ソフトウェア基礎論　18−4
（１９８６・９・３０）

オブジェクト指向言語Ｏｎｄｉｎｅの抽象化機構

荻原剛志　梶原洋一　長野伸一　有澤　誠
山梨大学工学部　計算機科学科

Ｏｎｄｉｎｅは、オブジェクトのモジュール化と並列実行を強く意識したオブジェクト指向言語である。
本稿では、まず、Ｏｎｄｉｎｅのクラス階層の仕組みについて述べる。このクラス階層の記述方式では、クラス間の継承関係を構造的に表すことにより、スーパークラスの変更や、クラスのインタフェース情報の管理を容易にすることができる。
続いて、このクラス階層を用いて型の導入を行うことについて述べる。ここでは、Ｏｎｄｉｎｅにおける型の意味と、強く型付けされたオブジェクトによるプログラミングを可能にする、型相互の「互換性」という概念について論じる。

Abstraction Mechanism of Object Oriented Language Ondine

Takeshi OGIHARA　Yoichi KAJIHARA　Shin-ichi NAGANO　Makoto ARISAWA
Faculty of Engineering, Yamanashi University, 4-3-11, Takeda, Kofu, 400 Japan

Ondine is an object oriented language, which aims at modularity and concurrent execution of objects.
The first half of the paper describes the class hierarchy mechanism, the between-classes inheritance information.
The second half of the paper explains about introduction of typed-objects using the class hierarchy and the semantics of type.
The major point is that the compatibility between types enables to program with strong typed objects.

1．はじめに

プログラムを書き、保守し、利用して行く上で、その正当性を維持するために注目すべきこととして、プログラムのモジュラリティや、情報隠蔽ということをあげることができるであろう。そして、これらの理想的な形のひとつをオブジェクト指向の原理の中に見いだすことができる。

しかし、現実にオブジェクト指向言語と言われている言語は、こういった要求を十分満たすものなのであろうか。また、オブジェクト指向の考え方に従った、これらとは別の形のプログラミング言語は考えられないのだろうか。

我々は、オブジェクト指向型と言われている言語を再検討、再構築することにより、オブジェクト指向言語の「別のあり方」を探ることとした。このために構築するシステムの名称を、Ｏｎｄｉｎｅ（オンディーヌ）とする。

Ｏｎｄｉｎｅは手続き型のコンパイラ言語である。現在の段階では、言語仕様は開発環境と独立可能とし、従来のコンパイラ言語のような開発環境を想定している。特定の計算機環境に依存しない、移植性のよさということにも留意したい。

Ｏｎｄｉｎｅは、主に以下の３つのことを支援することを目標としている。

1. プログラムのモジュラリティ、情報隠蔽
2. クラス階層をもとにした差分プログラミング
3. オブジェクトの並列処理

プログラム作成時の概念整理と、翻訳時のエラーチェックのために、オブジェクトに型を導入することについても検討する。

これらを実現するための、Ｏｎｄｉｎｅの中心となる構想は以下の３つである。

1. オブジェクトの性質についての解釈
   従来のオブジェクトの性質には、並列処理を行おうとする場合に問題になる点が多かった。オブジェクトとクラスの関係や、オブジェクトの生成、消滅などについても新しい考え方を導入する。
2. クラス階層と継承、型の実現方法
   今までの記述方法では、概念を系統的に表現したり、まとめて修正したりするのが困難であった。この点について改良を加え、さらにこの階層を利用して型の導入を行う。
3. オブジェクトの並列実行方式
   並列に処理を進める上で必要なメッセージ通信や同期、相互排除について、オブジェクトと親和性のよい方式を取り入れる。さらに、並列処理の単位を階層化することにより、信頼性の高いプログラミングができるようにする。

