関数型言語 Haskell による並列データベースの設計と実装

平成26年度 学位論文(修士)

琉球大学大学院　理工学研究科
情報工学専攻

當眞 大千

要旨

Haskell は純粋関数型プログラミング言語である。純粋であるため、引数が同じならば関数は必ず同じ値が返すことが保証されている。純粋性は、関数の正しさを簡単に推測できる、関数同士を容易に組み合わせることができるといったメリットをもたらす。

本研究では、Haskell を用いて並列に実行できるデータベースの設計と実装を行う。並列にデータへアクセスする手法として、非破壊的木構造を用いる。非破壊的木構造は、破壊的代入の存在しない Haskell と相性がよい。

Haskell の純粋性は並列実行と相性が良いが、実際に並列プログラムを書く際には遅延評価が問題となる。結果が必要となるまで評価しないため、いつ実行されるかが不明確で並列度を思うように高めることが難しい。また同じ呼び出しがあった場合、Haskell には結果をキャッシュし同じ式が再計算されない仕組みがあるが、並列実行時には各スレッド間での同期コストが大きくなってしまう。

Haskell での並列実行について考察し、並列データベースの設計及び実装を行った。データベースの書き込み及び読み込みについて性能を計測し、並列データベースを評価する。また、簡易掲示板システムを開発し、既存の Java の並列データベースの実装との性能比較を行う。

目 次

図 目 次

表 目 次

第1章　序論

1.1　研究背景と目的

ブロードバンド環境やモバイル端末の普及により、ウェブサービスの利用者数は急激に伸びている。リクエスト数の増加を予想することは困難であり、負荷が増大した場合に容易に拡張できるスケーラビリティが求められる。ここでいうスケーラビリティとは、利用者や負荷の増大に対し、単なるリソースの追加のみでサービスの質を維持することのできる性質のことである。

ウェブサービスにおけるスケーラビリティを実現するための難点の一つとして、データベースが挙げられる。本研究の目的は、スケーラビリティを実現するデータベースの実装である。ウェブサービスにおけるスケーラビリティを実現するためには、並列にデータにアクセスできる設計が必要となる。本研究では並列にデータへアクセスする手法として、非破壊的木構造を利用する。非破壊的木構造では、排他制御をせずにデータへアクセスすることが可能でありスケーラビリティを確保できる。

データベースの実装には、純粋関数型言語 Haskell を用いる。Haskell を用いることで、表現力や純粋性のメリットを享受することができる。Haskell では、高度な型を一からつくり上げることができ、型情報を利用してコンパイル時に多くのエラーを捕捉できる。また、並列処理において副作用に依存する問題から解放され処理が簡潔になるといったメリットがある。

本研究で実装したデータベースを用いて、簡易掲示板システムを開発し、既存の Java による非破壊的木構造データベースと性能比較を行う。

1.2　本論文の構成

本論文では、データベースの開発によって得られた知見について述べる。

第 2 章では、純粋関数型言語 Haskell の特徴及び Haskell を用いた並列プログラミングについて述べる。第 3 章では、非破壊的木構造を用いたスケーラビリティ確保の方法について考察し、Haskell を用いたデータベースの設計について述べる。第 4 章では、Haskell を用いた非破壊的木構造データベースの実装と、その利用方法について述べる。第 5 章では、実装したデータベースのベンチマークを行う。簡易掲示板システムを開発し、既存の Java との性能比較を行う。

第2章　Haskellとは

Haskell とは純粋関数型プログラミング言語である。Haskell では、変数の代入は一度のみで書き換えることはできない。また、引数が同じならば関数は必ず同じ値を返すことが保証されている。このような特徴から既存の手続き型言語と同じようにプログラムを書くことはできず、関数を組み合わせることでプログラミングを行う。

2.1　関数型プログラミング

関数とは、一つの引数を取り一つの結果を返す変換器のことである。関数型プログラミング言語では、引数を関数に作用させていくことで計算を行う。

Haskell は純粋であり、引数が同じならば関数は必ず同じ値を返すことが保証されている。これはつまり、関数の正しさを簡単に推測でき、また関数同士を容易に組み合わせられることを意味している。実際、Haskell では小さな関数をつなぎ合わせることでプログラミングを行う。

