プログラミング-言語・基礎・実践 14－4
（1993．10．29）

状態抽象：モジュール化のもう１つの可能性

久野　靖

筑波大学大学院経営システム科学専攻

現在のプログラム言語におけるモジュール化では、モジュールの内部状態を隠蔽しているが、どの操作をどの時点で呼び出しても構わないというわけには行かないので、その意味ではすべての状態を隠蔽してしまうのは適切でない。しかし内部状態をそのまま公開することは望ましくない。本稿では内部状態とは別個にモジュール開発者が「抽象的な状態」を規定し、それに基づいて操作適用の可否を指示することを提案する。これにより内部の詳細に立ちいることなく操作適用の順序を制御でき、翻訳時や実行時の検査に役立てられる。さらに、並行プログラムにおいて抽象状態を実行主体間の同期条件として利用することも検討し、いくつかの例題を示した。

State Abstraction: Yet Another Possibility of Modules

Yasushi Kuno

Graduate School of Systems Management,
The University of Tsukuba, Tokyo
3-29-1 Ootsuka, Bunkyo-ku, Tokyo 152 JAPAN

Module facility of modern programming languages encapusulates internal states of module data structures from outside, but some of the states still must be dealt with from module clients. In this paper, I propose to associate "abstract state" with modules and include in the module interfaces information about which operation is allowed on which abstract state. Such facility enables expression of constraints on operation sequences without going into internal details, and is useful for compile- and run-time validity checking. Possibility of using abstract state constraints for synchronization description of concurrent programs is also discussed.

0 はじめに

現在のプログラム言語において、モジュール機構の位置づけはとても重要である。そして、プログラム言語の進歩につれてモジュール機構の機能自体も単純な名前空間の分離から情報隠ぺい (information hiding ないし encapsulasion)、データ抽象 (data abstraction) と進歩してきた。これら一連の進歩は一貫してモジュール内部の状態は外部に見せないという方針に基づいている。しかし実際にモジュールの提供する機能を外部から利用する際、その内部状態によっては正しく動作しないという制約が常に存在する。そのような状態情報は現存の言語においてはモジュールのインタフェースの一環としては利用可能でなく、文書の一部として記され、モジュールを利用するプログラマが自律的に従うことによって制約を充足していた。本稿ではこのような状況を改善する手段として、状態抽象 (state abstraction) の概念を提案し、それを言語および処理系に組み込むことによってプログラマの負担を減らし、誤りの少ないプログラムを作成しやすくすることの可能性について検討する。

以下第1節では現存の言語におけるモジュール機構とその機能についてまとめ、状態の扱いについて欠けている点を指摘する。第2節ではモジュール機構に組み込むべき新しい機能である状態抽象について説明する。第3節では状態抽象の言語への組み込みと対応する言語処理系の機能について検討する。第4節では並列プログラムに対する状態抽象の適用について検討する。最後に第5節において議論とまとめを行う。

1 モジュール機構の機能と意義

Euclid、Modula などモジュール機構を導入した初期のプログラム言語においては、モジュールを取り入れることの目的は次の2つであった。

(a) 情報隠ぺい — モジュール内部のデータ構造のアクセスをモジュールに包含される手続きからのみに限定する。これにより、プログラム要素間の不必要な依存関係を排除し、データ構造の正しさを保証しやすくできる。

(b) 部品化 — モジュール内部の名前の使用を局所的なものとし、外部からはモジュール名を介して参照できるようにする。これにより、多数のモジュールを名前の衝突の心配なしに多人数で並行して開発できるようになり、また一度作成したモジュールを他のアプリケーションにおいて再利用することも容易になる。

その後、型とモジュールの概念を統合することにより、抽象データ型 (abstract data types) の概念が提唱された。

(c) 抽象データ型 — 型をデータ構造から分離し、提供する機能 (操作群) の集まりのみによって特徴づけられるものと考える。これにより、多様な機能を持つデータ構造を単一の「値」として取り扱うことができ、プログラムが概念的に簡潔に記述できる。

