Vol. 39　No. 0　情報処理学会論文誌　1998

動的スコープの利用による並列言語の同期・例外処理の階層的構造化

八杉昌宏[†]

実用的な並列処理のための高水準な並列言語には，不規則な計算や副作用を含む実用的な並列プログラムを簡潔に誤りなく記述でき，また，並列計算機上で効率良く実行可能であることが要求される．並列言語の記述性を向上させるには，プログラムを簡潔に記述できること，様々なタイプの並列処理や例外処理に対応した記述ができること，そして逐次言語における goto 文の使用制限に対応するようなバグを防止するための制限といったことが重要になる．本論文では，比較的少数の言語コンストラクトを用いて様々なタイプの並列処理を簡潔に誤りなく記述するためには，動的スコープの利用により同期処理・例外処理を階層的に構造化した並列言語が有効であることを示す．提案方式では，構造化された同期処理・例外処理は個々の操作ではなく構文により記述される．構造化された構文を用いた言語では，並列プログラム実行中の実行状況をユーザが把握しやすく，また並列処理中の例外を自然に処理できる．また，同期処理が操作により記述される従来の場合と比べて，記述量やそれに伴うバグ混入の可能性が削減される．本論文では，提案する言語の設計とその有効性を示すとともに，言語のセマンティクスと実装上望まれる工夫について述べる．

Hierarchically Structured Synchronization and Exception Handling in Parallel Languages using Dynamic Scope

Masahiro Yasugi[†]

In high-level parallel programming languages for practical parallel processing, it is desired to be able to describe practical parallel programs with irregular computation and/or side effect easily and safely as well as to execute them on parallel computers efficiently. Important properties for good description include the ease of writing programs, the support of describing a variety of parallel processing and exception handling, and the restriction on the use of harmful operations such as the one corresponding to `goto` statement in some sequential languages. This paper shows that the effectiveness of the parallel languages with hierarchically structured synchronization and exception handling using dynamic scope to describe a variety of parallel processing easily and safely with a small set of language constructs. The structured synchronization and exception handling are specified by the syntax rather than individual operations. In the structured languages using those syntax, the user can easily picture the configuration of the current execution context and the exceptions thrown in the course of parallel execution can be naturally handled. The description length and the possibility of the presence of bugs are also reduced compared to the conventional approach that uses some synchronization operations. This paper shows the proposed language design and its effectiveness and also describes the language semantics and the implementation issues.

1. はじめに

共有メモリ型アーキテクチャ，分散メモリ型アーキテクチャなど様々な並列アーキテクチャの違いを吸収しつつ信頼性・再利用性・実行効率の高いソフトウェアを開発するには，並列処理のための高水準プログラミング言語が有効である．実用的な並列処理のための高水準並列言語には，不規則な計算や副作用を含む実用的な並列プログラムを簡潔に誤りなく記述でき，また，並列計算機上で効率良く実行可能であることが要求される．

本論文では，並列言語の記述性を向上する—具体的にいえば，比較的少数の言語コンストラクトを用いて様々なタイプの並列処理を簡潔に誤りなく記述可能とする—ためには，動的スコープの利用により同期処理・例外処理を階層的に構造化した並列言語が有効で

[†] 京都大学大学院情報学研究科通信情報システム専攻
Department of Communications and Computer Engineering, Graduate School of Informatics, Kyoto University

あることを示す．またそのような考え方に基づいたオブジェクト指向並列言語の設計の一つについて述べる．また，動的スコープを利用した単純化された言語についてそのセマンティクスを考察し，同期処理構文を例外処理構文と双対にした場合について議論する．さらには実装上望まれる工夫について述べる．

並列プログラムを簡潔に記述するには，対象とする並列処理のタイプを限定してそれにあわせた構文を用意することが考えられる．例えば，並列処理のタイプをデータ並列に限り，forall 構文を用意することが考えられる．並列に処理をしたスレッド間の同期が forall 文の完了時に自動的に取られるとすることで，明示的な同期の記述が必要な方法と比較して，誤りが入りにくいものとなる．また，forall の処理中の例外は forall 文が起こした例外として適切に処理できる．しかし，このような構文だけでは，不規則な処理を含む様々なタイプの並列処理の記述はできない．

そこで，実行時のスレッドの生成操作や，スレッド間の同期操作を記述可能とすることで不規則な問題に対応した言語も数多く設計されている．我々は，これに関して，スレッドの生成操作については対象となる問題の不規則さに合わせるために必要なものと考えるが，明示的なスレッド間の同期操作についてはプログラムを分かりにくくさせるとともに記述量を増加させ，しばしばバグの原因となると考えている．これはスレッド間の同期操作が，逐次言語における goto 文の使用にみられるように，プログラム中の離れたポイントをトップダウン的に把握できない形で結びつけ，プログラムの構造が階層的になることを妨げるためである．

そこで，提案方式では，構造化された同期処理・例外処理は個々の操作ではなく動的スコープを利用した構文により記述するものとする．これは，逐次言語で goto 文を制限して while 構文などの制御構造を導入したり，オブジェクト指向言語でデータに対する自由なアクセスを制限してカプセル化を行うというような，バグを防止するための制限の並列言語版の一つと考えている．つまり，本研究では並列言語版の構造化プログラミングを扱う．本論文で対比させる「明示的な操作による記述」と「構文による記述」について C 言語におけるループを例に説明すると次のようになる：「明示的な操作による記述」では繰り返し実行する部分を分岐先を表すラベルと条件分岐文とで挟んで記述するが，その繰り返し部分がプログラムの構造上一つ内側にあることにはならない．一方「while 構文による記述」では，繰り返し部分がプログラムの構造上一つ内側にあることが示される．そこで，入れ子のレベルに応じた continue 文の制御の移動先の自動決定などが可能となる．

構造化された構文を用いた言語では，並列プログラム実行中の実行状況をユーザが把握しやすく，また並列処理中の例外を自然に処理できる．対象となる並列処理のタイプは，問題の分割は不規則だが，分割された問題の処理の完了の同期により，元の問題の処理の完了が達成されるような分担型である．すべての並列処理が分担型に属すると主張しているわけではないが，「問題の解を求める」や「次のステップに到達する」といった目的を持った並列処理の多くがこのタイプに属すると考えられる．

以下，2 章では並列処理の記述上の問題点を整理し，3 章では動的スコープを用いた同期・例外処理構文とその有効性をオブジェクト指向並列言語の例などを交えて述べる．4 章では単純化された言語上でのセマンティクスを考察し，5 章では同期処理構文を例外処理構文と双対にした場合について議論する．6 章では実装上望まれる工夫について述べ，7 章では関連研究を述べる．

2. 並列処理の記述上の問題点

この章では並列処理の記述に関して，(1) その簡潔さ，(2) 記述可能な並列処理のタイプ，(3) 例外の取り扱いの容易さ，などについて議論する．対象とする言語として次の前提をおく：

- 並列処理のためのコンストラクトを明示的に指定
- C 言語や Java 言語のように副作用が利用可能
- 並列処理の粒度に関する問題は処理系で吸収

2.1 並列実行の構文による記述

構文による並列処理の記述の一つには，データ並列に対応した `forall` を考えることができる：

`forall(i=1 to` N`)` ⟨文⟩

これにより，N 個のスレッドが作られ，それぞれが ⟨文⟩を実行した後，スレッドの実行完了の同期が自動的にとられるとする．また，同様の処理であるが，それぞれのスレッドが異なる文を実行するような記述には Concurrent Pascal 流の `cobegin ...coend` を考えることができる：

`cobegin` ⟨文⟩$_1$`;` ⟨文⟩$_2$`; ...` ⟨文⟩$_N$ `coend`

これにより，N 個のスレッドが作られ，それぞれが ⟨文⟩$_i$ を実行した後，スレッドの実行完了の同期が自動的にとられるとする．

これらの構文による方法の特徴を述べると，(1) 記述は簡潔である．(2) 単純な fork-join 型の並列処理なの

で不規則な処理を含む様々なタイプの並列処理の記述はできない．ただし⟨文⟩のところに再帰的に `forall` や `cobegin ...coend` を記述することで階層構造をとることはできる．(3) 例外処理については個々の⟨文⟩の実行中に発生した例外も `forall` 文全体の例外として適切に処理できる．つまり，`forall` 文に対して `try-catch` を指定しておくことで並列実行中に発生する例外にも対処可能であり，例外発生時には並列実行全体が abort されると定めておけばよい．

