JAIST Repository
https://dspace.jaist.ac.jp/

Title	Java 言語における非同期通信の抽象化
Author(s)	阿部, 修
Citation	
Issue Date	1997-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/1046
Rights	
Description	Supervisor:渡部 卓雄, 情報科学研究科, 修士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

修 士 論 文

Java 言語における非同期通信の抽象化

指導教官　渡部 卓雄 助教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

阿部 修

1997 年 2 月 14 日

Copyright © 1997 by Osamu Abe

要 旨

本研究では、 Java で並行計算をより抽象的に扱えるように言語を拡張し、そして、その言語を Java バーチャルマシン上で動作させることを目的とする。Java というオブジェクト指向言語は Web ブラウザによってリモートマシンにあるプログラムをロードして実行することができる。 Java はそれだけで画期的な言語ではあるが、並行計算に適した言語ではない。また、RMI や HORB などの Java で分散オブジェクトを抽象的に扱うためのシステムは現れたが、並行計算を抽象的に扱えるようにするためのシステムは現れていない。分散計算においても並行計算の扱いやすさは重要なことである。

本研究で作成した言語には MILK と名付けた。 MILK は Java を拡張している。MILK では、並行計算を抽象的に扱うために、 Java に並行オブジェクト指向モデルを組み込んでいる。並行オブジェクト指向モデルでは、各オブジェクトが自立して並行に動作することができる。従って、オブジェクトを擬人化して捉えることができる。このことから、人間にとって並行性を直観的に考えることがしやすく、プログラムの記述が容易になると考えられている。

並行オブジェクト指向モデルでは、オブジェクト間でメッセージによる非同期通信を行うことで、計算を進めていく。 MILK ではメッセージ通信の方式に 2 種類の方式を採用している。それらの方式は過去型と未来型と呼ばれている。過去型はオブジェクト間でメッセージ通信を行った時に、送信側が受信側の処理の終了を待たずにすぐに計算を再開する方式である。過去型ではメッセージ送信後、受信側から戻り値が返されることはない。未来型はフューチャーと呼ばれるプレイスホルダを、送信側から受信側にメッセージと共に渡し、過去型と同様、送信側はすぐに計算を再開する。そして、受信側は処理の結果をフューチャーに渡す。送信側は結果が必要になった時にフューチャーを調べ、結果が戻っていればその値を使う。結果が戻っていない場合には、そこで戻ってくるまで待つという方式である。 MILK は本研究で作成したトランスレータで Java に変換し、実行することができる。

本論文では MILK の設計とトランスレータの実装についての諸問題とその対処方法について説明している。そして、実際に MILK を使ってアプリケーションを作成し、この言語について評価考察を行っている。

目 次

1 はじめに **1**

1.1 本研究の目的 1

1.2 本論文の構成 3

2 基礎概念 **4**

2.1 並行オブジェクト指向モデル 4

3 既存の並行分散システム **9**

3.1 Java 9

3.2 RMI 10

3.3 HORB 12

3.4 Obliq 13

3.5 CORBA 15

3.6 ConcurrentSmalltalk 15

3.7 Multilisp 16

3.8 Schematic 17

4 MILK の設計 **18**

4.1 設計方針 18

4.2 言語仕様の設計 19

4.2.1 並行オブジェクト 19

4.2.2 フューチャー 25

4.3 MILK を用いた並行計算の例 29

5 MILK のアーキテクチャ **36**
5.1 並行オブジェクト 36
5.1.1 ConcurrentObject 37
5.1.2 変換 41
5.2 フューチャー 44
5.2.1 宣言 45
5.2.2 参照 48
5.2.3 代入 50
5.2.4 その他の演算子 52
5.3 プロパティファイル 52

6 分散環境における MILK **56**
6.1 RMI と MILK 56
6.1.1 並行オブジェクト 56
6.1.2 フューチャー 58
6.2 HORB と MILK 61
6.3 分散環境での例 62

7 MILK の評価 **74**
7.1 トランスレータの実装による問題 74
7.2 Java による問題 75
7.3 分散環境での問題 76

8 おわりに **79**
8.1 今後の展望 79

謝辞 **81**

A MILK トランスレータ操作説明 **85**
A.1 動作環境 85
A.2 入手方法 85
A.3 インストール 86
A.4 トランスレータの実行方法 86

A.5 プロパティのデフォルト値 87
A.6 RMI と HORB の入手方法 87

B Weather Information の実行方法 **91**
B.1 コンパイル手順 91
B.1.1 サーバ側 92
B.1.2 ウィンドウ側 92
B.2 実行方法 93
B.2.1 ウィンドウ 93
B.2.2 サーバ 94
B.2.3 アプレット 95

図 目 次

2.1 逐次オブジェクトのメッセージ通信 5
2.2 過去型のメッセージ通信 6
2.3 現在型のメッセージ通信 7
2.4 フューチャーによるメッセージ通信 8

4.1 並行オブジェクトの概念図 20
4.2 最も簡単な例 29
4.3 Java で書いた場合の最も簡単な例 31
4.4 第 3 のオブジェクトへのフューチャーの譲渡 32
4.5 フューチャーによる副作用 33
4.6 素数を生成するプログラム 34

5.1 並行オブジェクトのクラス定義 37
5.2 Java に変換された並行オブジェクトのクラス定義 38
5.3 メソッド呼び出しの流れ 39
5.4 クラス milk.runtime.ConcurrentObject の定義 40
5.5 プロパティファイルのフォーマット 54

6.1 RMI を使用した並行オブジェクトのクラス定義 57
6.2 int 型フューチャーオブジェクトのインターフェース 59
6.3 int 型フューチャーオブジェクトの実装の一部 60
6.4 Weather Information の分散オブジェクトの関係 63
6.5 Weather Information でのフューチャーの流れ 64
6.6 Weather Information のアプレット 64
6.7 ファイル WeatherInfoServer.milk 65

6.8 ファイル WeatherInfoServerImpl.milk 67
6.9 ファイル WeatherInfoWindow.milk 69
6.10 ファイル WeatherInfoWindowImpl.milk 69
6.11 ファイル WeatherInfoApplet.milk 71
6.12 ファイル InfoWriter.milk 73

7.1 同一クラスのオブジェクトの変数のアクセスに関する問題 77

B.1 ファイル FixedLayout.java 96
B.2 ファイル weather.html 98

表 目 次

4.1 フィールドのアクセスレベル 22

5.1 フューチャーの代入文と変換後のメソッド名の対応 50
5.2 フューチャーに関係するプロパティとフューチャーとの対応 55

A.1 プロパティのデフォルト値（コンパイルオプション無しの場合）. 88
A.2 プロパティのデフォルト値（-rmi オプション有りの場合） 89
A.3 プロパティのデフォルト値（-horb オプション有りの場合） 90

第 1 章

はじめに

本論文は Java における非同期通信の抽象化のための 1 つの手法について述べたものである。この章ではまず本研究の目的について述べ、その後で本論文の構成について述べる。

1.1 本研究の目的

現在、コンピュータネットワークが急速に普及し、並行計算や分散計算の応用範囲も急速に増えてきている。しかし、これらの開発はほとんどの場合、未だに逐次計算のためのパラダイムに頼っている。とりわけ計算機言語ではそうである。例えば、並行計算においては、アプリケーションの開発に実際に用いられる言語のほとんど全部が、逐次計算のために開発された言語にプロセスやスレッドまたはコルーチンなどの機構を組み合わせることで実現している。また、分散計算においても、ストリームやもっと低レベルの機構や、少し抽象度の高いものでもリモートプロシージャコールによって、アプリケーションを開発することがほとんどである。このようなやり方は既存の知識を活かすことができるという点では優れている。そして、人間にとってはこの方がとっつきやすい。従って、このやり方が今まで多く採用されてきた。

しかし、並行計算や分散計算には、逐次計算には無いこれら特有の問題が内包されている。例えば、並行計算では同期のとり方やデータの共有の仕方、クリティカルセクションの排他制御などの問題がある。分散計算では通信の方式やセキュリティの管理、データの共有の仕方、ネットワーク特有の遅延やリモートで発生した例外の問題などがある。このような問題を全て逐次計算のパラダイムだけに押し込むことは、最適な方法とはいえない。

逐次計算のパラダイムで設計された言語を用いて、並行分散計算のためのプログラムを書くことは、人間にとってあまり直観的でなく、理解しにくいものとなってしまう。例えば、並行計算をプロセスを使ってプログラムを記述する場合、プログラマは fork や wait といった命令で、プロセスを直接制御する必要があり、プロセス間の同期のとり方やデータの共有方法などについても気を配る必要がある。そして、そのようなプロセスを使って記述されたプログラムを読む場合、プロセスのライフサイクルや同期のタイミング、特定のプロセスが実行するプログラムの範囲などについて、なかなか理解することはできない。これは実装に使った言語自体が逐次計算だけをうまく表現できるように設計されており、並行計算を的確に表現する能力を欠いているからである。

しかし、並行分散計算に適したパラダイムで言語を設計し、それを用いれば、人間にとってもっと理解しやすいものとなるに違いない。実はこのようなパラダイムは特に新しいものではない。この分野の研究は 20 年以上も前から行われている。特に並行オブジェクト指向モデルというパラダイムはとても興味深い。並行オブジェクト指向モデルとは一言で述べると、オブジェクト指向モデルの一種で、各オブジェクトが自立して並行に動作することのできるモデルである。各オブジェクトの持っている情報はオブジェクトによって隠蔽されている。人間はそのオブジェクトについて、並行に動作するということと、そのオブジェクトの入出力だけ知っていれば良い。従って、人間にとってとても直観的な形で並行動作を表現することが可能である。

そのような背景の中で Java という分散計算に適した性質を持った計算機言語が現れた。Java はオブジェクト指向言語である。この言語は Web ブラウザで動作するプログラムをリモートマシンからローカルマシンに安全にロードすることができる。従って、たちまち脚光を浴びた。しかし、分散計算のための機能や並行計算のための機能はまだまだ低レベルのものであった。そこで Java の分散計算をより強力にかつ抽象的に扱えるようにするために RMI や HORB などのシステムが開発された。並行計算をより抽象的に扱えるようにするためのシステムは今のところ現れていない。分散環境は分散した計算機を並行に動作させることができ、むしろそのように動作させる方が自然なので、並行計算の抽象度は分散環境においても大変重要である。

そこで本研究では Java で並行計算をより抽象的に扱えるよう、 Java に並行オブジェクト指向モデルのセマンティクスを付加した言語を設計し、その言語を Java バーチャルマシン上で動作させることを目的とした。

1.2 本論文の構成

本論文の構成は次のようになっている。第 2 章では本研究の基盤となった並行オブジェクト指向モデルとその非同期通信の方法について紹介する。第 3 章では既存の並行分散システムのいくつかについて簡単に紹介し、それらのシステムが並行計算と分散計算においてどのような利点、欠点があるのかを考察する。第 4 章では本研究で開発した言語 MILK をどのように設計し、どのような仕様を採用したのかについて説明する。また、実際に例となるプログラムを作成し、Java との比較も行っている。 MILK はトランスレータによって Java に変換することができる。第 5 章ではそのトランスレータの変換の仕組みや実行時に使用されるクラスライブラリについて説明する。第 6 章では MILK の分散環境での振舞について説明する。第 7 章では MILK について評価し、 MILK の問題点とその改善方法について論じる。第 8 章では MILK と並行オブジェクト指向言語の今後の展望について述べる。付録 A では本文では紹介できなかったトランスレータの使い方等について説明する。付録 B では例題で取り上げた天気情報のプログラムの動作方法について説明する。

第 2 章

基礎概念

プログラムを並行に動作させる場合、プロセスやスレッド、コルーチンなどの機構が良く使われる。しかし、これらで並行プログラミングを行うと、データの共有方法や同期の問題について実装者はいつも頭を悩ませなくてはならない。並行オブジェクト指向モデルではオブジェクトを用いて、これらの問題をもっと抽象的に扱えるようにしている。この章では本研究の基盤となった並行オブジェクト指向モデルの考え方について紹介する。並行オブジェクト指向モデルについては文献 [1, pp. 403–436] でより詳しく説明されている。

2.1 並行オブジェクト指向モデル

一般的な逐次オブジェクト指向モデルではオブジェクトを静的な物として説明することが多い。それはオブジェクトが他から作用を及ぼされない限り、活動を始めることがないからである。また、オブジェクト外部から、そのオブジェクト内部の状態を直接操作できるモデルも多い。逐次オブジェクトのメッセージ通信は図 2.1 のように 2 方向であり、呼び出し側は呼び出した処理が終るのを必ず待たなければならない。

一方、並行オブジェクト指向モデルでのオブジェクトは擬人化された者として説明されることが多い。それはオブジェクトの振舞が人間の振舞によく似ていて、擬人化することで説明がしやすくなるからである。人間の社会では人と人が会話をすることで、ある取り決めをし、そして、仕事を行っていく。仕事を請け負った人はその仕事を請け負った時にすぐ処理することもできるし、何か他の仕事の後に行うこともできる。また、その仕事を行わなくても良い。更に、その仕事を第 3 の人に依頼することもできる。請け負った仕事

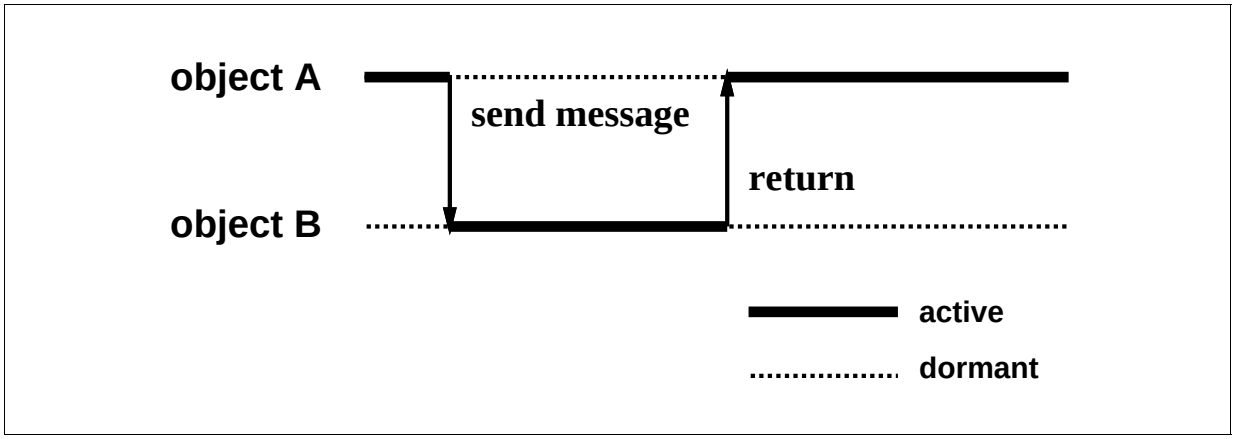

図 2.1: 逐次オブジェクトのメッセージ通信

をどのように処理するかは請け負った人の自由である。

仕事を依頼した人はその仕事が終るのを待っている必要はなく、何か他の仕事をしていても良い。依頼した仕事の終了の連絡方法もいろいろある。仕事を引き受けた人に直接教えてもらう方法や、郵便による方法、それから、特に連絡の必要がない場合もある。また、自分の住所と名前を連絡しておけば、仕事を引き受けた人とは違う人から連絡が入るかも知れない。また、自分の名前と住所をうまく宣伝しておけば、いろいろなところから仕事の依頼があることもある。

それから、人間社会では他人の心の中まではのぞき込むことはできない。全ては会話でやりとりされ、他人が本当は何を考えているのかを知ることはできないし、ましてや他人の考え方を変更することはできない。

並行オブジェクト指向モデルは、まさにこのような人間社会のモデルである。従って、人間にとって直観的な表現ができ、並行動作を理解しやすい。並行オブジェクトはそれぞれが自立して並行に活動することができる。そして、それらの状態はオブジェクト内にしっかりとカプセル化されている。これは状態をしっかりと保護することで、安全に自立して並行に活動することを保証するためである。

オブジェクト同士はメッセージを介して互いに通信を行う。届いたメッセージの処理の仕方は個々のオブジェクトに委ねられる。届いた順番に処理をするものもあれば、順番を変えて処理するものもあるし、並行オブジェクト内で更に並行に処理を行うものもある。

メッセージ通信の方式について、並行オブジェクト指向言語 ABCL [2, 3] を例にとって

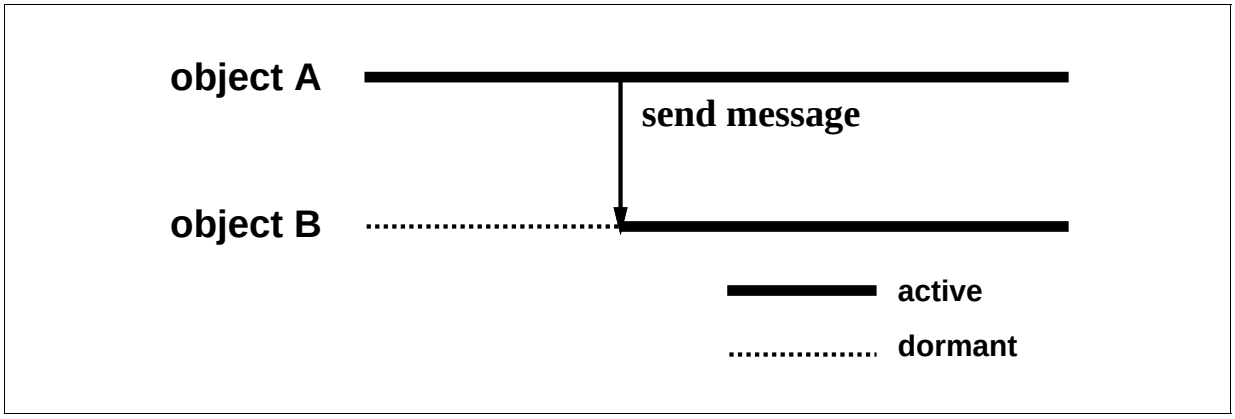

図 2.2: 過去型のメッセージ通信

説明する。 ABCL は、並行オブジェクト指向モデルの元祖と考えられているアクタモデルに基本を置いている。

ABCL のメッセージ通信の方式には 3 種類ある。過去型、現在型、未来型である。過去型は 1 方向のメッセージ通信である。図 2.2 は過去型のメッセージ通信のモデル図である。送信オブジェクトはメッセージを送信した後、すぐに他の処理を行うことができる。送信オブジェクトが処理の結果を返信してもらいたい場合には、メッセージとして送信オブジェクトの識別子を渡し、その識別子宛に再びメッセージを送信してもらうことで行う。

現在型のメッセージ通信は 2 方向のメッセージ通信である。図 2.3 は現在型のメッセージ通信のモデル図である。このメッセージ通信は逐次オブジェクトのメッセージ通信に似ている。送信オブジェクトは戻り値が返ってくるまで他の処理をすることができない。しかし、受信オブジェクトは戻り値を返した後も作業を続けることができる。そしてまた、依頼された処理を第 3 のオブジェクトに依頼することもできる。

未来型のメッセージ通信は現在型のメッセージ通信とよく似ている。図 2.4 は未来型のメッセージ通信のモデル図である。受信オブジェクトは処理結果を返した後もまだ作業を続けることができる。また、受信オブジェクトは依頼された処理を第 3 のオブジェクトに依頼することもできる。しかし、送信オブジェクトは処理結果を受け取るために必ずしも待つ必要がない。送信オブジェクトはメッセージ送信後に別の処理をすることができる。このメッセージ通信ではフューチャーと呼ばれるプレイスホルダをメッセージの一部として受信オブジェクトに送っている。受信オブジェクトが処理結果を送信オブジェクトに渡す場合にはこのフューチャーに値を格納する。受信オブジェクトが依頼された処理を第 3

