プログラミング-言語・基礎・実践- 16－3
(1994. 3. 9)

並行言語 Harmony/2 とその一級継続機構

脇田建
東京工業大学理学部情報科学科
〒１５２東京都目黒区大岡山２－１２－１

あらまし

メッセージを介した通信によって情報を交換する分散計算モデルで、メッセージを一級のオブジェクトとして扱うことによって、広いクラスの情報交換の形態を記述できる計算の枠組みを提案する。一級メッセージに対する操作の意味はメッセージの同一性という概念に基づいて設計された。提案された枠組みに基づいて設計された並行プログラミング言語 Harmony/2 を紹介する。計算のモデルは非同期的メッセージ送信と同期的メッセージ送信をともに備えた、single-thread のオブジェクト指向並行計算モデルである。Harmony/2 を用いることで並行計算で必要とされる、さまざまな通信形態を容易に記述できることが確認された。

和文キーワード　分散プログラミング、オブジェクト指向並行計算、通信形態記述、一級継続、一級メッセージ

A Concurrent Language Harmony/2 and Its First Class Continuation Mechanism

WAKITA Ken
Department of Information Science
Tokyo Institute of Technology
2-12-1 Oh-okayama, Meguro-ku, Tokyo 152 Japan

Abstract

Distributed computation framework is proposed, where message can be treated as first class objects in a message-based distributed computation model. First class messages allows the programmer to describe wide variety of communication scheme. A concurrent programming language, Harmony/2, is designed based on this computatio model. Harmony/2 is based on a single-thread object-oriented concurrent computation model and it incorporates both asynchronous and synchronous message passing.

英文 key words　Distributed Programming, Concurrent Computation, Communication, 1st Class Message

1 はじめに

分散環境で動作するアプリケーションは、アプリケーション自身に並行性が内在こと、プログラムのモジュール間での情報交換ためにネットワークを介した通信を行うこと、さまざまな要因による実行の非決定性、などの特徴がある。これらの点は、分散環境におけるアプリケーション開発を、普通の非分散環境下で逐次プログラミング言語を用いた場合に比べて困難なものにしている。したがって、分散環境におけるプログラミングを効率的に行う上では、通常の逐次プログラミング言語ではなく、上述の分散環境における計算の特徴を念頭に設計された言語、つまり分散プログラミング言語を用いることが望ましい。

ところで、分散プログラミング言語の設計においては従来のプログラミング言語を設計する際に考慮された点に加えて、つぎの2点が重要であるとされている[3]:(1)いかに、並行実行単位の動作をプログラムで記述するか、(2)いかに並行に動作するプログラム間での情報交換(広義の通信)を記述するか。後者の情報交換は(2．1)情報交換の手段と(2．2)情報の送り手と受けての待合せ(同期)の手段をともに含む。

分散プログラミングにおいては、アプリケーションの実行形態からの要請によって、さまざまな情報交換の形態が必要である。また、プログラミング技法の観点からも、アプリケーションに内在する並行性を抽出して、実行効率を向上するためにも、プログラミング言語が柔軟な情報交換のためのプリミティブを提供することが必須である。

情報交換のためのプリミティブとして、分散共有記憶を仮定した言語モデルでは semaphore, monitor, path expression などの同期機構を利用して、共有記憶を用いた情報交換の際の並行性を制御する方法が提案された。共有記憶を仮定しない言語モデルでは、遠隔手続き呼びだし、rendezvous などのさまざまな形態のメッセージ通信による情報交換の方法が提案されてきた。これらの情報交換のプリミティブの記述性は強力ではあるが、アプリケーションが求めるさまざまな高度な通信形態を実現するためには、プログラムの中でこれらの情報交換のプリミティブを複雑に駆使する必要があり、(1)プログラムの中でアプリケーションの機能面の記述と混在する、(2)ひとつの情報交換の形態を実現するコードがその情報交換に携わるすべてのコードに散在するため、プログラムの可読性を著しく損なう、(3)情報交換形態を実現するコードを再利用することを難しくするなどの問題点がある。