2.2　操作によるスレッドの生成と同期

不規則な処理を含む様々なタイプの並列処理の記述のためには，実行時のスレッドの生成操作や（同期のための何らかの場所を介した操作を含む）スレッド間の同期操作を記述可能とした言語も数多く設計されている．例えば Java 言語 $^{1)}$ では実行時にスレッドを生成し，そのスレッドへの参照を用いた様々な操作が可能である．ここでは次のような操作を考える：

```
thr = spawn ⟨文⟩;
```

これにより，⟨文⟩を実行するスレッドが生成され，その参照を`thr` に得るとする．

```
join(thr);
```

これにより，`thr` で参照されるスレッドの終了を待つとする．これらを組み合わせることで，例えば次のような，生成するスレッドの個数が固定でない並列処理の記述が可能になる：

```
{
  int i, n = 0;
  thread_t thr[N];
  for(i=0;i<N;i++)
    if(...) thr[n++] = spawn ⟨文⟩;
  for(i=0;i<n;i++) join(thr[i]);
}
```

これにより，条件が満たされるときのみスレッドを生成して⟨文⟩を実行し，個々のスレッドの実行完了の同期は`join` 文による明示的な操作で行われる．

同様に，スレッドへの参照を用いずデータフロー同期（ある場所の値を取り出そうとしたスレッドが，その値が確定されるまで待たされる）を採用した言語であれば次のような操作が考えられる：

```
ch = future ⟨式⟩;
```

これにより，⟨式⟩の値を計算するスレッドが生成され，並列に計算した値を`ch` で参照される場所に格納するとする．

```
val = touch(ch);
```

これにより，`ch` で参照される場所の値を`val` に取り出すとする．生成するスレッドの個数が固定でない並列処理の記述は`spawn`，`join` を用いたものと同様にできる：

```
{
  int i, n = 0, sum=0;
  int_channel ch[N];
  for(i=0;i<N;i++)
    if(...) ch[n++] = future ⟨式⟩;
  for(i=0;i<n;i++) sum += touch(ch[i]);
}
```

これにより，条件が満たされるときのみスレッドを生成して⟨式⟩を計算し，個々のスレッドの実行完了の同期は`touch` 式による明示的な操作で行われる．

これらのスレッドの生成（`spawn`, `future`），同期（`join`, `touch`）に操作を用いる方法の特徴を述べると，(1) 記述は簡潔とはいえない．特に同期操作のコード（プログラム例での`join`, `touch` に関するループ）や，同期を正しく行うためにスレッドを管理するコード（配列`thr`, `ch` による管理）が余分に必要になる．記述量が増えることで誤りを混入させる危険性が増大し，スレッドの同期の記述が不完全な場合にはゾンビスレッド（プログラマからすれば生きているはずがないのに実際には実行を続けているスレッド）による予期しない副作用というバグの特定が困難な症状が発生する．(2) 不規則な処理を含む様々なタイプの並列処理の記述ができる．(3) 例外処理については，個々の⟨文⟩の実行で発生した例外をその外に伝える方法が自明ではない．例外を適切に処理できるようにするには，例外を親のスレッドに伝えるための操作や処理を続ける必要のなくなったスレッドを停止させるための操作を準備し，タイミングなどを考慮した複雑な記述を明示的に行う必要がある．スレッドを停止させるための操作に関していえば，Java 言語の場合であれば `ThreadGroup` オブジェクトで関連するスレッドを管理しておくことで停止させる操作の記述は短くできるが，記述そのものを無くすことはできない．また，例外を親のスレッドに伝えることに関していえば，例外が発生したスレッドに関して`join` を行ったときや（例外を同期場所に伝えておくこととして）例外が伝えられた同期場所に`touch` を行ったときに，それらの操作を行ったスレッドに例外が伝わるという方法もある．ただし，この場合でも，不要となったスレッドを停止させる記述は必要であり，同期操作が行われるまで例外が伝わらないといった問題もある．またスレッドが操作により同期場所の値を確定させる言語であれば，そもそもスレッドで発生した例外を同期場所に伝えておくことができない．

スレッドの生成と同期を操作により記述する方法では，例外処理が難しいが，同時に並列実行の実行状況のユーザによる把握も難しくなる．スレッドがどういう目的で実行させられているかをユーザがはっきり知るためには，スレッドの実行結果を必要とするプログラム上のポイント，すなわち同期が取られるポイントが分かっている必要があるが，これが実際に同期操作が行われるまで確定しないためである．

2.3 構文による同期処理

操作によるスレッドの生成と同期を行う言語の，スレッド管理の煩わしさと例外処理の困難さの問題のうち，前者の問題を軽減するために構文の助けを借りることができる．例えば，Cilk14) は，C 言語を拡張した並列言語であるが，この言語では関数定義においてキーワード `cilk` を付加することで関数内で生成したスレッドの管理を自動的に行う．例えば次のように記述できる：

```
cilk void foo(...) {
  int i;
  for(i=0;i<N;i++)
    if(...) spawn ⟨関数コール⟩;
  sync;
}
```

これにより，条件が満たされるときに`spawn` により生成したスレッドで⟨関数コール⟩を実行し，個々のスレッドの実行完了の同期は`sync` 文による明示的な操作で行う．ただし，関数の本体というスコープ内で生成されたスレッドは自動的に管理され，`sync` 文の実行によりそれまでに生成されたスレッドが同期される．Cilk 言語では，`cilk` が付加された関数のみスレッドを生成することができ，またその関数の return の際には自動的に`sync` 文に相当する同期がとられることになっている．

Cilk のアプローチの特徴を，2.2 節のスレッド管理と同期を明示的に行う方法と比較して述べると，(1) スレッドを管理するコードが不要になった分，記述は簡潔となり，同期の記述を誤る危険性は減少した．(2) スレッド数が可変の並列処理は記述できるが，関数が終了したときにスレッドが残るような記述はできない．つまり，スレッド生成を行うための記述量が多くてもそれを別の関数として独立させることはできない．(3) 例外処理については，`sync` 文は操作であるため，例外を親スレッドに伝えるための操作が必要という点は改善されない．ただし，スレッドが自動的に管理されているため，Cilk に用意されている`abort` 文を用いて親スレッドはそれまでに生成したスレッドを停止させることはできる．

ここで，Cilk のアプローチの同期の記述を構文で行うように変更したものについて検討する．すなわち，次のような構文を考える：

```
waitfor ⟨文⟩;
```

これにより，⟨文⟩の中に記述された`spawn` 文により生成された個々のスレッドの実行完了の同期は明示的な操作無しで行われるものとする．例えば次のように記述できる：

```
{
  int i;
  waitfor for(i=0;i<N;i++)
           if(...) spawn ⟨関数コール⟩;
}
```

これにより，条件が満たされるときのみスレッドを生成して⟨関数コール⟩を実行し，個々のスレッドの実行完了の同期は`waitfor` 文の実行完了という形で自動的に行われる．ここで，`waitfor` 文の本体というスコープ内で生成されたスレッドは自動的に管理されている．

`waitfor` 文の特徴を，Cilk の方法と比較して述べると，(1) さらに記述は簡潔となり，同期の記述を誤る危険性は減少した．(2) スレッド生成は操作で記述するのでスレッド数が可変の並列処理は記述できるが，同期は構文で行われるのでスレッドの生成時に同期するポイントが定まっているものしか記述できなくなった．(3) 例外処理については，`forall` 文と同様に，個々の⟨関数コール⟩の実行中に発生した例外も `waitfor` 文全体の例外として適切に処理できる．つまり，`waitfor` 文に対して `try-catch` を指定しておくことで並列実行中に発生する例外にも対処可能であり，例外発生時には並列実行全体が abort されると定めておけばよい．

ここで仮定した`waitfor` 文では，そのスレッド管理のスコープ（有効範囲）は lexical なものとしたが，次の章で述べるように，これを動的なものとすることで，`waitfor` 文の lexical スコープの外にある別の関数などでの`spawn` の記述が許されることになる．

3. 動的スコープを用いた同期・例外処理構文

動的スコープとは，Common Lisp の用語でいえば動的なエクステントかつ indefinite なスコープであることを表す．つまり，ある実体の確立と廃止までの間なら，プログラム中のどの場所からでもその実体への参照を行ってよいことを表す．この章では catch-throw 型の例外処理では，例外処理先が動的スコープで参照されるといえることを述べ，同期処理でも同様に同期処理先が動的スコープで参照されるとした場合についい

て議論する．

3.1 catch-throw 型の例外処理

例外処理の記述を Java 言語を用いて説明する．例外を発生させるには`throw`文を用いる：

`throw` ⟨式⟩;