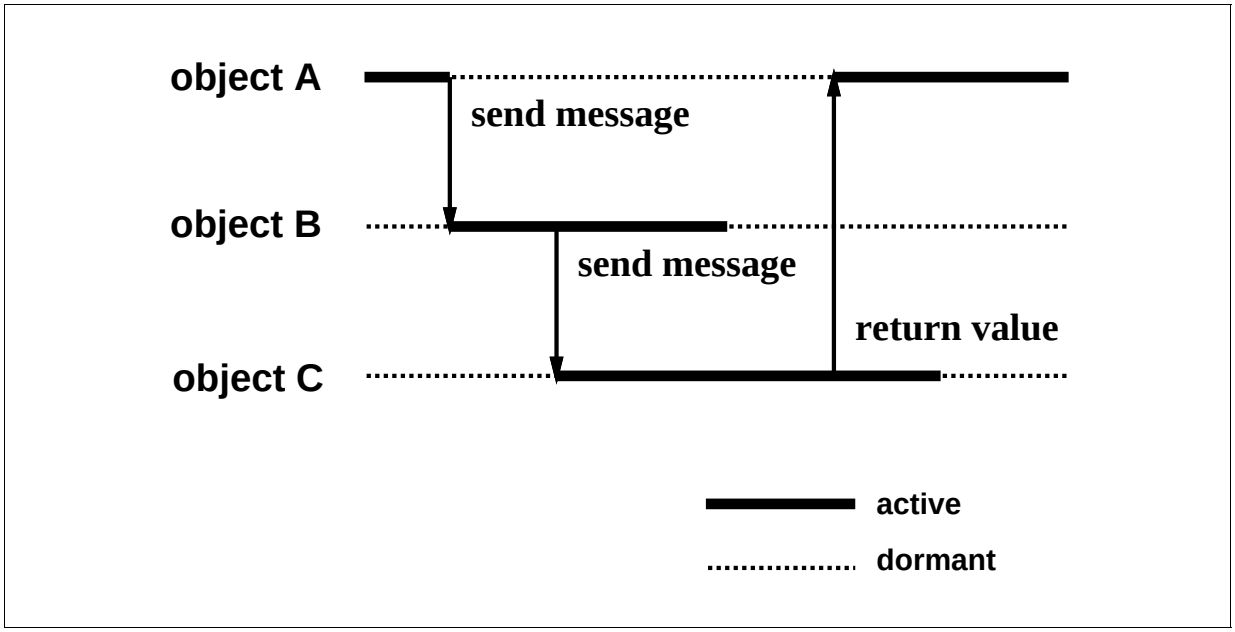

図 2.3: 現在型のメッセージ通信

のオブジェクトに依頼する場合には、メッセージとしてこのフューチャーも渡すことで行うことができる。送信オブジェクトは処理結果の値が必要になった時にフューチャーを参照する。この時、すでにフューチャーに値が格納されていれば、そのままその値を使うことができる。もし、フューチャーに値が格納されていない時には、送信オブジェクトは値が格納されるまで待つことになる。そして、フューチャーに値が格納されると、送信オブジェクトはそのことに気付き、値を取り出し、処理を再開する。未来型のメッセージ通信を単にフューチャーと呼ぶこともある。

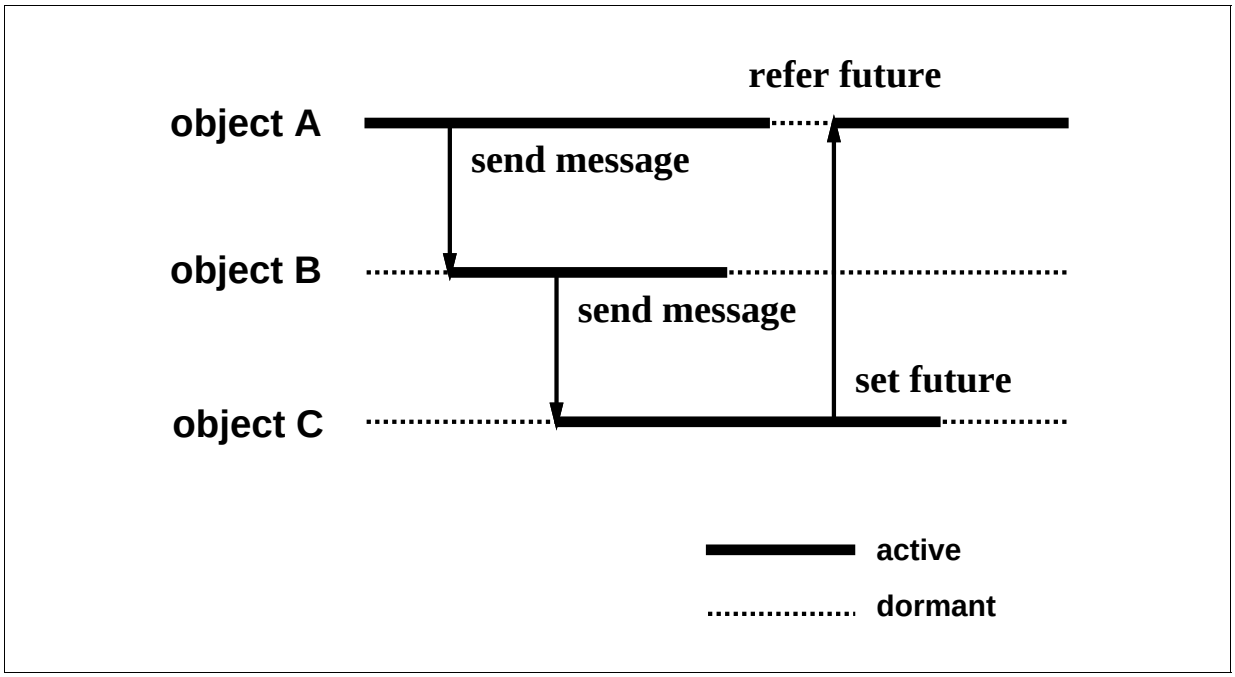

図 2.4: フューチャーによるメッセージ通信

第 3 章

既存の並行分散システム

新しく並行言語を設計するにあたって、まず既存の並行分散システムがどのようになっているのかをみておく必要がある。この章では既存の並行分散システムのいくつかについて、並行計算や分散計算をどのように扱っているのかを簡単に紹介する。

3.1 Java

Java はとりわけ並行分散言語というわけではないが、本研究は Java に基づいて行っているので、この節で Java がどのような言語であるかを紹介する。

Java は C++ をベースにして設計されたオブジェクト指向言語である。従って、多くの人に馴染みやすい言語となっている。しかし、簡易性を追求したため C++ の以下の機能は除去されている。

> プリプロセッサ、typedef 文、構造体、共用体、関数、多重継承、goto 文、演算子のオーバーロード、自動強制型変換、ポインタ

これらを除去したおかげで簡潔で理解しやすいプログラムが書けるようになっており、バグが入りにくくなっている。

逆に C++ には無い機能も多く備えている。まず、 Java はインタプリタ形式で実行される。 Java はソースコードを 1 度コンパイルしてバイトコードに落す。このバイトコードのファイルのことをクラスファイルと呼ぶ。そして、 Java バーチャルマシン（VM）がこのクラスファイルをロードして実行を行なう。従って、VM があれば、 Java をどのマシ

ンでも実行することが可能である。この方式のおかげで、マシンに依存しない中立性、移植性の高い言語となっている。従って、分散環境での使用には適しており、インターネットで広く利用されている。また、 VM がガベージコレクションを自動で行なってくれるので、ユーザがメモリ管理について気を使う必要がなく、バグの混入の可能性が小さい。また、これにより、プログラムを簡潔に書けるようになっている。

また、 Java は言語レベルでマルチスレッド機能を実現している。従って、その安定した正しい動作が保証されているだけでなく、スレッドを扱うための構文があらかじめ用意されている。従って、スレッドを扱ったプログラムをとても簡潔に書くことができる。

それから Java には例外処理をするための構文が組み込まれており、コンパイル時に発見することのできる全ての例外の発生する可能性のある箇所に対して、例外処理をしておかなくてはならない。従って、例外が発生してもすぐには止まらない信頼性の高いアプリケーションを書くことができる。

その他に分散環境でのセキュリティについてもよく考慮されている。 Java は Java を動作させることのできる Web ブラウザで、ホストから送られてきたクラスファイルをロードする場合、厳密な型チェックなどを行っている。従って、リモートから送られてきたプログラムを安全に実行することができる。

Java の特徴は文献 [4] で詳しく紹介されている。 Java の言語仕様については文献 [5] で、バーチャルマシンについては文献 [6] で、 API ライブラリについては文献 [7] で詳しく説明されている。Java を実際に使う上でのノウハウは文献 [8] がとても参考になる。

3.2 RMI

RMI [9, 10] は Sun Microsystems が開発した Java のためのリモートオブジェクトにアクセスするためのシステムである。リモートオブジェクトとは分散環境においてリモートにあるマシン上で活動しているオブジェクトのことである。 RMI は Modula-3 のネットワークオブジェクト [11, 12] を参考にして設計されている。 RMI は独自のセキュリティマネージャとオブジェクトシリアライザを用いることで、リモートオブジェクトのメソッド呼び出しを可能にしている。

Java では普通、クラスファイルのロードはローカルマシンからのロードだけが許されている。また、アプレットを用いた場合にはアプレットの存在するホストからのみ、クラスファイルをロードすることが可能である。従って、あちこちに分散したリモートマシンにあ

るクラスファイルをロードすることはできない。このため、Java で分散環境を扱う場合には各リモートマシンにあらかじめ必要なクラスファイルを準備しておく必要がある。 RMI では独自のセキュリティマネージャによって分散したマシンからのクラスファイルのロードを可能にしている。

また、 Java ではネットワークをストリームとして扱っている。従って、オブジェクトはストリームでないから、オブジェクトを直接リモートマシンに渡すことはできない。 RMI のオブジェクトシリアライザはオブジェクトを自動でストリームの形に直すものである。これにより、どんなオブジェクトのコピーでもリモートに転送することが可能となる。

RMI は、あるインターフェースを実装したクラスを元にして、自動でそのインターフェースを実装したスタブクラスとスケルトンクラスを生成する。元となるクラスはユーザが定義し、サーバクラスと呼ばれる。そして、このスタブクラスとスケルトンクラスを介することでサーバクラスのオブジェクトのメソッドをリモートから呼び出すことを可能にしている。これらのクラスに共通に使われたインターフェースは、インターフェース java.rmi.Remote を実装していなくてはならない。インターフェース Remote を実装したインターフェースのことをリモートインターフェースと呼ぶ。 RMI では、リモートインターフェースを実装して分散オブジェクトとして機能するオブジェクトのことを、リモートオブジェクトと呼んでいる。

RMI はこれらの機構を利用することで、ユーザに対してリモートオブジェクトを抽象度の高い形で扱えるようにしている。分散環境でのリモートオブジェクトの参照渡しは、そのスタブオブジェクトを渡すことで実現される。リモートオブジェクトの参照はメソッドの引数、またはメソッドの戻り値としてリモートに渡すことができる。この時、リモートオブジェクトそのものを渡したとしても、受け取った先のリモートマシンではそのスタブオブジェクトとなるように RMI が自動で変換してくれる。リモートオブジェクトとそのスタブオブジェクトとスケルトンオブジェクトは同じインターフェースで扱われており、ユーザにはあたかも同一のオブジェクトを扱っているように感じられる。リモートオブジェクトの参照、つまり、そのスタブオブジェクトを得たオブジェクトはそれを更に別のマシン上で動作している第 3 のオブジェクトに渡すことも可能である。第 3 のオブジェクトは、たとえそのオブジェクトのクラスファイルをローカルマシン上に持っていなくても、そのオブジェクトの物理的なアドレスやそれがどのように渡ってきたのかなどを知らなくても、そのオブジェクトへアクセスすることができる。リモートオブジェクトはネームサーバに

登録することで、どのリモートマシンからでも、その参照を得ることができるようになる。しかし、リモートからオブジェクトを生成することはできない。オブジェクトは必ずローカルで生成し、そして、それをリモートに渡すようにしている。

RMI はリモートオブジェクトを抽象度の高い優れた形で扱えるようにしているが、それは逐次処理に適したオブジェクトとしてである。リモートオブジェクトのメソッド呼び出しは通常のメソッド呼び出しと同じように、リモートでの計算が終了して戻り値が戻ってくるまで待たなくてはならない。リモートでの計算の返答速度はネットワークを介する分、ローカルで行なうよりもかなり遅くなる。更に悪いことにリモートで計算を行なっている間、ローカルでは何も計算を行なっていない。 RMI でリモートオブジェクトを並行に動作させたい場合、ユーザがスレッドを使ってそのように動作するようにプログラムを作らなくてはならない。この問題は分散環境では必ず起きる問題であるから、システムでもっと並行性について考えるべきである。

その他に RMI は独自のセキュリティマネージャを入れてしまったために Java との互換性が少し損なわれてしまっており、Java の動作する Web ブラウザでは動作せず、RMI のために特殊化した Web ブラウザが必要である。しかし、JDK 1.1 のライセンスでは、RMI をフルサポートする予定になっている。JDK とは Sun Microsystems の Java 開発キットのことである。従って、将来 RMI の動作する Web ブラウザが増えてくる可能性がある。

3.3 HORB

HORB [13, 14] もまた Java のためのリモートオブジェクトにアクセスするためのシステムである。 HORB は、リモートオブジェクトにプロキシオブジェクトとスケルトンオブジェクトを用意してやることで、リモートからそのオブジェクトにアクセスできるようにしている。しかし、 RMI のように独自のセキュリティマネージャを用意していないので、分散したマシン上にあるクラスファイルをロードすることはできない。従って、各リモートマシンにクラスファイルをあらかじめ用意してやるか、またはアプレットを利用するしかない。

HORB ではメソッドの引数または戻り値としてリモートオブジェクトの参照、またはそのコピーをリモートに渡すことができる。しかし、 RMI のようなオブジェクトシリアライザが無いので、プロキシを用意していないオブジェクトはリモートにそのコピーすら転送することができない。従って、Java にあらかじめ用意してあるライブラリのオブジェク

トなどはプロキシがないので転送することができない。しかし、配列だけは、 Java 内部ではオブジェクトとして扱われているにも関わらず、転送できるようになっている。しかし、配列をクラス Object にタイプキャストしてしまうと結局転送できなくなってしまう。

HORB では分散したクラスファイルをロードできない反面、ローカルマシンからリモートマシン上にリモートオブジェクトを生成することができる。しかし、ローカルにリモートオブジェクトを生成し、そのオブジェクトをリモートに渡すような場合には、リモートに生成する場合と同様にネットワーク上のアドレスを指定して、そのプロキシをローカルに生成しなくてはならない。そうやってオブジェクトを生成するとローカルにあるオブジェクトに対してもネットワークを介してアクセスすることになり、不自然である。また、プロキシの生成ではリモートオブジェクトのコンストラクタに引数で初期値を渡すことができない。その他にも、リモートオブジェクトとプロキシオブジェクトでは異なるところが多く、そのことが原因で分散計算の抽象度はあまり高くない。

しかし、並行性についてはいくらか考慮されている。リモートオブジェクトのメソッド名の後ろに _Async がついたメソッドは並行に動作させることができる。そのメソッドを並行に動作させるには _Async の部分を _Request に置き換えた名前でプロキシのメソッドを呼び出すことで行う。そして、計算結果を受け取る時にはメソッド名の後ろを _Receive に置き換えたメソッドを呼ぶことで受け取ることができる。しかし、計算結果を受け取る前にプロキシの他のメソッドを呼び出してはならないという規則がある。また、複数のリモートオブジェクトから 1 つのリモートオブジェクトに同時にアクセスする場合の同期の問題については考慮されていない。

その他にリモートオブジェクトのメソッドの名前の付け方などに制限があり、 RMI と比較するとぎこちなく、抽象度が低い。しかし、 Java と互換性があるため Java の動作する Web ブラウザで動作させることが可能である。

3.4 Obliq

Obliq [15] は Modula-3 のネットワークオブジェクト [11, 12] に似た分散オブジェクト指向言語である。Obliq で使われる名前の参照は全て静的にレキシカルスコープで解決することができる。実行はインタプリタによって行われる。 Obliq ではレキシカルスコープとインタプリタ方式により、名前の参照を正しく解決することで、ネットワークのセキュリティを保証している。 Obliq には型はない。型による複雑さを無くすことで分散処理が簡

単になるからである。

オブジェクトはプロトタイプに基づいて定義され、クラス階層はない。分散環境ではクラス階層よりもプロトタイプを用いたオブジェクト定義の方が適している。それは継承によるオブジェクト定義だと継承した全てのクラスについて知っている必要があるので、リモートから取り寄せなくてはならない情報が多くなる。このため、分散環境の負荷が重くなるからである。

オブジェクトのフィールドはメソッドとエイリアスと変数から構成されている。エイリアスは他のオブジェクトのフィールドに、別名をつけてそのオブジェクトのフィールドとして使う仕組みである。オブジェクトの操作には変数の参照とメソッド呼び出し、メソッドの更新、変数の更新、クローンの作成、エイリアスの追加更新がある。メソッドの更新はオブジェクトにすでに定義されているメソッドを定義し直す操作である。オブジェクト外部からの操作はオブジェクトを保護する目的で制限することが可能である。また、あるオブジェクトを同時に操作できるスレッドの数を 1 つに制限することも可能である。これらの制限はキーワードを 1 つ指定するだけで簡単に行うことができる。

Obliq ではオブジェクトはネットワーク上を移動することはできないが、その参照をクロージャと共に渡すことで、その参照をどこへでも渡すことができる。自由変数のスコープはその変数を定義したローカルマシンの環境での値が使用される。クロージャはそのために使用される。Obliq はクロージャを使うことで分散環境での自由変数の問題をうまく解決している。

Obliq ではクローンの作成とエイリアスの更新をうまく使うことでオブジェクトマイグレーションを実現することができる。オブジェクトマイグレーションとは通信を行っている 2 つのオブジェクトがあり、そのうちの 1 つがあるマシンから別のマシンに移動してもなお通信を行うことができるように移動することを指す。 Obliq は言語レベルでオブジェクトマイグレーションを実現しているわけではないが、それを簡単にプログラミングできることは非常に優れている。

Obliq は言語レベルで並行性もある程度意識した構造になっている。前述したスレッドのオブジェクトへのアクセスの制限がそうである。これにより、クリティカルセクションを排他的に制御することができる。しかし、オブジェクト間のメッセージ通信の方式は逐次オブジェクトのメソッド呼び出しと全く同じである。

3.5 CORBA

CORBA [16] はオブジェクトマネージメントグループ（OMG）によって仕様をまとめられている The Common Object Request Broker : Architecture and Specification の略称である。Specification という言葉からも分かるとおり CORBA はある特定のシステムを指すのではなく、分散オブジェクト環境のためのアーキテクチャと仕様を定義したものである。CORBA は現在大半が出来上がっているが、まだ未完成である。現在もまだ仕様の検討作業は続けられている。

CORBA の主な目的はマシンアーキテクチャや実装言語の異なったリモートオブジェクトのメソッドを透過的に呼び出すことを可能にすることである。これをするためにオブジェクトのネットワークインターフェース部分を、インタフェース定義言語（IDL）で記述する。現在 C、C++、Smalltalk、Ada へのマッピングが定められており、COBOL、Java へのマッピングが検討中である。これにより、これらの言語間での移植性、相互運用性は非常に高めることができる。しかし、ユーザは新たに IDL を憶えるという労力をはらわなくてはならない。

CORBA では並行計算についてはあまり考慮されておらず、メソッド呼び出しはリモート・プロシージャ・コール（RPC）を拡張した程度に留まっている。それと、CORBA によって定義されている仕様はとても膨大であり、フルスペックで動作する ORB を実用的なものとして作成することが可能かどうかを疑問視する声も出ている。

3.6 ConcurrentSmalltalk

ConcurrentSmalltalk [17, 18, 19] （以下 CST と略す）は Smalltalk-80 に並行計算能力を付加した並行分散オブジェクト指向言語である。CST ではプロセスとオブジェクトを結びつけ、オブジェクトを並行に動作できるようにしている。このオブジェクトを CST ではアトミックオブジェクトと呼んでいる。アトミックオブジェクトでは複数のオブジェクトからのメッセージの受信を、正しく行うために相互排除を行っている。CST ではメッセージ通信の処理に関して以下の 3 つを実現している。

- リモートプロシージャコール（RPC）の受信側は返答を返した後にも実行を継続する
- RPC を同時に複数起動し、全ての受信側からの返答を待って実行を再開する

- 送信側は返答を待たずに実行を再開し、返答が必要になった時に返答を受け取るための待ち合わせをする

しかし、第 3 のオブジェクトが送信側に対して返答することはできない。このことは並行性の記述の柔軟性を損なわせている。

Smalltalk-80 にはクラス変数のように複数のオブジェクト間で共有できる共有変数が存在する。共有変数がある場合に並行動作を行うには、共有変数に対して相互排除を行わなければならない。本来ならばオブジェクトは自己完備な性質のものであるので、共有変数は無い方がよい。しかし、 CST では lock と release というプリミティブメソッドによって共有変数の相互排除をできるようにしている。CST の設計方針は Smalltalk-80 における並行計算の記述容易性と理解容易性の向上であった。しかし、この部分に関しては依然記述性、理解容易性共に良くなってはいない。

3.7 Multilisp

Multilisp [20] は lisp から派生した言語 Scheme を並行計算に適した形に拡張した言語である。この言語はオブジェクト指向言語ではない。しかし、フューチャーは Multilisp で初めて登場した。Multilisp にはフューチャーの他に、与えられたオペランドを並行に処理するオペレーション pcall がある。

