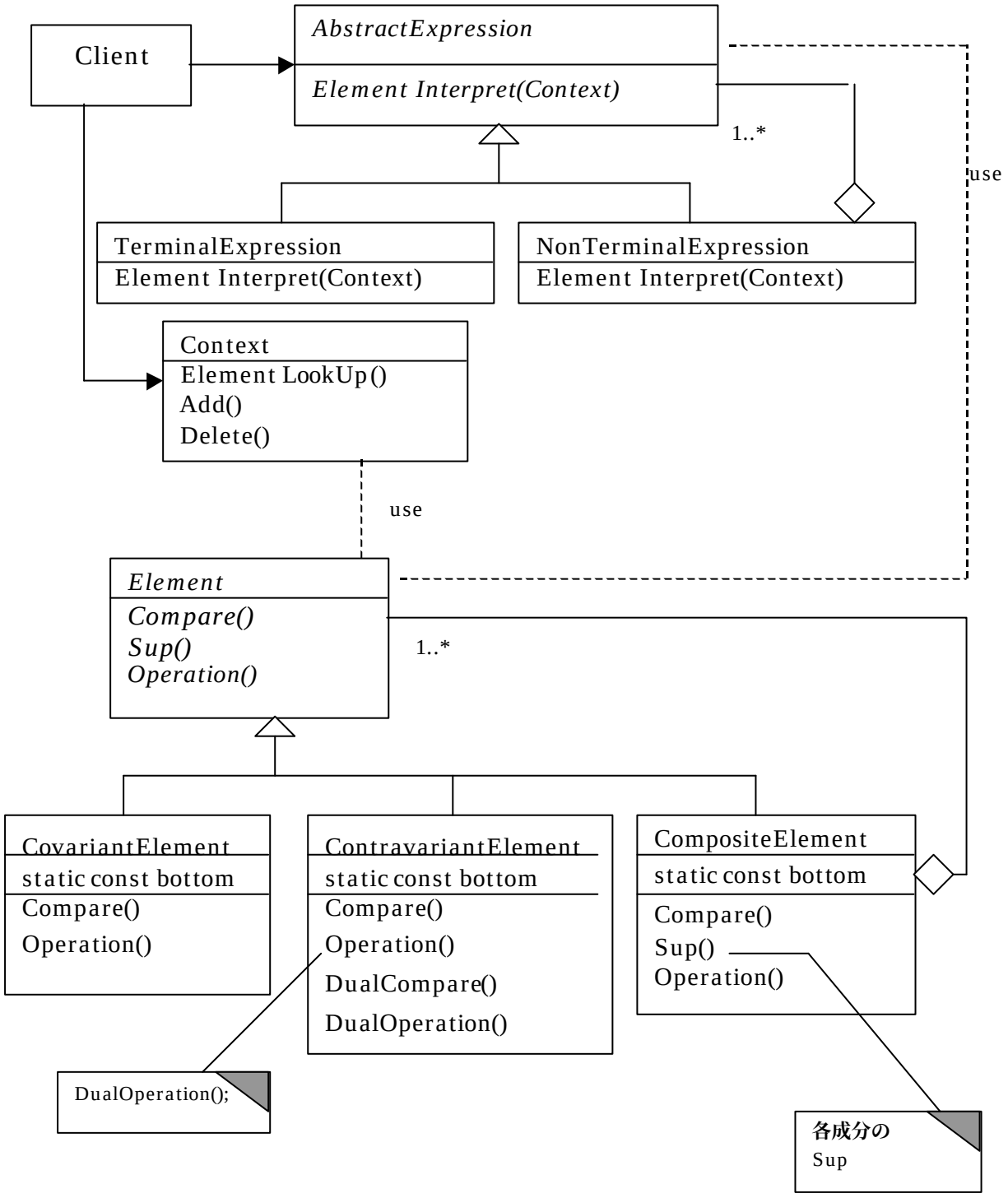
評価のためのメソッドは評価式の構文解析結果である抽象構文木に与え、評価領域と分離する。評価εは抽象構文木の集合 AST から評価領域 D への写像として実現される。　　$\varepsilon : AST \rightarrow D.$