もちろん、データ抽象の機能には情報隠ぺいや部品化も包含されている。

これらと並行して、Simula にはじまるオブジェクト指向言語におけるモジュール (クラス) の考え方があった。オブジェクト指向においては、クラスはモデル化の手段であり、現実世界の対象に対応するものとして考えられる。また、現実世界の対象の包含関係に対応したクラス間の包含関係 (サブクラス機能) や、包含関係にあるクラス間で操作のコードを共有する機構 (継承機構) を持つことが一般的である。プログラム言語の機構として見た場合、クラスは情報隠ぺいや部品化の機能も提供しており、またクラスを型に対応するものとして考えた場合、抽象データ型の機能も持つと見なせる (ただしオブジェクト指向言語には強い型を持たないものも多い)。

これらの流れとは別に、並行プログラミングの世界では並行性の制御の手段としてのモジュール機構という考え方が古くから存在する。

(d) モニタ、並行オブジェクト — モジュール内部においては実行の流れ (スレッド) を一時に1つだけに限定することにより、データ構造への順次的なアクセスを可能にしその一貫性を保証しやすくする。

例えば Hoare と Brinch Hansen のモニタはその内部のデータ構造に排他的にアクセスすることを目的としたモジュール機構である。また Actor、ABCL/1、PLASMA などの並行オブジェクト指向言語ではモ

ジュールの各実体をアクタと呼ぶ。アクタはモニタと同様、その内部は一時に1つの実行の流れしか持たない(一部そうでないものもある)が、基本的に個々のアクタが能動的な並行実行の単位である点がモニタと異なっている。

これらいずれのアプローチにおいても、モジュール機構を外部から参照する際のインタフェースには状態の概念が含まれていない。すなわち、モジュール(抽象データ型ないしオブジェクト)の状態がどのようである場合にどのような操作が適用できるかはモジュール外部から隠されている。モニタやアクタではその目的上、操作呼び出しやメッセージを内部状態に応じて一時保留するなどの機能が提供されるが、それはあくまでも呼び側からは見えないように(透明に)行われる。

もちろん、内部状態の細部を隠ぺいすることはモジュールを用いる主要な目的の1つであり、このような状況はある面では納得できる。しかし、すべての状態を外部から隠ぺいするのが正しいかどうかについては議論の余地があると思われる。例えば入力ファイルに対応するモジュールを考えると、その主要な操作はデータを読み込むことであるが、ファイルの終端まで到達した状態ではそれ以上読むことは不可能である。またスタックを考えた場合、その主要な操作の1つであるpopはスタックが空であれば実行できない。以下本稿ではこのように、ある状態において適用できない操作を呼んでしまうことを本稿では状態不適合(state mismatch)と呼ぶ。

現在のところ、既存の言語において状態不適合に対処する方法としては次の2つがある。

(x) 現在の状態を調べる操作(is_eof、is_emptyなど)を用意し、それらによって現状態を絶えず調べながらモジュールを利用する。

(y) 現在の状態において適用可能でない操作が呼び出された時には「失敗」させ、例外機構やフラグなどを用いて処理する。

いずれのやり方も必ずしも望ましいものではない。絶えず状態を調べるやり方は、手続き呼び出しの回数が増加し効率上望ましくない上、コードが煩雑になりやすい。フラグを用いる場合は呼び出し回数は増えないがコードの煩雑さはむしろ大きくなりがちである。例外機構はこれらの中ではもっともデメリットが少ないが、例外機構を備えていない言語も多い。そしていずれの方法も実行時における検査である以上、状態不適合を予め(翻訳時に)検出することは本質的に不可能である。

一方、実はこれらとは別の、そしてしばしば取られる方策として次のものがある。

(z) 状態不適合が起こらないようにプログラムを設計し、それ以降は状態不適合の存在を無視する。

これはすなわち、「ここではスタックは2段以上積まれている『はず』だから…」「ファイルにはまだ3行以上残っている『はず』だから…」といった設計上の判断に基づいてプログラムを書くことを意味する。これはもしその判断が正しければ最も効率が良いことは明らかである。しかし判断が間違っていれば(あるいは後日の改訂によってその判断が正しくなくなるようなことがあれば)悲惨な結果をもたらす。