しかし、複数のタスクに共有されているデータに対する相互排除は低レベルなものである。Multilisp には、相互排除を行うために replace(L, V) と replace-if-eq(L, V, X) というアトミックに処理することのできる特別なオペレーションが用意されている。 replace(L, V) は L に格納されている値をアトミックな操作で V に置き換えるというものである。戻り値には L に格納されていた古い値が返される。 replace-if-eq(L, V, X) は replace(L, V) によく似たオペレーションである。異なるところは L に格納されている値が X の時にだけ置き換えがされるということである。置き換えがされなかった時には戻り値は nil になる。Multilisp ではこの replace(L, V) と replace-if-eq(L, V, X) を使うことでセマフォを定義している。プログラムのある領域に対する相互排除はこのセマフォを使うことで行う。

3.8 Schematic

Schematic [21] も Scheme を並行計算のために拡張した言語である。Schematic ではプロセス間の通信にチャネルを使い、フューチャーを実現している。また、並行計算がしやすいように Schematic にも Multilisp の pcall と同じものがある。その他に plet, pbegin, pmap, pfor-each がある。これらはそれぞれ let, begin, map, for-each の並行処理版である。

Schematic の Multilisp と大きく異なるところは共有データの相互排除を行うためにオブジェクト指向を取り入れていることである。このオブジェクトはクラスによって定義され、継承もすることができる。 Schematic の相互排除のやり方は興味深い方法で行っている。オブジェクトのメソッドはユーザからは同時に実行されることがないように見える。しかし、実際には並行に実行しても構わないメソッドは並行に実行している。これを実現するためにメソッドの種類を 2 種類に分けている。1 つはインスタンス変数を参照するだけで値を変更しないタイプのものである。もう 1 つはインスタンス変数の値を更新するタイプのものである。前者のタイプのメソッドはこれから述べる方式により、他のメソッドと並行に実行することが可能である。

後者のタイプのメソッドはメソッドのボディを更に 2 種類に分類している。インスタンス変数の値を更新する前の部分と更新する部分にである。インスタンス変数の更新は become と呼ばれる特別なオペレーションによって行われる。 become はそのメソッドの評価値となる位置にのみ記述することができ、高々1 回だけ評価される。 become は引数で与えられた複数のインスタンス変数の更新をアトミックに行い、その評価値は更新を行ったオブジェクトとなる。つまり、インスタンス変数をメソッドのボディの途中で更新することはしない。このようにメソッドをインスタンス変数の更新の前段階と後段階に分けておき、前段階の処理は同じオブジェクトの他のメソッドの前段階の処理と重複して処理することを禁止する。後段階はどのメソッドと重複して処理をしても構わない。それともう 1 つ規則がある。それはメソッド呼び出しの処理の途中でインスタンス変数の値は変化しないというものである。これにより、インスタンス変数の値は、そのメソッドを処理し始めた時の値で固定される。

この方式により、並行オブジェクト内でも並行に実行できる部分は実行するようにしている。これで並行処理能力はかなり向上するが、逆にプログラムの記述容易性は低下すると考えられる。

第 4 章

MILK の設計

本研究で設計した言語は Java を拡張したものである。この言語には MILK と名前を付けた。これはコーヒーにミルクを入れると苦みが消えてマイルドになるのと同じように、Java を拡張した MILK は、 Java の抽象度の低いスレッド機能を覆い隠して、より並行計算に適した言語となっているからである。この章ではまず MILK の設計方針について述べ、次に言語仕様をどのように設計したのかについて説明する。そして、最後に MILK の簡単なプログラム例を紹介する。

4.1 設計方針

並行計算に適した言語を設計するにあたり、開発に使用する言語として Java を選んだ。選定理由には低レベルとはいえ、マルチスレッドを言語レベルで備えており、並行計算のためのプログラミングが他の言語よりも容易と考えられたことや、Java のもつプラットフォームに依存しないポータビリティの高さなどが挙げられる。

新しく設計した言語 MILK は Java 言語を拡張することで作成した。そして、Java が持っているセマンティクスをなるべく崩さないようにした。こうすることで Java の持っている長所を多く取り入れることができる。また、 MILK のユーザが Java を知っていれば、全く新しい言語を覚えるよりも早く MILK について覚える事ができる。幸いなことに Java は C++ に似せて設計してあるので、多くのユーザが容易に MILK を覚える事ができる。

MILK の実装は MILK ソースコードを Java ソースコードに変換するトランスレータを作成する事で実現した。これはコンパイラを作るよりも実装が容易だからである。またそ

れだけでなく、Java に変換したソースコードが Java で正しく動作することで、MILK で作成したアプリケーションが Java の上で正しい事が証明できるからである。

その他分散環境については、RMI や HORB などの他のシステムと組み合わせて使用することで実現できるようにした。

4.2 言語仕様の設計

MILK は Java を拡張し、並行オブジェクトを組み込んでいる。このオブジェクトは過去型と未来型のメッセージ通信ができる。しかし、Java に初めからある逐次オブジェクトとスレッドを取り除くようなことはしていない。これは並行オブジェクトが万能ではなく、逐次オブジェクトやスレッドを使うことの方が適した場合があるからである。例えば、並行オブジェクトは逐次オブジェクトに比べると呼び出しに伴うオーバーヘッドが多い。従って、処理速度が重視される問題に関しては逐次オブジェクトが適していることもある。また、並行オブジェクトよりもスレッドを直接操作する方が、問題に適したきめ細やかな同期を取ることが可能である。従って、これらを適材適所、使い分けていく方が良いと考えた。

言語仕様について説明をする前に述べておかなくてはならない表記上の規則が 1 つある。Java はオブジェクト指向言語であり、メソッドのオーバーロードが許されている。従って、メソッドを正しく区別するためには、メソッドの引数の個数やその型について述べる必要がある。しかし、メソッドをいちいち引数まで書いて説明していると文章が大変読みにくくなる。そこで、混乱が特に生じない場合は、メソッドを引数を省略して名前に () をつけたもので表記する。

この節では MILK の言語仕様について、Java から拡張した部分だけを詳しく説明する。

4.2.1 並行オブジェクト

MILK では並行オブジェクトを扱えるように Java を言語拡張してある。この並行オブジェクトはこれだけで過去型のメッセージ通信が可能で、受信したメッセージを順番に 1 つずつ処理する。並行オブジェクトの内部構造は、メッセージキューとスレッドを持っており、他のオブジェクトからメッセージが送られてくると、そのメッセージを一時キューに蓄え、そしてスレッドがキューからメッセージを 1 つずつ取り出して実行していくよう

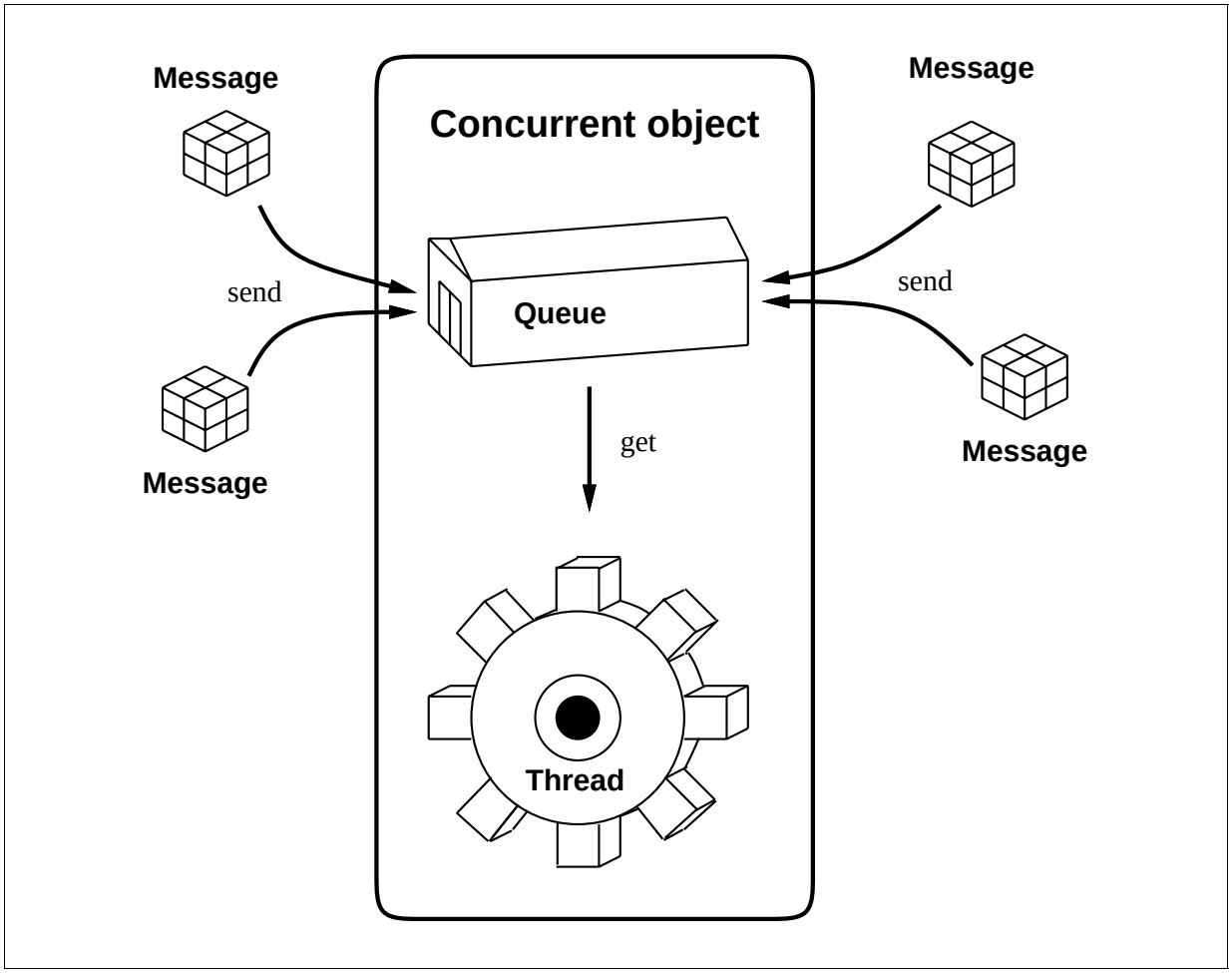

図 4.1: 並行オブジェクトの概念図

になっている。図 4.1 はこのモデルを図に示したものである。

並行オブジェクトのクラス定義は Java のクラス定義と似ていて、クラス定義の修飾子に concurrent を付けることで定義する。以下は並行オブジェクトのクラス定義の例である。

```
concurrent class A extends B implements C, D {
    ....
}
```

並行オブジェクトのフィールドの定義は並行オブジェクトを保護する目的で、フィールド変数とメソッド、それにスタティックイニシャライザの定義に制限が加えられている。フィールドの定義にはその他にコンストラクタ定義があるが、これは Java の定義と同じである。コンストラクタ以外のフィールドにどのような制限が加えられているのかについては、後で順番に詳しく説明する。その前にまず、これらに深く関連するアクセスレベルの問題について説明する。そして最後に並行オブジェクトの継承について説明する。

アクセスレベル

並行オブジェクトは独立して他のオブジェクトと並行に動作するものであるから、インスタンス変数はオブジェクト外部からは直接アクセスされるべきものではない。もし、直接インスタンス変数にアクセスすることができると、正しく同期がとれなくなるからである。 Java のインスタンス変数へのアクセスレベルは表 4.1 に示すように 4 段階ある。文献 [22] によると Java の β バージョンでは、インスタンス変数のアクセスレベルを、その変数を宣言したクラスとそのサブクラスだけがアクセスできるように宣言することができた。しかし、その後のバージョンからはそのような宣言はできないようになっている。これは残念なことである。なぜなら、このようなアクセスレベルを指定できないので、次のような問題が生じるからである。

インスタンス変数を 1 番厳しいアクセスレベル private で宣言した場合、並行オブジェクト保護の立場からすれば何も問題はない。しかし、インスタンス変数を private にすると、親クラスのインスタンス変数を直接アクセスすることができなくなる。これではプログラムの記述に柔軟性が無くなってしまう。そこで今度はアクセスレベルを protected にすると、同一パッケージの他のクラスのオブジェクトから並行オブジェクトのインスタンス変数にアクセス可能となる。パッケージとは Java のコンパイルの単位である。これでは並行オブジェクトを保護することができない。

修飾子	アクセス可能な範囲
public	全てのクラスから参照可能
protected	同一パッケージ内の全てのクラスと パッケージ外のそのクラスのサブクラスから参照可能
（なし）	同一パッケージ内の全てのクラスから参照可能
private	そのクラスからのみから参照可能

表 4.1: フィールドのアクセスレベル

どちらのアクセスレベルも一長一短があり、今のところどちらが最善であるのかは、はっきりしない。そこで、言語の振舞をどちらにするかはユーザが簡単に変更できるようにした。変更の仕方は 5.3 節で説明する。説明の便宜上、以後並行オブジェクト内でのみ使うことを目的とした変数を内部変数と呼び、並行オブジェクトの外部からアクセスしても良い変数を外部変数と呼ぶことにする。

同様なことはメソッドにも述べることができる。並行オブジェクトのメソッドには並行オブジェクト外部から呼び出されることを目的とし、並行オブジェクトにメッセージを送ることでそのオブジェクトに逐次処理されるメソッドと、並行オブジェクト内部で補助的に使われることを目的とし、逐次オブジェクトのメソッド呼び出しと同じ形態で使われるメソッドの 2 種類がある。こちらも便宜上、前者を外部メソッド、後者を内部メソッドと呼ぶことにする。

並行オブジェクトのインスタンス変数とメソッドのアクセスレベルは連動して変化するものとし、そのアクセスレベルを private と protected の 2 段階に分類した。並行オブジェクトのフィールドに対するアクセスレベルが private の時にはインスタンス変数は private のものが内部変数となり、メソッドも private のものが内部メソッドとなる。アクセスレベルが protected の時にはインスタンス変数は protected までのものが内部変数となり、メソッドも protected までのものが内部メソッドとなる。

全てのメソッドをメッセージキューを介した非同期通信の形にせずに、逐次オブジェクトのメソッド呼び出しの形態をした内部メソッドを採り入れている理由は、全てをメッセージキューを介した非同期通信にすると簡単にデッドロックが起きてしまうからである。

例えば、並行オブジェクトがある仕事を処理している最中に新しいメッセージがキュー

に入ったとする。このメッセージは今行なっている処理が終了するまで、キューから取り出して処理することはできない。なぜなら、1 つの仕事を処理している最中ではオブジェクトの状態は過渡的な状態になっており、意味のある正しい状態ではないからである。そのような状態の時に別の処理を行った場合、正しく計算をすることはできない。この状態で並行オブジェクトが内部処理のために、この並行オブジェクトのメソッドを呼んだとする。メソッド呼び出しが全てメッセージキューを介した非同期通信だとすると、このメソッド呼び出しにより新しいメッセージがキューに入る。このメッセージはすでにキューに他のメッセージが入っているので、すぐに実行されることはない。従って、デッドロックになる。

このデッドロックは後から呼び出したメソッドを、メッセージキューを介さずにすぐに実行することで回避することができる。そこで並行オブジェクト内部だけで使うことを目的としたメソッドの呼び出しは、逐次オブジェクトのメソッド呼び出しと同じ通信方法とした。

フィールド変数

フィールド変数には、クリティカルセクションを並行オブジェクトの中へカプセル化する目的で、制限が加えられている。外部変数は並行オブジェクト外部からアクセス可能なものであるから、このような変数が存在すると、並行オブジェクトのメソッド呼び出しを正しく計算することができない。なぜなら、メソッドの処理中にオブジェクト外部から変数の値を変更することができるからである。従って、外部変数の定義を禁止した。

逐次オブジェクトのクラス定義では、フィールド変数を定義する時に static を付けることで、クラス変数にすることができる。クラス変数は複数のオブジェクトで共有されるものであるから、同じクラスの複数のオブジェクトを並行に動作させる場合には、正しく同期をとる必要がある。しかし、この種の同期を自動的に判定して正しく行うことは、非常に困難なことである。従って、 static 宣言は final 宣言と共に用いる場合以外は、これを禁止した。 final 宣言は Java の修飾子の 1 つで、定数を宣言するためのものである。定数は同期をとる必要が無いので、これを許可した。以下は static 宣言の使い方の例である。

```
static final int i; // OK
static       int j; // ERROR
```

また、頭に $MILK$ を冠した名前の変数はシステムで使うので、ユーザの使用を禁止

した。

メソッド

内部メソッドは逐次オブジェクトのメソッドの定義と同様に定義することができるが、外部メソッドにはいくつかの制限がある。外部メソッドは並行オブジェクトにメッセージを送ることで非同期に逐次処理される。従って、外部メソッドには逐次オブジェクトのメソッドのような戻り値はない。そこで、外部メソッドの戻り値の型は必ず void とする。

また、Java ではメソッドの throws 節にメソッド内で発生する例外を書くことで、メソッドの呼び出し側にメソッド内で発生した例外を渡すことができる。しかし、外部メソッドの処理は、メソッドの呼び出し側のスレッドとは別の並行オブジェクト内部のスレッドが行うので、呼び出し側に例外を直接渡すことはできない。従って、外部メソッドでは throws 節は意味をなさないものとなるが、外部メソッドから throws 節を除去することはしていない。この理由はインターフェースで throws 節が定義されているメソッドを実装できるようにするためである。つまり、throws 節を書くことはできるが、その本来の用途のように呼び出し側に例外を渡すことはできない。従って、外部メソッドの内部で発生した例外は外部メソッド内で処理する必要がある。この throws 節の規則は MILK を RMI と組み合わせて使う場合にその効果を大きく発揮する。このことについては 6.1 節で説明する。

Java ではメソッドを static 宣言すると、同一クラスのオブジェクトで共有して使うことのできるメソッドを宣言することができる。このメソッドをスタティックメソッドと呼ぶ。しかし、MILK の並行オブジェクトにはクラス変数が無いので、スタティックメソッドの必要性はほとんど無い。それとスタティックメソッドは並行オブジェクトのプライベートなインスタンス変数に直接アクセスできるので、並行オブジェクトの同期を保つことが困難となり、大変危険である。従って、並行オブジェクトではスタティックメソッドの定義を禁止した。

メソッド run() は並行オブジェクト内部でシステムが使うので、ユーザが定義することはできない。メソッド run() は通常、スレッドが最初に実行するメソッドとして使われる。しかし、ユーザがこのメソッド run() からスレッドをスタートさせたとしても、何も実行しないで終了する。

並行オブジェクト内部でシステムが使うその他のメソッドの名前には、頭に $MILK$ を冠したものが使われる。ユーザは $MILK$ を冠した名前のメソッドを定義することはでき

ない。

スタティックイニシャライザ

Java では、クラス変数を初期化する目的で、クラスがインタプリタにロードされた時に 1 度だけ実行されるブロックを書くことができる。このブロックをスタティックイニシャライザと呼ぶ。スタティックイニシャライザはスタティックメソッドと同じ理由で並行オブジェクトでの使用を禁止した。

継承

並行オブジェクトの親クラスの指定は、逐次オブジェクトのクラス定義と同様にクラス定義の extends 節で行なう。 extends 節が省略された場合には、逐次オブジェクトのクラス定義では、 java.lang.Object が指定されたことになるが、並行オブジェクトのクラス定義では、 milk.runtime.ConcurrentObject が指定されたことになる。クラス ConcurrentObject では並行オブジェクトが共通して持っている性質が定義してあり、このクラスはクラス Object を継承している。全ての並行オブジェクトのクラスは、直接または間接的にクラス ConcurrentObject を継承しなくてはならない。従って、並行オブジェクトのクラスは逐次オブジェクトのクラスを継承することはできない。

しかし、逆に並行オブジェクトのクラスを逐次オブジェクトのクラスが継承することは実装の都合により制限していない。さらにこのような形で定義されたクラスを並行オブジェクトのクラスで継承することが可能である。これらのオブジェクトは並行オブジェクトとしては正しく機能しない。例えば、インスタンス変数を正しく保護することはできない。また、オーバーライドされたメソッドを正しく実行することもできない。しかし、このような継承関係で意味のある使い方ができるとは考えられない。このような継承はユーザのモラルで行わないようにするべきである。

インターフェースの継承は逐次オブジェクトのクラス定義と同様に行なうことができる。その他の点についても逐次オブジェクトのクラス定義と同様である。

4.2.2 フューチャー

MILK ではフューチャーを扱うことができる。MILK でのフューチャーの扱いは変数の扱いに似ている。この節ではフューチャーの宣言、参照、代入とその他の演算子について

順に説明し、その後でフューチャーで発生する例外について説明する。

宣言

フューチャーの宣言はローカル変数とインスタンス変数で行うことができる。ローカル変数での宣言では修飾子として future をつけることで行なう。型の指定、初期値の書き方などその他の規則は Java のローカル変数の宣言と同じである。

インスタンス変数での宣言もローカル変数での場合とほぼ同じであるが、アクセスレベルに対して制限が加わる。インスタンス変数をフューチャーにする場合には、修飾子に public, protected, private をつけることを禁じる。その代わりにインスタンス変数のフューチャーには private よりも強いアクセス制限を与える。通常のインスタンス変数は、例えば this.a のように、'.' で区切った名前でアクセスすることができるが、インスタンス変数のフューチャーではこのような '.' で区切った名前でのアクセスを禁止する。これにより、同じクラスのオブジェクトであっても、他のオブジェクトのインスタンス変数のフューチャーをアクセスすることはできない。この制限は MILK の実装の都合である。また、クラス変数をフューチャーにすることはできない。これも MILK の実装の都合である。このことについては 5.2.1 節で説明する。

こうして定義されたフューチャーを、並行オブジェクトのメソッド呼び出しのときに、引数として渡すことで未来型のメッセージ通信を実現している。現在型のメッセージ通信はフューチャーで代用することができるので特に実装をしなかった。以下はフューチャーの宣言の例である。

```
future int i, j;
future String str = "ABCD";
```

メソッドの仮引数でフューチャーを宣言する場合には、引数に修飾子 future をつけるだけで良い。以下はメソッドの仮引数をフューチャー宣言した場合の例である。

```
void foo(future int i) { ... }
```

参照

フューチャーは変数と同様のやり方で参照を行なうことができる。参照はフューチャーの値が定まっている場合には、すぐにその値を参照することができる。しかし、値がまだ定まっていない段階でフューチャーを参照した場合には、参照をしたスレッドは値が定ま

るまで待つことになる。そして、フューチャーの値が定まると、すぐにその値を参照し処理を再開する。

スレッドがフューチャーの値が定まるまで待っている間に、待っているスレッドに対してクラス Thread のメソッド interrupt() が呼ばれた場合には、例外 FutureException が発生する。例外 FutureException については後で説明する。

メソッドにはフューチャーの参照結果を渡すだけでなく、フューチャーそのものを渡すことができる。この場合には、実引数の前にキーワード future をつけることでそのことを示す。歴史的にはフューチャーに格納されている値を参照する場合にはキーワード touch を使い、フューチャーそのものを参照する場合には何もつけない言語が多い。しかし、MILK ではフューチャーを受けとるメソッドの定義は仮引数に future をつけて定義しているので、このことに合わせて実引数でもフューチャーそのものを渡す場合にキーワード future をつけるようにした。このように統一することで、どのメソッドを呼んでいるのかが直観的に理解しやすくなる。

代入

代入は通常の変数と同様に行なわれる。*= のような複合代入が可能な型の場合には、それらも正しく行なうことができる。フューチャーには通常の代入演算子の他に演算子 <- がある。<- はアトミックな操作である。そして、その機能は、フューチャーの値が未決定のときにのみ代入を行ない、値が決定している場合には何も行なわないというものである。<- の計算優先順位は他の代入文と同じである。

演算子 init!