構造 (Structure)

Element クラスが評価領域Dをあらわす。いいかえれば、Element クラスのインスタンスが近似評価の結果として得られる値を表している。評価領域は通常 CPO(Complete Partial Order)を採用する。これは半順序集合(Partially Ordered Set)で、その半順序に関する上限がまたその集合に属するという意味の完備性を満足するもののことである。さらに、その半順序に関する最小限⊥の存在を仮定する。

Element の持つ Compare メソッドが半順序を、Sup メソッドが上限をとるオペレータを表している。２つの評価領域 D_1,D_2 が与えられた時、直積 $D_1 \times D_2$ や関数空間 $D_1 \rightarrow D_2$ をとることで新しい評価領域 D を構成することができる。このように既存の評価領域から新たに構成された評価領域を CompositeElement で表している。一般にはこのような構成が何回か繰り返される可能性がある。この階層構造を表現するために Composite Pattern を用いた。すなわち、複合型の CompositeElement は単純型の CovariantElement、ContravariantElement から構成される。

Element のもつ基本オペレータはそれを構成する CompositeElement, CovariantElement, ContravariantElement の基本オペレータから誘導される。CompositeElement の持つ基本オペレータはそれを構成する成分の基本オペレータから誘導されたものであるから、結局、CovariantElement と ContravariantElement 上の基本オペレータから他のすべての基本オペレータが誘導されていることになる。CovariantElement と ContravariantElement の違いは基本オペレータの誘導の仕方の違いであり、CovariantElement では共変的に誘導するのに対し、ContravariantElement では反変的に誘導する。いいかえると、Element クラスのサブクラスとして CovariantElement クラスと ContravariantElement クラスは基本オペレータを（継承して）持つが、CovariantElement クラスではこのクラスが本来持つ基本オペレータに一致するのに対し、ContravariantElement クラスではその双対オペレータに一致する。例えば、ContravariantElement 上の Sup オペレータ（上限をとる作用子）は実は下限をとる Inf オペレータを大小比較を逆向きにみて、Sup に


Client
AbstractExpression
Element Interpret(Context)
1..*
use
TerminalExpression
Element Interpret(Context)
NonTerminalExpression
Element Interpret(Context)
Context
Element LookUp()
Add()
Delete()
use
Element
Compare()
Sup()
Operation()
1..*
CovariantElement
static const bottom
Compare()
Operation()
ContravariantElement
static const bottom
Compare()
Operation()
DualCompare()
DualOperation()
CompositeElement
static const bottom
Compare()
Sup()
Operation()
DualOperation();
各成分の
Sup

読み替えたものである。このように読みかえることで、CovariantElement と ContravariantElement を同列に扱うことができ、CompositeElement 側でこれら２つのクラスの違いを意識する必要がなくなる。また、基本オペレータの組合せとして得られるメソッドに関してはこれを Element クラスで定義しておき、基本オペレータを上述のようにオーバーライトすることでこのようなメソッド定義を何度も記述する必要がなくなる。(Template Method パターン）