これらの問題の背景には、プログラミング言語が提供する情報交換のためのプリミティブが、アプリケーションの通信形態に比べて抽象度が低いこと、さらに情報交換のプリミティブを組み合わせて新たなプリミティブを作るような各調整を言語が提供していないことがある。これらの問題の本質的な解決のためには、分散プログラミング言語の中で、情報交換の形態を実現する部分を他のプログラム部分から切り放して、独立に記述できるようにすること、さらにその記述によって実現される情報交換の形態をプログラムの各部から容易に利用できるようにすることが必要である。

[7]において、multi-thread のオブジェクト指向並行計算モデルにおいてメッセージを一級オブジェクトとして扱うことで、さまざまな通信形態を表現できることを示した。本稿では、同じ一級メッセージの概念を single-thread のオブジェクト指向並行計算モデルにも導入できることを示し、そのモデル、モデルに基づいた並行プログラミング言語について説明し、いくつかのプログラム例を紹介する。本稿において紹介する並行プログラミング言語 Harmony/2 のプロトタイプはすでに完成しており、さまざまな記述実験のために用いられている。

本稿の残りの構成は、以下の通りである。2節でオブジェクト指向並行計算とその中の single-thread のモデルにもとづいた並行プログラミング言語 Harmony/2 の基本的な計算モデルについて説明する。3節では、Harmony/2 でもっとも特徴的な一級メッセージの機能の概要を述べ、4節でメッセージの同一性という概念を導入し、それに基づいた厳密な議論をし、5節では一級メッセージの多相的な扱いについて述べる。6節で本稿をまとめる。

2 基本計算モデル

Harmony/2 は、オブジェクト指向並行計算モデルにもとづく分散プログラミング言語である。ひとくちにオブジェクト指向並行計算モデルといっても、いろいろな流儀がある。ここでは、まずオブジェクト指向並行計算モデル一般について述べ(2.1節)、そのあとで Harmony/2 の計算モデルを Harmony/2 の言語の紹介とともにする(2.2節)。

2.1 オブジェクト指向並行計算モデル

オブジェクト指向並行計算モデルは、大域的な名前空間を仮定しない分散計算のモデルの一種である。オブジェクトと呼ばれる状態を持った実行単位によってシステム全体が構成される点は、Smalltalk-80 や C++ に代表される逐次オブジェクト指向言語と同様である。それぞれのオブジェクトは固有の局所状態変数群(インスタンス

変数）と局所手続き群（メソッド）を持つ。オブジェクトの状態はインスタンス変数への値の束縛で表現され、インスタンス変数への値の代入がオブジェクトの状態変化に相当する。逐次オブジェクト指向言語と同様に、オブジェクトのインスタンス変数は他のオブジェクトから隠蔽されている。インスタンス変数の値の参照や変更は、インスタンス変数を所持しているオブジェクトにおけるメソッドの実行にのみが許されている。

他のオブジェクトは、このオブジェクトと通信することによって間接的にインスタンス変数の値の参照や変更ができる。オブジェクト同士はメッセージを介して、1対1の通信を行う。メッセージを送ったオブジェクト（メッセージの送信者、あるいは単に送信者）は、メッセージ送信の際に、メッセージの行き先（メッセージの受信者、あるいは単に受信者）、受信者で起動するメソッド（メソッドセレクタ）とそのメソッドに与える実引数を指定する。メッセージが届くと受信者はメッセージのメソッドセレクタに対応したメソッドを起動する。メッセージの引数はメソッド起動時にメソッドの実引数として渡される。

メッセージが受信者に届いた場合の処理は、言語ごとに大きく2通りある。multi-thread の言語では、メッセージは受信者に届くとすぐにメソッドセレクタに対応したメソッドを起動する (POOL/T, Sina/ST[2, 1])。したがって、受信者オブジェクトの内部での複数のメソッドが同時に並行実行する可能性がある、すなわち並行性がある。single-thread のモデルでは、オブジェクトに届いたメッセージを一つずつ処理することによりオブジェクト内部のメソッド実行の並行性を排除している (ABCL/1, Concurrent Smalltalk, Hybrid, Harmony/2[10, 9, 5])。