このように考えると、現在のモジュール機構が状態について明示的に扱わないことはいくつもの問題点を生じさせているように思える。以下本稿ではこの点を改良するようなプログラム言語上の可能性について検討していく。

2 状態抽象

データ抽象の主要なアイデアは、型の性質としてその型の値が提供する操作群のみを考え、実際にそれを実現するデータ構造からは分離したところにある。このため、内部実現がどんなに複雑な構造や機能を持っていても、外部からは提供された操作を通した挙動しか観察されない。従って、モジュール内部の複雑さは外部からは観測されず、そのモジュールの外部にいるプログラマにとって扱わなければならない複雑さが低減されることになる。言い替えれば、モジュールの内部と外部で複雑さにギャップを生じさせることが重要だということになる。

ここで前節で挙げた状態不適合の問題を考える。抽象データ型の値ないし実体の状態は、外部からは隠ぺいされた内部構造の値によって規定されている。しかし状態不適合が存在するということは、状態のうち少なくともある部分はモジュールの利用者にとっても意識されなければならないことを示している。その「ある部分」とは内部構造のうち特定の欄の値かも知れないし、複数の欄の値の関係について計算をすることで定まるようなものかも知れない。

しかし、これらの欄の値を外部から直接参照可能にすることは、特定の内部構造の知識を外部に公開してしまい、内部実現の変更が困難になるため好ましくない。前節の方法(x)はどの状態にいるかという情報をこのような問題を生じさせずに外部から参照させるためだけに用意された操作ということになる。ただし、(x)の方法における問題点は前述の通りである。そこで、データ抽象において外部に提供される操作とその内部構造の上での対応を分離したのと同様に、抽象データ型の値が持つ、外部から見える状態(abstract state)を内部構造とは分離したものとして用意する。これを状態抽象(state abstraction)と呼ぶ。

例えば、整数値を積むことの出来る型intstackを考えると、その状態として

```
{empty, nonempty}
```

が考えられる(ここでは記憶領域は主記憶がある限り増やせることにして、オーバーフローは考えない)。実際には例えばemptyかどうかは内部的には実現によって「先頭要素を指す添字が0である」(配列による実現の場合)とか「先頭セルを指すポインタがnilである(リストによる実現の場合)などに対応しているのだが、そのような詳細は外部からは観測されない。

次に、抽象データ型に備わる各操作に対して、その操作がどの状態で適用可能であり、その結果としてどの状態が生じるかを規定する。これを状態仕様(state signature)と呼ぶ。状態仕様は通常の界面の仕様と一緒に記述するのが素直であろう。以下ではMisty[5]ふうの構文を適宜拡張して実例に用いる。

```
intstack = class spec
  states {empty, nonempty}  … ★
  defines new, push, pop, top
 new = proc() returns(self{empty})
 push = proc(x:{*}self{nonempty}, i:int)
 pop = proc(x:{nonempty}self{*})
     returns(int)
 top = proc(x:{nonempty}self{-})
     returns(int)
end intstack
```

まず★でintstackの値には2つの抽象状態emptyとnonemptyがあることを宣言している。そして各操作仕様においては呼び出し時や戻り時の抽象状態を引数や戻り値の型指定の前後に{}で囲んで記した。({*}は任意の状態を表し、{-}はその引数の状態を変更しないことを表す。また{~ …}は「…以外の状態」を意味する。)

例えばnewによってintstackが作られた時にはその状態はemptyである。pushは引数として受け取るintstackの状態は問わないが、実行後にはその状態はnonemptyとなる。一方popが呼べるのはintstackがnonemptyの時に限られ、その結果としてintstackの状態はemptyになることもnonemptyになることもある。topについては呼ぶ時の条件は同じであるが、実行後もintstackの状態は元と変わらないことを意味する。