その他にフューチャーには特別な演算子 init! と determined? が用意されている。init! はフューチャーの値を未代入の状態に戻すものである。 init! は代入文のように単独でプログラムのステートメントにすることができる。 init! の計算優先順位は ~ や ! などの単項演算子と同じである。

演算子 determined?

演算子 determined? はフューチャーの値が決定しているかどうかを boolean 型で返すものである。 determined? は値が決定している場合には true を返し、値が未決定の場合に

は false を返す。 determined? はそれだけでは単独でステートメントにすることができない。これは init! のように副作用を起こす演算ではないので、単独でステートメントにする価値がないからである。 detemined? の計算優先順位は~ や！ などの単項演算子と同じである。以下はフューチャーの例である。

```
future int x;
x = 3;
System.out.println(x);
init! x;
if (determined? x) {
    System.out.println("x is determined.");
}
else {
    System.out.println("x is not determined.");
}
```

例外 FutureException

Java ではプログラム実行中に何か異常が発生すると、例外を生成する。例外は普通 try catch 文で捕捉して適切な処理をするか、メソッドの throws 節で例外が発生する可能性のあることを示して、そのメソッドの呼び出し側に例外を渡さなくてはならない。

フューチャーからは例外 milk.runtime.FutureException が発生する可能性がある。FutureException はフューチャー内部で何らかの例外が発生した時に生成される。実際に発生した例外は FutureException のインスタンス変数 ex に格納される。変数 ex は public 宣言されているので、どのオブジェクトからでも参照することができる。また、メソッド getMessage() によって、どのような例外が発生したのかをある程度詳しく知ることもできる。この例外の発生理由の多くは分散環境によるものである。

FutureException は java.lang.RuntimeException を継承している。例外 RuntimeException は try catch 文で捕捉する必要や、呼び出しメソッドにその例外を投げる必要のない例外である。そして、この例外を継承した例外についても同様の性質がある。従って、例外 FutureException を必ずしも捕捉する必要はない。このような例外にした理由を以下に示す。

- 発生理由が分散環境での予期しない例外の場合、その最適な対処の方法が無い場合が多い

```
 1: class A {
 2:     public static void main(String args[]) {
 3:         future String str;
 4:         B b = new B();
 5:         b.foo(future str);
 6:         System.out.println("A: I can do another job.");
 7:         System.out.println("A: " + str);
 8:     }
 9: }
10:
11: concurrent class B {
12:     void foo(future String str) {
13:         str = "Hello World.";
14:         System.out.println("B: I can do a job yet.");
15:     }
16: }
```

図 4.2: 最も簡単な例

- フューチャーからの例外を必ず捕捉しなければならないとなると、ユーザの負担は大変重くなり、また、プログラムの可読性も悪くなる
- ユーザがクラス Thread のメソッド interrupt() を呼び出して例外を発生させる場合には、適切な箇所で例外を捕捉することができる

つまり捕捉しても対処の方法の無い例外多いので、例外を捕捉する価値がある時に必要な箇所だけで捕捉できるようにした。

4.3 MILK を用いた並行計算の例

図 4.2 は並行オブジェクトとフューチャーを使った最も簡単な例である。図の左側の番号は便宜上付けた行番号である。クラス B は並行オブジェクトなので クラス A のメソッド main() とクラス B のメソッド foo() は並行に動作する。5 行目でメソッド main() からメソッド foo() を呼び出すことで、並行オブジェクトにメッセージを送っている。この時、

フューチャー str を引数として渡している。メソッド foo() での計算結果がフューチャーに返ってくるまでの間、メソッド main() では別の計算を行うことができる。それが 6 行目である。メソッド foo() ではまず 13 行目でフューチャー str に値をセットしている。メソッド foo() がこの行を実行すると、メソッド main() はフューチャーの値を参照することができるようになる。つまり、7 行目を実行することができるようになる。メソッド foo() はフューチャーに値を返した後も計算を続けることができる。それが 14 行目である。

これと同じ意味のことはもちろん Java でも計算することができる。しかし、プログラムをここまで簡潔に書くことはできない。 Java で図 4.2 と同等の意味のプログラムを書くとすると、例えば図 4.3 のようになる。 Java ではユーザ自身が同期のとり方や計算結果の受け取り方などを決めなくてはならない。

フューチャーは図 4.4 のように第 3 のオブジェクトに渡すことができる。この例ではクラス A の知らないクラス C のオブジェクトがフューチャーに値を返している。

このように並行オブジェクトとフューチャーの導入による効果は大きいが、注意しなくてはならない点もある。並行オブジェクトとフューチャーによって起こされる副作用は当然のことながらレキシカルな順序になるとは限らない。図 4.5 のプログラムを実行すると、メソッド println() によって表示される値は 1 または 2 または 3 のどれかになり、実行した時によって異なってくる。また、2 つのメソッド println() の表示する値が同じになるとは限らないことにも注意されたい。

最初に得られた値を使いたい場合にはメソッド get() で行なわれるフューチャーへの代入の演算子 = を <- にすれば良い。こうすれば最初に行なわれた代入の結果だけを使うことができる。これにより、2 つのメソッド println() の表示結果も同じになる。特定のメソッド呼び出しの結果だけを使いたい場合には、もう 1 つ別にフューチャーを用意するしかない。

次に少し実用的な例を挙げてみる。図 4.6 は素数の生成プログラムである。エラトステネスのふるいによって逐次に生成しようとすると、あらかじめ求める数の上限を決めておき、ふるいに使われる配列などの初期化、素数の生成、結果の表示のそれぞれにおいて、ループを回すのが普通である。しかし、この例では発見された素数ごとにオブジェクトを割り当てており、ループは 1 つだけである。このやり方なら求める素数の上限の値を動的に変更することも可能である。この例ではフューチャーは特に使っていない。

その他にも並行オブジェクトを用いるとアプレットの描画を簡単に並行動作させることなどができる。 6.3 節ではもっと複雑な例を紹介している。その例ではフューチャーを用

```
class A {
    public static void main(String args[]) {
        Wrapper future = new Wrapper();
        B b = new B();
        Thread t = new Thread(b);
        b.future = future;
        t.start();
        System.out.println("A: I can do another job.");
        synchronized (future) {
            if (future.str == null) {
                try {
                    future.wait();
                }
                catch (InterruptedException e) {}
            }
        }
        System.out.println("A: " + future.str);
    }
}

class B implements Runnable {
    public Wrapper future = null;
    public void run() {
        if (future == null) {
            System.err.println("error");
            System.exit(1);
        }
        future.str = "Hello World.";
        synchronized (future) {
            future.notify();
        }
        System.out.println("B: I can do a job yet.");
    }
}
```

図 4.3: Java で書いた場合の最も簡単な例

```
class Wrapper {
    public String str = null;
}
```

図 4.3: Java で書いた場合の最も簡単な例 (つづき)

```
class A {
    public static void main(String args[]) {
        future String str;
        B b = new B();
        b.sayHello(future str);
        System.out.println(str);
    }
}

concurrent class B {
    void sayHello(future String str) {
         C c = new C();
         c.sayHello(future str);
    }
}

concurrent class C {
    void sayHello(future String str) {
        str = "Hello World.";
    }
}
```

図 4.4: 第 3 のオブジェクトへのフューチャーの譲渡

```
class A {
    public static void main(String args[]) {
        future int x;
        B b1 = new B(1);
        B b2 = new B(2);
        B b3 = new B(3);

        b1.get(future x);
        b2.get(future x);
        b3.get(future x);

        System.out.println(x);
        System.out.println(x);
    }
}

concurrent class B {
    private int x;

    B(int x) {
        this.x = x;
    }

    void get(future int x) {
        x = this.x;
    }
}
```

図 4.5: フューチャーによる副作用

```
import java.io.*;

concurrent class Sieve {
    private int num;
    private Sieve next;

    Sieve(int n) {
        num = n;
        next = null;
    }

    void work(int n, PrintStream out) {
        if (n == num) {
            out.println(num);
        }
        else if (n % num != 0) {
            if (next == null) {
                next = new Sieve(n);
            }
            next.work(n, out);
        }
    }
}

class Sample {
    public static void main(String args[]) {
        Sieve s = new Sieve(2);
        for (int i = 2; i < 1000; i++) {
            s.work(i, System.out);
        }
    }
}
```

図 4.6: 素数を生成するプログラム

いてアプレットの描画を並行に行わせている。また、分散環境でのフューチャーによる非同期通信を行っている。

第 5 章

MILK のアーキテクチャ

MILK のソースコードは、トランスレータで Java ソースコードに変換して、それを Java コンパイラでコンパイルすることで実行することができる。トランスレータは、まずソースコードをスキャナでトークンの列に切り分ける。切り分けられたトークンの列はパーサに渡され、構文木が作成される。木の各ノードはオブジェクトで構成され、各構文の必要な情報を保持している。そして、トランスレータはこの構文木をたどり、Java から拡張した構文を発見すると Java で動作するコードに変換しながら、再び文字列に戻していく。最後に、できた文字列をファイルに出力する。計算機言語の仕組みについては文献 [23] が参考になる。

トランスレータは JDK version 1.0.2 によって実装されている。JDK は Sun Microsystems がフリーで公開している Java の開発キットである。トランスレータのパーサ部分は CUP version 0.9e [24] によって実装されている。 CUP は YACC に良く似た Java のための LALR パーサのジェネレータである。

この章では MILK がどのようにして Java に変換され、 Java レベルではどのように実装されているのかを説明する。

5.1 並行オブジェクト

MILK では並行オブジェクトを図 5.1 のように定義する。この節ではこの例がどのように変換されるのかを見ながら説明していく。図 5.1 を Java に変換すると図 5.2 のようになる。図 5.3 は図 5.2 のメソッド呼び出しとメッセージの関係を図で現したものである。

```
concurrent class A {
    public void sayHello(String name) {
        System.out.println("Hello " + name);
    }
}
```

図 5.1: 並行オブジェクトのクラス定義

5.1.1 ConcurrentObject

図 5.1 では親クラスを指定していない。並行オブジェクトクラスでは親クラスが指定されていない時には milk.runtime.ConcurrentObject が指定された事になる。また、親クラスを指定する時には必ず ConcurrentObject を継承しているものを指定しなくてはならない。ConcurrentObject の実装は図 5.4 のようになっている。ConcurrentObject はインスタンス変数として、キューとスレッドを持っている。メソッド run() は並行オブジェクトのスレッドが最初に実行するメソッドである。このメソッドはこのクラスを継承したクラスで実装しなくてはならない。

メソッド $MILK$activate() はスレッド生成のためのメソッドである。 メソッド $MILK$checkQueue() はキューを調べ、キューが空の時にスレッドを破棄するためのメソッドである。この 2 つのメソッドは同時に実行される事があってはならないので synchronized 宣言してある。 Java では、同じオブジェクトの synchronized 宣言したメソッドは同時に実行されることはない。スレッドの扱い方には、必要な時に生成して必要が無くなった時に破棄する方法を採用した。この他に 1 回だけスレッドを生成してそのスレッドを必要に応じてサスペンドさせたりレジュームさせたりする方法が考えられる。この方法はサスペンドとレジュームのコストが破棄と生成のコストよりも安いように思えるが、実際はほとんど差が無い。また、サスペンドしたスレッドがあると Java プログラムは終了しないので、プログラムの終了条件についても考えなくてはならない。従って、前者の方法を採用した。メソッド $MILK$checkQueue() の変数 tmp はカレントスレッドが $MILK$thread と等しい時でも正しく動作させるために必要である。

$MILK$select() については後で説明する。 $MILK$error() は並行オブジェクトで発生したエラーを通告するためのメソッドである。

```
class A extends ConcurrentObject  {
    private void $MILK$sayHello(String name) {
        System.out.println("Hello " + name);
    }

    public synchronized void sayHello(String name) {
        $MILK$queue.put("A");
        $MILK$queue.put(new Integer(0));
        $MILK$queue.put(name);
        $MILK$activate();
    }

    protected void $MILK$select(String name, int num) {
        if (name.equals("A")) {
            switch (num) {
           case 0:
                $MILK$sayHello((String)$MILK$queue.get());
                break;
           default:
                $MILK$error(name, num);
                break;
            }
        }
        else {
            super.$MILK$select(name, num);
        }
    }
```

図 5.2: Java に変換された並行オブジェクトのクラス定義

```
    public void run() {
        if ($MILK$thread != Thread.currentThread()) {
            return;
        }
        while (true) {
            $MILK$checkQueue();
            $MILK$select((String)$MILK$queue.get(),
                         ((Integer)$MILK$queue.get()).intValue());
        }
    }
}
```

図 5.2: Java に変換された並行オブジェクトのクラス定義 (つづき)

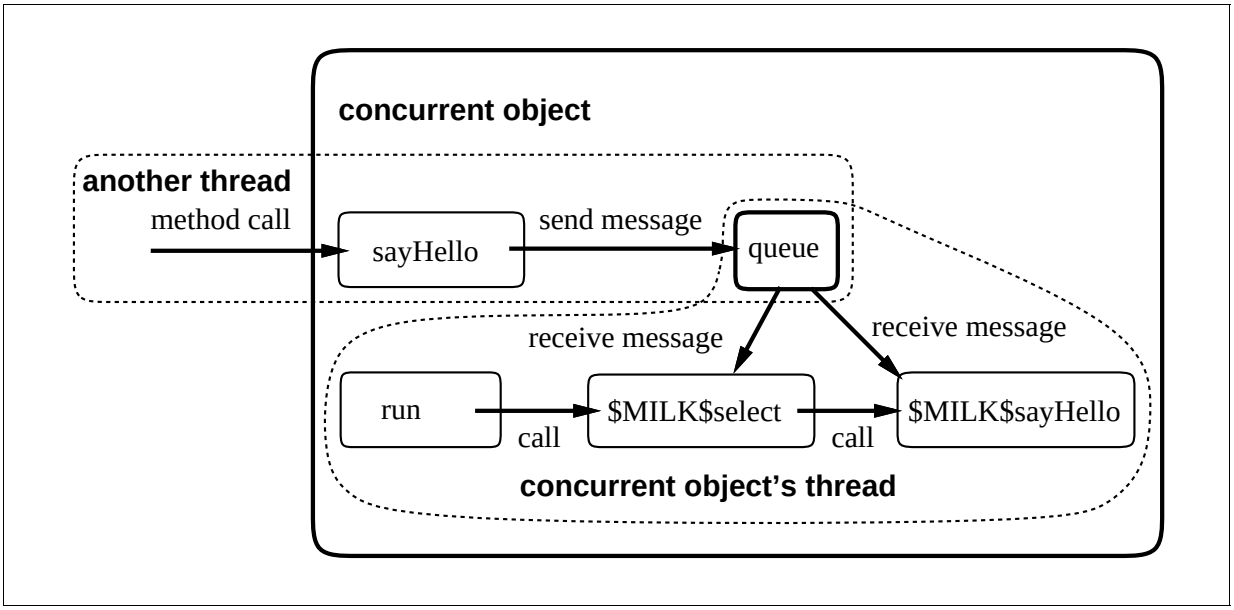

図 5.3: メソッド呼び出しの流れ

```
public abstract class ConcurrentObject implements Runnable {
    protected Queue $MILK$queue = new Queue();
    protected Thread $MILK$thread;

    public abstract void run();

    protected synchronized final void $MILK$activate() {
        if ($MILK$thread == null) {
            $MILK$thread = new Thread(this);
            $MILK$thread.start();
        }
    }

    protected synchronized final void $MILK$checkQueue() {
        if ($MILK$queue.isEmpty()) {
            Thread tmp = $MILK$thread;
            $MILK$thread = null;
            tmp.stop();
        }
    }

    protected void $MILK$select(String name, int num) {
        $MILK$error(name, num);
    }

    protected void $MILK$error(String name, int num) {
        System.err.println("ConcurrentObject: Error. Unknown message "
                           + "'" + name + " " + num + "'");
        System.exit(1);
    }
}
```

図 5.4: クラス milk.runtime.ConcurrentObject の定義

5.1.2 変換

トランスレータは並行オブジェクトクラス定義に対して以下の事をする事で変換を行う。この節ではこれらについて順に説明していく。

- メッセージ送信メソッドの生成
- メッセージ解釈メソッドの生成
- メッセージ処理メソッドの生成
- メソッド run() の生成

メッセージ送信メソッド

並行オブジェクトは外部からメッセージを受けとり、メッセージを逐次処理する事で活動する。メッセージ送信メソッドは並行オブジェクト外部から並行オブジェクトにメッセージを送信するためのものであり、並行オブジェクト外部からはこのメソッドを使って、このオブジェクトにメッセージを送信する。このメソッドは、並行オブジェクトクラス定義で定義されている外部メソッドに対応して作成される。メソッド名や引数の個数や型、メソッドの修飾子は外部メソッドの定義と同じものが使われる。

図 5.1 ではメソッド sayHello() が外部メソッドとして定義してあるので、このメソッドのためのメッセージ送信メソッドが作成される。以下に示すものは図 5.2 のものである。

```
public synchronized void sayHello(String name) {
    $MILK$queue.put("A");
    $MILK$queue.put(new Integer(0));
    $MILK$queue.put(name);
    $MILK$activate();
}
```

このメソッドはメッセージキューに複数回アクセスするため、他のキューにアクセスするメソッドと同期を取る必要がある。従って、このメソッドは synchronized 宣言してある。外部メソッドには、トランスレータによってクラスごとに 0 から始まる通し番号がふられる。メッセージキューにはクラス名とこの通し番号が入れられる。そして、外部メソッドに引数がある場合には、それもメッセージキューに入れられる。この例ではクラス名 A、通し番号 0、引数 name がキューに入れられている。そして最後に、メソッド $MILK$activate()

を呼んで、メッセージが届いた事を並行オブジェクトに内蔵されているスレッドに通知する。このメッセージのフォーマットは他のクラスに依存していないので、親クラスの変更が子クラスに波及するような事がない。従って、親クラスが変更されたからといって子クラスまでコンパイルする必要は無い。