メソッドの実行中には、インスタンス変数の参照と変更、メッセージの送信、メソッドが処理中のメッセージに対する。新しいオブジェクトの生成などができる[1]。

メッセージの送信形態にはさまざまな提案がある。代表的なものが、非同期的メッセージ送信と同期的メッセージ送信である。非同期的メッセージ送信では、送信者はメッセージの送信後すぐにメソッドの実行を続けることができる。やがて、メッセージは受信者側でメソッドを起動するが、このメソッドは送信者側のメソッドと並行に実行する。したがって、非同期的メッセージ送信を用いることによりシステム内の並行性が増す。同期的メッセージ送信は、メソッドを通常の手続きとみなせば、逐次言語の手続き呼び出しに似ている。メッセージを送信すると送信者は受信者におけるメソッド実行からの返答を待ち続ける。返答の扱いとしては、メソッド実行の最後の文の値を返答値とするものと、明示的に返答を返すものがある (early reply)。後者の場合、返答後もメソッドの実行を続けられるため、Ada の rendezvous で実現される送受信者間の並行性を表現できる。

この二つのメッセージ送信をともに提供する言語も多い。ABCL/1 と Concurrent Smalltalk は、これに加えて、非同期的メッセージ送信と同期的メッセージ送信の中間的な未来型メッセージ送信と呼ばれるものを持っている。未来型メッセージ通信については、3.3節で詳しく述べる。

2.2 オブジェクト指向並行言語 Harmony/2

Harmony/2 は、東京工業大学理学部情報科学科で設計と開発が行われているオブジェクト指向並行言語である。Lisp の1方言である Scheme で実装されており、字面も Scheme のオブジェクト指向的な拡張である。計算モデルは、single-thread であり、同期的メッセージ送信と非同期的メッセージ送信をともに持っている[2]。

オブジェクトは、クラスと呼ばれる雛型を元に作られる。以下に初期値0のカウンタの定義の例を示す。

```
[class Counter (count)
    [initialize () [count := 0]]
    [method (tick) [count := (+ count 1)]]
    [method (value) ^count]
    [method (set! c) [count := c]]]
```

この Harmony/2 の式、[class Counter ...] は Counter という名前のクラスを定義している。Counter をひな型として作られるオブジェクトは一つのインスタンス変数 (count) と [initialize () ...] で定義される初期化メソッドと、3つのメソッド (tick, value, set!) を持つ。初期化メソッドはオブジェクトが生成されるとすぐに実行される手続きで、この例ではインスタンス変数 count を0に初期化する。

メソッドの定義は Scheme での関数定義で (define ...) の代わりに [method ...] を使っているものと見なせる。Scheme の手続きと同様にメソッドへの引数を宣言することができる。この例では、tick と value は無引数のメソッドで、set! は1引数 (v) のメソッドである。三つのメソッドはそれぞれ、count の値を1増加、count の値をメソッド実行の結果として返答、count に引数 v の値を代入している。tick の定義のようにメソッドの本体には任意の Scheme の式に加えて、以下の Harmony/2 の特殊形式を書くことができる。

[1] 一部のオブジェクト指向並行プログラミング言語は動的なオブジェクトの生成を許さない。

[2] [7] では、multi-thread で、同期的メッセージ送信のみを持った標準 ML を拡張した同名のオブジェクト指向並行言語について議論した。

インスタンス変数の値の参照： *ivar*

インスタンス変数 *ivar* の値を返す。通常のインスタンス変数のほかに *self* という疑似インスタンス変数があり、これはメソッドを実行しているオブジェクトのオブジェクト識別子（ポインタを抽象化したもの）を表す。

インスタンス変数への値の代入： [*ivar* := *value*]

オブジェクト生成： [Create *class* arg_1 ...]

クラス *class* のインスタンスを生成する。この式の評価値は、新しく生成されたインスタンスをさすオブジェクト識別子である。新しく作られたオブジェクトでは、[initialize ...] で定義された初期化メソッドがオブジェクトを生成したメソッド実行に対し非同期的に起動され、arg_1 ... は初期化メソッドへの実引数として渡される。

非同期的メッセージ送信： [*target* <= (*method* arg_1 ...)]

target を評価して得られるオブジェクトに対して非同期的にメッセージを送信する。メッセージのメソッドセレクタとして *method* を、引数に arg_1 ... を指定する。この式の評価値は仕様では定義されていない。

同期的メッセージ送信： [*target* <== (*method* arg_1 ...)]

target を評価して得られるオブジェクトに対して同期的にメッセージを送信する。生成されるメッセージは非同期メッセージ送信と同様である。メソッド実行の途中で同期的にメッセージが送信されると、メソッド実行は一時的に中断し、受信者からの返答を待つ。受信者からの返答値が、この式の評価値になる。