次に、intstackを用いて2つの変数の値を交換するコードを書いたものとする。

```
x, y: int ...
...
s:intstack := intstack!new()
s!push(x); s!push(y); s!pop(x); s!pop(y)
```

ここでコンパイラはintstack!newにより返された新しいintstackの状態はemptyであることが分かる。そこで、非引き続く操作がpushであれば、これは問題なく適用可能である。もし間違って最初がpopであれば、これはコンパイラによってエラーとして検出できる。2番目のpush、3番目のpopまでは同様に、翻訳時に状態を検査できるので実行時の検査は不要である。

なお、ここで4番目のpopは3番目のpopの結果状態がemptyである可能性が生じるため実行時の状態検査が必要となる。これは状態として「いくつ積まれているか」を厳密に把握するようにすれば避けられるが、プログラマの大きな負担となることなしに、明らかな間違いを早期に発見し、できる範囲で検査を省くことを目指したため複雑な状態記述は避けている。

ところで、翻訳時に状態不適合を検出する他の方法として、IOTA、OBJ2、Larchなどに見られるように抽象データ型の上に代数的仕様記述を構築し、証明器によって操作適用の意味付けを追跡するものがある。しかし、自動的な証明付与は必ずしも容易でなく、この方法は現在のところ広範囲に用いられてはいない。これに対し、本稿のやり方では状態の情報のうちで外部から参照することが有効である(と作成者が判断した)部分だけを公開するため、コンパイラなどによる扱いも容易である。

また、CLU[2]やEiffel[3]などの言語では、抽象データ型(クラスタやクラス)の操作に付随して事前条件(precondition)、事後条件(postcondition)などを指定できるようになっている。このうち事前条件の方はその操作が適用可能であることを検査するものなので、ここで言う状態不適合の検出に利用できる。ただし、その検査は抽象データ型の内部表現を参照した論理式によるので、呼び側で外部から事前条件への適合の可否を知ることはできない(引数の値などに関する制約はこの限りではないが、それだけではあまり意味はない)。そして、検査自体は常に実行時に行なわれることになる。

3 プログラム言語における状態抽象機能

本節では前節で述べたような機能をプログラム言語に導入する際の諸側面について検討する。まず、状態抽象を言語機能として実現する際必要なことがらは基本的には次のように整理できる。

(1) 各型に対し、その値が持つ抽象状態を宣言する。

(2) 型の各操作に対し、抽象状態の遷移を宣言する。

(3) 翻訳時に値の状態をできる範囲で追跡し、状態不適合を検出する。

(4) (3)で検出できない操作呼び出しについては実行時に状態検査を行なう。

(5) 各型の実現においても実行時に現在の状態を設定する機能を用意する。

これらのうち(1)～(4)については既に前節で例を挙げたので、ここでは(5)をどのように設計するか考察する。ここで、次の2つのやり方が考えられる。

(a) 各抽象状態ごとの成否を論理式として与える。

(b) 列挙型の値のような形で内部表現中に蓄える。

これらのうち(a)は操作適用の可否を事前条件のような形で与えるという考え方によく合致する。一方、状態機械のようなものをプログラムとして実現するような場合には、状態が内部表現の特定の条件と必ずしも対応していないので、(b)の方がなじみがよい。どのみち、(a)の表現と(b)の表現の間で行き来するのは容易であるので、型の界面としても内部実現としても両方を用意することを考えてみる。

まず型の界面としては、次のようにする。

(a1) 各抽象状態xxxごとに操作is_xxxを呼ぶ。

(b1) 各型は操作get_stateを持つものとする。

いずれの操作も引数はその型の値1つだけとなる。is_xxxについてはその返値は論理型であればよい。一方、get_stateの返値は何であるべきだろうか。選択肢としては、Pascalの列挙型のようにある型の状態値のみから成る型(つまり型ごとに別の状態値型を用意する)と、LispやSmalltalk-80の記号型のようにすべての状態値を包含した型の2通りがある。ここでは当面簡単のため後者を前提としておく。