関数型プログラミング言語は、関数を第一級オブジェクトとして扱うことができ、高階関数を定義することができる。これは、引数として関数を取ったり返り値として関数を返すことができるということである。高階関数は問題解決の強力な手段であり、関数型プログラミング言語にはなくてはならないものである。Haskell では標準ライブラリに、リストを処理するための便利な高階関数がいくつも定義されている。

Haskell では、全ての関数は一度に一つの引数だけを取る。複数の引数を取るようにみえる関数は、実際には１つの引数を取り、その次の引数を受け取る関数を返す。このように関数を返すことで全ての関数を一引数関数として表すことをカリー化という。カリー化によって、関数を本来より少ない引数で呼び出した際に部分適用された関数を得ることができる。

再帰もまた関数型プログラミング言語において必要不可欠な要素である。再帰とは、関数を関数自身を使って定義することをいう。関数型プログラミング言語では、ループといった処理を行う場合再帰を用いる。また、リストの畳み込み関数といった再帰を用いた関数が標準ライブラリで提供されている。

2.2　型

Haskell は静的型付け言語である。全ての型に起因するエラーはコンパイル時に捕捉される。コンパイル時に多くのエラーを見つけてくれるため、プログラマがプログラムの正

しさを確認するのに役に立つ。また、Haskell は型を明示しなくても、処理系により自動的に型が推論される。型安全を保ちながら、ほとんどの部分で型宣言を省略することができる[注1]。

データ型

Haskell の標準なデータ型には、Int 型や、Float 型、Bool 型、Char 型などが存在する。data キーワードを用いることで、自作のデータ型を作ることもできる。標準ライブラリにおける Bool 型は 2.1 のように定義されている。

```
data Bool = False | True
```

ソースコード 2.1: Bool 型の定義

この型宣言は、「Bool 型は True または False の値を取り得る」ということを表している。

型変数

Haskell の型は多相的である。型変数を使い、型を抽象化できる。型変数は、型安全を保ちながら、関数を複数の型に対して動作するようにできる。Haskell の型の名前は全て大文字から始まるが、型変数は小文字から始まる。任意の型のリストを引数に取り、その型の先頭要素を返すという関数 head の型は 2.2 のように定義される。

```
head :: [a] -> a
```

ソースコード 2.2: 型変数

データ型も型変数を使うことができる。例えばリストといったデータ型は、その要素の型を型変数として定義する。これらのデータ型は、型引数を受け取って具体型を生成する。言うまでもなく、型推論があるため明示的に型引数を渡す必要はない。

型クラス

型クラスは型の振る舞いを定義するものである。ある型クラスのインスタンスである型は、その型クラスに属する関数の集まりを実装する。ある型を型クラスのインスタンスにしようとする場合、それらの関数がその型ではどのような意味になるのか定義する。例えば、Eq 型クラスのインスタンスとなる型は等値性をテストできる。Show 型クラスのインスタンスとなる型は文字列として表現できる。いくつかの関数には型クラスが制約となるものがある。演算子 * の引数は、数を表す型クラスである Num 型クラスである必要がある。

[注1]明示的な型宣言は可読性の向上や問題の発見に役に立つため、トップレベルの関数には型を明記することが一般的である。

```
(*) :: Num a => a -> a -> a
```

ソースコード 2.3: 演算子 *

１つの型はいくつもの型クラスのインスタンスとなることができる。自作のデータ型を作成する際に型クラスを利用することで、既存の Haskell ライブラリを利用できる。

2.3 モナド

Haskell では、さまざまな目的にモナドを使う。モナドを用いることで I/O 処理を行うこともできる。

モナドとなる型は、型変数として具体型をただ１つ取る。これにより何かしらのコンテナに包まれた値を実現する。モナドの振る舞いは型クラスとして実装し、メソッドとして return および >>= を定義する。return は値を持ち上げてコンテナに包む機能を実装する。>>= は、「コンテナに包まれた値」と、「普通の値を取りコンテナに包まれた値を返す関数」を引数にとり、コンテナに包まれた値をその関数に適用する。この 2 つの関数を利用することにより、状態を保ったまま関数を繋いでいくことができる。