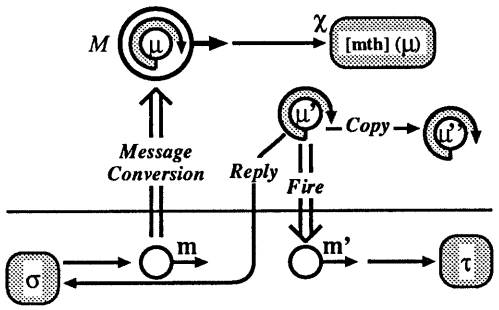
返答： ^*value*

現在のメソッド実行が同期的なメッセージ送信によって起動された場合には、*value* を評価して、その値を返答値としてメッセージの送信者に返す。非同期的メッセージ送信によって起動されたものの場合は、*value* を評価するだけである。

3 一級メッセージ

並列計算機や超並列計算機が市場に出るようになった近年までに、多くのオブジェクト指向並行言語が提案された。これらの中で Harmony/2 をとりわけ独特なものとして特徴づけているのは、メッセージを一級オブジェクトとして扱う機構である。通常の枠組みではモデルに存在する概念的な対象を、プログラミング言語で操作の対象となっていないものを、枠組みを拡張することで操作の対象とすることをその対象の一級化という。

Harmony/2 では、通常は操作の対象となっていないメッセージを一級化し、通常のオブジェクトと同様に操作の対象とすることで、並行計算に現れる通信、同期、並行性制御などを一般に記述することを試みている。



σ:	sender	τ:	target
$m^{(\prime)}$:	message	$\mu^{(\prime)}$:	1st class message
M:	msgcont carrier	χ:	Communicator

図 1: メッセージ変換と一級メッセージ

継続とは逐次プログラムの実行の任意の時点における「残りの計算」を形式化したものである。継続の概念は手続き型プログラミング言語の制御構造の意味づけのために考案された。のちに Scheme などのプログラミング言語のデータ構造として取り入れられ、制御構造を一般的に記述できることが知られている [6, 4, 8]。

Harmony/2 では、メッセージが並行計算において継続と同じ役割を果たす点に着目し、メッセージを一級のオブジェクトとして扱えるようにすることで、並行計算に現れる通信、同期、並行性制御などを一般に記述することを試みている。われわれは、メッセージを媒介にして継続を表現しているという観点から、一級のメッセージをメッセージ継続と呼ぶ。メッセージ継続を導入することで、Harmony/2 では並行性、情報交換、同期などのさまざまな並行計算の形態を表現することができるようになった [7]。

[7] では、multi-thread で同期的なメッセージ送信を持つオブジェクト指向並行言語に一級メッセージを導入した。その後 single-thread で同期的なものに加え、非同期的メッセージ送信を含んだ計算モデルに対しても導入できることが分かった。本稿は、後者のモデルに基づいたメッセージ継続の枠組みについて論ずる。

メッセージ継続の枠組みは基本的にはメッセージを一級オブジェクトとして扱う枠組みである。したがって、全体の構成はメッセージを特定して一級オブジェクトに変換する部分と、一級オブジェクトになったメッセージに対する操作からなる。枠組みを図 1に図解した。

プログラマはクラス定義の中に、特定のメッセージ (m)

を一級メッセージ (μ) に変換するための宣言（メッセージ変換宣言）を記述できる。一級メッセージは、新たに作られたメッセージ (M) の引数として、メッセージ変換宣言によって指示されたオブジェクト (χ) に送られる。M によって起動されるメソッドの中では一級メッセージを自由に操作することができる。一級メッセージに対しては3つの操作、copy, fire, reply が定義されている。これらは、それぞれ一級メッセージの複製 (copy)、一級メッセージからのメッセージの再生 (fire)、メッセージに対する返答の送信 (reply) を行う。以降では、3.1節でメッセージ変換について、3.2節で一級メッセージに対する操作を説明し、3.3節では一級メッセージを用いたプログラミングの例として未来型通信の実現について述べる。