次に、実現について考える。これも先と同様、

(a2) 各抽象状態xxxごとに操作is_xxxを作る。

(b2) 操作get_stateを作る。

の2通りが考えられる。そして、(a2)か(b2)のいずれかを与えれば他方は自動的に用意されるものとする。例えばintstackで(a2)は次の通り。

```
intstack = class body
  rep = array[int]
  is_empty = proc(x:cvt) returns(bool)
    return(rep$size(x) = 0)
  end is_empty
  is_notempty = proc(x:cvt) returns(bool)
    return(rep$size(x) ~= 0)
  end is_notempty
  ...
end intstack
```

一方、(b2)による場合には「状態を保持することだけを目的とした」型生成子statesを用意するものとする。すなわち、型states [名前1, 名前2, …]は操作be_名前$_i$とis_名前$_i$を持ち、現在どの名前の状態にあるかを記憶するものとする。各クラスでは状態をこの型によって記憶させることにして、操作is_xxx、get_stateをこの型に委譲して実現することができる。

```
intstack = class body
  st = states[empty, notempty]
  rep = record{a:array[int], s:st}
  use ref.s for
    get_state, is_empty, is_notempty
  ...
end intstack
```

state型のようなものを用いる代わりに、Pascalの列挙型のようなものを用いてその値を内部表現の一部に格納することで状態を表すこともできる。(ある

意味では、列挙型は書き換え不能な状態型のようなものだと見ることもできる。)

4 並行プログラムへの適用

ここまでに見てきた直列言語における状態抽象の場合、状態不適合が発見された場合にはそれは例外的な場合として検出し、実行を中止するかまたは修復動作を試みることが前提となっていた。

しかし、並行プログラムの場合には、これとは異なる方策を取ることができる。すなわち、ある実行主体があるオブジェクトの操作を呼ぼうとして状態不適合が発見された場合、その呼び出しを単に遅らせて待機することで、その間に別の実行主体がそのオブジェクトの別の操作を実行することにより不適合を解消してくれる可能性が存在するからである。例えば有限バッファを例にとって考える。

```
class bbuffer
  states {empty, mid, full}
  defines new, put, get
 new = proc(s:int) returns(self{empty})
 put = proc(x:{~full}self{~empty}, i:int)
 get = proc(x:{~empty}self{~full})
     returns(int)
end bbuffer
```

これを参照する実行主体は例えば次のようになろう。

```
producer = proc(..., b:bbuffer)
  while ... do
    % produce value into i
    b!put(i)
  end
end producer

consumer = proc(..., b:bbuffer)
  while ... do
    i:int := b!get()
    % consume i
  end
end consumer
```

ここで生成者においては、b!put を呼ぼうとした時に b の状態が full であればその操作は「呼べない」ので生成者の実行全体が待たされる。その間に消費者が b!get を実行してくれれば b の状態は full でなくなるので、生成者の実行は再開できることになる。

この種の問題は多くの言語設計者によってアプローチされているが、そのやり方は基本的に次のいずれかである。

(a) get や put などの操作はいつでも呼ぶことができ、その実行の内部で状態不適合を発見して待ち合わせを行なう機構を用意する。

(b) 操作の受け付け部に特殊な機構を設け、ある操作を呼べたり呼べなくしたり選択する。

(a) のアプローチは Hoare や Brinch Hansen のモニタや、Argus など 1 つのオブジェクト内部で複数の実行の系列を許す言語のアプローチである。(b) には Ada、Occam などガード機構を持つ言語、および sequencer など特殊なオブジェクトによってオブジェクト本体のメッセージ受理を制御する機構を備えた並行オブジェクト言語などが相当する。

これらのいずれもが、どのような場合に呼び側の実行が (結果的に) 待たされるかを呼ばれ側だけで管理するため、前に述べた呼び側での制御が行なえないという弱点を持つ。例えば複数のバッファのうち現在書き込み可能などれか 1 つに書く、というような制御はこれらのモデルでは行なえない。(Ada のようにランデブ型の言語で選択的な複数エントリ呼び出しを許すならば可能であるが、現在の Ada はそうなっていない。)