本稿ではこのうち、オブジェクト、クラス、型についての概略を述べる。

2．オブジェクトの性質

2.1. データ、制御構造とオブジェクト

プログラムの中でオブジェクトをどのように位置づけして考えるべきか、ということがまず問題となる。今までの言語での例を見ると、オブジェクトを一番根源的な概念としてとらえた言語はSmalltalk-80であると言えよう。Smalltalk-80では、条件分岐や繰り返しにあたるものまでをオブジェクトの考え方で説明する。これに対し、既存の言語にオブジェクトを導入したものの中には、オブジェクトをひとつの型として扱う感じのものもある。これらのさまざまなアプローチには、それぞれに理由があり、どれが正しい、という言い方はできない。

Ｏｎｄｉｎｅの場合には、従来型の言語でデータとかデータ構造と言われていた部分、手続きと言われていた部分をオブジェクトとして統一的に扱う。その他の、制御部分、宣言部分についてはオブジェクトとは見なさない。

オブジェクトであるデータ、そうでないデータ、というような区別をせず、すべてをオブジェクトに含めることにより、プログラムの作成や修正がしやすく、概念の上からも統一的に考えることができる。

制御部分、つまり条件分岐や繰り返しなどの制御構造をオブジェクトのはたらきで説明することは不可能ではないが、そのようにすることの必然性はないと言える。さらに、制御構造をオブジェクトの性質にゆだねると、制御構造の抽象化という問題が生じてくる。むしろ、従来の手続き型言語の構造の方が、よりわかりやすく、簡潔に記述できるのではないだろうか。

従って、Ｏｎｄｉｎｅでは、制御構造部分は手続き型言語と同様な構文を使用することにする。すべてをオブジェクトで説明しなければならないという必要はないのである。

2.2. オブジェクトの構成

オブジェクトは、次のものからなる。

a. 内部オブジェクト
b. インタフェース
c. メソッド
d. メッセージの待ち行列

オブジェクトの内部には、そのオブジェクトだけから見える内部オブジェクト（０個以上）があってよい。

内部オブジェクトの中には以下の種類がある。

- 構成オブジェクト……それを含むオブジェクトが生成した時から消滅するまで存在する。いわゆるインスタンスオブジェクトのこと。
- 一時オブジェクト……各メソッドの実行時に生成し、メソッドの終了とともに消滅する。メッセージの引数オブジェクトもこの中に含む。
- 無名オブジェクト……式の評価にともなって生成され、評価の終了によって消滅する。このオブジェクトは表だって現れることはない。

インタフェースとは、そのオブジェクトに対し、どのようなメッセージが送れるのか、返る結果は何か、という情報をクラスごとにまとめたものである。オブジェクトを使う時、原則的にはこれ以外の情報は必要でない。

オブジェクトを生成しないスーパークラスでインタフェースだけを記述し、メソッドの実現はサブクラスに任せるということもできる。

メソッドには、次の種類がある。

- オブジェクトとしてもともと備えているメソッド(primitive method)
- インタフェースの一部として外部に公開するメソッド(public method)。
- オブジェクトの内部でだけ使用し、公開しないメソッド(private method)。

Ｏｎｄｉｎｅシステムは、オブジェクトによる計算に必要な低水準のメソッドを持っている。これらをプリミティヴメソッド(primitive method)という。これは、Ｏｎｄｉｎｅのインプリメンテーションをメソッドという形で表したものである。

プログラマが作成するメソッドは、 public か private かである。インタフェースの一部として公開し、オブジェクトの外部から、メッセージの送信によって起動することができるものをpublicメソッドという。一方、オブジェクトの内部で使用するだけで、外部へ公表しないメソッドをprivateメソッドという。このような構成とすることで、オブジェクト内部での手続きを構造的に構成しても、不必要な部分は外部へ見せない、ということが可能となる。

さらに、継承の際、スーパークラスのインタフェースのうち、特定のものだけを継承することを指定できるが、この時継承したメソッドはすべてpublicになり、継承しなかったメソッドはprivateとなる。