ここで，⟨式⟩の値は`Throwable`クラスのサブクラスのオブジェクトでなければならない．このオブジェクトを例外として`throw`することで例外を発生する．例外発生時には現在の実行を中断し，動的スコープで参照される例外ハンドラに制御を戻すことになる．

例外処理用のコードと有効範囲の記述には try-catch-finally 構文を用いる，try ブロックの中で発生した例外について，対処可能な例外ハンドラを catch 節としてそれぞれ記述する：

```
try {
   ... // 例外が発生しうる処理の記述
}catch(Exception1 ex1){
   ... // 例外 1 発生時実行
}catch(Exception2 ex2){
   ... // 例外 2 発生時実行
}finally{
   ... // 必ず実行
}
```

catch 節が処理できる例外は，その引数のクラス（例えば最初の catch 節であれば`Exception1`クラス）のサブクラスのオブジェクトとなる．try ブロックの実行中に例外が発生しなければ catch 節は実行されない．また，try-catch 構文は（動的な意味においても）入れ子に記述することができ，発生した例外に対応する catch 節がない場合は，より外側の try ブロックに付加された catch 節が例外ハンドラとなる．また例外ハンドラの処理の途中で発生した例外については，より外側の catch 節が例外ハンドラとなる．

例外発生の有無や例外の種類に関わらずに必ず実行したい処理があれば finally 節に記述する．つまり発生した例外が catch 節にたどり着くまでに finally 節が付加された try ブロックを脱出しなくてはならないときは，先に finally 節を実行する．このとき finally 節で新たに例外が発生したときは，元の例外を忘れて新しい例外を処理する．

以上のように，finally 節が原因となり例外が忘れられる可能性を除いていえば，例外が throw されたときに制御を移すべき例外ハンドラは動的スコープで参照される．つまり，try ブロックの中から呼び出されたメソッドやそこからさらに呼び出されたメソッドの実行中に例外が発生すれば，その場所がプログラム上離れていてもそこから例外ハンドラを参照することができる．

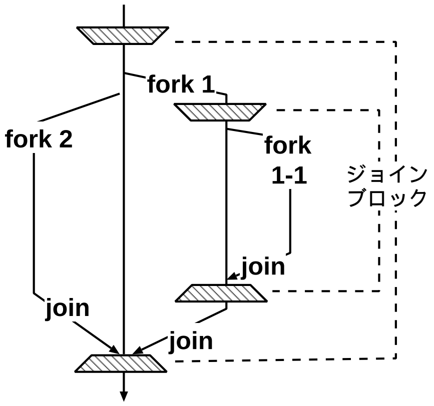

図 1 構造化された同期処理
Fig. 1 Structured synchronization

3.2 動的スコープを用いた同期処理

同期処理も動的スコープにすることが可能である．具体的には，すでに 2.3 節の最後で述べた`waitfor`コンストラクトのスレッドを管理し同期させる機能の有効範囲を lexical スコープではなく動的スコープとしてやるだけでよい．例えば：

```
waitfor { spawn f1(); f(); }
```

という並列処理において，`f1`や`f`が次のように定義されていた場合：

```
f1(){ ...
  waitfor {... spawn f1_1(); ...}
}
f(){ ... spawn f2(); ... }
```

図 1 に示すような並列処理が行われる．図中の join ブロックとは`waitfor`内部の実行開始から終了までを表す．ここで，`f1`において`waitfor`を入れ子で用いている．また，`f`の実行中に`f2`を`spawn`する際に，`waitfor`による同期先（join 先）を動的スコープで参照できる．また図には示していないが，`f`の呼び出しを`spawn`をつけて新しいスレッドで行った場合でも，`f2`の join 先は同じものとなる．このような`waitfor`コンストラクトは，C++を並列処理用に拡張した COOL[2] で用いられている．ただし，COOL ではスレッドの生成は parallel 関数（キーワード`parallel`付きで定義された関数）の呼び出し時に行われ，また例外処理については考慮されていない．

動的スコープによる同期処理構文を用いた並列処理記述の特徴を，2 章で議論した点から述べると，(1) 記述は簡潔である．(2) スレッド生成は操作で記述するのでスレッド数が可変の並列処理が記述できる．同期は

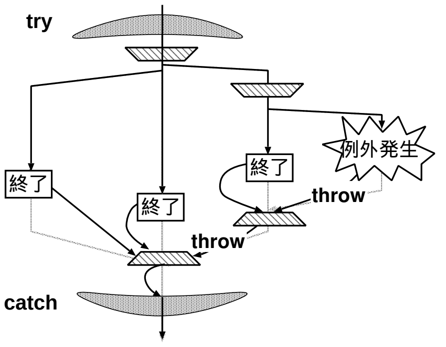

図 **2** 並列処理中の例外処理
Fig. 2 Handling an exception which is thrown during parallel processing

構文で行われるのでスレッドの生成時に同期するポイントが定まっているものしか記述できないが，`f` から `f2` を `spawn` したようにスレッド生成を行うための関数を記述することができる．(3) 例外処理については，個々のスレッドの実行中に発生した例外も `waitfor` 文全体の例外として適切に処理できる．つまり，`waitfor` 文に対して `try-catch` を指定しておくことで並列実行中に発生する例外にも対処可能であり，例外発生時には並列実行全体が abort されると定めておけばよい：

```
try{
  waitfor { spawn f1(); f(); }
}catch( ... ){
  ...
}
```

この様子を図 **2** に示す．あるスレッド内で例外が処理できなかったときは，join 先に例外を伝えていき，そのとき join 先を共有する他のスレッドは停止させられるとする．このように同期に操作を用いた言語では極めて複雑な記述が必要であった例外処理とそれに伴うスレッドの停止処理が簡単な構文を用いて実現される．

また，並列処理の実行状況の把握という点について考えてみる．逐次処理では実行途中の実行状況は呼び出しのスタックにて把握することができる．実際にはスタック上にデータの領域も同時に取られることが多いが，ここでは制御の移動に関係するデータのみを考える．スタックの先頭には現在の実行している文や関数などが完了したときに次に実行するべき文や関数などに関するデータが積まれている．現在の実行している文や関数などを実行するために，その部分文や呼び出し先の実行が必要な場合には，部分文や呼び出し先の実行の後に実行するべき文や関数などに関するデータを更にスタックに積むことになる．よって逐次言語ではスタックをみることで，現在どういう文脈で，つまりどういう目的で実行が行われているかをユーザが把握することができる．一方，図 1 のような並列処理の場合のスタックは cactus stack となり図 **3** に示すように変化する．生成されたスレッドは join 後の処理を続けるという目的をもって処理を行っていると考えることができ，図 3 のように生成時に定まる同期先である join frame 上でその部分スタックを持つと考えることができる．同期に関していえば，join frame はその上のスタックが全て空になったときにその制御を戻すと考えればよい．このように cactus stack を用いると実行状況を直観的に把握できる，また，図 2 のような並列処理中に発生した例外に対する cactus stack の変化は図 **4** のようになる．ここでも例外発生に伴う実行状況の変化を直観的に把握できる，

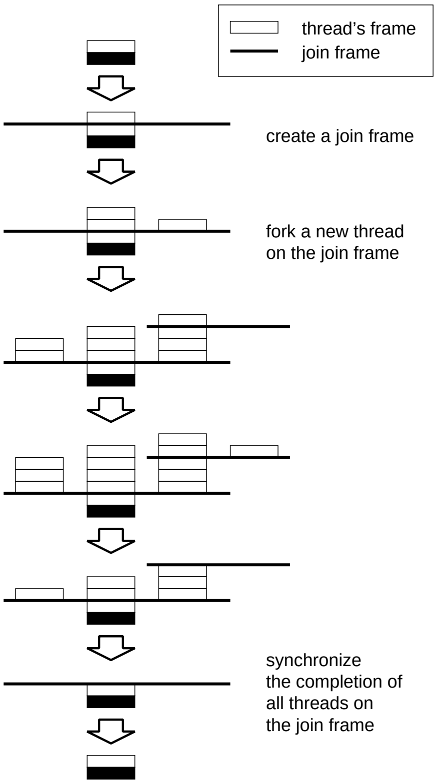

図 **3** 構造化された同期処理に基づく cactus stack の変化
Fig. 3 Transition of the cactus stack based on the structured synchronization

これまで議論してこなかったが，並列処理中の例外処理ではもう少し深い議論が必要である．並列処理中に join 先を共有するいくつかのスレッドでほぼ同時に複数の例外が発生したときに，どの例外が優先されるかという問題がある．これは先に join 先に辿り着いた例外のほうを優先するとするのが自然であると考えられる．似た問題は逐次処理の場合でも，対処しよ