3.1 メッセージ変換

Harmony/2 では、クラス宣言の中でメッセージ変換宣言を用いることで、特定のメッセージ (m) は全て一級オブジェクト (μ) に変換される。この変換をメッセージ**変換**と呼ぶ。一級メッセージは、あとで再びメッセージを再生できるように、そのもととなったメッセージが保持していた全ての情報、すなわち送信者と受信者のオブジェクト識別子、メソッドセレクタ、実引数などを含んでいる。

メッセージ変換宣言には、以下の2種類がある。

[Convert-outgoing *mth* => χ]

[Convert-incoming *mth* => χ]

Convert-outgoing 宣言は、メッセージセレクタが mth のメッセージがこのクラスのオブジェクトから送信されたときに、そのメッセージを一級メッセージに変換し、χ に送ることを指定する。Convert-incoming 宣言は、このクラスのオブジェクトにメッセージセレクタが mth のメッセージが到着したときに、メソッドを起動する代わりにメッセージを一級メッセージに変換し χ に送ることを指定する。

メッセージ変換が起きる際には、新たにメッセージ (M) が作られ、一級メッセージ (μ) は M の引数として、メッセージ変換宣言によって指定されるオブジェクト (χ) に送られる。もともとのメッセージ (m) のメソッドセレクタが mth であったとすると、M のメソッドセレクタは [mth] となる[3]。したがって、χ は、M を受理して、メソッド [mth] を起動する。

メソッド [mth] が起動すると、引数として一級メッセージ (μ) を受け取るため、メソッドの中で、一級メッセージのインスタンス変数に保存、メッセージの引数に与えることによる他のオブジェクトへの送信、後述の3種類の操作などが可能となる。

[3]いかなるメソッド名とも衝突しない特殊な名前にする目的と M に一級メッセージの型情報を与えるためにシステムで許される名前空間を拡張した。

3.2 一級メッセージに対する操作

メッセージ変換によってつくられた一級メッセージは、通常のオブジェクトと同様にクラスを持ち、一級メッセージに対する操作はそのクラスのメソッドとして定義されている。一級メッセージのクラスは MsgCont と呼ばれ、3つのメソッド copy, fire, reply が定義されている。

copy 一級メッセージは、もともとのメッセージの保持していた全ての情報を表現している。この情報には、メッセージの送信者、受信者、メソッドセレクタ、メッセージの引数、などが含まれる。copy 操作は一級メッセージの複製を生成する。ただし、メッセージの同一性（4節）を保つために、複製が表現するメッセージの送信モードはもとのメッセージの送信モードに関わらず非同期となる。

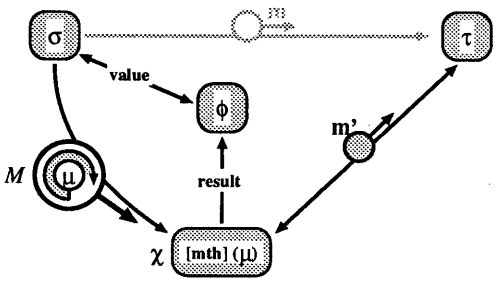
fire 一級メッセージが表現するメッセージを再生する。メッセージの同一性（4節）を保証するために、fire 操作は一級メッセージに対して副作用を持ち、一級メッセージが持つ送信モードの情報を非同期に変更する。

reply 一級メッセージが表現するメッセージ送信の送信モードが同期的送信の場合には、送信者に返答を行う。メッセージの同一性（4節）を保つために reply 操作は一級メッセージに対して副作用を持ち、一級メッセージが持つ送信モードの情報を非同期に変更する。

3.3 一級メッセージを用いた通信機構の実現（未来型通信）

[7] において同期的メッセージ送信のみを持った multi-thread の計算モデルにおいて、非同期的メッセージ送信、逐次／並行マルチキャスト、マルチプロセッサ上でのプログラム実行のエミュレーションなどがを容易に記述できることを示した。同様の技法を用いることによって、本稿に述べる single-thread のモデルにおいても、これらを実現できることが分かっている。ここでは、Actor 言語のプログラミング技法として用いられ、ABCL/1 や Concurrent Smalltalk では言語機構として採用された未来型通信が一級メッセージを用いて実現できることを述べる。

メッセージの非同期的通信では、送信者と受信者の間で同期による待合せは一切ない。メッセージ通信の種類として非同期通信のみを提供する並行言語を用いる場合は、別の同期のためのプリミティブを用いて明示的に同