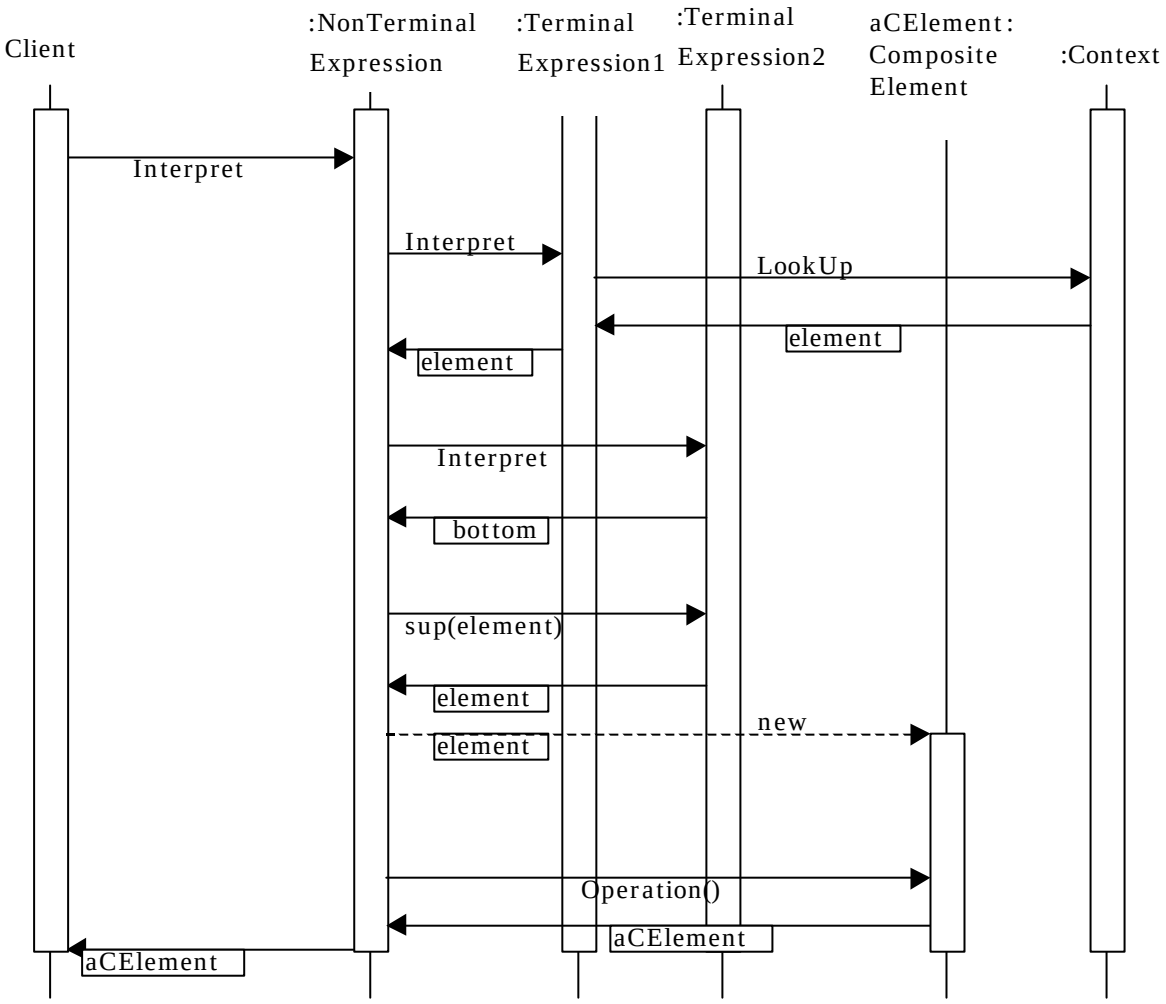
評価する表現を木構造であらわしたものが抽象構文木の集合 AST である。終端記号をあらわす Terminal クラスと非終端記号をあらわす NonTerminal クラスを用意し、これらをまとめる抽象クラス AbstractExpression を用意することで AST を表現する。（これは Composite パターン[2]の適用である。）Abstract Expression クラスは近似評価を行うメソッド Interpret を持つ。Interpret メソッドは文脈情報をもったクラス Context を利用しながら近似評価を行う。この部分のクラス構成は Interpreter パターン[2]のクラス構成と同じである。Interpret メソッドと Context オブジェクトはともに Element を利用している。

CovariantElement、ContravariantElement はそれぞれの比較オペレータに関する最小値 bottom を持ち、CompositeElement, Element クラスはこれらから誘導される最小値 bottom を持つ。Element の最小値 bottom が解(Solution)で述べた「未定義の値」として用いられる。

近似評価での Element の役割は近似に直接関係ない情報を捨象して必要な性質の抽出を行うことであるから、通常はあまり複雑な構造はとらない。したがって、実際にはここで与えたクラス構造が退化したものを採用することも多い。

動的側面(Dyamics)