メッセージ解釈メソッド

並行オブジェクト内部のスレッドは、メソッド $MILK$select() によってメッセージを解析して、実行すべきメソッドを選択し実行する。このメソッドは各クラスごとに１つ作成される。以下に示すメソッドは図 5.2 のものである。

```
protected void $MILK$select(String name, int num) {
    if (name.equals("A")) {
        switch (num) {
       case 0:
            $MILK$sayHello((String)$MILK$queue.get());
            break;
       default:
            $MILK$error(name, num);
            break;
        }
    }
    else {
        super.$MILK$select(name, num);
    }
}
```

仮引数の name と num は実行すべきメソッドのクラス名と通し番号である。name がこのメソッドの属するクラスの名前と異なる時には親クラスのメッセージ解釈メソッドを呼び出す。 name がこのクラスの名前と一致している時には、 switch 文で通し番号 num と一致するメソッドを選択する。この例では name が A で num が 0 のときに $MILK$sayHello() が実行される。そして、この時、実行すべきメソッドに渡す実引数がメッセージキューから取り出される。

このようにクラス名とメソッドの通し番号によるメッセージと、子クラスから親クラスへと探索していく方法によって、オブジェクトの継承によって生じるメソッドのオーバーライドやオーバーロードの問題をうまく処理することができる。

外部メソッドの定義で throws 節がある場合には、そのための try catch 文が挿入される。例えば、仮にメソッド sayHello() の throws 節で例外 Exception が定義されているとすると、上のソースコードの case 0: のところは以下のようになる。

```
case 0:
    try {
        $MILK$sayHello((String)$MILK$queue.get());
    }
    catch (Exception e) {
        System.err.println("Exception ocurred "
                           + "in a cuncurrent object: "
                           + e.getMessage());
        e.printStackTrace();
        System.exit(1);
    }
    break;
```

メッセージ処理メソッド

メッセージ処理メソッドはメッセージ解釈メソッドから呼び出され、実際にメッセージの処理を行なうためのメソッドである。このメソッドは並行オブジェクトクラス定義で定義されている外部メソッドを元にして作成される。外部メソッドとの差異は、まずメソッド名が $MILK$ を冠したものになる事と、アクセスレベルが必ず private になる事である。このメソッドは並行オブジェクトの内部だけで使用されるので、アクセスレベルを private にして保護してある。メソッドのボディやその他の部分は外部メソッドの定義と同じである。以下に示すものは図 5.2 のものである。

```
private void $MILK$sayHello(String name) {
    System.out.println("Hello " + name);
}
```

メソッド run()

Java ではオブジェクトがインターフェース Runnable を実装している場合、そのクラスのメソッド run() を、スレッドが 1 番最初に実行するメソッドとすることができる。クラス ConcurrentObject はインターフェース Runnable を実装しているので、並行オブジェク

トによって生成されたスレッドはまずこのメソッドから実行を開始する。このメソッドは各クラスに１つ同じものが生成される。

```
public void run() {
    if ($MILK$thread != Thread.currentThread()) {
        return;
    }
    while (true) {
        $MILK$checkQueue();
        $MILK$select((String)$MILK$queue.get(),
                     ((Integer)$MILK$queue.get()).intValue());
    }
}
```

このメソッドは Java の取り決めで public にしなくてはならない。従って、並行オブジェクトの外部からこのメソッドをアクセスすることができる。しかし、このメソッドの最初の if 文によって、不正にこのメソッドにアクセスした場合には何もしないで戻るようになっている。その後の while 文はメッセージがあるかどうかを調べ、あればそのメッセージを取り出して、メッセージ解釈メソッドを実行するという事を、繰り返し行なっている。並行オブジェクトはスレッドを一時に高々1 つしか生成しない。従って、メッセージが逐次処理されていく事を保証することができる。

5.2 フューチャー

フューチャーはあらかじめ用意しておいたクラスライブラリのコンストラクタ呼び出し、またはメソッド呼び出しに変換される。このクラスライブラリは Java のプリミティブな型である boolean, byte, char, double, float, int, long, short の全てとオブジェクトに対して、それぞれ用意してある。配列は Java の内部ではオブジェクトとして表現されるので、オブジェクト用のものを使う。これらのライブラリはほとんど同じ構造になっているので、代表して int 型のものを例にとって説明し、異なる部分についてはその都度説明することにする。

これらのクラスでは value と determined という２つのインスタンス変数を定義している。value はフューチャーの値を保存しておくための変数で、型はそれぞれのライブラリで適切な型が使われている。 determined はフューチャーの値が定まっているかどうかを表

す boolean 型のフラグである。これらの変数は private で外部から直接アクセスすることはできない。

それではフューチャーがどのように変換されるのかを順番に説明していく。

5.2.1 宣言

フューチャーの宣言では、そのフューチャーをアクセスした時のために、名前や型について記憶しておく必要がある。トランスレータは、1 つのクラス定義の中だけで解決することのできる名前の参照についてのみ調べる。これは名前の参照先の実体がクラスの外部に存在すると、探索するのが非常に困難になるので、実装を簡単にする目的でクラス内に限定している。

トランスレータはプログラムのブロック構造を 1 つの単位としてフレームを作り、フレームをスタックすることで変数に関する情報を記憶している。フレームはハッシュテーブルを持っており、変数に関する情報がその名前をキーとしてそこに記憶されている。変数の情報はその変数の型、フューチャーであるかどうか、フィールド変数であるかどうかについて記憶されている。

トランスレータはクラス定義の構文木を発見すると、まずそのクラスで定義されているインスタンス変数のフューチャーをフレームとして記憶する。その後でクラス内で定義されているメソッドをそれぞれ調べていき、ローカル変数が宣言されていれば、それをブロックごとに別々のフレームで記憶していく。

このような方法で名前の参照を解決しているので、ローカル変数またはメソッドの仮引数で宣言したフューチャーは、その参照関係を解決することができる。一見するとインスタンス変数も、プライベートなものならばこのやり方で解決することができるように思える。しかし、 Java は同一のクラスであれば、他のオブジェクトのプライベートな変数にアクセスすることが可能である。この場合、名前は '.' で区切られた複合的な名前が使われる。例えばクラス名を A とし、そのクラスに属するオブジェクト名を obj とし、そのオブジェクトのインスタンス変数名を v とすると以下のように参照することができる。

```
obj.v;
(new A()).v;
```

その他にも、配列を使った参照など複雑な参照の仕方がいろいろとある。これをみて分かるように、複合的な名前の解決は単純な名前の解決よりもかなり難しい。そこで、フューチャー

には複合的な名前ではアクセスすることができないという制限を設ける。これを実現するためにインスタンス変数のフューチャーの名前は元の名前に $MILK$INSTANCE_VAR$ を冠したものに置き換え、そのアクセスレベルを private にする。 $MILK$ を冠した名前はスキャナによってユーザが使えないようにしているので、トランスレータで正しく変換すれば、たとえ同じクラスのオブジェクトであったとしても、インスタンス変数のフューチャーにアクセスすることはできなくなる。これにより名前の参照の解決方法を単純化することができる。

ローカル変数またはインスタンス変数でのフューチャーの宣言の変換は初期値がある場合と無い場合で異なってくる。また、宣言が配列の場合にはさらに異なった変換となる。まずは簡単な変換でできる配列でない場合について説明する。この場合の宣言は次のようになる。

```
future int x;
future int y = 10;
```

これらの宣言がローカル変数のものであるとして、変換を行うと次のようになる。

```
IntFuture x = new IntFuture();
IntFuture y = new IntFuture(10);
```

また、インスタンス変数の場合には次のようになる。

```
private IntFuture $MILK$INSTANCE_VAR$x = new IntFuture();
private IntFuture $MILK$INSTANCE_VAR$y = new IntFuture(10);
```

int 型のフューチャーは IntFuture という型に置き換えられ、宣言に初期値があろうとなかろうと IntFuture というオブジェクトが生成される。このオブジェクトはフューチャーの振舞を管理しているオブジェクトで、以後このオブジェクトのことをフューチャーオブジェクトと呼ぶことにする。フューチャーに初期値がある場合には、フューチャーオブジェクトのコンストラクタにそれが引数として渡される。このコンストラクタの実装は以下のようになっている。

```
public IntFuture() {
    determined = false;
}

public IntFuture(int x) {
```

```
        value = x;
        determined = true;
    }
```

初期値がある場合と無い場合で変数 determined の初期化の仕方が異なっている。

フューチャーの型と変換されて生成されるフューチャーオブジェクトの型との関係は 5.3 節で説明する。

次に配列の宣言の場合である。配列は以下のように宣言される。

```
future int a[];
future int b[] = new int[5];
future int c[] = {3, 4, 5};
```

配列は Java の内部ではオブジェクトとして扱われているので、IntFuture ではなくて ObjectFuture に置き換えられる。a と b の場合は先の例と同じやり方で置き換えられる。しかし、c の場合はコンストラクタに引数として直接 {3, 4, 5} を渡すことはできない。そこで、とりあえず配列を別の名前で宣言してから、 ObjectFuture のコンストラクタにその変数を渡すことで変換する。配列につける別の名前にはその変数の名前に $MILK$ を冠したものを使う。この場合は $MILK$c となる。以下は宣言がローカル変数のものだとした場合の変換結果である。

```
ObjectFuture a = new ObjectFuture();
ObjectFuture b = new ObjectFuture(new int[5]);
int[] $MILK$c = { 3, 4, 5 };
ObjectFuture c = new ObjectFuture($MILK$c);
```

次元の異なる変数が 1 つの宣言文でまとめて宣言されている場合、変換される型や変数の持つ情報が異なってくるので、1 つの宣言文に変換することはできない。そこで、複数の変数をまとめてフューチャーとして宣言している構文の変換は、1 つずつ宣言する構文に変換される。つまり、

```
future int x, y, z[];
```

が、ローカル変数の宣言だとすると

```
IntFuture x = new IntFuture();
IntFuture y = new IntFuture();
ObjectFuture z = new ObjectFuture();
```

のように変換される。

メソッドの仮引数でフューチャーが宣言されている場合には単に型の変換だけが行なわれ、フューチャーオブジェクトの生成は行なわれない。以下は仮引数での宣言の例である。

```
void foo(future int x, future int y[]) {
}
```

これを変換すると以下のようになる。

```
void foo(IntFuture x, ObjectFuture y) {
}
```

5.2.2 参照

フューチャーの参照の変換にはローカル変数またはインスタンス変数の参照と、メソッドの実引数としてフューチャーそのものを渡す場合の 2 種類がある。ローカル変数またはフューチャーであるインスタンス変数はトランスレータによってその情報が記憶されているので、ソースコード中の変数の参照がフューチャーの参照であるかどうかを知ることができる。フューチャーで無い変数は変換する必要が無いので、フューチャーで無いインスタンス変数は記憶していない。しかし、フューチャーで無いローカル変数はフューチャーであるインスタンス変数と同名であることもあるので、全て記憶している。

トランスレータは参照されている変数がフューチャーだと分かると、その変数をフューチャーオブジェクトのメソッド ref() の呼び出しに変換する。この時、参照されている変数がインスタンス変数の場合には、 $MILK$INSTANCE_VAR$ を冠した名前に置き換える。オブジェクトを扱っているフューチャーの場合には、型をそれぞれの型で扱っているわけではなく、Object 型に変換して共通のフューチャーオブジェクトで扱っている。従って、それが参照された時にはふさわしい型にタイプキャストしてやる必要がある。その型はトランスレータがフューチャーの宣言時に記憶しているので、正しくタイプキャストしてやることができる。

メソッドの実引数にフューチャーそのものを渡す場合には、メソッド ref() の呼び出しは行なわない。以下はこれらの参照例である。

```
future int x;
future A y;
foo(x, future x, y);
```

これを変換すると以下のようになる。ただし、フューチャーの宣言はすべてローカル変数だとする。

```
IntFuture x = new IntFuture();
ObjectFuture y = new ObjectFuture();
foo(x.ref(), x, ((A)y.ref()));
```

メソッド呼び出しの 3 番目の実引数に余分な括弧がついているのはタイプキャストされる範囲をはっきりさせるためのものである。これにより複雑な式の中であっても正しくタイプキャストされる。

フューチャーオブジェクト IntFuture のメソッド ref() の実装は以下のようになっている。

```
public synchronized int ref() throws FutureException {
    if (! determined) {
        try {
            wait();
        } catch (InterruptedException e) {
            throw new FutureException("InterruptedException", e);
        }
    }
    return value;
}
```

フューチャーオブジェクトは、正しく同期をとるために、全てのパブリックなメソッドが synchronized 宣言されている。メソッド ref() は変数 determined を調べて、値が決定していたら即その値を返し、値がまだ未決定の場合には値が決定されるまで待つこと。そして、値が決定したところでその値を返す仕組みになっている。待っている間には例外 InterrruptedException が発生することもある。この例外は他のスレッドが、待っているスレッドに対してクラス Thread のメソッド interrupt() を呼ぶことで発生する。この例外が発生した場合には代わりに例外 FutureException を投げている。

Java の例外には大きく分けて 2 種類あり、例外の発生する可能性のある箇所では必ず try catch 文で捕捉するかまたは呼び出しメソッドに投げなくてならない例外と、特に何も例外処理をしなくても良い例外がある。後者の例外は RuntimeException を継承していなくてはならない。例外 InterruptedException は前者の例外であり、発生する可能性のある箇所では必ず何らかの処理をしなくてはならない。例外 FutureException は 後者の例外である。通常、例外 InterruptedException はユーザが明示的に発生させない限り発生するこ

演算子	メソッド名	演算子	メソッド名	演算子	メソッド名
=	assign	+=	plusEq	>>>=	fillShiftRightEq
<-	determine	-=	minusEq	&=	andEq
*=	mulEq	<<=	shiftLeftEq	^=	xorEq
/=	divEq	>>=	shiftRightEq	|=	orEq
%=	modEq				

表 5.1: フューチャーの代入文と変換後のメソッド名の対応

とはない。このような例外を必ず捕捉しなくてはならないのは、ユーザにとって大変煩わしい作業である。そこで、代わりに例外 FutureException を発生させて必要に応じて捕捉すれば良いようにした。

5.2.3 代入

代入のための演算子には = <- *= /= %= += -= <<= >>= >>>= &= ^= |= があるので、これらをそれぞれフューチャーオブジェクトの適切なメソッド呼び出しに置き換える。変数がインスタンス変数の時には名前の置き換えも行う。表 5.1 は演算子と変換されるフューチャーオブジェクトのメソッド名の対応表である。以下は代入の例である。

```
future int x;
x = 1;
x <- 2;
x *= 3;
```

これを変換すると以下のようになる。ただし、フューチャー x はローカル変数とする。この変換では特に複雑な処理は何もしない。単に置き換えるだけである。

```
IntFuture x = new IntFuture();
x.assign(1);
x.determine(2);
x.mulEq(3);
```

メソッド assign() の実装は以下のようになっている。

```
public synchronized void assign(int x) {
    if (! determined) {
        value = x;
        determined = true;
        notifyAll();
    }
    else {
        value = x;
    }
}
```

メソッド assign() は変数 determined を調べ、値が未決定の場合には値を代入して変数 determined を true にし、値が決定するのを待っているスレッドを起こしている。値が決定している場合には単に代入だけを行なっている。

メソッド determine() の実装は以下のようになっている。

```
public synchronized void determine(int x) {
    if (! determined) {
        value = x;
        determined = true;
        notifyAll();
    }
}
```

メソッド determine() はメソッド assign() に似ている。しかし、値がすでに決定している場合には何も代入を行なわない。

メソッド mulEq() の実装は以下のようになっている。

```
public synchronized void mulEq(int x) {
    if (! determined) {
        error();
    }
    else {
        value *= x;
    }
}
```

*= などの複合代入を実装する場合には副作用の問題があるので、評価を正確に 1 回だけ行なうように注意をしなくてはならない。この実装では Java が用意している複合代入の演算子をそのまま使っているので、副作用も通常の演算子と同じように正しく起きる。

Java には、複合演算子で右側オペランドの型が byte 型 char 型 short 型のものは用意されていない。これらは int 型で演算が行なわれる。従っ て、byte 型 char 型 short 型のフューチャーオブジェクトのこれらのメソッドの仮引数の型は int 型で扱っている。

5.2.4 その他の演算子

フューチャーにはその他に演算子 init! と演算子 determined? がある。演算子 init! はフューチャーオブジェクトのメソッド init() の呼び出しに置き換えられる。オペランドの変数がインスタンス変数の場合にはその名前の置き換えも行う。メソッド init() は単に変数 determined を false にするだけである。演算子 determined? はフューチャーオブジェクトのメソッド isDetermined() の呼び出しに置き換えられる。この場合もオペランドの変数がインスタンス変数の場合には、その名前の置換えを行う。メソッド isDetermined() は単に変数 determined の値を返すだけである。以下はメソッド init() とメソッド isDetermined() の実装である。

```
public synchronized void init() {
    determined = false;
}

public synchronized boolean isDetermined() {
    return determined;
}
```

5.3 プロパティファイル

トランスレータの振舞はプロパティファイルによって若干変更することができる。プロパティファイルはトランスレータの起動時に初期化の目的で 1 回だけ読み込まれる。そしてトランスレータはプロパティファイルに書かれているプロパティとその値を記憶し、その値に従った振舞をするようになる。プロパティファイルはテキストファイルであり、そのフォーマットを BNF で表現すると、図 5.5 のようになる。図 5.5 の式 5.1 は空行である。空行は無視される。式 5.2 はコメント行である。コメント行は無視される。式 5.3 はコマンド行である。コマンド行はプロパティ名 ＜name＞ が ＜value＞ という値で、トランスレータによって記憶される。この時、 ＜value＞ で与えられる文字列の中にメタキャ

ラクタ % によって囲まれている文字列があった場合、その文字列はそれをキーとするプロパティの値に置き換えられる。そのプロパティはこのコマンド行よりも先に登録しておかなくてはならない。登録していない場合にはデフォルト値が使われる。デフォルト値がない場合にはエラーとなる。 % を入力したい場合には %% とする。

次にプロパティの種類とその役割について説明する。プロパティ AccessLevel は private または protected の値を取り、並行オブジェクトの内部変数と内部メソッドのアクセスレベルを決める。

プロパティ Imports で、システムで必ず必要となるクラスファイルの import 文を定義することができる。これにより、 MILK ソースコードでの import 文を省略することができる。トランスレータは変換された Java ソースコードにプロパティImports に書かれた import 文を自動で挿入してくれる。

プロパティ Prefix ではトランスレータが生成する変数名やメソッド名の頭に付けられる文字列を指定する。デフォルトでは $MILK$ が使われる。この文字列を変更すると、使用するライブラリ内の名前も変更する必要がある。従って、なるべく変更するべきではない。

プロパティ Rmi に文字列 on をセットすると、トランスレータは分散環境として RMI を使用することを前提とした変換を行なう。同様にプロパティ Horb に on をセットすると、分散環境として HORB を使用することを前提とした変換を行なうようになる。プロパティ Host はプロパティ Horb が on になっている時に意味をなすプロパティで、フューチャーオブジェクトが生成されるマシン名を指定するために使われる。分散環境でのトランスレータの振舞については第 6 章で詳しく説明する。

トランスレータで使用するライブラリはプロパティによって変更することができる。プロパティ Parent には並行オブジェクトのルートとなるクラスを指定することができる。デフォルトではクラス ConcurrentObject になっている。フューチャーで使用するライブラリはオブジェクト名とハンドルのために使用する型名をプロパティで指定することができる。オブジェクト名と型名の２つに分けたことで、あるインターフェースを実装したフューチャーオブジェクトをそのインターフェース名でハンドルすることができる。フューチャーを RMI や HORB で用いる場合、フューチャーオブジェクトをあるインターフェースでハンドルしなくてはならない場合がある。その問題をこの 2 種類のプロパティで回避することができる。これらのプロパティは Java のプリミティブな型とオブジェクトのそれぞれについて用意されていて、その名称との対応は表 5.2 のようになっている。このように、プ

$$
\begin{aligned}
\text{<property file>} &::= \text{<lines>} \\
\text{<lines>} &::= \varepsilon \\
&\mid \text{<lines><line>} \\
\text{<line>} &::= \text{<white space><eol>} \qquad (5.1) \\
&\mid \text{<comment>} \\
&\mid \text{<command>} \\
\text{<comment>} &::= \text{<white space><comment symbol>} \\
&\quad \text{<string><eol>} \qquad (5.2) \\
\text{<command>} &::= \text{<white space><name><white space>} \\
&\quad \text{<delimitor><white space><value><eol>} \qquad (5.3) \\
\text{<name>} &::= \text{<symbol>} \\
&\mid \text{<name><nondelimitor>} \\
\text{<value>} &::= \text{<nondelimitor>} \\
&\mid \text{<delimitor symbol>} \\
&\mid \text{<value><character>} \\
\text{<white space>} &::= \varepsilon \\
&\mid \text{<white space><white>} \\
\text{<string>} &::= \varepsilon \\
&\mid \text{<string><character>} \\
\text{<character>} &::= \text{<delimitor>} \mid \text{<nondelimitor>} \\
\text{<delimitor>} &::= \text{<delimitor symbol>} \mid \text{<white>} \\
\text{<nondelimitor>} &::= \text{<comment symbol>} \mid \text{<symbol>} \\
\text{<eol>} &::= '\backslash\mathtt{n}' \mid '\backslash\mathtt{r}' \\
\text{<comment symbol>} &::= '\#' \mid '!' \\
\text{<delimitor symbol>} &::= '=' \mid ':' \\
\text{<white>} &::= ' ' \mid '\backslash\mathtt{t}' \\
\text{<symbol>} &::= \text{その他の文字}
\end{aligned}
$$