状態抽象の場合は、ある操作が呼べるかどうかは呼ばれ側でしか評価しようのない論理式ではなく抽象的な状態として外部に公開されているので、呼び側でこれを参照して選択することを認めるのはごく自然である。例えば次のような構文を導入することでこれを行なうことができる。

```
select b1!put(...); i := 1
or     b2!put(...); i := 2
end

foronce k in 1 to n do
  b[k]!put(...); i := k
end
```

このように、並行プログラムにおいては抽象状態を同期の記述手段として使うことができるのは、そもそも同期を取ることの目的の 1 つが操作対象のオブジェクトに対して各操作を不適当な時期に実行しないためであることを考えれば、当然であるといえる。さらに進んで考えると、あるオブジェクトに

対する一連の操作が各実行主体によって不可分に行なわれなければならないような場合にもこれを抽象状態によって記述することができる。例えば次の例を考える。

```
class account
  states {normal, excl}
  defines open, close,
          deposit, withdraw,peek
  open = proc(x:{nomal}self{excl})
    returns(key{valid})
  close = proc(x:{excl}self{normal},
    a:{valid}key{invalid})
  deposit = proc(x:{excl}self{-},
    a:{valid}key{-}, m:int)
  withdraw = proc(x:{excl}self{-},
    a:{valid}key{-}, m:int)
  peek = proc(x:{normal}self{-})
    returns(int)
end account
```

すなわち、アカウントを変更のため open すると、そのアカウントは excl になり、それ以降の open や peek に対する呼び出しは待たされる。open した実行の系列は open で返される鍵を利用してのみアカウントを変更できる。これを参照するコードの断片は次の通り。

```
  ...
  k:key := a!open()
  ...
  a!deposit(k, ...)
  a!withdraw(k, ...)
  ...
  a!close(k)
```

なお、ここで目指しているのはあくまでも誤ったコーディングを防ぐことなので、別途 valid な鍵を入手してアクセスするような場合には対処していない。

もう１つの例として哲学者の食事を挙げておく。

```
phil = class body
  ...
  life = proc(x:self, l,r:fork)
    while ... do
      self!think(...)
      fork!pick_both(l, r)
      self!eat(...)
      fork!release_both(l, r)
    end
  end life
end phil
```

fork の界面は次の通り。

```
fork = class spec
  defines pick_both, release_both
  pick_both = proc(l:{down}self{up},
                   r:{down}self{up})
  release_both = proc(l:{up}self{down})
                      r:{up}self{down})
end fork
```

すなわち、本稿で提案する方式の場合、操作の入口点で引数となっている全オブジェクトの状態を同期させることができるので、複数のセマフォのような工夫は必要ない。見かたを変えればこれは至る所にある操作呼び出しを Commited-Choice 型言語のガード機構や DinnerBell の neck 同期機構のように働かせられるものと言える (ただしこれらの言語では同期においてオブジェクト内部の状態を参照できるが、そうでない方がかえって望ましいことは既に述べた。)

類似の研究として、ActorSpace モデル [4] ではメッセージ送信の際送り手が複数の受け手の属性に対するパターンマッチによって実際の受け手を定めることができ、また synchronizer と呼ばれる特殊なアクタにより一般のアクタのメッセージ受理の可否を制御できる。これらの機構を用いれば本節で挙げた抽象状態と同様の制御を行なうことができるが、ただし属性をアクタの状態を抽象化したものとして捉えてはいず、メッセージ受理の可否の判定も直接アクタの内部状態を参照することが基本になっている。