まず、client が Context を初期化する。次に、評価すべき式の Interpret メソッドを呼ぶ。通常、この評価すべき式は抽象構文木の NonTerminalExpression であり、その Interpret メソッドは一段下の部分構文木それぞれの Interpret メソッドを呼び出した後、これらの部分的評価結果を統合するため NonTerminalExpression に対応する Element 上のオペレーション、例えば Sup を呼ぶ。
TerminalExpression に対する Interpret メソッドは、Context 上の値を LookUp で引いたり、指定された Element の値に置き換えることなどで処理される。特に、いま考えている近似の観点から無視すべき表現に対しては bottom を対応させる。



実装(Implementation)

・近似評価のモデル化

・評価対象となる表現の決定

まず、評価対象となる表現を決定する。Interpreter **パターンの場合はこの表現をデザインすることが、主眼点であったが、近似評価パターンの場合は本評価の対象となる表現があるはずなので、基本的にはそれを用いればよい。**

◆ データベースラッパーの例では SQL **の検索条件式が評価対象である。より詳しく述べると、**x,y **を変数、**c **を定数、**op **を比較演算子（**=,<,>,≦,≧**）として** x op y **または** x op c **の形の条件式を** and, or, not **の論理演算子で結合したものである。**

・「近似」概念の明確化

次に何のためにどのような近似評価を行うのか、すなわち近似評価の目的と観点

を明らかにする。これにより抽出したい情報と無視してよい情報の区別が明確となる。その観点から見て意味のない情報あるいは計算可能でない情報は「未定義の値」に写す。

◆データベースラッパーの例では制御フィールドの情報を抽出して、検索範囲を限定することが近似評価の主目的である。したがって、制御フィールド以外の情報は基本的に「未定義」であるとみなし、「未定義の値」に写す。記号 U でこの「未定義の値」をあらわす。x が制御フィールドを表わす変数、y, z はそれ以外のフィールドを表わす変数、c は定数とするとき、y>z, y=c などは U に写されることになる。さらに制御フィールドを含んでいても x <y のように他のフィールドも含む条件式は x に関する束縛条件として見たとき、具体的にどういう束縛であるかを（データベースアクセス以前の状態では）特定できない。このためこのような条件式も U に写す。

・評価領域の設計

評価領域を設計する。一般に、評価領域は CompositeElement の集合であり、CompositeElement はいくつかの Element クラスのオブジェクトを複合したものである。さらに、Element クラスと CompositeElement クラスの関係が共変的であるか反変的であるかの区別をつける必要がある。Element クラス上のオペレーションから CompositeElement クラス上のオペレーションが誘導(induce)されるが、その誘導のされ方が異なってくるからである。共変的な Element の場合には CompositeElement 上のオペレーションに対応するオペレーションをそのまま適用すればよいが、反変的な場合には双対的なオペレーションをとる必要がある。

◆$D = D_1 \rightarrow D_2$ という構成では D_2 は共変的であるが、D_1 は反変的である。

◆データベースラッパの場合には、制御フィールドの値は非負整数とみなせる。したがって、その範囲は非負整数内の区間の有限個の和集合としてモデル化できる。（１点も閉区間とみる。検索条件式に現れる比較オペレータの数が有限であるから区間の数も有限個となる。）区間は左端点と右端点の組とみることができるが、端点どうしの数としての大小関係と区間の包含関係との間の関手性は、右端点に関しては共変的であるが、左端点に関しては反変的である。

・基本オペレーションの抽出

評価を実行するために必要な基本オペレーションを抽出する。特に、「未定義の値」に対する基本オペレーションを定義する。基本オペレーションは評価対象から評価領域への写像を実現できるよう選択する。

◆データベースラッパの場合、基本オペレーションは以下の通り。

(1)集合としての包含関係 $\subseteq$

(2)和集合　$\cup$

(3)共通集合　∩

(4)差集合　＼

最後の３つはそれぞれ条件式の　OR, AND, NOT を評価領域上で実現するために使われる。U を含む集合演算は[0,∞)の任意の部分集合 A に対して U∩A = A∩U =A, U∪A=A∪U=U、[0,∞) ＼U = U と定める。また、包含関係⊆についても、A⊆U であるとする。特に、[0,∞) ⊆U である。