図 5.5: プロパティファイルのフォーマット

フューチャーの型	ハンドル名	実装名
boolean	BoooleanFutureType	BooleanFutureImpl
byte	ByteFutureType	ByteFutureImpl
char	CharFutureType	CharFutureImpl
double	DoubleFutureType	DoubleFutureImpl
float	FloatFutureType	FloatFutureImpl
int	IntFutureType	IntFutureImpl
long	LongFutureType	LongFutureImpl
object	ObjectFutureType	ObjectFutureImpl
short	ShortFutureType	ShortFutureImpl

表 5.2: フューチャーに関係するプロパティとフューチャーとの対応

ロパティを使うことで、トランスレータの実装とはある程度独立した形で、ライブラリを変更したり拡張したりすることができる。プロパティのデフォルト値は付録 A.5 でまとめておく。

第 6 章

分散環境における MILK

MILK はそれだけでは分散環境で使うことはできない。しかし、RMI や HORB を利用することで、分散環境でも使うことができるようになる。この章では MILK に RMI や HORB をどのように組み込んでいるのかを説明する。

6.1 RMI と MILK

RMI と MILK を組み合わせて使う場合、MILK トランスレータは RMI に適した変換をする必要がある。トランスレータに -rmi オプションを付けて変換すると RMI に合わせた変換をしてくれる。この節では MILK を RMI と組み合わせて使う場合、並行オブジェクトとフューチャーをどのように扱う必要があり、どのような変化があるのかを説明する。

6.1.1 並行オブジェクト

RMI を使って並行オブジェクトを分散環境でも使えるようにするためには、並行オブジェクトクラスがリモートインターフェースを実装している必要がある。リモートインターフェースとは RMI の用語で、リモートオブジェクトが実装しなくてはならないインターフェースのことである。並行オブジェクトでもリモートインターフェースを実装しておけば、生成されたクラスファイルを元にして RMI のコンパイラ rmic によってスタブクラスとスケルトンクラスを生成することができる。スタブクラスとスケルトンクラスはリモートオブジェクトのメソッドを呼び出すために必要なクラスである。

このようにすれば RMI のシステムはインターフェースを使って並行オブジェクトにリ

```
interface A extends Remote {
    void sayHello(String name) throws RemoteException;
}

concurrent class AImpl implements A {
    public void sayHello(String name) throws RemoteException {
        System.out.println("Hello " + name);
    }
}
```

図 6.1: RMI を使用した並行オブジェクトのクラス定義

モートオブジェクトとしてアクセスすることができる。しかし、リモートからはインターフェースで定義したメソッドにしかアクセスすることができない。これは、インターフェースで定義したメソッドは必ず public になるから、 protected までのメソッドには、たとえメソッドが並行オブジェクトの外部メソッドであっても、リモートからはアクセスすることはできないからである。そこでトランスレータに -rmi オプションをつけた場合、プロパティ AccessLevel のデフォルトを protected に変更する。

RMI による分散並行オブジェクトの例として、sayHello の例をリモートオブジェクトとして使えるようにすると、図 6.1 のようになる。RMI でリモートオブジェクトを実際にネットワーク上で使えるようにするためにはクラス定義で java.rmi.server.UnicastRemoteObject を継承しておくか、メソッド UnicastRemoteObject.exportObject() によって、そのオブジェクトを明示的にネットワークにエクスポートしておかなくてはならない。並行オブジェクトは ConcurrentObject を継承しなくてはならないので、前者の方法はできない。従って、並行オブジェクトは必ず後者の方法でエクスポートしなくてはならない。

RMI のリモートインターフェースでは例外 java.rmi.RemoteException をメソッドの throws 節に必ず書かなくてはならない。この例外は RMI が生成するスタブクラスまたはスケルトンクラスで発生する可能性があるからである。ユーザーが定義するリモートオブジェクトのクラスでは、その中で他のリモートオブジェクトを扱っていない限り、この例外が発生することは無い。従って、並行オブジェクトクラス定義で特にこの例外について気を使う必要は無い。メソッドの throws 節にこの例外を書いておくだけで良い。メソッド定

義の中で他のリモートオブジェクトを扱っている場合には、そこから例外 RemoteException が発生する可能性がある。 4.2.1 節で並行オブジェクトの外部メソッドは呼び出し側に例外を渡すことができないことを説明した。従って、この場合にはメソッド内で例外処理をする必要がある。

並行オブジェクトの変換は通常の変換と全く同じに行うことができる。 RMI のための特別な処理は何もしていない。

6.1.2 フューチャー

RMI によってフューチャーを分散環境で使えるようにするためには、ライブラリを書き直す必要がある。それにはまず、リモートインターフェースでフューチャーオブジェクトのメソッドを定義する。そして、このインターフェースを実装したクラスを定義する。フューチャーはローカル変数のように振舞うことにしたので、そのエクステントから抜ければ消滅してもよい実体である。しかし、クラス UnicastRemoteObject を継承したリモートオブジェクトは Java VM が動作中はずっと存在し続ける。また、クラス UnicastRemoteObject はコンストラクタを生成した時に例外を発生する可能性がある。これはフューチャーにとって好ましくない。そこで、フューチャーオブジェクトでは UnicastRemoteObject では無く、クラス java.rmi.server.RemoteObject を継承することにする。クラス RemoteObject は上記の条件にあった RMI のリモートオブジェクトのためのクラスである。後は RMI の通常の手順でコンパイルすることで、フューチャーがリモートオブジェクトとして利用できるようになる。

しかし、このままだとフューチャーは RMI の普通のリモートオブジェクトであるから、例外 RemoteException を発生させる可能性がある。この例外は必ず捕捉しなくてはならない種類の例外であるので、 MILK ソースコードのフューチャーを使用する箇所では、この例外を必ず捕捉しなくてはならない。しかし、この例外は 4.2.2 節で説明した理由で捕捉したくはない。そこで RMI が生成したスタブクラスのソースコードを一部改変して RemoteException の代わりに FutureException を投げるようにした。

図 6.2 と図 6.3 は int 型フューチャーオブジェクトのインターフェースと、そのリモートオブジェクトのソースコードの一部である。トランスレータは -rmi オプションがついている時には、フューチャーオブジェクトの型にこのインターフェースを使い、フューチャーオブジェクトの生成にはこのリモートオブジェクトを使うようにプロパティの値を変更する。

```
public interface IntFuture extends Remote, Future {
    int ref() throws FutureException;
    void determine(int x) throws FutureException;
    void assign(int x) throws FutureException;
    void mulEq(int x) throws FutureException;
    void divEq(int x) throws FutureException;
    void modEq(int x) throws FutureException;
    void plusEq(int x) throws FutureException;
    void minusEq(int x) throws FutureException;
    void shiftLeftEq(int x) throws FutureException;
    void shiftRightEq(int x) throws FutureException;
    void fillShiftRightEq(int x) throws FutureException;
    void andEq(int x) throws FutureException;
    void xorEq(int x) throws FutureException;
    void orEq(int x) throws FutureException;
}
```

図 6.2: int 型フューチャーオブジェクトのインターフェース

```
public class IntFutureImpl extends FutureImpl implements IntFuture {
    private int value;

    public IntFutureImpl() throws FutureException {
        super();
        determined = false;
    }

    public IntFutureImpl(int x) throws FutureException {
        super();
        value = x;
        determined = true;
    }

    public synchronized int ref() throws FutureException {
        if (! determined) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        return value;
    }

    public synchronized void determine(int x) throws FutureException {
        if (! determined) {
            value = x;
            determined = true;
            notifyAll();
        }
    }
 .
 .
 .
```

図 6.3: int 型フューチャーオブジェクトの実装の一部

6.2 HORB と MILK

HORB によって並行オブジェクトを分散オブジェクトにするには、トランスレータに -horb オプションをつけて変換後、 Java ソースコードを HORB のコンパイラ horbc でコンパイルすることで行うことができる。 トランスレータは HORB のために特別な変換をすることは無い。ただ、フューチャーのために使うライブラリを変えるだけである。

プリミティブな型のフューチャーオブジェクトは、ライブラリのソースコードを horbc でコンパイルしてプロキシクラスとスケルトンクラスを用意するだけで、分散オブジェクトとして機能する。プロキシクラスとスケルトンクラスは HORB でリモートオブジェクトのメソッドを呼び出せるようにするためのクラスである。

オブジェクトのためのフューチャーは、今まではクラス Object にタイプキャストすることで扱ってきた。しかし、HORB はプロキシオブジェクトの無いオブジェクトをリモートに転送できないので、 Object にタイプキャストしてしまうと、たとえそのオブジェクト自身にプロキシオブジェクトがあったとしても、転送できなくなってしまう。そこでフューチャーで転送するオブジェクトはクラス milk.runtime.horb.Remote を継承しなくてはならないことにする。このクラス Remote はタイプキャストのためのクラスで、何も実装はしていない。このオブジェクトにはプロキシオブジェクトとスケルトンオブジェクトを用意しておく。そして、オブジェクトのためのフューチャーは Object ではなく、この Remote にタイプキャストする。こうすることで、オブジェクトもフューチャーで転送できるようになる。

しかし、配列は Object を継承しているが、Remote を継承していない。従って、直接フューチャーに転送することができない。しかし、タイプキャストせずに配列を直接転送することはできるので、次のようにして間接的に転送する。

```
public class Wrapper extends milk.runtime.horb.Remote {
    public int a[];
}
```

インスタンス変数 a に配列を入れてこのオブジェクトをフューチャーに渡すことで、リモートに配列を転送することができるようになる。

HORB の場合、フューチャーオブジェクトには特にインターフェースを用意する必要はないが、 MILK の RMI との利用の仕方に似せる目的と、フューチャーの汎用性を上げる目的でフューチャーオブジェクトのインターフェースを用意した。

HORB ではインターフェースを介さなくても分散オブジェクトのメソッド呼び出しができるので、並行オブジェクトの内部メソッドをリモートから呼び出すことができる。これは危険であるので、ユーザは RMI のように、インターフェースを用いてアクセスする方が望ましい。

6.3 分散環境での例

分散環境での例として、天気情報システムを作成した。この例の実行方法については付録 B で述べることにし、この節ではこの例のプログラムについて説明する。この例は RMI と組み合わせて作成されており、以下の 3 種類の分散オブジェクトからなっている。

- 天気情報のサーバ
- アプレット
- サーバとアプレットの窓口（ウィンドウ）

これらのオブジェクトの関係は図 6.4 のようになっている。サーバとアプレットは不特定多数存在してもよいが、ウィンドウは 1 つだけが存在する。アプレットとサーバはお互いの存在については間接的にしか知らない。このシステムでは天気情報のサーバがいろいろな地域に存在し、いつでもその地域の最新の天気情報を持っている。そして、アプレットからウィンドウに、ある地域の天気情報の取得の依頼があると、ウィンドウは適切な地域のサーバを選び出し、そこへ情報の取得の依頼を回す。依頼が回ってきたサーバは最新の天気情報をアプレットに直接返す。

このような動作の実装を簡単にするために、天気情報の受渡しにはフューチャーを使っている。図 6.5 はフューチャーの流れを示している。アプレットはフューチャーに値が返ってくるまでの間、“Searching now, please wait for a moment.” とメッセージを表示して待機し、フューチャーに値が返ってくると、すぐにその内容を表示する。図 6.6 はアプレットが情報を得たときのものである。

サーバのソースコードは図 6.7 と図 6.8 である。サーバは並行オブジェクトであり、メソッドには updateInfo() と getInfo() がある。メソッド updateInfo() は天気情報を新しい情報に更新するためのものである。メソッド getInfo() は引数で与えられたフューチャーに最新の天気情報を返すためのものである。このメソッドはウィンドウから呼び出される。

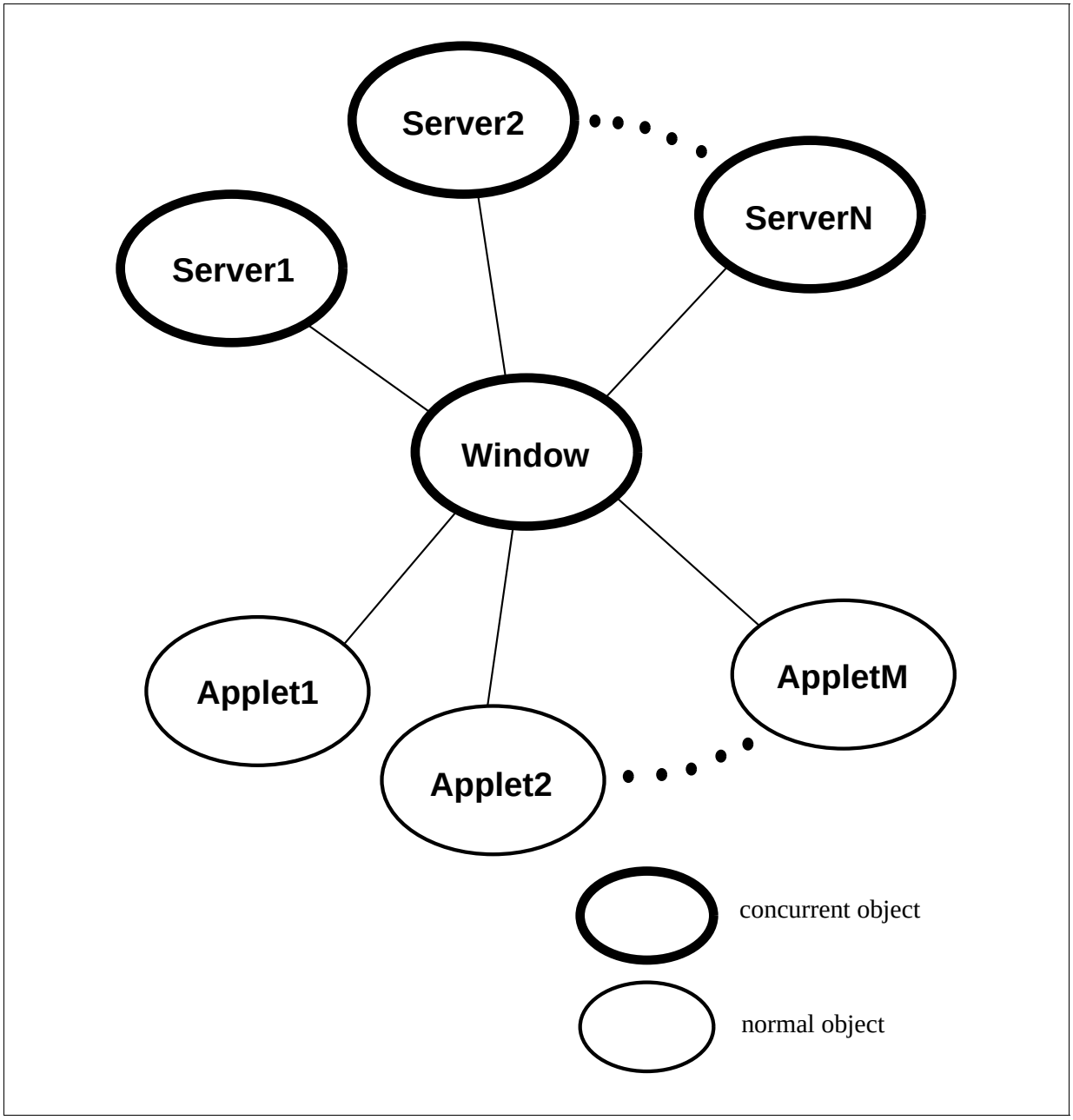

図 6.4: Weather Information の分散オブジェクトの関係

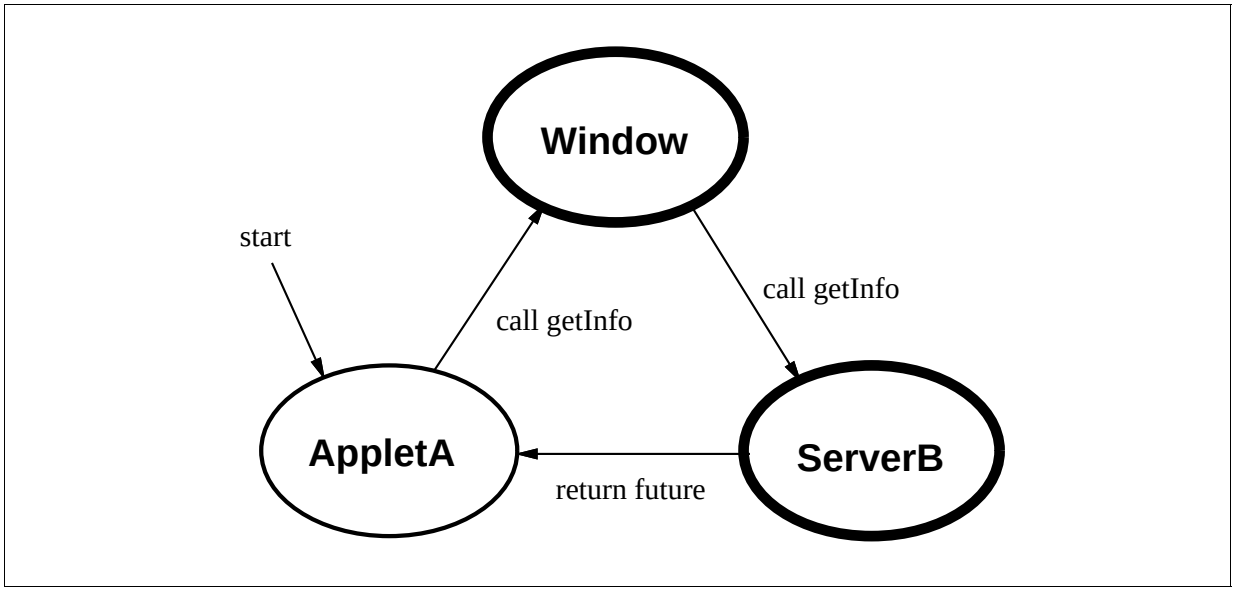

図 6.5: Weather Information でのフューチャーの流れ

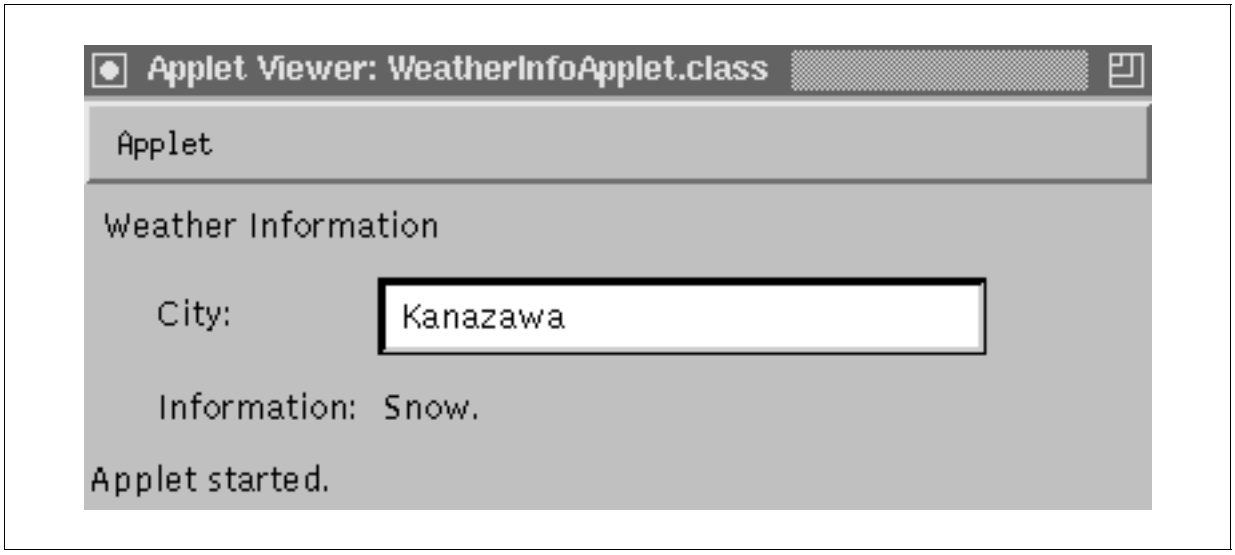

図 6.6: Weather Information のアプレット

```
public interface WeatherInfoServer extends Remote {
    void getInfo(future String info) throws RemoteException;
}
```