Ｏｎｄｉｎｅには、指定したスーパークラスのメソッドを起動するという機能があり、これを用いてメソッド結合に準じたメソッドの実行方式の記述を行うことができる。

2.3. オブジェクトの状態

オブジェクトは、生成直後は休眠状態にある。この状態でメッセージを受け取ることができる。

メッセージを受け取ると、オブジェクトは活性状態になり、メッセージに対応した処理を行う。

処理の終了時点、あるいは途中で処理を中止して、送り元と同期をとるために待ち状態に入ることがある。この状態を待機状態という。（図１）

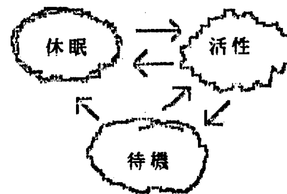

図１　オブジェクトの状態遷移

あるオブジェクトのメソッドを実行する制御の流れはひとつだけであるとする。しかし、オブジェクトとその内部オブジェクト、または、内部オブジェクト同士を並列に実行することはできる。

３．クラスとオブジェクト

3.1. オブジェクトの生成

Ｏｎｄｉｎｅではクラスはオブジェクトではなく、インスタンスであるオブジェクトの内容や性質を定義した宣言としてとらえる。すべてのオブジェクトはクラス、あるいは他のオブジェクトのコピーとして生成し、必要に応じて修正を加える。

オブジェクトからオブジェクトをコピーするには、そのオブジェクトに対してメッセージを送ってやればよい。では、クラスからオブジェクトを生成するにはどうすればよいのか。これら自体は単なる定義であり、メッセージを受け取ったりすることはできないはずである。

Ｏｎｄｉｎｅでは、式の中にクラス名が現れた場合、そこにそのクラスのインスタンスがあるものと見なして評価を行う。このオブジェクトは、式の評価にともなって一時的に生成し、処理の終了とともに消滅する。例えば、

```
A := B;
```

という式があると、B がオブジェクトならそのコピーが A に代入されるのだが、B がクラス名なら、クラス定義のままのオブジェクトが作られて A に代入されることになる。つまり、式の中のクラス名は、評価のたびにクラス定義通りのオブジェクトを作り出すのである。

ここで、クラス定義通りのオブジェクトという言葉を使ったが、その値、つまり、構成変数の内容はどうなっているのであろうか。(なお、以下の記述で「変数」という用語を使うが、これはオブジェクトではなく、プログラム記述上の構文要素のことである。通常は変数とオブジェクトの区別をしなくてもよいが、以下では両者を区別して論じる。)

変数はすべて型を宣言しており、必ずどれかのクラスに属している。変数が実際に値(オブジェクト)を持つのは、以下の場合である。

a. 構成変数の場合、それを含むオブジェクトが空でない値を持った時
b. 一時変数の場合、それを含むメソッド(あるいは文)が実行された時

オブジェクトは、特別の指定をしなければ、生成した時に空(null)を値として持つ。これは、再帰的なオブジェクトを生成する場合に必要な仮定である。

空の場合を含め、値を持った変数は既にその時点から宣言したクラスに属している。オブジェクト生成の際、構成オブジェクトの初期値を指定するには後述のコンスタントの機能を使用すればよい。

以上に述べたインスタンスとクラスの関係には、クラス変数、クラスメソッド、あるいはメタクラスといった概念がない。オブジェクトはクラスのインスタンスのみであり、すっきりした構成とすることができる。

クラス変数は並列処理の際、共有変数として振舞う。このことも、クラスをオブジェクトでないとしたひとつの理由である。クラス変数やクラスメソッドにあたる機能を実現するには、そのためのオブジェクトを別に定義しなければならない。しかし、このようなはっきりした形で相互関係を記述することが重要なのである。

3.2. コンスタント

オブジェクトは状態の変化について大きく2種類に分かれる。メッセージを受け取って状態を変えてゆくものと、どんなメッセージを受け取っても状態を変えないものである。前者は、構成オブジェクトを内部に含む一般のオブジェクトである。後者は構成オブジェクトを含まないオブジェクト、およびコンスタント(定数オブジェクト)である。

コンスタントには、まず、1とか3.14とか、あるいは'A'、"This is a string."などというものがある。これらはそれぞれ、整数、実数、文字、文字列のインスタンスであるが、そのことは宣言をする必要がない。