・正当性証明

以上のように定義した評価領域と基本オペレーションによる評価の実現を与える。さらにこの評価が最初に設定した近似概念に関して、評価対象に対する近似を与えることを確かめる。可能ならその証明を与える。

◆データベースラッパの例では、近似評価 ε は以下のような構造帰納法により、定義されている。P を検索条件式とし、C を制御フィールドとする。

(1) P が論理演算子を含まない時

(a)C 以外の変数を含むときは ε [P] = U とする。

(b)C 以外の変数を含まないとき。P を C の関数とみて、P=P(C)と書くことにして、 ε [P(C)] = {x∈[0,∞) | P(x)が成立つ}と定める。

(2) P = not(P')で P'が論理演算子を含まない時 ε [P] = [0,∞)＼ ε [P(C)].

(3) (2)以外の形で論理演算子を含む場合、次のように定める。

ε [P1 and P2] = ε [P1] ∩ ε [P2],

ε [P1 or P2] = ε [P1] ∪ ε [P2],

ε [not(P1 and P2)] = ε [not(P1)] ∪ ε [not(P2)],

ε [not(P1 or P2)] = ε [not(P1)] ∩ ε [not(P2)].

このようにして定義された ε [P]に対しては次の命題が成り立つ。ただし、D を任意のネットワーク型データベースとし、D に属するレコード r で制御キーの値が x であるものを r(x)と書く。また、レコード r がアクセス条件 P を満たすか否かの判定を、論理式の形で Satisfy(r,P)と書くことにする。Satisfy(r,P)=true なら P が r で満足され、Satisfy(r,P)=false なら満足されていない。

命題　ε [P]⊇{ x∈[0,∞) |　Satisfy(r(x),P)=true}

この命題は構造帰納法により容易に示せる。ε の値域は[0,∞) ∪{U}であるが、U を[0,∞)に写す射影 π : [0,∞) ∪{U}→[0,∞)との合成 π∘ ε を ε' とすると、U と[0,∞)内の区間との大小関係の定義から、ε'についても対応する命題が成り立つことがわかる。

・評価アルゴリズムの詳細化とプログラム（Interpret メソッド）実装

ここまでで行った評価プログラムの設計は、プログラムの表示的意味論を展開したということであり、フォーマルメソッドの枠組みで言えば（最初の）仕様記述を行った段階にあたる。実際に実行可能なプログラムを得るためには、この設計結果を詳細化し、さらに実行効率等についての考察を加える必要がある。この詳細化の際には、前段階で保障した正当性をくずさないように注意する必要がある。