図 6.7: ファイル WeatherInfoServer.milk

サーバは並行オブジェクトであるので、これらのメソッドが同時に実行されることはない。また、並行オブジェクトのメソッド呼び出しは、そのオブジェクトにメッセージを送るとすぐに次の処理を行うことができるので、メソッド getInfo() を呼び出すウィンドウはサーバにメッセージを送ると、その処理の終了を待たずに次の処理を行うことができる。また、RMI での分散オブジェクトのアクセスは図 6.7 のインターフェースを介して行うので、リモートからメソッド updateInfo() を呼び出すことはできない。従って、情報の更新はローカルの Java VM だけが安全に行うことができる。ただし、トランスレータに -rmi オプションを付けて変換すると、デフォルトではメソッド update() は外部メソッドにならないので、トランスレータのプロパティ AccessLevel を private にしておく必要がある。この例では 5 秒おきに天気情報が更新されるようになっている。

ウィンドウのソースコードは図 6.9 と図 6.10 である。ウィンドウも並行オブジェクトであり、メソッドには register() と getInfo() がある。メソッド register() はサーバから呼ばれ、引数に都市名とサーバの参照を取り、都市名をキーとしてサーバの参照をテーブルに登録する。メソッド getInfo() はアプレットから呼ばれ、引数に都市名とフューチャーを取り、都市名をキーとしてテーブルからサーバの参照を探しだし、そのサーバに天気情報の取得を依頼する。ウィンドウは並行オブジェクトであるので、どんなにたくさんの依頼が集中したとしても、依頼主はそれほど待たずに自分の処理に戻ることができる。また、依頼されたメッセージは逐次に処理されるので、正しく同期のとれた情報を管理することができる。もし、並行に動作せずに依頼されたメッセージを逐次に処理しようとしたら、依頼主はキューに溜っているメッセージの量に比例して待たされることになる。つまり、メッセージがたくさん溜っている時には、依頼主はかなりの時間待たされることになる。

アプレットのソースコードは図 6.11 と図 6.12 である。図 6.11 の中でクラス FixedLayout を使っているが、このクラスはアプレットのレイアウトのためのクラスで並行計算に直接関係しない。従って、付録 B で掲載する。アプレットではメソッド init() でウィンドウオ

ブジェクトの参照を取得し、GUI を構築している。 Java では GUI のことを awt と呼んでいる。アプレットは、 awt から都市名が入力されると、メソッド action() を呼び出す。メソッド action() ではウィンドウのメソッド getInfo() を、引数に都市名とフューチャーを渡して呼び出す。ウィンドウは並行オブジェクトであったので、アプレットはすぐに自分の処理に復帰し、“Searching now, please wait for a moment.” と表示して待機する。天気情報の表示は並行オブジェクト InfoWriter に任せている。 Java では、 awt の更新が実際の画面にすぐに反映されるわけではない。普通なら、メソッド action() の中で天気情報についても表示すれば良いように思える。しかし、その場合には “Searching now, please wait for a moment.” のメッセージは表示されずに天気情報だけが表示される。そこで、並行オブジェクトに後の描画を任せることで、簡単に awt の再描画を行うことができる。そして、アプレットはすぐに別の処理をすることができる。

このような例のプログラムを Java で書くとすると、情報の共有の仕方、同期の取り方についてプログラマが考える必要がある。また、分散環境では通信の方式についても考える必要がある。これらは逐次計算にはない問題を内包しており、いろいろな例外が発生する可能性がある。従って、これらについてプログラミングすることは非常に骨の折れる作業となる。しかし、 MILK では並行計算をオブジェクトという抽象的な形で扱うことができる。、そしてそれは、計算環境が分散環境であっても、そうでない場合とほとんど同じように扱うことができる。

```
import java.rmi.server.UnicastRemoteObject;
import java.util.*;

public concurrent class WeatherInfoServerImpl
     implements WeatherInfoServer {
    private String info;

    void updateInfo(String info) {
        System.out.println("update to " + info);
        this.info = info;
    }

    public void getInfo(future String info) throws RemoteException {
        try {
            Thread.currentThread().sleep(3000);
        }
        catch (InterruptedException e) {}
        info = this.info;
    }
}

class WeatherInfoServerStarter {
    private static final String WINHOST = "isc-sol2";
    private static final String
         weather[] = {"Sunny skies.", "Cloudy skies.", "Rain.",
                          "Snow."};

    public static void main(String args[]) {
        System.setSecurityManager(new RMISecurityManager());

        WeatherInfoServerImpl server = new WeatherInfoServerImpl();
        server.updateInfo(weather[0]);
```

図 6.8: ファイル WeatherInfoServerImpl.milk

```
        try {
            WeatherInfoWindow window = (WeatherInfoWindow)
                 Naming.lookup("rmi://" + WINHOST
                               + "/WeatherInfoWindow");
            UnicastRemoteObject.exportObject(server);
            window.register("Kanazawa", server);
        }
        catch (Exception e) {
            System.out.println("WeatherInfoServerStarter exception: "
                               + e.getMessage());
            e.printStackTrace();
            System.exit(1);
        }
        System.out.println("WeatherInfoServer is registered.");

        Random rand = new Random();
        int size = weather.length;
        int i;
        while (true) {
            try {
                Thread.currentThread().sleep(5000);
            }
            catch (InterruptedException e) {}
            i = Math.abs(rand.nextInt()) % size;
            server.updateInfo(weather[i]);
        }
    }
}
```

図 6.8: ファイル WeatherInfoServerImpl.milk (つづき)

```
public interface WeatherInfoWindow extends Remote {
   void register(String city, WeatherInfoServer server)
        throws RemoteException;
   void getInfo(String city, future String info)
        throws RemoteException;
}
```

図 6.9: ファイル WeatherInfoWindow.milk

```
import java.rmi.server.UnicastRemoteObject;
import java.util.*;

public concurrent class WeatherInfoWindowImpl
     implements WeatherInfoWindow {
    private Hashtable table = new Hashtable();

    public void register(String city, WeatherInfoServer server)
         throws RemoteException {
        table.put(city, server);
    }

    public void getInfo(String city, future String info)
         throws RemoteException {
        WeatherInfoServer server = (WeatherInfoServer)table.get(city);
        if (server != null) {
            try {
                server.getInfo(future info);
            }
            catch (RemoteException e) {
                System.out.println("WeatherInfoWindowImpl exception: "
                                   + e.getMessage());
                e.printStackTrace();
```

図 6.10: ファイル WeatherInfoWindowImpl.milk

```
            }
        }
        else {
            info = "Sorry, I don't know " + city +".";
        }
    }
}

class WeatherInfoWindowStarter {
    public static void main(String args[]) {
        System.setSecurityManager(new RMISecurityManager());

        try {
            WeatherInfoWindowImpl window
                 = new WeatherInfoWindowImpl();
            UnicastRemoteObject.exportObject(window);
            Naming.rebind("rmi:WeatherInfoWindow", window);
            System.out
                 .println("WeatherInfoWindowStarter is completed.");
        }
        catch (Exception e) {
            System.out.println("WeatherInfoWindowStarter exception: "
                               + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

図 6.10: ファイル WeatherInfoWindowImpl.milk (つづき)

```
import java.applet.Applet;
import java.awt.*;

public class WeatherInfoApplet extends Applet {
    private static final String WINHOST = "isc-sol2";
    private WeatherInfoWindow window;
    private Label output;
    future String info;

    public void init() {
        try {
            window = (WeatherInfoWindow)
                 Naming.lookup("rmi://" + getCodeBase().getHost()
                              + "/WeatherInfoWindow");
        }
        catch (Exception e) {
            appletException(e);
        }

        Label title = new Label("Weather Information");
        Label city = new Label("City:");
        TextField input = new TextField(26);
        Label info = new Label("Information:");
        output = new Label();

        setLayout(new FixedLayout());
        add(title);
        add(city);
        add(input);
        add(info);
        add(output);
```

図 6.11: ファイル WeatherInfoApplet.milk

```
        title.move(5, 5);
        city.move(25, 38);
        input.move(110, 35);
        info.move(25, 73);
        output.move(110, 73);
    }

    public boolean action(Event e, Object o) {
        if (e.target instanceof TextField) {
            String str = (String)o;
            if (str.equals("")) {
                return true;
            }

            init! info;
            try {
                window.getInfo(str, future info);
            }
            catch (RemoteException re) {
                appletException(re);
            }

            output.setText("Searching now, please wait "
                           + "for a moment.");
            output.resize(output.preferredSize());
            (new InfoWriter()).work(this);
            return true;
        }
        else {
            return false;
        }
    }
```

図 6.11: ファイル WeatherInfoApplet.milk (つづき)

```
    void writeInfo() {
        output.setText(info);
        output.resize(output.preferredSize());
        repaint();
    }

    private void appletException(Exception e) {
        System.out.println("Applet Exceptin: " + e.getMessage());
        e.printStackTrace();
        System.exit(1);
    }
}
```

図 6.11: ファイル WeatherInfoApplet.milk (つづき)

```
concurrent class InfoWriter {
    public void work(WeatherInfoApplet applet) {
        applet.writeInfo();
    }
}
```

図 6.12: ファイル InfoWriter.milk

第 7 章

MILK の評価

MILK では並行オブジェクトと、過去型、未来型のメッセージ通信を Java に組み入れた。これにより、並行計算を人間の直観的な表現方法で実現できるようになった。また、MILK は Java を拡張し、トランスレータは MILK を Java に変換することから、 MILK は Java が持っている長所をそのまま受け継いでいる。例えば、移植性や相互運用性の高さは MILK でもそのまま受け継いでいる。もともと Java はプログラムの記述性容易性が高い言語であるが、 MILK によりそれがさらに高まったと考えられる。 MILK で記述した並行計算と同等の意味の計算を Java で記述しようとすると、図 4.3 のように大抵の場合プログラムの記述の手間は増え、そしてまた、プログラムの可読性も悪くなる。このことは MILK が並行計算をうまく抽象化して扱っていることを裏付けている。

しかし、MILK にもいくつか問題点がある。この章では MILK の問題点と改善方法について説明していく。

7.1 トランスレータの実装による問題

MILK にはトランスレータの実装の都合による問題点がある。フューチャーは、ローカル変数と制限の加えられたインスタンス変数でだけ宣言することができる（5.2.1 節参照)。これはトランスレータが 1 つのクラス定義の中だけで判別できる名前の参照だけを解決しているからであった。この問題はトランスレータの名前の解決法を改良すれば直すことができる。また、4.2.1 節で説明したように並行オブジェクトの継承にも少し欠陥がある。この問題もクラス名の参照関係を調べれば解決することができる。

しかし、名前の参照を完全に解決することは簡単なことではない。それをするためには'.'で区切られた名前の参照を解決しなくてはならない。そのためにはクラスやパッケージそれにフィールド変数などの名前について解決する必要がある。また、当然他のクラスファイルを参照しなくてはならないこともある。その場合にはその名前のアクセスレベルについても知っておく必要がある。また、他のクラスファイルがまだ生成されていないこともあり、そして、そのクラスのソースコードでも、当然また他のクラスファイルを参照している可能性がある。また、ひょっとしたら相互に参照しあっていることもある。このようなことを全てトランスレータのレベルで解決させることは現実的では無い。もし、名前の参照関係を完全に解決するのであれば、それによって静的に調べることのできるエラーをほとんど調べることができるので、コンパイラとして実現する方が望ましい。その方がソースコードをパースする回数が減り、実行効率も良くなる。

また、並行オブジェクトの内部で発生した例外を、呼び出し側に渡すことはできない(4.2.1 節参照)。並行オブジェクト指向モデルではオブジェクトを自立して動作することのできる動作主体と考えているので、このこと自体は特に問題ではない。しかし、外部メソッドの throws 節で書かれた例外と同じ例外かまたはそのサブクラスの例外が、外部メソッドのボディ内で発生する可能性がある場合、MILK トランスレータでは、そのことを検知することができない。従って、この種の例外が正しく捕捉され、その例外処理が行われているのかを判断することができない。この問題は、 Java コンパイラが例外について行っている処理を MILK でも行えば、解決することができる。しかし、その場合にも他のクラスファイルを参照することになるので、トランスレータで解決させることは現実的では無い。

7.2 Java による問題

MILK は Java を拡張しており、MILK のソースコードは Java のソースコードに変換される。従って、 MILK は Java の言語仕様の制約もまたそのまま受け継いでいる。並行オブジェクトのメソッドのアクセスレベルの問題(4.2.1 節参照)はその１つである。この問題は MILK ソースコードを Java ソースコードに変換しているため、 MILK のアクセスレベルの制限が Java のアクセスレベルの制限と同じになってしまうので、発生する。この問題を改善するためにはトランスレータというレベルではなく、コンパイラを作成して Java クラスファイルを MILK ソースファイルから直接生成するようにすることで改善すること

は可能である。しかし、その場合でもインタプリタでの実行時の動的なチェックまでは変更することはできない。

また、Java のオブジェクトは厳密な意味で情報を完全にカプセル化していない。これは Java を拡張している MILK においても同じである。 Java は同一クラスに属するオブジェクトならば、他のオブジェクトのプライベートなインスタンス変数にアクセスすることができる。図 7.1 はその例である。クラス A のメソッド main() において、オブジェクト b2 がメソッド copy1() を実行する時、オブジェクト b1 はメソッド suc() を実行している途中であり、 x++ を実行して 3 秒間スリープしている。この状態で copy1() を実行してしまうので、メソッド copy1() はメソッド suc() の処理の途中の意味のなさない値を取り出してしまうことになる。また、メソッド copy2() は他のオブジェクトのプライベートなインスタンス変数に、直接代入を行っているので、本来並行オブジェクトとして保護されるべき値が全く保護されていないことになる。従って、このようなアクセスは禁止すべきであるが、MILK トランスレータの持つ名前空間ではそこまで調べることはできない。また、MILK では Java の持つ言語の意味をなるべく尊重して設計したので、特にこの問題については対処しなかった。MILK ではこのような使い方はしないよう、ユーザが注意するしかない。プライベートなメソッドに関しても同様のことがいえる。この問題も先ほどの問題と同じように、コンパイラを作ることで改善することは可能である。

その他に Java の VM は今のところパラレルマシン上には実装されていない。従って、スレッドの処理は結局逐次に処理される。処理時間にはスレッド処理のためのオーバーヘッドが加わる。更に MILK では並行オブジェクトの処理のためのオーバーヘッドが上乗せされる。従って、ローカルマシンで並行オブジェクトを多数動作させた場合、そのオーバーヘッドでかなり動作が遅くなる。しかし、分散環境にあるオブジェクトでは、独立した複数のマシンで動作するので、並行性が高く、情報隠蔽の度合も高い。従って、MILK の並行オブジェクトは今のところ、ローカルの環境よりもむしろ分散環境において、よりその真価を発揮する。

7.3 分散環境での問題

MILK を分散環境で使う場合には、まだたくさんの問題が残っている。例えば、フューチャーオブジェクトのライフサイクルについては考慮の余地が残されている。 RMI ではサーバとなるオブジェクトはローカルの Java VM が停止するまで、そのオブジェクトを

```
class A {
    public static void main(String args[]) {
        B b1 = new B(1, 2);
        B b2 = new B(2, 1);

        b1.suc();
        b2.copy1(b1);
        b2.print();
        b2.copy2(b1);
        b1.print();
    }
}

concurrent class B {
    private int x;
    private int y;

    B(int x, int y) {
        this.x = x;
        this.y = y;
    }

    void suc() {
        x++;
        try {
            Thread.currentThread().sleep(3000);
        }
        catch (InterruptedException e) {}
        y++;
    }

    void copy1(B b) {
        this.x = b.x;
        this.y = b.y;
    }
```

図 7.1: 同一クラスのオブジェクトの変数のアクセスに関する問題

```
    void copy2(B b) {
        b.x = this.x;
        b.y = this.y;
    }

    void print() {
        System.out.println(x + " " + y);
    }
}
```

図 7.1: 同一クラスのオブジェクトの変数のアクセスに関する問題 (つづき)

なるべく破棄しないようにしている。しかし、 MILK ではローカル変数のフューチャーオブジェクトのライフサイクルをローカル変数のエクステントと同じにしたので、そのエクステントから抜ける時に RMI を使用している場合でも破棄するようにした。しかし、リモートではフューチャーが破棄されたことについては気がつかないので、リモートから破棄されたフューチャーオブジェクトにアクセスしようとすることも起こりうる。その場合、リモートでは例外が発生するだろう。

また、リモートにおいてフューチャーを参照したためにウエイトしているスレッドがあるとして、それに対して、割り込みができるのか、またはできたとして何が起きるのかは MILK とともに用いる分散環境によって変わってくる。分散環境におけるオブジェクトの振舞は、おそらくその他にも多くの問題をはらんでいると考えられる。しかし、本研究では並行計算に重点を置いたので、分散環境での問題についてはあまり探求しなかった。

第 8 章

おわりに

この章では MILK と並行オブジェクト指向言語の今後の展望について述べる。

8.1 今後の展望

前章で挙げた問題の多くは Java に依存しているところが多いので、コンパイラを作成して直接 Java クラスファイルを生成することで、改善することができる。このようなコンパイラにすると、コンパイラで静的に調べる型のチェックなどは Java の規則とは全く異なったものとして実現することが可能である。この場合、Java との互換性のトレードオフをどの辺りにするかで、かなりいろいろな実装ができると考えられる。例えば、フューチャーはより自然な形で表現できるようになる。フューチャーは当初以下のような形で表現しようとした。