σ:	sender	τ:	target
$m^{(\prime)}$:	message	$\mu^{(\prime)}$:	1st class message
M:	msgcont carrier	χ:	future communicator
ϕ:	future object		

図 2: 未来型メッセージ通信の実現

期を指定しなくてはならない。一方、同期的通信はオブジェクト間の情報交換（メッセージの送信とメッセージに対する返答）および同期を組み合わせたものとみなすことができる。送信者は、メッセージの送信から返答を受け取るまでの間、返答を待ち続ける。

未来型通信は意味的に同期的通信と非同期的通信の中間に位置するものと考えられる（図 2）。未来型のメッセージ送信は、受信者にメッセージを配送しつつ、送信者に対してはすぐに未来オブジェクト (future object) と呼ばれるオブジェクトをメッセージ送信に対する返答として返す。受信者は、受け取ったメッセージを同期的に送信されたものとして処理し、やがてメッセージに対して返答する。この返答は、同期的通信のようにメッセージの送信者ではなく、未来オブジェクトに送られ、未来オブジェクトに保存される。

送信者は、未来型通信に対する返答を未来オブジェクトに問い合わせのメッセージを送ることで得ることができる。受信者からの返答が未来オブジェクトに到達して折らず、未来オブジェクトに値が設定されていない場合は、送信者が未来オブジェクトに送った問い合わせのメッセージは返答の到着を待たなくてはならない。

未来型通信は、同期型通信におけるメッセージの送信と返答を受け取るための同期の意味を切り放して独立に扱えるようにしたものといえる。逆に、同期型通信は未来オブジェクトへの問い合わせがメッセージ送信の直後に実行される特殊な場合と考えることもできる。

ここでは、一級メッセージを用いたプログラミング例として、未来型通信が容易に実現できることを述べる。

```
[Convert-outgoing mth => χ]
  ...
  (let ((φ [τ <== (mth)]))
    ...
    [result := [φ <== (value)]]])
```

上のコードの [τ <== (mth)] では、τ に対してメッセージを同期的に送信しようとする。しかし、コードの先頭でのメッセージ変換が宣言されているため、メッセージ (m) が τ に送られるかわりに、このメッセージは一級化され (μ)、未来型通信を実現するオブジェクト χ に送られる。ここで、χ はつぎの Future-communicator クラスのインスタンスとする。

```
[class Future-Communicator ()
  [method ([mth] μ)
    (let* ((μ' [μ <== (copy)])
           (φ [Create Future μ']))
      [μ <== (reply φ)])]]
```

χ は、μ を受けとると、未来オブジェクト (ϕ) を作り、一級メッセージに対する reply 操作により、もともとのメッセージ送信者に対して ϕ を返す。ϕ を生成する際の初期化メソッドへの引数に一級メッセージが与えられ、fire 操作によって、以下に見るようにもともとのメッセージの処理が始まる。

```
[class Future (result)
  [initialize (μ)
    [result := [μ <== (fire)]]]
  [method (value)
    ^result]]
```

上の定義は未来オブジェクト (ϕ) のクラス定義である。未来オブジェクトが生成されると、初期化メソッドに与えられた引数 μ に対して fire 操作を加えることにより、もともとのメッセージ (m') を再生する。やがて、m' は処理され、それに対する返答は ϕ の初期化メソッドに対して返され、初期化メソッドはその返答値を ϕ のインスタンス変数 *result* に記憶する。

さて、メッセージの送信者は得られた未来オブジェクトに対して、メッセージを送ることによって ([ϕ <== (value)]) 未来型通信に対する返答を得ることができる。value メッセージが送られたときの未来オブジェクトの動作には、result の値が定めっているか否かによって、2 通りが考えられる：(1) すでに定まっているとき、value メソッドが起動され、result の値を返答値として返す。(2) result の値がまだ定まっていないとき、初期化メソッドの実行は、まだ終了していない。計算モデルの single-thread 性より、value メソッドの実行は初期化メソッドが修了し

て、result の値が定まるまで待たされる。あとの実行は (1) と同様である。

4 メッセージの同一性と副作用

一級メッセージの設計においては、並行計算に現れる通信構造一般を記述するのに十分な記述力を持たせることはもちろん、プログラマに対して十分な抽象レベルの高いプリミティブを提供することを重視した。この抽象レベルの指標として、用いたものはメッセージの同一性の保持であった。