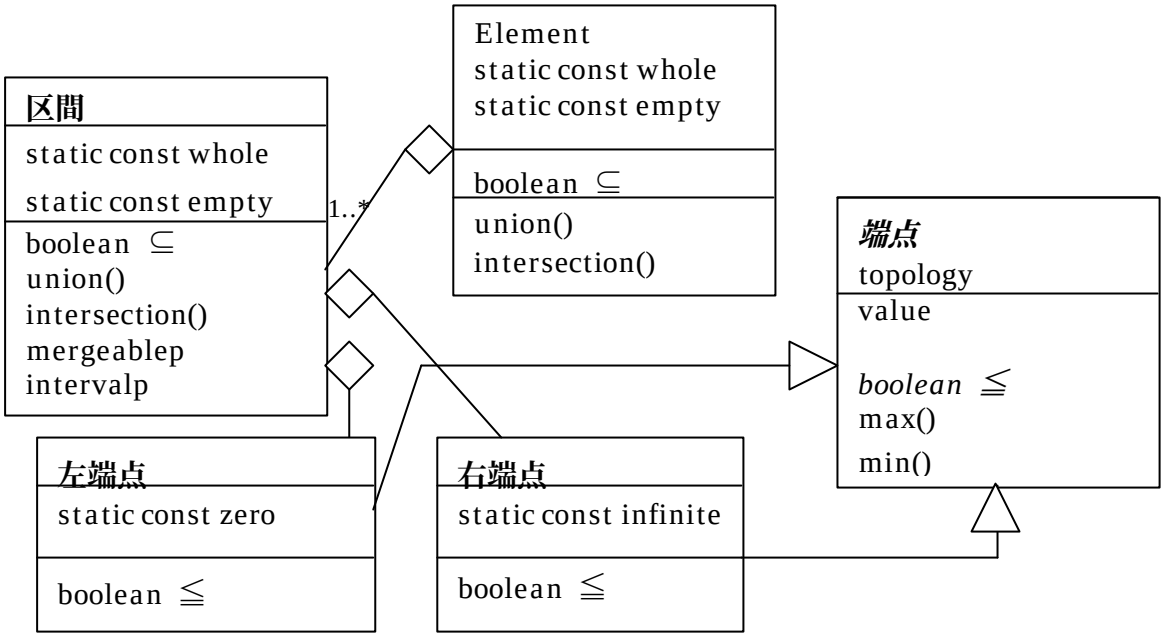
- データベースラッパーの場合には Element は区間の並びを表しているため union,intersection の実装には工夫が必要である。（並びに属している区間の組合せの数だけ、union, intersection を実行すると計算量がかかる。）データベースラッパーの実装では検索条件式を最初に選言標準形に変形しておき、区間の並びは左端点の大きさの順にソートしたリストとして実現することでこの問題を解決した。ただし、このソートに用いる大小関係は通常の数値の大小関係ではなく、左端点に対する≦を用いる必要がある。

例題への適用(Example Resolved)

既に述べたように、アクセス条件に現れる集合は、[0,∞)に含まれる有限個の区間の和集合として表現できる。区間は基本的には a≦b なる数値の組(a,b)で表現できる。a を左端点、b を右端点と呼ぶことにする。ただし、端点が区間に属しているか否かで、集合演算の実現は異なる。端点が属する時その端点は閉(closed)、属していない時は開(open)であるということにする。データベースアクセスで扱うのは[0,∞)に属する整数であるから、端点の種類を開または閉に正規化するやり方も考えられる。これは例えば、(a,∞)を[a+1,∞）に書きかえる方法である。しかし、例題のネットワーク型データベースでは制御キーの最大長が非常に大きく設定できる仕様になっており、データベースラッパでは整数ではなく文字列として扱うことにした。このため、大小比較や等号は判定できるが、1 を足すといった数値演算を効率よく行うことは難しい。そこで、今回は開閉を区別したモデル化を採用した。

さて、このように開閉を区別すると、２つの端点が開であるか閉であるかにしたがい、区間には 4 つの種類があることになり、集合演算の実現にも多くの場合わけが生じ煩雑となる。この問題は開閉を区間の性質と考えるのではなく、端点の性質と考えることで解決できる。このアイデアに基づくクラス構造を図に示す。

区間には包含関係から半順序が定まり、左右端点にはそれぞれ数の大小関係から半順序が定まる。既に注意したように、左端点と区間との関係は反変的であり、右端点と区間との関係は共変的である。



ここでは構造(Structure)の節で述べたクラス構造を特化して展開したクラス構造を採用している。この例では Element は区間から構成され、さらに区間は２つの端点から構成されているが、これらの構成要素をまとめるスーパークラスは設けなかった。区間における⊆, union,intersection を端点における≦,max,min にそれぞれ対応付ければインターフェイスを統一することができるが、クラス階層が一段だけで浅いため、あまりメリットはない。ここではそれよりも理解しやすさを重視して数学で慣用的に用いられている記法を採用した。同じ理由から反変的なサブクラスである左端点のオペレータを共変的に読み替えることはしなかった。この例題では min と max の（双対的な）両方のオペレータを使うため、読み替えをしなくとも右・左端点をまとめる抽象スーパークラス「端点」を定義することができる。また、区間クラスは空集合をあらわす empty と想定している範囲の全区間をあらわす whole の２つのクラス変数を持っている。（これらも互いに双対である。）端点クラスは min, max の定義を≦に関連づけるのに用いている。（Template Method パターン）