5 議論とまとめ

本稿では、抽象データ型ないしそれに相当するモジュール機構に対し、そのモジュールが実現している値の「抽象的な状態」を内部表現とは分離して扱うことを提案した。これによって、モジュール操作を正しくない順序で呼んだためにそれ自身としては完成しているモジュールが適切に動作しないという誤りを防止することができる。検査はコンパイラによって静的に行なえる部分は翻訳時に、それ以外は実行時に行なうことになる。この方式を適用する場合、モジュール作成者は各操作の界面として各引数のあるべき抽象状態を記述し、また内部表現や呼び出し系列と抽象状態の対応関係を記述することになる。モジュール利用者やコンパイラは界面の記述の

みを参照して正しい呼び出し系列を知ることができる。

同様の事柄をめざした別のやり方として、操作ごとに事前条件を記述させ実行時に検査する方法や、代数的仕様記述と証明器によって不適切な呼び出しを検出する方法がある。しかし、前者は実行時のみにしか検査が行なえず、また条件の大部分は内部表現に依存したものとなりそのモジュールをブラックボックスとして扱いたい利用者には適さない。後者はモジュール作成者にとっても利用者にとっても記述が繁雑であり、証明の手間などを考えると大規模な実用には現在のところ適さない。

一方、並行プログラムについて考えた場合、操作呼び出し時に値の状態が期待されるものと一致しないことは全面的な誤りというよりは、同期の必要性を示唆する場合が多いと考えられる。そこでこれを積極的に同期機構として利用し、操作呼び出し時に抽象状態が適合しない場合には待ち合わせを行なう方式を提案するとともに、この記法によるいくつかの例題記述を示した。

並行プログラムにおける同期機構は、大きく分けてセマフォのように実行主体の中断 / 再開を明示的に制御するようなものと、CSP やアクタのようにメッセージを主体としたものとがあり、データ抽象 / オブジェクト指向のモデルには後者の方がなじみがよい。そのようなモデルにおいては単に送られたメッセージを順に受け取るだけでは記述力が不足しており、そのためメッセージを選択的に受信するような機構が多く用いられる。しかし、受け取り側の情報に基づいた選択機構では上述の事前条件と同様の問題がある上、送り側で非決定的な選択を行なう余地がない。また、複数の値に関わる同期を記述することも直接的ではない。これに対し、抽象状態の一致を同期機構として用いた場合、送り側での非決定的な選択や複数の値に関わる同期などもごく自然に記述できる。

最後に、本稿で述べた事柄はまだ基本的なアイデアの段階であり、やるべきことは多く残っている。具体的な課題としては例えば次のものが考えられる。

- 継承 / 副型との関係 — ある型の値に対して抽象状態を規定した場合、その副型の値はそれと同じか、またはそれを細分化した (上位互換性を持つ) 状態群を持つべきだと思われるが、検討が十分でない。また多重継承の場合も考慮するべきである。
- 実装と経験 — もちろん、実際にこの機構を実装した言語と処理系において経験を積む必要がある。その場合、例えば既存の言語に皮をかぶせる (抽象状態の記述と翻訳時 / 実行時検査を追加する) ことも検討してみたい。
- 評価 — 既存のコードをこの方式で書き直した場合、どれくらいよくなるかどうかも定量的に評価してみたい。

参考文献

[1] Mark Cashman: Edicates — A Specification of Calling Sequences, SIGPLAN Notices, vol. 28, no. 8, pp. 77-80, 1993.

[2] Barbara Liskov, John Guttag: Abstractions and Specification in Program Development, MIT Press, 1986.

[3] Bertrand Meyer: Object-Oriented Software Construction, Interactive Software Engineering, 1988. (酒匂訳: オブジェクト指向入門, アスキー, 1990)

[4] Gul Agha, Svend Frølund, Woo Young Kim, Rajendra Panwar, Anna Patterson, Daniel Sturman: Abstraction and Modularity Mechanisims for Concurrent Computing, IEEE Parallel and Distributed Technology, vol. 1, no. 2, pp. 3-14, 1993.

[5] 久野　靖: 多重継承と強い型付けを持つオブジェクト指向言語 Misty, コンピュータソフトウェア, vol. 6, no. 3, pp. 9-18, 1989.