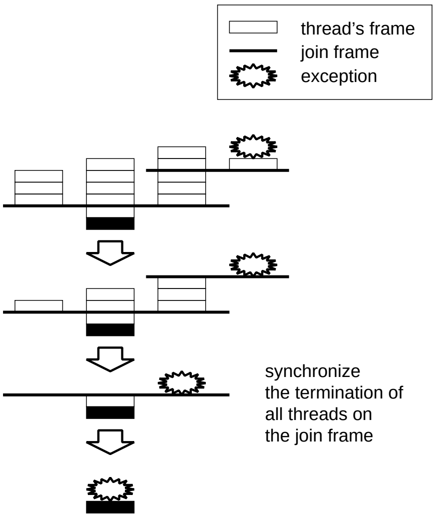

図 4 並列処理中の例外発生時の cactus stack の変化
Fig. 4 Transition of the cactus stack after an exception is thrown during parallel processing

うとしている例外とそのために finally 節を実行していて throw された例外のどちらを優先するかという問題があった．Java の場合は finally 節で再度発生した例外のほうを優先するが，これの性質は並列処理が行われている場合に当てはめることができる．また，join 先を共有する他のスレッドの例外が原因で停止させられるスレッドの振る舞いを定義するのが難しいという問題もある．この問題は，停止させられるスレッドが，停止させられること表す例外（以後 `stopped` と呼ぶ）を自動的に throw すると定めることで巧みに定義できる．ただし，すでに `stopped` を throw した後や，finally 節の実行中には自動的に `stopped` の throw を行わないと定める必要がある．後者は，他のスレッドの例外が原因で finally 節の処理が途中で終わらされることはないようにするためである．また一度 throw された `stopped` が脱出途中の finally 節で発生した例外が優先されて捨てられてしまう問題がある．この問題は一度 `stopped` が捨てられても再度自動的に throw されると定めることで解決できる．

3.3 オブジェクト指向並列言語 OPA

我々はオブジェクト指向並列言語 OPA[12),15)] を開発しており，本論文のテーマとしている動的スコープを用いた同期・例外処理を採用している．OPA は Java[1)] の言語仕様をベースとしつつ，スレッドとモニタの仕様を取り除いて，代わりに使い易く効率の良い並列処理を組み込んだオブジェクト指向並列言語である．

OPA では並列処理を，スレッド間の関係に応じて，分担型並列処理，協調型並列処理，排他型並列処理の三つの型に分類して記述させる．分担型並列処理は，問題（タスク）を複数の部分問題（サブタスク）に分割し，それぞれのサブタスクを並列に実行する処理方法である．スレッドの生成により分割を動的に行うことで不規則な問題にも対応できる．同期はその問題を処理するのに用いたすべてのスレッドの終了を待つことになる．協調型並列処理は，複数のスレッドが互いに通信/同期をとりながら処理を進める方法であり，分担型並列処理で捉えられないパイプライン的な並列処理はこの型に属する．排他型並列処理は，他のスレッド実行の排除を伴いながら処理を進める方法である．

OPA では分担型並列処理に関しては，動的スコープを利用した構文で構造化されているため，従来の並列言語と比べて同期処理の記述が容易であり，誤りが入りにくい．協調型並列処理ではデータフロー同期的な同期場所を提供するクラスのオブジェクトを利用するように記述する．また，排他型並列処理のためには `synchronized` メソッドの他に `instant` メソッドを準備しており，メソッドをその読み書きする変数によって読み出し専用メソッド・読み書き両用メソッドに分類して，オブジェクトに対する複数のスレッドからの並行アクセスをできるだけ許しているため，高速な実行を可能としている．

本論文で提案している例外処理手法や実装方針は，OPA 独自の言語機構には依存していない．ただし，4 章の最後で少し述べるように，OPA では変数の更新などの副作用をオブジェクトや排他制御を用いて空間的・時間的に閉じ込める手段を提供している．

OPA ではスレッドの fork と join に関係する 3.2 節における `spawn`, `waitfor` に対応するコンストラクトとして `par` コンストラクト，`join` コンストラクトを用いる．本論文ではこれ以降この OPA のキーワード `par`, `join` を用いるものとする．並列実行させたいメソッド呼び出しや文に `par` を付加すると，新たにスレッドが生成され，そのメソッドや文を新しいスレッドで実行することができる．また，`join` が付加された文は，その文の実行開始から終了まで（join ブロックと呼ぶ）に fork されたスレッドは，`join` 文の終了においてその終了が待たれるため，`join` 文の終了後にスレッドが生き残ることはない：

```
join{
    par obj1.m1();  // スレッド生成
    par obj2.m2();  // スレッド生成
}  // 生成したスレッドの終了と同期
```

また，生成したスレッドで計算した値を用いて処理を続ける必要がある場合，次のように複文を join ラ

ベルで区切ることで，join ラベルより前の部分を join ブロックとした処理が記述できる：

```
{
  int x = par f1(n);
  int y = par f2(m);
join:
  z = x + y;
}
```

ここで，並列に初期値が与えられる x などの変数のスコープは join ラベル以降として，値の確定した後のみ参照を許す．並列の初期化を用いる変数宣言より後に join ラベルのないものはコンパイル時にエラーとする．

OPA での例外処理の構文は，基本的には Java と同じである．ただし，すでに 3.2 節で述べたように try ブロック内部での並列処理中に発生した例外に対処できる．例えば次のような記述が可能である．

```
try {
  join{
     par obj1.m1();  // スレッド生成
     par obj2.m2();  // スレッド生成
  }  // 生成したスレッドの終了と同期
}catch(Exception1 ex1){  // 例外ハンドラ
}
```

例外が原因となり，停止させられるスレッドは，停止する前に finally 節を実行する．このとき使用していたオブジェクトのロックなども解放する．`instant` メソッドではオブジェクトのデータは一括して更新するが，一括更新より前に停止させられた場合には特に更新を行わない．しかし，例外を throw する前に一括更新を行いたい場合は `vflush` 文により更新することができる．また，オブジェクト間にまたがった処理の中断でのオブジェクト間の関係の一貫性保持に関しては自動的には実現されないので明示的に記述する必要がある．

3.4 OPA による記述例

同期処理のための join-par の記述と，例外処理のための catch-throw の記述という比較的少数の言語コンスラクトを用いて様々な記述が可能であることを例を用いて示す．

配列 `objs` の要素のオブジェクトすべてについてデータ並列的に処理をさせたい場合は次のように記述できる：

```
join for(i=0; i<objs.length; i++)
         par objs[i].doit();
```

`BinTree` クラスのオブジェクトで構成される二分木の全ての leaf の item の総和を並列に求める処理は次のように記述できる：

```
class BinTree {
  int item; boolean isLeaf;
  BinTree left, right;
  ...
  int getSum(){
    if( isLeaf ) return item;
    else{
      int x = par left.getSum();
      int y = par right.getSum();
     join:
      return x + y;
    }
  }
}
```

また以下のように記述することで，万一，総和の並列計算中に例外が起こった場合でも，並列計算を中断してその例外を処理できる：

```
try{
  sum = root.getSum();
}catch(...){ ...
}
```

複数のスレッドが並列に解を探索し，必要個数の解を見つけたときに探索全体を終了させることを考える．従来の言語では並列探索を行っているすべてのスレッドの実行停止といった制御の記述が必要であるためプログラミングが複雑になる．OPA では例外処理を用いて並列実行しているスレッドの終了を容易に行うことができる．以下に例として OPA による解を二つ探索する問題を示す：

```
// メインクラス
//    （例外を受け止める）
public class SearchStart {
  public static
   void main(String argv[]) {
    Table table = new Table();
    try{
      join{
        Node node = new Node(0);
       // 並列探索開始
        node.search(table);
      }
      System.out.println("NotFound");
    }catch(Found excp){   // 例外処理
      System.out.println("Found");
```

```
    }
  }
}

// 解探索のノード
class Node {
  int p;
  Node(int p0){ p = p0; }
  void search(Table table)
       throws Found {
     ...
    if (解発見)
       table.add(解);
    else{
      Node node1 = new Node(2*p+1),
           node2 = new Node(2*p+2);
      par node1.search(table);
      par node2.search(table);
    }
  }
}

// 解を保存するクラス ( 例外が発生する )
class Table {
  int[] answers = new int[2];
  int n = 0;
  instant void add(int ans)
           throws Found {
    answers[n++] = ans;
    if (n == 2) {
      vflush;
      // 例外を発生させる
      throw new Found();
    }
  }
}
```