```
future x = obj.foo(a, b, c);
```

これはフューチャーをメソッドの戻り値として渡すという形である。この方が人間には直観的に理解できると考えられる。しかし、これを実現するには名前の参照を完全に解決する必要がある。また、前章で述べたように Java のオブジェクトはオブジェクトの状態を完全に隠蔽することができない。しかし、コンパイラによって Java クラスファイルを直接生成する形ならば、これらを実現することが可能である。このようにコンパイラで実現することはいろいろな可能性を秘めている。

今回 MILK で実現した並行オブジェクトにはプライオリティというものはない。しかし、プライオリティを取り入れると、並行計算の記述がより一層豊かになると考えられる。

ローカルの環境の場合には、プライオリティを並行オブジェクトに取り入れることは難しいことではない。しかし、分散した並行オブジェクトにプライオリティをつけることは、今の MILK の実装では難しい。

MILK ではフューチャーによる非同期通信の方式に、大きく分けて = と <- の 2 種類の方式を用意した。しかし、その他にもいろいろな方式が考えられる。例えば、フューチャーをキューのように使い、フューチャーに返された値を全て保存しておく方式や、フューチャーの値を参照したのと同時にフューチャーを初期化する方式や、複数のフューチャーに対して同時に待ち、値が返ってきたものから処理を始めていく方式などが考えられる。このようにいろいろな通信方式が考えられるが、それらの多くは既存の構文を組み合わせることで作ることが可能であるだろう。従って、他の通信方式を取り入れる場合には、あくまでもそれを実現したことによって、プログラムの記述が簡潔にならなくてはいけない。

現在、 Java の VM はパラレルマシン上では実装されていない。従って、並行オブジェクトは本当の意味でその真価を発揮することができていない。しかし、近年のハードウェアの進歩と普及の速度を考えると、パラレルマシンが普及してくる日もそう遠くはないと考えられる。そうなってくると、MILK のような並行オブジェクト指向言語もその活躍の場を増やしていくことだろう。

謝辞

渡部卓雄助教授には本研究について多大な御指導と御助言をして頂きました。ここに感謝の意を表し、心より御礼申し上げます。二木厚吉教授にはよく御指導して頂き、私を導いて下さいました。心より感謝致します。緒方和博助手にはいつも貴重な助言をして頂きました。深く感謝致します。言語設計学講座の皆様には議論によく乗って頂きました。深くお礼申し上げます。コマツソフト株式会社には私を快く大学に出して頂き、研究の機会を与えて頂きました。また、それだけでなく生活面を全面的に援助して頂きました。心より感謝致します。

1997 年早春

情報科学研究科棟にて

阿部 修

参考文献

[1] 所 真理雄, 松岡 聡, 垂水 浩幸, **オブジェクト指向コンピューティング**, 岩波書店, 1993.

[2] Akinori Yonezawa, Jean-Pierre Briot and Etsuya Shibayama, **Object-Oriented Concurrent Programming in ABCL/1**, OOPSLA '86 Proceedings pp.258-268, September 1986.

[3] Takuo Watanabe and Akinori Yonezawa, **Reflection in an Object-Oriented Concurrent Language**, OOPSLA '88 Proceedings pp.306-315, Semptember 25-30, 1988.

[4] James Gosling, Henry McGilton, **Java 言語環境 技術白書**, internet publication (http://www.sun.co.jp/smi.jp/tech/java/index.html), May 1995.

[5] James Gosling, Bill Joy, Guy Steele, **The Java Language Specification, Version 1.0**, internet publication (http://sunsite.sut.ac.jp/java/jdk/docs/), August 1996.

[6] **The Java Virtual Machine Specification**,
internet publication (http://sunsite.sut.ac.jp/java/jdk/docs/), August 21, 1995.

[7] Doug Kramer, **Java API Documentation 1.0.2**,
internet publication (http://sunsite.sut.ac.jp/java/jdk/docs/), April 14, 1996.

[8] Ken Arnold and James Gosling, **The Java^{TM} Programming Language**, Addison-Wesley Publishing Company, Inc. , May 1996.

[9] **Java^{TM} Remote Method Invocation Sepcification**, Beta Draft Revision 1.2, internet publication (http://chatsubo.javasoft.com/current/), December 2, 1996.

[10] **Java^{TM} Object Serialization Sepcification**, Prebeta Release Revision 1.1 internet publication (http://chatsubo.javasoft.com/current/), November 1, 1996.

[11] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber, **Network Objects**, internet publication (http://ftp.digital.com/pub/DEC/SRC/reseach-reports/abstractssrc-rr-115.html), February 28, 1994, Revised December 4, 1995.

[12] Andrew Birrell, David Evers, Greg Nelson, Susan Owicki, and Edward Wobber, **Distributed Garbage Collection for Network Objects**, internet publication (http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts /src-rr-116.html), December 15, 1993.

[13] HIRANO Satoshi, **ネットワークコンピューティングの魔法のじゅうたん：HORB Flyer's ガイド**, internet publication, September 13, 1996.

[14] 平野 聡 **HORB: ワールドプログラミングのための並列分散オブジェクト指向言語**, WOOC '96,

[15] Luca Cardelli, **A Language with Distributed Scope**, internet publication (http://www.research.digital.com/SRC/Obliq.html), May 30, 1995.

[16] 小野沢 博文, **分散オブジェクト指向技術 CORBA**, ソフト・リサーチ・センター, 1996.

[17] 横手 靖彦, 所 真理雄, **並行オブジェクト指向言語 ConcurrentSmalltalk**, コンピュータソフトウエア, Vol 2, No.4, pp.2-18, Oct. 1985.

[18] Yasuhiko Yokote and Mario Tokoro, **The Design and Implementation of ConcurrentSmalltalk**, OOPSLA '86 Proceedings pp.331-340, September 1986.

[19] Yasuhiko Yokote and Mario Tokoro, **Experience and Evolution of Concurrent-Smalltalk**, OOPSLA '87 Proceedings pp.406-415, October 4-8, 1987.

[20] Robert H. Halstead, Jr, **Multilisp: A Language for Concurrent Symbolic Computation**, ACM Transactions on Programming Languages and Systems, Vol.7, No.4, October 1985.

[21] Kenjiro Taura and Akinori Yonezawa, **Schematic: A Concurrent Object-Oriented Extension to Scheme**,

internet publication (http://web.yl.is.s.u-tokyo.ac.jp/pl/schematic.html), December 1995.

[22] **The Java™ Language Specification, Version 1.0 Beta**, internet publication (http://sunsite.sut.ac.jp/java/jdk/docs/), Octorber 30, 1995.

[23] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes, **Essentials of Programmming Languages**, The MIT Press, McGraw-Hill Book Company, 1992.

[24] Scott E. Hudson, **CUP User's Manual**, internet publication (http://www.cc.gatech.edu/gvu/people/Faculty/hudson/java_cup/home.html), March 1996.

付録 A

MILK トランスレータ操作説明

この付録で本文ではふれられなかった MILK トランスレータの使い方についてまとめておく。

A.1 動作環境

MILK トランスレータは Java の上で実装されているので、理論上は Java VM の実装されているマシンの上でならどのマシンでも動作することになる。しかし、改行コードや、ファイルパスの階層を区切るために使われる文字は OS に依存している。MILK トランスレータの実装は solaris 上で行われ、他の OS での動作テストは行っていない。従って、solaris での動作は保証されるが、その他の OS での動作の保証はできない。なお、実装に使った Java のバージョンは JDK 1.0.2 である。

A.2 入手方法

MILK トランスレータは Java 上で動作する。もし、まだ Java を手に入れてない場合には JDK 1.0.2 を以下の URL から入手することができる。

http://sunsite.sut.ac.jp/java/

MILK トランスレータは Web ブラウザにより以下の URL から入手することができる。

http://www.jaist.ac.jp/~o-abe/milk.html

A.3 インストール

MILK トランスレータは Java の上で実装されているので、まず Java をインストールしておく必要がある。 MILK トランスレータのインストールは大変簡単である。入手したファイルは gzip と tar によって圧縮されている。まず、これを展開する。展開作業をしたディレクトリを $HOME とする。次はパスの設定を行う。パスに $HOME/milk/bin を加える。最後に Java のクラスバスを設定する。環境変数 $CLASSPATH に $HOME/milk を加える。これだけで MILK トランスレータを動作させることができるようになる。以下に典型的なインストール手順の例を示す。この例でのシェルは csh を仮定している。

```
% gunzip milk.tar.gz
% tar xvf milk.tar
% set path = ($HOME/milk/bin $path)
% setenv CLASSPATH $HOME/milk:$CLASSPATH
%
```

A.4 トランスレータの実行方法

MILK トランスレータを動作させる前に MILK ソースファイルを用意しておく。ソースファイルの拡張子は .milk としておく。 MILK トランスレータはシェルのプロンプトよりコマンドを入力することで実行される。コマンドの形式は以下のようになっている。

milk [-rmi | -horb [-host host_name]] [property-file ...] source-file ...

オプション -rmi と -horb は MILK を RMI または HORB と組み合わせて使う時に指定する。これらは必要がなければ指定しなくても良い。指定する時にはどちらか片方のみを指定する。これらのオプションを指定するとプロパティのデフォルト値が変更される。プロパティのデフォルト値については後で説明する。オプション-horb を指定した時には、オプション -host かまたは次に説明する property-file で、フューチャーオブジェクトを生成するホスト名を与えなくてはならない。

オプション property-file でトランスレータで使用されるプロパティの値を変更することができる。まず、更新するプロパティの値を拡張子 .properties で終るファイルに用意しておく。このファイルをプロパティファイルという。そして、このプロパティファイルへのパスをこのオプションで指定する。このオプションは省略しても構わない。また、プロパ

ティファイルを複数指定してもよい。同じプロパティの値が複数回更新されている場合には最後に更新した値が有効となる。

オプション source-file でトランスレートする MILK ソースファイルを指定する。ソースファイルの拡張子は .milk でなければならない。このオプションは省略することができない。このオプションを複数指定することは構わない。

トランスレートが無事行われると Java ソースファイルが MILK ソースファイルの拡張子 .milk の部分を .java に置き換えた名前で出力される。トランスレート時にシンタックスエラーが発生した場合にはそのことを伝えるメッセージが表示される。トランスレータではシンタックスしかチェックしていないので出力された Java ソースファイルを javac でコンパイルした時にエラーが発生することがある。その場合には生成された Java ソースコードに MILK ソースコードの対応する行番号がコメントとして入れてあるので、それを元に MILK ソースコードを直していく。

A.5 プロパティのデフォルト値

MILK トランスレータで使われるプロパティのデフォルト値をオプション -rmi -horb を与えないで起動した時、オプション -rmi を与えて起動した時、オプション -horb を与えて起動した時のそれぞれについて表 A.1 A.2 A.3 にまとめておく。 MILK トランスレータにオプション -horb を与えた時にはオプション -host かまたはプロパティファイルでプロパティ Host の値を変更してやる必要がある。表 A.2 のプロパティ Imports の値が 2 行に分けて書かれているが、実際の値は 1 行である。

A.6 RMI と HORB の入手方法

RMI または HORB と、 MILK を併用する場合にはこれらを以下の URL から入手する必要がある。

RMI	http://chatsubo.javasoft.com/current/
HORB	http://ring.etl.go.jp/openlab/horb/

これらを入手したら、後はそれぞれのマニュアルに従ってインストールすれば、MILK と併用することができる。

プロパティ名	値
Prefix	$MILK$
Imports	import milk.runtime.*;\nimport milk.runtime.concurrent.*;
Parent	ConcurrentObject
AccessLevel	private
ObjectFutureType	ObjectFuture
ByteFutureType	ByteFuture
CharFutureType	CharFuture
DoubleFutureType	D oublefuture
FloatFutureType	FloatFuture
IntFutureType	IntFuture
LongFutureType	LongFuture
ShortFutureType	ShortFuture
BooleanFutureType	BooleanFuture
ObjectFutureImpl	ObjectFuture
ByteFutureImpl	ByteFuture
CharFutureImpl	CharFuture
DoubleFutureImpl	Doublefuture
FloatFutureImpl	FloatFuture
IntFutureImpl	IntFuture
LongFutureImpl	LongFuture
ShortFutureImpl	ShortFuture
BooleanFutureImpl	BooleanFuture
Rmi	off
Horb	off
Host	HOST

表 A.1: プロパティのデフォルト値（コンパイルオプション無しの場合）

プロパティ名	値
Prefix	$MILK$
Imports	import milk.runtime.*;\nimport milk.runtime.rmi.*;\n import java.rmi.*;
Parent	ConcurrentObject
AccessLevel	protected
ObjectFutureType	ObjectFuture
ByteFutureType	ByteFuture
CharFutureType	CharFuture
DoubleFutureType	D oublefuture
FloatFutureType	FloatFuture
IntFutureType	IntFuture
LongFutureType	LongFuture
ShortFutureType	ShortFuture
BooleanFutureType	BooleanFuture
ObjectFutureImpl	ObjectFuture
ByteFutureImpl	ByteFutureImpl
CharFutureImpl	CharFutureImpl
DoubleFutureImpl	DoublefutureImpl
FloatFutureImpl	FloatFutureImpl
IntFutureImpl	IntFutureImpl
LongFutureImpl	LongFutureImpl
ShortFutureImpl	ShortFutureImpl
BooleanFutureImpl	BooleanFutureImpl
Rmi	on
Horb	off
Host	HOST

表 A.2: プロパティのデフォルト値（-rmi オプション有りの場合）

プロパティ名	値
Prefix	$MILK$
Imports	import milk.runtime.*;\nimport milk.runtime.horb.*;
Parent	ConcurrentObject
AccessLevel	protected
ObjectFutureType	ObjectFuture
ByteFutureType	ByteFuture
CharFutureType	CharFuture
DoubleFutureType	D oublefuture
FloatFutureType	FloatFuture
IntFutureType	IntFuture
LongFutureType	LongFuture
ShortFutureType	ShortFuture
BooleanFutureType	BooleanFuture
ObjectFutureImpl	ObjectFuture_Proxy
ByteFutureImpl	ByteFuture_Proxy
CharFutureImpl	CharFuture_Proxy
DoubleFutureImpl	Doublefuture_Proxy
FloatFutureImpl	FloatFuture_Proxy
IntFutureImpl	IntFuture_Proxy
LongFutureImpl	LongFuture_Proxy
ShortFutureImpl	ShortFuture_Proxy
BooleanFutureImpl	BooleanFuture_Proxy
Rmi	off
Horb	on
Host	HOST

表 A.3: プロパティのデフォルト値（-horb オプション有りの場合）

付録 B

Weather Information の実行方法

この付録では 6.3 節で取り上げた例 WeatherInformation のコンパイル手順と実行方法について述べる。この付録で説明している計算機の OS は solaris を仮定している。

B.1 コンパイル手順

この例を実行する時に使うクラスファイルはサーバ側とウィンドウ側の 2 つに分けて保管する。アプレットはウィンドウ側に保管され、Web ブラウザによって、リモートマシンにロードされる。従って、コンパイルはサーバ側とウィンドウ側の 2 つに分けて行う。それぞれで必要になるファイルを以下にまとめておく。

サーバ側	WeatherInfoServer.milk WeatherInfoServerImpl.milk
	WeatherInfoWindow.milk
ウィンドウ側	WeatherInfoWindow.milk WeatherInfoWindowImpl.milk
	WeaterhInfoServer.milk WeatherInfoApplet.milk
	InfoWriter.milk FixedLayout.java weather.html

ファイル WeatherInfoServer.milk と WeatherInfoWindow.milk はインターフェースを定義したファイルなので、どちらの側でも必要である。ファイル FixedLayout.java を図 B.1 に載せておく。ファイル weather.html はアプレットを実行するための html ファイルである。このファイルを図 B.2 に載せておく。

この例ではウィンドウ側のホスト名を isc-sol2 として、プログラムを作成してある。この部分はコンパイルを行う前に適当に直しておく必要がある。それでは、それぞれについ

てどのような手順でコンパイルするのかを見ていく。

B.1.1 サーバ側

サーバ側は次の手順でコンパイルする。MILK ソースファイルはカレントディレクトリにあると仮定する。

1. MILK ソースコードを Java ソースコードに変換する

ファイル WeatherInfoServerImpl.milk を変換する時にはトランスレータのプロパティ AccessLevel を private にする必要がある。そこで、ファイル weather.properties に

```
AccessLevel private
```

と書いて用意しておく。そして、以下の要領で変換する。途中で改行がしてあるが、1 行で入力する。以下、他のコマンドも同じである。

```
milk -rmi weather.properties WeatherInfoServer.milk
          WeatherInfoServerImpl.milk WeatherInfoWindow.milk
```

2. Java ソースコードをコンパイルする

これは以下の要領でコンパイルする。

```
javac WeatherInfoServer.java WeatherInfoServerImpl.java
      WeatherInfoWindow.java
```

3. スタブクラスとスケルトンクラスを生成する

これは以下の要領でコンパイルする。

```
rmic WeatherInfoServerImpl
```

以上により、サーバ側のクラスファイルを用意することができる。

B.1.2 ウィンドウ側

ウィンドウ側もサーバ側とほぼ同じ手順でコンパイルすることができる。ウィンドウのためのソースファイルは WeatherInfoWindow.milk と WeatherInfoWindowImpl.milk と

WeatherInfoServer.milk である。それ以外はアプレットのためのソースファイルである。コンパイルは以下の手順で行う。ソースファイルはカレントディレクトリにあるものとする。

1. MILK ソースコードを Java ソースコードに変換する

オブジェクト InfoWriter はリモートオブジェクトでは無いので、これだけ別に変換する必要がある。変換は以下の要領で行う。

```
milk -rmi WeatherInfoWindow.milk WeatherInfoWindowImpl.milk
          WeatherInfoServer.milk WeatherInfoApplet.milk
milk InfoWriter.milk
```

2. Java ソースコードをコンパイルする

これは以下の要領でコンパイルする。

```
javac WeatherInfoWindow.java WeatherInfoWindowImpl.java
      WeatherInfoServer.java WeatherInfoApplet.java
      InfoWriter.java FixedLayout.java
```

3. スタブクラスとスケルトンクラスを生成する

これは以下の要領でコンパイルする。

```
rmic WeatherInfoWindowImpl
```

以上により、ウインドウ側のクラスファイルを用意することができる。

B.2 実行方法

この例を実行するには、ウィンドウ、サーバ、アプレットの順番で実行を行っていく。この節では、このそれぞれについてどのようにして実行するかを見ていく。

B.2.1 ウィンドウ

ウィンドウを実行する前に、しっかりと必要なクラスファイルにパスを通しておかなくてはならない。この例ではウィンドウを実行するのに必要なクラスファイルが置いてあるディレクトリを以下のところとして話を進めていく。

~o-abe/public_html/window

このディレクトリは URL で以下のように指定できるとする。

http://www.jaist.ac.jp/~o-abe/window/

そしてまた、環境変数 CLASSPATH を、このディレクトリにあるクラスファイルをロードできるように、セットしておく。

ウィンドウは RMI のネームサーバに登録する必要があるので、ウインドウを起動する前にコマンド rmiregistry を実行しておく。

このように準備をしておいてから、以下のようにするとウィンドウを起動することができる。

```
java -Djava.rmi.server.codebase=http://www.jaist.ac.jp/~o-abe/window/
     WeatherInfoWindowStarter
```

プロパティ java.rmi.server.codebase は RMI で必要なプロパティで、リモートからウインドウにアクセスした時、このプロパティの値のところからクラスファイルがロードされる。

B.2.2 サーバ

サーバもウィンドウとほぼ同じ手順で実行する。サーバのクラスファイルが置いてあるディレクトリを以下のところとする。

~o-abe/public_html/server

そして、ウインドウと同様にこのディレクトリは以下の URL で参照することができるとする。

http://www.jaist.ac.jp/~o-abe/server/

また、環境変数 CLASSPATH を、このディレクトリを探索するようにセットしておく。

そして、以下のようにすることで、このサーバをウィンドウに登録し、実行することができる。

```
java -Djava.rmi.server.codebase=http://www.jaist.ac.jp/~o-abe/server/
     WeatherInfoServerStarter
```

B.2.3 アプレット

RMI を動作させることのできる Web ブラウザは限られている。ここでは appletviewer でアプレットを動作させてみる。

まず、アプレットのクラスファイルの置いてあるディレクトリをカレントディレクトリにする。この例では以下のところである。

```
~o-abe/public_html/window
```

カレントディレクトリを変更したら、以下のようにすることでアプレットを起動することができる。

```
appletviewer weather.html
```

アプレットが起動されたら、アプレット上のテキストフィールドに Kanazawa と入力してみる。するとまず、“Searching now, please wait for a moment.” と表示され、しばらくしてから、天気がランダムに表示される。 Kanazawa 以外の文字列 ???? を入力すると、“Sorry, I don't know ????.” と表示される。

サーバはいくつでも登録できるので、いろいろな名前のサーバを登録して試してみるとよい。

```
import java.awt.*;

public class FixedLayout implements LayoutManager {
    int hgap;
    int vgap;

    public FixedLayout() {
        this(0, 0);
    }

    public FixedLayout(int h, int v) {
        hgap = h;
        vgap = v;
    }

    public void addLayoutComponent(String name, Component comp) {
    }

    public void removeLayoutComponent(Component comp) {
    }

    public Dimension preferredLayoutSize(Container target) {
        Dimension dim = new Dimension(0, 0);
        int nmembers = target.countComponents();

        for (int i = 0 ; i < nmembers ; i++) {
            Component m = target.getComponent(i);
            if (m.isVisible()) {
                Point p = m.location();
                Dimension d = m.preferredSize();
                dim.height = Math.max(dim.height, p.y + d.height - 1);
                dim.width = Math.max(dim.width, p.x + d.width - 1);
            }
        }
```

図 B.1: ファイル FixedLayout.java

```
    Insets insets = target.insets();
    dim.width += insets.left + insets.right + hgap * 2;
    dim.height += insets.top + insets.bottom + vgap * 2;
    return dim;
}

public Dimension minimumLayoutSize(Container target) {
    Dimension dim = new Dimension(0, 0);
    int nmembers = target.countComponents();

    for (int i = 0 ; i < nmembers ; i++) {
        Component m = target.getComponent(i);
        if (m.isVisible()) {
            Point p = m.location();
            Dimension d = m.minimumSize();
            dim.height = Math.max(dim.height, p.y + d.height - 1);
            dim.width = Math.max(dim.width, p.x + d.width - 1);
        }
    }
    Insets insets = target.insets();
    dim.width += insets.left + insets.right + hgap * 2;
    dim.height += insets.top + insets.bottom + vgap * 2;
    return dim;
}

public void layoutContainer(Container target) {
    Insets insets = target.insets();
    int nmembers = target.countComponents();
```

図 B.1: ファイル FixedLayout.java (つづき)

```
        for (int i = 0 ; i < nmembers ; i++) {
            Component m = target.getComponent(i);
            if (m.isVisible()) {
                Dimension d = m.preferredSize();
                m.resize(d.width, d.height);
            }
        }
    }

    public String toString() {
        return getClass().getName() + "[hgap=" + hgap
             + ",vgap=" + vgap + "]";
    }
}
```

図 B.1: ファイル FixedLayout.java (つづき)

```
<HTML>
<TITLE>Weather Information</TITLE>
<CENTER><H1>Weather Information</H1></CENTER>
<APPLET CODEBASE="http://jaist.ac.jp/~o-abe/window/"
        CODE="WeatherInfoApplet.class"
        WIDTH=300
        HEIGHT=100>
</APPLET>
</HTML>
```

図 B.2: ファイル weather.html