これに対し、PI という名前で円周率を表すとか、ATAN という名前で、ある確定した数表を表す、というような使い方をするコンスタントもありうる。

従来の言語では、これらについてはっきりした定義や構成法を述べていない。以下では、コンスタントをクラス階層の一部とする考え方について述べる。

コンスタントには、どんなメッセージを送っても、その内容が変化することはない。コンスタントがオブジェクトだとすると、このような性質を説明することはできない。そこで、コンスタントにメッセージを送った時だけオブジェクトが生成され、メッセージの処理を行い、処理がおわると消滅してしまうのだと考える。この考え方は、クラス名が式の中に現れた場合の考え方と同じである。

いま、クラスは、構成変数の初期値をクラスの外から静的に指定できる機能を持っていてもよいとする。すると、初期値を指定したクラスを表すものが、上で論じたコンスタントそのものである。

このように、コンスタントをクラス階層の中で説明することにより、概念を統一することができ、また、複雑な構造を持っているコンスタントを構成することもできるようになる。

コンスタントを、メッセージに対して値だけを返し、副作用を残さないものであると考えることもできる。これは即ち、コンスタントを関数として見なしてもよいことを示している。コンスタント、関数については今まであまり明確な定義がなかったが、この考え方はこれらに対し、統一的な見解を与えるものであると考えられる。

構成変数の初期値をクラスの外から静的に指定できるという機能は、汎用体の考え方で説明できる。

しかし、汎用体の機能は強力なだけに、無制限に取り入れることは避け、一定の規則、制限を設けることを検討している。

3.3. コピーと実体

Ｏｎｄｉｎｅの大きな特徴のひとつは、メッセージの引数の受渡しを、原則としてコピーで行うことである。また、オブジェクト内部の構成オブジェクトが他のオブジェクトを示すこともできないようになっている。

このような考え方を採用したのは、オブジェクトを渡した先での副作用、特に並列処理時の相互排除を考慮したためである。また、オブジェクトのモジュラリティを高める上でも効果があると考えられる。

しかしこれは、明らかに通常のオブジェクトにおける、値とアドレスの区別を考えなくてもよいという利点を損なっている。どちらかというと SIMULA に近い考え方である。

（オブジェクトのコピーと実体、並列処理方式については本稿の範囲を越える）

4． クラス階層

4.1. 継承

Ｏｎｄｉｎｅでは、クラスの多重継承を許す。継承の対象となるものは、インタフェース、構成変数、メソッドである。

スーパークラスのインタフェースのうち、継承したいものだけを指定することができる。通常、何の指定もしなければすべてを継承する。

構成変数は、すべてのスーパークラスのものを全部継承する。

インタフェースに対応するメソッドがスーパークラスにある場合、継承を指定したものは public なメソッドとなる。ただし、サブクラスでメソッドを記述し直してもよい。継承を指定しなかったものは private なメソッドとなる。

スーパークラスではインタフェースだけを記述するということができるため、対応するメソッドがない場合もある。同様に、インタフェースを継承する場合でも、そのメソッドを必ずしも記述する必要はない。

スーパークラスで privateなメソッドはサブクラスでも private である。

通常のクラス定義の省略形式を図２のように定めておく。 inherit の後には、スーパークラス名を書く。

```
class <class-name>
  inherit <class-name>; ...;
  ...
endclass
```

図２　クラス定義（省略形）

4.2. クラス階層の記述法

オブジェクト指向言語の強力な差分プログラミングの能力は、継承を利用したクラス階層によるものである。この能力により、以前書いたクラスを利用して新しいクラスを作成したり、同じようなクラスをまとめて扱うなどのことが可能になる。

しかし、問題点も多い。主なものとして、スーパークラス、サブクラス間の情報隠蔽が不完全であること、クラス管理をする言語上の機能が不十分であること、サブクラスをプログラム中のあちこちに記述できるため、『ひとまとまりのデータに対する手続きはまとめて書く』という原則が損なわれやすいこと、がある。