4. 言語のセマンティクス

図 3 や図 4 に示した cactus stack は，並列処理中の残りの計算を表していると考えることもでき，階層的並列継続と考えることができる．この章では，まず単純な命令型の逐次言語 (真偽値のみを扱い，また場所を直接扱う) についての継続 (continuation) を形式化し，次に，これを拡張して階層的並列継続について形式化を試み，階層的並列継続の概念を明らかにする．以下では 3) で用いられている表記法に従う．

単純な命令型言語の抽象構文：

$\Gamma \in \mathrm{Com}$	commands
$\mathrm{E} \in \mathrm{Exp}$	expressions
$\mathrm{L} \in \mathrm{Loc}$	locations
$\mathrm{K} \in \mathrm{Con}$	constants
$\mathrm{T} \in \mathrm{Tag}$	catch-throw tags

$\Gamma \ ::= \mathrm{L}\ \texttt{:=}\ \mathrm{E} \mid \Gamma\ \texttt{;}\ \Gamma$

$\quad \mid \texttt{if}\ \mathrm{E}\ \texttt{then}\ \Gamma\ \texttt{else}\ \Gamma \mid \texttt{while}\ \mathrm{E}\ \texttt{do}\ \Gamma$

$\quad \mid \texttt{throw}\ \mathrm{T} \mid \texttt{catch}\ \mathrm{T}\ \texttt{in}\ \Gamma \mid \texttt{try}\ \Gamma\ \texttt{finally}\ \Gamma$

$\mathrm{E} \ ::= \mathrm{K} \mid \mathrm{L}$

意味領域：

$\epsilon \in \mathtt{E} = \{\mathit{false}, \mathit{true}\}$	values
$\sigma \in \mathtt{S} = \mathrm{Loc} \to \mathtt{E}$	stores
$\theta \in \mathtt{C} = \mathtt{S} \to \mathtt{A}$	continuations
$\kappa \in \mathtt{K} = \mathtt{E} \to \mathtt{C}$	expression continuations
$\rho \in \mathtt{U} = \mathrm{Tag} \to \mathtt{C}$	tag environments
$\mathtt{A}$	answers

意味関数：

$\mathcal{K} : \mathrm{Con} \to \mathtt{E}$

$\mathcal{E} : \mathrm{Exp} \to \mathtt{K} \to \mathtt{C}$

$\mathcal{C} : \mathrm{Com} \to \mathtt{U} \to \mathtt{C} \to \mathtt{C}$

$$\mathcal{C}\,[\![\mathrm{L}\ \texttt{:=}\ \mathrm{E}]\!]\,\rho\,\theta = \mathcal{E}\,[\![\mathrm{E}]\!]\,\lambda\epsilon.\lambda\sigma.\theta(\sigma\{\mathrm{L} \mapsto \epsilon\})$$

$$\mathcal{C}\,[\![\Gamma_1\ \texttt{;}\ \Gamma_2]\!]\,\rho\,\theta = \mathcal{C}\,[\![\Gamma_1]\!]\,\rho\,(\mathcal{C}\,[\![\Gamma_2]\!]\,\rho\,\theta)$$

$$\mathcal{C}\,[\![\texttt{if}\ \mathrm{E}\ \texttt{then}\ \Gamma_1\ \texttt{else}\ \Gamma_2]\!]\,\rho\,\theta$$
$$= \mathcal{E}\,[\![\mathrm{E}]\!]\,(\lambda\epsilon.\epsilon \to \mathcal{C}\,[\![\Gamma_1]\!]\,\rho\,\theta,\ \mathcal{C}\,[\![\Gamma_2]\!]\,\rho\,\theta)$$

$$\mathcal{C}\,[\![\texttt{while}\ \mathrm{E}\ \texttt{do}\ \Gamma]\!]\,\rho\,\theta$$
$$= \mathit{fix}\ \theta'.\,(\mathcal{E}\,[\![\mathrm{E}]\!]\,(\lambda\epsilon.\epsilon \to \mathcal{C}\,[\![\Gamma]\!]\,\rho\,\theta',\ \theta))$$

$$\mathcal{C}\,[\![\texttt{throw}\ \mathrm{T}]\!]\,\rho\,\theta = \rho(\mathrm{T})$$

$$\mathcal{C}\,[\![\texttt{catch}\ \mathrm{T}\ \texttt{in}\ \Gamma]\!]\,\rho\,\theta = \mathcal{C}\,[\![\Gamma]\!]\,\rho\{\mathrm{T} \mapsto \theta\}\,\theta$$

$$\mathcal{C}\,[\![\texttt{try}\ \Gamma_1\ \texttt{finally}\ \Gamma_2]\!]\,\rho\,\theta$$
$$= \mathcal{C}\,[\![\Gamma_1]\!]\,\{\mathrm{T}_1 \mapsto \mathcal{C}\,[\![\Gamma_2]\!]\,\rho\,\theta_1, \ldots, \mathrm{T}_n \mapsto \mathcal{C}\,[\![\Gamma_2]\!]\,\rho\,\theta_n\}$$
$$(\mathcal{C}\,[\![\Gamma_2]\!]\,\rho\,\theta)$$
$$\text{where } \rho\ = \{\mathrm{T}_1 \mapsto \theta_1, \ldots, \mathrm{T}_n \mapsto \theta_n\}$$

$$\mathcal{E}\,[\![\mathrm{K}]\!]\,\kappa = \kappa(\mathcal{K}\,[\![\mathrm{K}]\!]\,)$$

$$\mathcal{E}\,[\![\mathrm{L}]\!]\,\kappa = \lambda\sigma.\kappa(\sigma(\mathrm{L}))\sigma$$

ただし，$\sigma\{\mathrm{L} \mapsto \epsilon\}$ は，σ の場所 L の値を ϵ で置き換える．また，$\epsilon \to \theta_1, \theta_2$ は ϵ が trueのとき θ_1，ϵ が falseのとき θ_2 であるとする．

以上の表示的意味論で定められる命令言語を，次にその文脈を用いて扱う．文脈書き換え規則により操作的意味論を定義することができる．文脈書き換え規則で用いるストア s は $s : \mathrm{Loc} \to \mathrm{Con}$ とする．

文脈の抽象構文：

$\mathrm{G} \in \mathrm{CCtxt}$	command contexts
$\mathrm{F} \in \mathrm{ECtxt}$	expression contexts
$\mathrm{C} \in \mathrm{Ctxt}$	contexts

$\mathrm{C} \quad ::= \mathrm{G} \mid \mathrm{E} :: \mathrm{F} \mid \texttt{protected} :: \mathrm{C}$

$\mathrm{F} \quad ::= \texttt{update}(\mathrm{L}) :: \mathrm{G} \mid \texttt{select}(\Gamma, \Gamma) :: \mathrm{G}$

$\quad \mid \texttt{then}(\Gamma) :: \mathrm{G}$

$\mathrm{G} \quad ::= \Gamma :: \mathrm{G} \mid \texttt{mark}(\mathrm{T}) :: \mathrm{G}$

$\quad \mid \texttt{finally}\ \Gamma :: \mathrm{G} \mid \texttt{pmark} :: \mathrm{G}$

文脈書き換え規則 $(s, \mathrm{C}) \rightarrow (s', \mathrm{C}')$：

$(s, \mathrm{L}\ \texttt{:=}\ \mathrm{E} :: \mathrm{G}) \rightarrow (s, \mathrm{E} :: \texttt{update}(\mathrm{L}) :: \mathrm{G})$

$(s, \Gamma_1\ \texttt{;}\ \Gamma_2 :: \mathrm{G}) \rightarrow (s, \Gamma_1 :: \Gamma_2 :: \mathrm{G})$

$(s, \texttt{if}\ \mathrm{E}\ \texttt{then}\ \Gamma_1\ \texttt{else}\ \Gamma_2 :: \mathrm{G})$

$\quad \rightarrow (s, \mathrm{E} :: \texttt{select}(\Gamma_1, \Gamma_2) :: \mathrm{G})$

$(s, \texttt{while}\ \mathrm{E}\ \texttt{do}\ \Gamma :: \mathrm{G})$

$\quad \rightarrow (s, \mathrm{E} :: \texttt{then}(\Gamma\ \texttt{;}\ \texttt{while}\ \mathrm{E}\ \texttt{do}\ \Gamma) :: \mathrm{G})$

$(s, \texttt{throw}\ \mathrm{T} :: \Gamma :: \mathrm{G}) \rightarrow (s, \texttt{throw}\ \mathrm{T} :: \mathrm{G})$