左端点、右端点クラスに対する≦の定義はそれぞれ次の通り。

rp1, rpt2 を右端点、lpt1,lp2 を左端点とするとき

rpt1 ≦ rpt2

⇔ def

rpt1.value < rpt2.value or
rpt1.value = rpt2.value and
(rpt1.topology = rpt2.topology or
rpt1.topology = open and rpt2.topology = closed)

lp1 ≦ lpt2

⟺ lpt1.value < lpt2.value or

def lpt1.value = lpt2.value and

(lpt1.topology = lpt2.topology or

lpt1.topology = closed and lpt2.topology = open)

とそれぞれ定める。

また、rpt を右端点、lpt を左端点とするとき、組 (lpt, rpt)が区間をなすための条件 intervalp(lpt, rpt)は以下の論理式で与えられる。

intervalp(lpt, rpt)=

(lpt. value < rpt.value) or

(lpt.value = right..value and lpt.topology = rpt.topology = closed)

2 つの区間 (lpt1, rpt1), (lpt2, rpt2)が「連結」しており、２つをあわせて一つの区間とできるための条件判定 mergeablep は次の論理式で定義できる。

(lpt1.value < rpt2.value or

lpt1.value=rpt2.value and

(lpt1.topology =closed or rpt2.topology = closed))

and

(lpt2.value < rpt1.value or

lpt2.value = rpt1.value and

(lpt2.topology =closed or rpt1.topology = closed))

これらの判定子を用いると、２つの区間(lpt1,rpt1),(lpt2,rpt2)に対する intersection, union はそれぞれ ((max(lpt1,lpt2), min(rpt1,rpt2))、((min(lpt1,lpt2),max(rpt1,rpt2)) という新しい区間を返すよう定めればよい。ただし、intersection, union を呼ぶ側でこれらが実際に区間を定義しているかどうかを確かめておく必要がある。メソッド interpvalp, mergeablep はこの判定のために用いられる。

Element は区間の有限和を表しており。実装上は mergeablep が偽となる区間達の並びで表現する。Element に対して区間上の union,intersection が自然に拡張される。

「実装」の章で述べたようにデータベースラッパーの場合の評価対象は SQL の検索条件式であり、これを構文解析した構文木の各ノードに interpret メソッドを与える。構文木のノードは AND,OR 等の論理演算子に対応しており、interpret メソッドは AND,OR を Element クラスの intersection, union にそれぞれ対応付けることで評価を実行する。Context は変数束縛として実現され、変数に対する値を保持している。

Element は区間の並びを表しているため union,intersection の実装には工夫が必要

である。（並びに属している区間の組合せの数だけ、union, intersection を実行すると計算量がかかる。）データベースラッパーの実装では検索条件式を最初に選言標準形に変形しておき、区間の並びは左端点の大きさの順にソートしたリストとして実現することでこの問題を解決した。ただし、このソートに用いる大小関係は通常の数値の大小関係ではなく、左端点に対する≦を用いる必要がある[8]。

既知の使用例(Known Uses)

・束縛時点解析

部分計算の実装に抽象解釈を用いた例である。部分計算に関しては自己適用を実現すればコンパイラやコンパイラコンパイラが自動的に得られるという二村の結果[4]があり、自己適用可能な部分計算器の実現が挑戦的な課題とみなされている。この課題が困難なのは自己適用を行うと部分計算処理が無限ループする危険があるためである。Jones ら[5]はこの困難の解決を目指して、抽象解釈を用いてプログラムのうち部分計算可能な部分を近似的に求める手法を提案した。これが束縛時点解析である。束縛時点解析では評価領域として static と dynamic の 2 値からなる束をとるのが基本的である。

・Strictness Analysis

関数型言語で並列性を実現する有力な手法の一つに遅延評価がある。抽象解釈により、どの引数に対して評価を遅延できるかどうかの解析を行うのが、Strictness Analysis[6]である。この場合も評価領域としては 2 値の束をとるのが基本的である。