メッセージの同一性とは、メッセージ送信における送信者と受信者との間を関係であり、具体的には次の性質としてまとめられる。

- 同期的メッセージ送信の送信者に対しては、たかだか一つの返答が返される。
- 非同期的メッセージ送信の送信者に対しては、返答は返されない。

一級メッセージの枠組みの素朴な設計においては、一級メッセージに対して fire 操作や reply 操作を複数回実行することによって、メッセージの送信者に対して二つ以上の返答が返されてしまい、メッセージの同一性を破壊してしまう可能性がある。この問題を解決するために、[7] では、一級メッセージのモード（非同期的／同期的一級メッセージ）を与え、fire, reply の実行の際に、そのモードを変更するような副作用を与えた。ここでは、同様のモード付けに加え、非同期的に送信されたメッセージを変換して得られる一級メッセージに対する意味付け与える。

メッセージ変換 非同期的に送信されたメッセージ、同期的に送信されたメッセージは、それぞれ同期的一級メッセージ、非同期的一級メッセージに変換される。同期的一級メッセージは非同期的メッセージ送信、すなわち送信者が返答を待っていない状態を表現しているものとみなせる。したがって、fire, reply 操作に際しては、返答がメッセージの送信者に対して送られないような意味付けをするべきである。一方、非同期的一級メッセージは同期的メッセージ送信、すなわち送信者が返答を待っている状態を表現しているものとみなせる。したがって、fire, reply 操作に伴う返答が送信者に対して送られるような意味付けをするべきである。

一級メッセージへの操作に対する意味付け

fire 同期的一級メッセージは、非同期的送信を表現している。この場合、送信者は返答を期待していない。したがって、fire 操作によって、再生したメッセージの処理に対する返答を送信者に対して送ることはメッセージの同一性を破壊することになる。本枠組みでは、同期的一級メッセージに対する fire 操作では、fire 操作を行ったメソッド実行に対して返答が返される。この場合の fire 操作は再生されたメッセージが処理され、返答が返されるのを待つことになる。

一方、非同期的一級メッセージは、同期的送信を表現しており、送信者はメッセージ送信に対する返答を待っている。この場合の fire 操作では、再生されたメッセージに対する返答は返答を待っている送信者に送られる。fire 操作を行ったメソッドには返答を待つ必要がないために、fire 操作はただちに実行を終了する。したがって、fire 操作を行ったメソッドからは、再生されたメッセージは非同期的を送信されているようにみえる。

一級メッセージの同期的／非同期的という名前は fire 操作に伴う、動作の違いから名付けられた。

copy copy 操作は一級メッセージに対する副作用を持たない。copy 操作によって生成される一級メッセージは同期的一級メッセージとなる。これは、非同期的一級メッセージとその複製に対して fire 操作が加えられたときに、送信者に二つの返答が送られることを避けるためである。

reply 同期的一級メッセージに対する reply 操作では、一級メッセージが表現するメッセージの送信者に対して、返答を送った後、一級メッセージを非同期なものに変更する。これは、それ以降の fire や reply 操作によって、これ以上の返答が送信者に送られないようにするためである。

非同期的一級メッセージに対する reply 操作は何も行わない。

5 多相性

3.1節で、メソッドセレクタ mth のメッセージが変換されてできたメッセージのメソッドセレクタが [mth] になり、communicator に配達されて [mth] という名前のメソッドを起動すると述べた。[mth] を記述することでさまざまな通信機構を記述できることについて述べ、その例として未来型通信が実現できることを示した。この例で定義した Future-communicator はメソッドセレクタが mth の場合のみに対応しているだけである。したがって、他のメソッドセレクタ（たとえば、mth′）を持ったメッセージに対しても未来型通信を行いたい場合には、送信者のコードにあらたに mth′ に対する Convert-outgoing 宣言を加え、Future-communicator に [mth′] の定義を加える必要があり、非常に煩わしい。