$(s, \texttt{throw}\ \mathrm{T} :: \texttt{mark}(\mathrm{T'}) :: \mathrm{G}) \rightarrow (s, \texttt{throw}\ \mathrm{T} :: \mathrm{G})$

$(s, \texttt{throw}\ \mathrm{T} :: \texttt{mark}(\mathrm{T}) :: \mathrm{G}) \rightarrow (s, \mathrm{G})$

$(s, \texttt{throw}\ \mathrm{T} :: \texttt{finally}\ \Gamma :: \mathrm{G})$

$\quad \rightarrow (s, \texttt{finally}\ \Gamma :: \texttt{throw}\ \mathrm{T} :: \mathrm{G})$

$(s, \texttt{throw}\ \mathrm{T} :: \texttt{pmark} :: \mathrm{G})$

$\quad \rightarrow (s, \texttt{pmark} :: \texttt{throw}\ \mathrm{T} :: \mathrm{G})$

$(s, \texttt{catch}\ \mathrm{T}\ \texttt{in}\ \Gamma :: \mathrm{G}) \rightarrow (s, \Gamma :: \texttt{mark}(\mathrm{T}) :: \mathrm{G})$

$(s, \texttt{mark}(\mathrm{T}) :: \mathrm{G}) \rightarrow (s, \mathrm{G})$

$(s, \texttt{try}\ \Gamma_1\ \texttt{finally}\ \Gamma_2 :: \mathrm{G})$

$\quad \rightarrow (s, \Gamma_1 :: \texttt{finally}\ \Gamma_2 :: \mathrm{G})$

$(s, \texttt{finally}\ \Gamma :: \mathrm{G})$

$\quad \rightarrow (s, \texttt{protected} :: \Gamma :: \texttt{pmark} :: \mathrm{G})$

$(s, \texttt{protected} :: \texttt{pmark} :: \mathrm{G}) \rightarrow (s, \mathrm{G})$

$(s, \mathrm{C}) \rightarrow (s', \mathrm{C}')$ ならば

$\quad (s, \texttt{protected} :: \mathrm{C}) \rightarrow (s', \texttt{protected} :: \mathrm{C}')$

$(s, \texttt{true} :: \texttt{select}(\Gamma_1, \Gamma_2) :: \mathrm{G}) \rightarrow (s, \Gamma_1 :: \mathrm{G})$

$(s, \texttt{false} :: \texttt{select}(\Gamma_1, \Gamma_2) :: \mathrm{G}) \rightarrow (s, \Gamma_2 :: \mathrm{G})$

$(s, \texttt{true} :: \texttt{then}(\Gamma) :: \mathrm{G}) \rightarrow (s, \Gamma :: \mathrm{G})$

$(s, \texttt{false} :: \texttt{then}(\Gamma) :: \mathrm{G}) \rightarrow (s, \mathrm{G})$

$(s, \mathrm{L} :: \mathrm{F}) \rightarrow (s, s(\mathrm{L}) :: \mathrm{F})$

$(s, \mathrm{K} :: \texttt{update}(\mathrm{L}) :: \mathrm{G}) \rightarrow (s\{\mathrm{L} \mapsto \mathrm{K}\}, \mathrm{G})$

`finally` 節の文の実行の際は文脈のその文に関する部分を `protected` と `pmark` で挟んで保護する．

文脈は，ほぼそのまま継続に対応させることができる．つまり，文脈は継続をコンパクトなデータ構造として表現している．さらにいえば，catch-throw タグを用いた非局所脱出用の継続も同じデータ構造の中でコンパクトに表現している．このように継続がコンパクトなデータ構造で表現できるのは，対象とした命令型言語が適切な制御構造で構造化されていたためである．

さて，並列言語について文脈の点から，逐次言語から拡張された部分の意味をみることができる．我々の用いる並列言語は，次のようにコマンド Γ に par コマンド，join コマンドを追加したものとなる．

並列言語の抽象構文：

$\Gamma \quad ::= \cdots \mid \texttt{par}\ \Gamma \mid \texttt{join}\ \Gamma$

また，我々が逐次における文脈 C の代わり用いる並列文脈は，次のような階層的な文脈 H となる．

並列言語の文脈の抽象構文：

$\mathrm{H} \in \mathrm{PCtxt} \quad$ hierarchical parallel contexts

$\mathrm{G} \quad ::= \cdots \mid \texttt{term}$

$\mathrm{H} \quad ::= \mathrm{G} \mid \mathrm{E} :: \mathrm{F} \mid \texttt{protected} :: \mathrm{H}$

$\quad \mid \langle\!\langle \mathrm{H} \| \cdots \| \mathrm{H} \rangle\!\rangle :: \mathrm{G} \mid \langle\!\langle \mathrm{H} \| \cdots \| \mathrm{H} \rangle\!\rangle_{\mathrm{q}} :: \mathrm{G}$

ここで，$\langle\!\langle \mathrm{H} \| \cdots \| \mathrm{H} \rangle\!\rangle$ の文脈の順序は重要でなく（自由に入れ替えてよい），またそれぞれの文脈は並列に書き換え可能である．$\langle\!\langle \cdots \rangle\!\rangle$ は図 3 における join frame に対応する．また $\langle\!\langle \cdots \rangle\!\rangle_{\mathrm{q}}$ は図 4 における停止処理中の join frame に対応する．

並列文脈書き換え規則として，逐次の $(s, \mathrm{C}) \rightarrow (s', \mathrm{C}')$ の C を H に読み替えたものからの追加分を示す．

文脈書き換え規則 $(s, \mathrm{H}) \rightarrow (s', \mathrm{H}')$：

$(s, \texttt{join}\ \Gamma :: \mathrm{G}) \rightarrow (s, \langle\!\langle \Gamma :: \texttt{term} \rangle\!\rangle :: \mathrm{G})$

$(s, \langle\!\langle \texttt{par}\ \Gamma :: \mathrm{G} \| \cdots \rangle\!\rangle :: \mathrm{G}_0)$

$\quad \rightarrow (s, \langle\!\langle \Gamma :: \texttt{term} \| \mathrm{G} \| \cdots \rangle\!\rangle :: \mathrm{G}_0)$

$(s, \langle\!\langle \texttt{term} \| \cdots \rangle\!\rangle :: \mathrm{G}) \rightarrow (s, \langle\!\langle \cdots \rangle\!\rangle :: \mathrm{G})$

$(s, \langle\!\langle \rangle\!\rangle :: \mathrm{G}) \rightarrow (s, \mathrm{G})$

$(s, \langle\!\langle \texttt{throw}\ \mathrm{T} :: \texttt{term} \| \cdots \rangle\!\rangle :: \mathrm{G}_0)$

$\quad \rightarrow (s, \langle\!\langle \texttt{throw}\ \mathrm{T} :: \texttt{term} \| \cdots \rangle\!\rangle_{\mathrm{q}} :: \mathrm{G}_0)$

$(s, \langle\!\langle \texttt{throw}\ \mathrm{T} :: \texttt{term} \| \texttt{throw}\ \mathrm{T}' :: \texttt{term} \| \cdots \rangle\!\rangle_{\mathrm{q}}$

$\quad :: \mathrm{G}_0) \rightarrow (s, \langle\!\langle \texttt{throw}\ \mathrm{T} :: \texttt{term} \| \cdots \rangle\!\rangle_{\mathrm{q}} :: \mathrm{G}_0)$

where $\mathrm{T} \neq \texttt{stopped}$

$(s, \langle\!\langle \texttt{throw}\ \mathrm{T} :: \texttt{term} \rangle\!\rangle_{\mathrm{q}} :: \mathrm{G}_0)$

$\quad \rightarrow (s, \texttt{throw}\ \mathrm{T} :: \mathrm{G}_0)$

$(s, \langle\!\langle \texttt{throw stopped} :: \texttt{term} \| \cdots \rangle\!\rangle_{\mathrm{q}} :: \mathrm{G})$

$\quad \rightarrow (s, \langle\!\langle \cdots \rangle\!\rangle_{\mathrm{q}} :: \mathrm{G})$

$(s, \langle\!\langle \rangle\!\rangle_{\mathrm{q}} :: \mathrm{G}) \rightarrow (s, \texttt{throw stopped} :: \mathrm{G})$

$(s, \langle\!\langle \langle\!\langle \cdots \rangle\!\rangle :: \mathrm{G} \| \cdots \rangle\!\rangle_{\mathrm{q}} :: \mathrm{G}_0)$

$\quad \rightarrow (s, \langle\!\langle \langle\!\langle \cdots \rangle\!\rangle_{\mathrm{q}} :: \mathrm{G} \| \cdots \rangle\!\rangle_{\mathrm{q}} :: \mathrm{G}_0)$