これらの問題点を解決するために、まず、クラス階層を、継承の意味を表現する構造的な方法で定義するべきであると考えた。そのようにすれば、クラス間の継承関係はおのずから明らかであるし、あるクラスのサブクラスをまとめて記述することができる。従って、スーパークラスの仕様を修正した場合にも、サブクラスの修正が比較的容易に行える。

スーパークラスとサブクラス間の関係を端的に表現するには、木構造をとるのが自然である（図３）。

```
class A ...{define}... endclass A
subclass
  class B ... endclass B
  subclass
    class B1 ... endclass B1;
    class B2 ... endclass B2;
  end B;
  class C ... endclass C;
end A;
```

図３　クラス階層の木構造表現

Ｏｎｄｉｎｅではオブジェクトのインタフェース（仕様）と本体の定義を別々に書くことができる。クラス階層の宣言にはインタフェースのみを記述してもよい。

しかし、多重継承を認めるとすると木構造では対応できない。多重継承の場合には、スーパークラスの機能を拡充するというよりも、既存のクラスから新たなクラスを作成するといった意味あいが強いと考えられる。そこで、多重継承の場合も含め、スーパークラスの具体化と見るのはふさわしくないと考えられるものは、インタフェースの記述だけで、本体とそのサブクラスは外部へ出すこともできることにする（図４）。

```
class X ... endclass X
subclass
  class X1 ... endclass X1
  subclass
    class Z ...{interface}...endclass Z;
  end X1;
end X;

class Y ... endclass Y
subclass
  class Z ...{interface}...endclass Z;
end Y;

class Z
  inherit X1;Y;
  ...{body}...
endclass Z;
```

図４　多重継承の表現

このクラス階層の構成方法は、サブクラスをスーパークラスの具体化が進んだものとする見方と、スーパークラスをサブクラスを構成するための部品とする見方の統合となっているともいえる。

4.3. クラスの情報隠蔽

従来のオブジェクト指向言語では、サブクラスのメソッドからスーパークラスやメッセージの引数の内部情報にアクセスすることができ、情報隠蔽が完全ではなかった。この点については、原則としてインタフェースしか見せないというように、厳しくすることとした。ただし、スーパークラスの内容は、継承の記述の際にその旨を宣言すればサブクラスで参照できる。

クラス名は、通常はグローバルであるが、あるクラス内だけで有効な型やコンスタントを宣言することもできる。

クラスには、オブジェクトを生成できるものと、そうでないものがある。オブジェクトを生成できないクラスとは、スーパークラスとしてのインタフェースだけを記述しているクラス、サブクラスを生成するために作られた中間的なクラスなどである。これらの中には、グローバルな名前を持つ必要のないものもある。そのようなクラスは、階層の部分木の中でローカルにすることもできる。

オブジェクトを生成できるためには、すべてのメソッドが定義されていることが必要である。

５．型の導入

5.1. 型の相互関係

オブジェクトを使用する際、そのオブジェクトのインタフェースを知ってさえいれば、メッセージを送り、処理を行うことができる。しかし、実際にはさらに、メッセージの引数としてもよいクラス、指定してはならないクラスの別がある。プログラムの翻訳時にこの情報を利用できれば、正しいプログラムかどうかチェックする上で大変有力な方法となる。即ち、型の導入である。

極めて単純な方法では、メッセージの仮引数のクラスを宣言し、これと異なるクラスのオブジェクトは、その実引数としては使用してはならないことにすればよい。しかしそれでは、実際のプログラミングでいろいろな問題が起きてくる。

例えば、整数のサブクラスとして林檎の個数を表す林檎型と、時刻を表す時刻型を定義したとする。林檎型へのメッセージの引数は林檎型、あるいは整数型にしたいが、時刻型は禁止したい、というような場合がある。

また、要素の登録、参照、削除についてのインタフェースだけを記述した表型を作り、そのサブクラスとして、二分木型とハッシュ型を定義したとする。どちらもインタフェースは同じであるから、表型としてまとめて扱いたい、という場合もある。