表 1: メッセージとメソッドの対応

メソッドセレクタ	メソッド
mth	mth
[mth]	[mth], [ ]
[[mth]]	[[mth]], [[ ]], [ ]
⋮	⋮

ここで求められるのはメソッドセレクタとして、[mth] と [mth'] のいずれを持ったメッセージでも処理できるような多相的なメソッドの記述である。Harmony/2 では、[ ] という名前の多相メソッド宣言を許している。このメソッドは [mth] と同様、引数として一級メッセージを受けるメソッドである。[mth] メソッドが、[mth] をメソッドセレクタとして持ったメッセージを処理するのに対し、[ ] メソッドはメッセージ変換によって生成されたメッセージを一般的に処理できる。

メッセージとメソッドとの対応づけは、単純な名前の一致ではなく表 1に表されるパターンの対応となる。メソッドセレクタとして [[mth]] を持ったメッセージは、3 つのメソッド [[mth]], [[ ]], [ ] のいずれかによって処理される。Harmony/2 では、このうち最も具体的なメソッド、すなわち表の中で左に位置するものを優先してメソッドを探索し起動する。未来型通信の例では以下のように [mth] の代わりに [ ] という名前のメソッドを定義すれば、さまざまな型のメッセージに対して未来型通信を提供することが出来るようになる。

```
[class Future-Communicator ()
  [method ([ ] μ)  ;;; [ ] メソッドの定義
    (let* ((μ' [μ <== (copy)])
           (φ [Create Future μ']))
      [μ <== (reply φ)])]]
```

6 おわりに

オブジェクト指向並行プログラミング言語において、メッセージを一級オブジェクトとして扱う枠組みを提案した。並行／分散計算において、逐次計算において一級継続が果たしたのと同じような役割を、並行計算においては一級メッセージが果たすこと、および一級継続を用いてさまざまな通信形態を記述することができることについて述べた。具体的なプログラミングの例として、一部の並行言語に取り入れられている未来型通信を一級継続を用いて記述した。最後に、一級メッセージを処理するメソッドに多層的なものを導入することにより、いっそう記述面の柔軟性が得られることを示した。

本稿で説明した枠組みをもとにオブジェクト指向並行言語 Harmony/2 が設計され、プロトタイプが完成している。今後はまだ実装されていない多層メソッドを組み込み、より多くの通信機構の記述に利用していきたい。

謝辞 この研究の初期の段階で多くのコメントを下さった Twente 大学 M. Aksit 教授のグループの皆さんに感謝します。メッセージ変換のアイデアについて議論してくださった、米沢明憲教授を初めとして東京大学情報科学科の皆さんはメッセージ変換ついて議論してくださいました。東京工業大学情報科学科の皆さんは Harmony/2 の設計と一級メッセージのアイデアについて議論してくださいました。特に、森實裕人さんは Harmony/2 のプロトタイプを完成してくれました。

参考文献

[1] Mehmet Aksit, Lodewijk Bergmans, and Sinan Vural. An object-oriented langauge-database integration model: the composition-filters approach. Technical report, The university of Twente, 1992.

[2] P. America. POOL-T: A parallel object-oriented language. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199 – 220. MIT Press, Cambridge, Mass., 1987.

[3] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[4] C. T. Haynes and D. P. Friedman. Embedding continuations in procedural objects. *ACM Transactions on Programming Languages and Systems*, 9(4):582–598, April 1987.

[5] O. M. Nierstrasz. Active objects in Hybrid. In *Object-Oriented Programming Systems, Languages and Applications*, volume 22(12), pages 243 – 253. SIGPLAN Notices (ACM), December 1987.

[6] J. Rees and W. Clinger. Revised^3 report on the algorithmic language Scheme. *ACM SIGPLAN Not.*, 21(12):37–79, December 1986.

[7] K. Wakita. First class messages as first class continuations. In *proceedings of ISOTAS '93 (LNCS 742)*, pages 442–459, Kanazawa, November 1993.

[8] M. Wand. Continuation-based multiprocessing. In *Conference Record of the 1980 Lisp Conference*, pages 19–28, 1980.

[9] Y. Yokote and M. Tokoro. The design and implementation of ConcurrentSmalltalk. In *Object-Oriented Programming Systems, Languages and Applications*, volume 21(11), pages 331 – 340. SIGPLAN Notices (ACM), November 1986.

[10] A. Yonezawa. *ABCL: An Object-Oriented Concurrent System*. MIT Press, Cambridge, Mass., 1990.