$(s, \langle\!\langle \Gamma :: \mathrm{G} \| \cdots \rangle\!\rangle_{\mathrm{q}} :: \mathrm{G}_0)$

$\quad \rightarrow (s, \langle\!\langle \texttt{throw stopped} :: \Gamma :: \mathrm{G} \| \cdots \rangle\!\rangle_{\mathrm{q}} :: \mathrm{G}_0)$

where $\Gamma \neq \texttt{throw}\ \mathrm{T}$

最後の二つの規則が例外が原因となり停止させられる場合の規則である．ただし finally 節のコードを実行中に，他のスレッドの例外が原因となり停止させられることがないように，`protected` された並列文脈については停止させられるための規則はない．

文脈書き換え規則には，非決定的な並列実行のため

に上記に加えて，

$(s, \mathrm{H}) \to (s', \mathrm{H}')$ ならば

$$(s, \langle\!\langle \mathrm{H} \| \cdots \rangle\!\rangle :: \mathrm{G}) \to (s', \langle\!\langle \mathrm{H}' \| \cdots \rangle\!\rangle :: \mathrm{G})$$

を追加しておく．

以上で，非決定的な操作的意味論を階層的並列文脈について与えた．単一の記憶 s について読み出し/書き込み操作を行っているため，階層的並列継続とは，s に対する可能な操作列を得て，s から「可能な操作列を実行した結果の `A`」の集合へ写像としてとらえることもできる．しかしながら，このような定義を正確に与えたとしてもそれはあまり役に立たず，むしろ，階層的並列継続という関数よりは，階層的並列文脈というデータ構造のまま理解した方がよいことが多い．例えば，階層的並列文脈上での `throw` がどのような効果を持つのかは，階層的並列文脈上で十分理解できる．

ただし，階層的並列継続が非決定性の影響をあまり受けない場合は，階層的並列文脈はその継続をかなりよく近似する．それは，並列に動作するスレッドが互いに干渉せずにストア s を操作する場合である．そのためにあるスレッドが書き込む場所を，並列に動作する他のスレッドが読み書きしないように空間的にストアを分離させるか，他のスレッドの読み書きが終わるまで読み書きをしないように操作を時間的に分離させるかが重要になる．われわれの開発しているオブジェクト指向並列言語では，空間的分離をオブジェクトやそのカプセル化によって促進し，また，時間的分離をオブジェクトに対する処理系による適切な排他制御によりサポートしている．

5. 同期構文の拡張

例外処理の構文では例外ハンドラが catch-throw のタグを用いて選択できたのに対し，ここまでの同期構文では，スレッドの join 先はそれを取り囲む一番内側の join ブロックに対応するものと決まっていた．つまり，新たにスレッドが生成されるとき，生成元のスレッドと共通の join 先になると決まっていた．

比較的簡単な拡張で，同期構文でも join 先をタグを用いて選択できるようにできる．それには，基本的に cathc-throw による例外処理の双対の場合を考えればよい．具体的な構文としては次のものが考えられる：

```
do {
  f1();
}join(Total1 s1, Total2 s2, ...){
  ... // 同期完了時の処理
}
```

つまり，do-join 構文を準備し，do ブロック内部の `f1` の実行中に生成されたスレッドのうち，計算結果を `Total1`, `Total2`, ... のいずれかのクラス（またはサブクラス）のオブジェクトで返すものすべてを同期させる．また（後述の `reduce` メソッドにより）トータルした結果を `s1`, `s2`, ... の値として，同期完了後の処理を join 節に記述する．

このとき，並列に値を計算するスレッドを生成する方法は次のようなものが考えられる：

```
void f1(){ pthrow f2(); ... }
Total1 f2(){ ... return new Total1(...); }
```

ここで，`pthrow` の返す値を新しいスレッドを用いて計算するものとする．この新しいスレッドの join 先は，`pthrow` される式の型を用いて選択される．ここで `pthrow` される値の型（クラス）とすることはできない．これはスレッドの生成時点で join 先は定まらなければならないためである．join 先が最終的に求められる値で定まるとした場合，値が求められるのを待つこと自体が同期を行っていることになってしまう．

ここで，同期先を指定するための `Total1` のようなクラスは，リダクションのための `reduce` メソッドを持つとする．例えば次のような Sum クラスが考えられる：

```
class Sum extends Total {
  internal int sum=0;
  instant int getSum(){ return sum; }
  instant void reduce(Sum a){
    sum += a.getSum();
  }
}
```

これにより総和を求めるために生成されたスレッドの結果をまとめることができる．言語の仕様としては，リダクションの順番は実装で自由であるとしておくことで様々な最適化ができると考えられる．例えば，各プロセッサでローカルに総和をとるといった実装も許されるとする．もちろんそのためには各プロセッサで単位元を用意しないといけないが，これは無引数のコンストラクタで準備されるとすればよい．

`pthrow` コンストラクトでは，必ず式を記述しなくてはならないので，文が記述できるようにすることも検討する．例えば次のような記述を可能とする：

```
void f1(){
  Total1 par{ ... return new Total1(...);}
  Total2 par{ ... return new Total2(...);}
}
```

これは，`par` ブロックの中を実行するスレッドを生成するが，生成したスレッドが計算する値のクラスを `par`

コンストラクトの前に指定し，計算した値は `return` で返すものである．

以上のようにして join 先を選択できるようにすることで次のメリットが生じる．一つは，join 文の完了後も生き続けるスレッドの生成を join ブロック内に記述できるようになったことである．現在，Java 言語用に書かれたライブラリの OPA における再利用を検討しているが，join 先がプログラム全体に対応するようなスレッドを用いれば Java のスレッドに近いものとなり再利用が容易になると考えられる．もう一つの利点は「ある並列処理のためにスレッドを生成する」という処理を並列に実行できるようになったことである．例えば，木構造のデータを入力としてその leaf 全てについてスレッドを生成することを目的とするメソッドを記述しなくてはならないとき，木構造を並列に辿るという記述が可能になる．

6. 実装上の留意点

6.1 同期処理の実装

スレッド終了同期の判定には join frame とスレッドのもつ重みを用いる．join ブロックの開始時，join frame を生成したスレッドと生成された join frame は等しい重みを持つ．その join ブロック内で新たなスレッドが生成された時は，自スレッドの持つ重みを新しいスレッドに分け与え，join する時は，join frame に自分の重みを返す．これにより，join ブロック内のすべてのスレッドが終了した時，各スレッドによって返された重みと join frame が最初に持っていた重みが等しくなり，スレッド終了同期の判定が行える．

6.2 例外処理の実装

例外処理の実装は，4 章のセマンティクスからもわかるように，並列処理中に発生した例外をどう扱うかが実装のポイントとなる．

発生した例外を現在のメソッド内で受け止めることができる場合は，その例外に対応する処理に移行する．現在のメソッド内で受け止めることができない場合は，呼び出し元のメソッドに例外を返し，現在のスレッド内での処理を試みる．現在のスレッド内で処理できないときはその join 先に例外が伝えられ，join 先の join frame にスレッド停止フラグを立てる．スレッド停止フラグが立つ join frame 上のスレッドは finally 節の実行中を除き，処理をできるだけ早く中断すべきである．

処理の中断のためには，ポーリングを行うような実装が必要となり，ポーリングのオーバヘッドとスレッドが停止されるまでの遅延のトレードオフになる．一回のポーリングを効率化する方法の一つには，各プロセッサに対して例外フラグも準備し，例外フラグが立っているときだけ例外が発生していないかをスタックの深くまで調査するという方法が考えられる．

また，例外を起こしたスレッドが，finally の処理や浅いところにある join frame でのスレッドの終了待ちを行うために，深いところにある join frame に例外を伝えるまでに時間がかかる可能性がある．これを実質上高速化するためには，あらかじめスレッド停止フラグが立ちそうな join frame に仮のフラグを立てておいて，真のフラグの状態が確定するまでは仮のフラグの立つ join frame 上のスレッドの処理の優先度を下げておくことが考えられる．ただし優先度を下げたスレッドの計算が進まないと全体の計算が止まってしまう可能性があるため，完全にサスペンドさせることはできない．この方法は「見込みのない投機的実行をなるべく実行しない」というタイプの投機的実行と考えられる．

7. 関連研究

7.1 Java

Java の並列処理の記述には `Thread` クラスを用いる[13)]．`Thread` オブジェクトを生成することにより fork を行い，join はそのオブジェクトの `join` メソッドを呼び出すことで行う．しかしこのようなスレッド制御を行うためには，ユーザが `Thread` オブジェクトを管理する必要があるため，誤りが混入しやすくなる問題がある．