・Query Subsumption

データベースの分野では、ある問い合わせより弱い問い合わせを定式化した query subsumption という概念が知られており、異種データベースへの問い合わせ集合を包括する問い合わせの生成などに利用されている[7]。明示的に意識されてはいないが、Query Subsumption を得る手続きは近似評価を実行しているとみなせる。

結果(Consequences)

近似評価パターンは以下のような利点を持っている。

- 高い実行効率

不要な情報を「未定義の値」に写像して無視することで、実際の評価に比べて近似評価は高速に実行できる。

- **見通しのよさ**

  評価という統一的な枠組みでアルゴリズムが記述でき見通しがよい。プログラムとしての実現も容易である。
- **正当性の保証**

  評価領域の構成に明確な数学的モデルを採用することで、正当性証明を行うことができる。

近似評価パターンの欠点としては以下のような点があげられる。

- **評価領域設定の難しさ**

  適切な評価領域設定を行うことは容易ではない。特に正当性証明を行うためにはそれなりの数学的素養が必要である。
- **評価結果の保守性**

  評価という統一的な枠組みの中で、正当性の証明されている範囲に近似を限定しているため、評価結果は一般的に「保守的」である。すなわち、事前に決定できない部分が少しでもある表現はすべて「未定義の値」に落としてしまう。

変種(Variations)

- **Visitor パターンの利用によるインタプリタの切り替え**

  近似評価パターンを利用した場合、近似評価の後に本評価を実行することになる。両者はともに同じ抽象構文木に対する評価手続きであるから、これらの間でプログラムの一部を共有したいという要求が考えられる。いくつかの観点から近似を行うため、数種類の近似評価を行うような場合にも、このような要求が生じる。こういった場合には、Visitor パターンを適用して Interpret メソッドを切り分ける手法が有用である。

- **Parser Generator の利用**

  評価対象がある程度複雑な場合には Interpreter パターンを用いるよりは Parser Generator 等を用いてインタプリタを生成する手法が有効である。この場合にも評価領域の設定、基本オペレーションの抽出等の技法は必要となる。

関連パターン(Related Pattern)

Interpreter Pattern

インタプリタの実現に Interpreter Pattern[2]**のクラス構造を用いた。**

Composite Pattern

Element **を構成する** Element **のクラス構造を表現するのに、**Composite Pattern[2]

を用いた。また、Interpreter Pattern中の抽象構文木のクラス構造にも用いている。

Template Method Pattern

基本オペレーションの組合せとして定義できるメソッド（のコード）を Element のサブクラスで共有するのに Template Method Pattern を用いている。基本オペレーションは各サブクラスで上書きされている。

参考文献

[1] 古舘丈裕, 小室　睦: データベースラッピングにおける問合せ処理の実現手法: 情報処理学会第 62 回全国大会,　No.3, pp.223-224,　2001.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides.　*Design Patterns – Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.

[3] J.E. Stoy. *Denotational Semantics : The Scott-Strachey Approach to Programming Language Theory,* MIT Press, 1981.

[4] Y. Futamura. Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler. *Systems, Computers, Controls*, vol.2, No.5 , pp.45-50, 1971.

[5] N.D. Jones, P. Sestoft, and H. Sφndergaard. An experiment in partial evaluation,　*Rewriting Techniques and Applications*, LNCS 202, pp.124-140: Springer-Verlag, 1985.

[6] J. Hughes.　Analysing strictness by abstract interpretation of continuations:　In *Abstract Interpretation of Declarative Languages*, pp.63-102　Ellis Horwood, 1987.

[7] K.C-C.Chang, H. Garcia-Molina, and A. Paepcke. Boolean Query Mapping Across Heterogeneous Information Sources, *IEEE Transaction on Knowledge and Data Engineering*, vol.8, No.4, 1996.

[8] 小室　睦: データベースアクセス最適化への抽象解釈の適用: to appear in SEAソフトウェアシンポジウム 2001