以上のことから、型チェックで行うべき事柄をまとめると、以下のようになる。

a. メッセージの引数としてとれるクラスを指定し、それと異なるクラスに属するオブジェクトが実引数となることを禁止する。

b. 同じ内部構造を持つクラスの間では、必要に応じて a.の制限を緩めることができる。

c. 同じインタフェースを持つクラス間では、必要に応じて a.の制限を緩めることができる。

b.は c.の特殊な場合と考えてもよいが、２つの型の間で仮引数、実引数の関係がどのようになるかで区別することができる。例えば、クラスＡと同じ構造をもつクラスＢ、クラスＡをスーパークラスのひとつとするクラスＣを考えてみる。クラスＡとクラスＢは相互に仮引数、実引数になることができる。クラスＡとクラスＣの間では、クラスＣのオブジェクトがクラスＡのインタフェースを持つことから、クラスＡが仮引数、クラスＣのオブジェクトが実引数となれるだけである。

Ｏｎｄｉｎｅでは、型を、関係整理の方法としてもとらえられるように位置づける。そのためには、

クラスの階層構造を利用して定義するのが適切な方法である。

まず、b.の場合について考える。このように2つの型が互いに仮引数、実引数になりうる関係のことを「互換性」という言葉で定義する。この場合、型の間の継承関係は単一継承に限り、また、サブクラスで新たな構成オブジェクトを付け加えることも許さない。

さらに、互換性について以下のような規定を設けることにする。

1. ある型の直接のサブクラスをその型と互換であるようにできる。
2. あるクラスと互換性を持つことができるのは、そのクラスと互換性のあるクラスのサブクラスだけである。

この再帰的な規定により、いわば直系の先祖だけを互換なクラスとして定義することができる。先ほどの例では、整数型と林檎型、整数型と時刻型をそれぞれ互換であるとし、しかも、林檎型と時刻型は互換としないことができる。このように、同じスーパークラスを持つクラスの間でも、必要に応じて、演算を禁止することができるという点が大きな特徴である。

また、このような構成はクラス階層を用いて比較的容易に達成できる。

次に、c.の場合について検討する。

この場合、b.でのような継承に関する制限事項はないが、相互に仮引数、実引数になることはできない。仮引数で指定した型のインタフェースを含むオブジェクトが実引数になれるだけである。例えば、仮引数を表型とし、実引数を二分木型にすることはできるが、その逆はできない。この関係を、二分木型は表型と「部分互換性」を持つという。これはいわゆる「上位互換性」のことであるが、この場合の「上位」とクラス階層についての「上位」が反対の意味を表し、混乱するため、この言葉を用いることにする。

上では同じインタフェースを持つという表現を用いたが、これはクラス階層で考えると、同じスーパークラスを持つということで統一的に表現できる。従って、継承の指定の際に、スーパークラスと部分互換性を持つかどうかを宣言すればよい。

5.2. 型の定義方法

最初に、クラスと型の関係について述べる。

クラスにはオブジェクトを生成できるものとできないものがあるということは前述した。生成したオブジェクトには必ず型がなければならないことから、オブジェクトを生成できるクラスが型である、と見ることもできる。しかし、さきほどの例の表型のように、オブジェクトを生成しない、インタフェースのみの型もあると都合がよい。また、何種類もの概念を導入することで全体がわかりにくくなるおそれもある。従って、クラスと型は同じものとして扱うことにする。

次に、上で述べた型の2種類の相互関係を、クラス階層の中でどのように定義するかについて論じる。

まず互換な型の定義方法である。この場合には型についての制限が厳しく、通常のクラス定義の枠の中で定義するのは難しい。そこで、単一継承で、構成変数を付け加えることができない、という宣言形式を作ることにする。これをクラス定義に対して、型定義と呼ぶ。

```
type <type-name> isa <class-name>
 ...
endtype
```

ここで、 type-name が新たに定義する型の名前、class-name が継承元のクラス名である。

次に部分互換な型の定義である。

こちらは、普通の継承によるクラス定義をそのまま部分互換であると考えればいいように見える。しかし、継承には、スーパークラスのインタフェースや内部構成を利用するだけで、型としての概念的なつながりがない場合もある。また、スーパークラスのインタフェースをすべて受け継いでいるかどうかという点も問題である。