Java の例外処理は，基本的に例外が発生したスレッド内でその例外を解決するように設計されている．また，`ThreadGroup` オブジェクトを導入することで，複数のスレッドに対する制御をある程度容易にしている．

OPA は，Java との互換性をできるだけ保つようにしているが，OPA ではさらに，スレッドで発生した例外を join 先に伝えるとともに，共通の join 先を持つスレッドを自動的に終了させることができる．

7.2 ABCL/1

並列オブジェクト指向言語 ABCL/1 の例外処理[9)]は，例外メッセージに対する動作として記述する．例外発生時には例外メッセージが作成されオブジェクトに送信される．例外メッセージの送信先には，例外が発生したメソッドのタイプや引数に応じて，そのメソッドを呼び出したオブジェクトや返答メッセージを受け取るオブジェクトが指定される．

ABCL/1 は「並列オブジェクト」に基づく言語であるため，例外処理は，例外発生時の並列オブジェク

トの振舞いを個々に定義することで対処している．それに対し，OPA はスレッドをベースとしているため，例外処理はオブジェクトとは独立に catch 節に記述でき，また他のスレッドの自動終了を実現している．

7.3 KL1

KL1 は並列論理型言語であり，例外処理には荘園11) を用いている．荘園とは，ある複数のプロセスのグループのことであり，グループはあるプロセスとそのサブプロセスすべてで形成されている．また，荘園は入れ子構造にできる．プロセスが発生させた例外は，プロセス単位でなく，そのプロセスが属する荘園単位で扱われる．荘園の内外との通信のために，報告ストリームと制御ストリームが用意されており，これを用いて荘園内部で発生した例外を外部に伝えることができ，また，荘園内部に制御命令を送ることができる．

KL1 と OPA では，並列処理における例外処理の考え方は似ている．しかし KL1 の荘園は，入出力をもつ独立したプロセスとみることができ，そのプロセスの例外を扱うと考えることができる．一方 OPA では，ある大きな処理全体の中の一部の処理中で発生する例外を扱うと考えている．

7.4 Qlisp

初期の Qlisp4) は比較的少数の言語コンストラクトを用いて様々なタイプの並列処理を簡潔に誤りなく記述可能とすることを目指しており，本研究と共通する部分が多い．特に，並列処理中の例外発生時に同期先が共通のスレッドを停止させるというアイデアは 4) の中で述べられている．また，その後 Qlisp には多くの言語コンストラクトが加えられ5)，中でも `qwait` コンストラクトは，動的スコープによりその実行中に生成したすべてのスレッドの終了を待つもので，OPA における `join` コンストラクトと同等の機能を持つ．一方，OPA では，5 章で述べたように `join` コンストラクトを拡張しており，例外処理でタグが使い分けられるのと同様に，同期処理でも join 先のためのタグが使い分けられるような構文を準備している．

7.5 一級継続を用いたもの

逐次言語においては残りの計算（＝継続）を一級データとして扱う一級継続を利用して，非局所脱出や例外処理やコルーチンなどの処理を記述できる．並列言語においては，コルーチンのような機能については継続を使わなくても実際に並列に動き得るものとして実現できる．また，非局所脱出や例外処理については一級継続を用いなくても本論文で述べてきたような catch-throw を用いて実現できる．このため我々は並列言語においては，一級継続を扱える必要性は小さいと考えている．ただし継続の概念は並列計算の意味論を記述するためのヒントになると考えている．

並列計算における一級継続の研究には Katz と Weise による研究10)，Hieb と Dybvig による研究7),8) がある．ともに，Scheme ベースの研究である．Katz と Weise による研究では，future6) と一級継続の両方を用いても future を除去した逐次言語で実行したときと同じ結果になるような範囲でのみ並列実行が許されるような方法を提案している．Hieb と Dybvig による研究7) では，図 3 のような木構造の cactus stack において，その一部（部分木）を切り取って一級データとして表現し，切り取った場所まで非局所脱出して，その一級データを渡す．一級データとして保存された部分木はそれを cactus stack 上の別の場所で呼び出すことで接ぎ木として実行できる．

8. おわりに

本論文では，並列言語の記述性を向上する—具体的にいえば，比較的少数の言語コンストラクトを用いて様々なタイプの並列処理を簡潔に誤りなく記述可能とするためには，動的スコープの利用により同期処理・例外処理を階層的に構造化した並列言語が有効であることを示した．特に並列処理中に発生した例外に対する例外処理の記述は極めて簡潔なものとすることができた．また，動的スコープを利用した単純化された言語についてそのセマンティクスを考察し，実装上望まれる工夫について述べた．

今後は，提案した機能を利用した言語処理系を実装し，実際の並列計算機を用いた評価を行っていきたい．

謝辞 日頃議論していただいた神戸大学瀧研究室の OPA グループの皆様と京都大学の小宮常康博士に感謝いたします．本研究の一部は文部省科学研究費（奨励 (A)09780278）による．

参考文献

1) Arnold, K. and Gosling, J.(eds.): The Java Programming Language, Addison-Wesley Publishing Company (1996).

2) Chandra, R., Gupta, A. and Hennessy, J. L.: COOL, *Parallel Programming Using C++* (Wilson, G. V. and Lu, P.(eds.)), The MIT Press, chapter 6 (1996).

3) Clinger, W. and Rees, J.(eds.): *Revised^4 Report on the Algorithmic Language Scheme*, MIT AI Memo 848b, MIT (1991).

4) Gabriel, R. P. and McCarthy, J.: Queue-based multi-processing Lisp, Technical Report STAN-CS-84-1007, Department of Computer Science,

Stanford University (1984).
5) Goldman, R. and Gabriel, R.P.: Qlisp: Parallel Processing in Lisp, *IEEE Software*, pp. 51–59 (1989).
6) Halstead, R. H.: New Ideas in Parallel Lisp: Language Design, Implementation, and Programming Tools, *Parallel Lisp: Languages and Systems* (Ito, T. and Halstead, R. H.(eds.)), Lecture Notes in Computer Science, Vol. 441, Sendai, Japan, June 5–8, Springer, Berlin, pp. 2–57 (1990).
7) Hieb, R. and Dybvig, R.K.: Continuations and Concurrency, *ACM Conf. on the Principles and Practice of Parallel Programming (PPOPP)*, pp. 128–136 (1990).
8) Hieb, R., Dybvig, R. K. and Anderson, III, C. W.: Subcontinuations, *Lisp and Symbolic Computation*, Vol. 7, No. 1, pp. 83–110 (1994).
9) Ichisugi, Y. and Yonezawa, A.: Exception Handling and Real Time Features in an Object-Oriented Concurrent Language, *Concurrency: Theory, Languages and Architecture*, Lecture Notes in Computer Science, Vol. 491, Springer-Verlag, pp. 92–109 (1990).
10) Katz, M. and Weise, D.: Continuing Into the Future: On the Interaction of Futures and First-Class Continuations, *ACM Conference on Lisp and Functional Programming*, pp. 176–184 (1990).
11) 瀧和男: 第五世代コンピュータの並列処理, 共立出版(1993).
12) 八杉昌宏, 瀧和男: 並列処理のためのオブジェクト指向言語OPAの設計と実装, 情報処理学会研究報告, Vol. 96, No. 82, pp. 157–162 (1996).
13) Oaks, S., Henry Wong. 戸松豊和, 西村 利浩訳: JAVAスレッドプログラミング, オーム社(1997).
14) Supercomputing Technologies Group: *Cilk-5.1(Beta1) Reference Manual*, MIT Laboratory for Computer Science (1997). http://theory.lcs.mit.edu/~cilk.
15) 八杉昌宏, 瀧和男: 実用的な並列処理のためのオブジェクト指向言語OPAの設計, 第13回オブジェクト指向計算ワークショップ(WOOC'97) (1997).

(平成10年10月10日受付)
(平成10年12月16日採録)

八杉　昌宏(正会員)

1967年生．1989年東京大学工学部電子工学科卒．1991年同大学大学院電気工学専攻修士課程修了．1994年同大学大学院理学系研究科情報科学専攻博士課程修了．1993～1995年日本学術振興会特別研究員(東京大学，マンチェスター大学)．1995年神戸大学工学部助手．1998年より京都大学大学院情報学研究科通信情報システム専攻講師．博士(理学)．並列処理，言語処理系などに興味を持つ．日本ソフトウェア科学会，ACM会員．