継承の際、スーパークラスのインタフェースのうちのいくつかを指定して継承する機能があることは既に述べた。この形式は、

```
inherit <class-name> ( <selector>, ... )
```

である。 selector はメッセージセレクタを表す。すべてのインタフェースを引き継ぐ場合、

```
inherit <class-name> ( all )
```

と記述する。ここで、この仕組みを利用し、

```
inherit <class-name> ( type )
```

のようにすれば、インタフェースをすべて継承し、しかもそのクラスと部分互換であることが宣言できることにする。

5.3. same型

以上のような定義方法で、クラス階層内の型定義を行うことができる。しかし、このままでは、イン

タフェースを継承したとしても、インタフェースやメソッドで指定されている型名はスーパークラスの型名である。サブクラスでは型名を新しい型の名前に書き直さなければならないという問題が残っている。例えば、林檎型の場合、整数型から加算を継承したとしても、加算の引数は整数型のままである。これを整数型ではなく、林檎型にしたい。

この問題を解決するため、インタフェース、メソッドの変数の型宣言でのみ有効な same型を導入する。

same型は、そのインタフェースやメソッドを持つクラス自身の型を表す。さらに、互換な型としてインタフェースを継承した場合にも、 same型はつねに自分自身の型を表すのである。これは、オブジェクトにおけるselfに対応する概念である。

この規則を利用して林檎型を定義してみる。まず、整数型は以下のようなインタフェースを持つものとする。

```
class int;
  interface
    (add: same)(sum: same);
    (sub: same)(rem: same);
    ...
endtype;
```

ここでは same は int のことである。しかし、ここで intとあからさまに記述してしまうと、サブクラスで引数として整数しかとれなくなってしまう。

林檎型は例えば、次のようになる。

```
type apple isa int
  method(sub:same)(rem:same)
    rem:=super'int(sub:sub);
    if(rem<0) then trap(minus) endif;
  endmethod;
  ...
endtype;
```

ここでは、 same は apple のことである。インタフェースと sub 以外のメソッドは、 int のものをそのまま使用するため、ここには記述しない。林檎型の内部では same が apple のことになるため、apple と互換性のある型だけが引数になれる。従って、時刻型が int と互換であったとしても、引数になることはできない。

same型は、型定義による互換な継承以外では、スーパークラスの型を示す。

6. おわりに

Ｏｎｄｉｎｅは、現在もまだ言語仕様の改良を進めており、多くの未解決部分を含んでいる。また、計画全体が非常に実験的であるため、言語として実用となるまでは長い道のりとなろう。

並列処理方式とオブジェクト指向言語の関係については、後日改めて報告したいと考えている。

現在、Ｏｎｄｉｎｅは VAX-11/785 の UNIX (4.2 BSD) 上で開発中である（ＵＮＩＸはＡＴ＆Ｔベル研究所の商標である）。

参考文献

[Dahl 68] O-J. Dahl, K. Nygaard : Class and subclass declarations, IFIP Working Conference on Simulation Programming Languages, (Oslo, May 1967), North-Holland Publishing Company (1968)

[Goldberg 83] A. Goldberg, D. Robson : SMALLTALK-80 The Language and Its Inplementation, Addison-Wesley (1983)

[荻原86] 荻原剛志：構造的な抽象化機能を持つオブジェクト指向言語Ｏｎｄｉｎｅの概要, 第33回情報処理学会全国大会, 5D-5 (1986)

[木村82] 木村泉, 米澤明憲：算法表現論, 岩波講座 情報科学-12, 岩波書店 (1982)

[久野86] 久野靖：プログラム言語Mistyの試作版処理系と使用経験, ＷＯＯＣ'86資料

[山本85] 山本喜一, 土居範久：Simulaと Smalltalk-80の比較, (鈴木則久編) オブジェクト指向, pp. 119-132, 共立出版 (1985)

[米澤84] 米澤明憲：オブジェクト指向型言語について, コンピュータソフトウェア, 1-1, 29-41 (1984)