Haskell で副作用を持つ処理を実行するには、IO モナドを利用する。IO モナドは、main という名前と関連づけることで初めて実行される。IO モナド自体は単なる命令書であり、引数として渡したりするだけでは実行されない。そのため、IO モナドをリストに格納したりすることも可能である。

2.4 並列実行

Haskell はデフォルトではシングルスレッドで走る。並列に動かしたい場合は、-threaded 付きでコンパイルし、RTS の -N オプションを付けて実行する。-N オプションで指定された数だけ、OS のスレッドが立ち上がり実行される。

```
$ ghc -O2 par.hs -threaded
$ ./par +RTS -N2
```

ソースコード 2.4: 並列実行の様子

当然これだけでは並列に動かず、並列に実行できるようにプログラムを書く必要がある。

Haskell は遅延評価を行うため、必要となるまで式の評価が遅延される。普段は気にする必要はないが、並列実行ではどのように評価されるか意識する必要がある。並列に動くように処理を分割したとしても、値が必要になるまで評価されない。この問題は、Control.DeepSeq モジュールにある deepseq 関数を使用し、式を即時評価することで解決できる。

Haskell の並列化手法はいくつかあるが、その中で最もシンプルなのは Eval モナドを利用することである。Control.Parallel.Strategies モジュールをインポートすることで使用できる。

```
data Eval a
instance Monad Eval

runEval :: Eval a -> a

rpar :: a -> Eval a
rseq :: a -> Eval a
```

ソースコード 2.5: Eval モナド

rpar は、引数が並列処理可能であることを示す。rseq は、引数を評価し、結果を待つように示す。どちらも式が完全に即時評価されるわけではないので、出力を挟んだり、deepseq 関数をうまく活用する必要がある。runEval は、評価を実行し結果を返す操作である。実際の利用方法として 2 つのパターンを紹介する。

```
runEval $ do
   a <- rpar (f x)
   b <- rpar (f y)
   return (a,b)
```

ソースコード 2.6: rpar/rpar

rpar/rpar パターンは単純に並列に動かしたい時に利用する。即座に return される。

```
runEval $ do
    a <- rpar (f x)
    b <- rseq (f y)
    rseq a
    return (a,b)
```

ソースコード 2.7: rpar/rseq/rseq

rpar/rseq/rseq パターンは、f x および f y の結果を待ってから return する。他の処理が結果を必要としている時に利用する。

第3章　Haskellによる並列データベースの設計

3.1　スケーラビリティのあるデータベース

本研究の目標はマルチスレッド環境でのスケーラビリティのあるデータベースの開発である。データベースは、ウェブサービスでの利用を前提とする。Twitter や Facebook などのタイムラインは多少遅延しても問題のないサービスである。これはデータの整合性を犠牲にできることを意味し、結果的に整合性が保たれればよいという考え方であり結果整合性と呼ばれる。本研究で開発するデータベースは結果整合性の特性を持つ。

開発するデータベースはデータに並列にアクセスできる。マルチスレッド間でデータを共有する方法はいくつかある。最も一般的なのは、共有変数に対して排他制御を行う方法である。データにアクセスしてよいスレッドを１つに制限することで、そのデータが常に正しいことが保証される。しかしながら、１つのスレッドしかアクセスできないため、大量にデータにアクセスする場合ボトルネックとなってしまい並列度がでない。本研究では非破壊的木構造を採用する。非破壊的木構造は、最新の木構造はどれかという情報を取り扱う部分にのみ排他制御が必要で、並列に編集可能なデータ構造として扱うことができる。そのためマルチスレッド環境でのスケーラビリティのあるデータベースと相性がよい。また、ウェブサービスのデータ構造は木構造で表現することができる。

3.2　非破壊的木構造

非破壊的木構造は、木構造を書き換えることなく編集を行う手法である。排他制御を行わず、並列に読み書きを行うことが可能である。元の木構造は破壊されることがないため、自由にコピーを行うことができる。コピーを複数作成することでアクセスを分散させることも可能である。

通常の破壊的木構造との違いを説明する。

3.2.1　破壊的木構造

破壊的木構造は、木構造を編集する際に木構造を書き換えることにより編集を実現する。図 3.1 では、ノード６をノード A に書き換える処理を行っている。

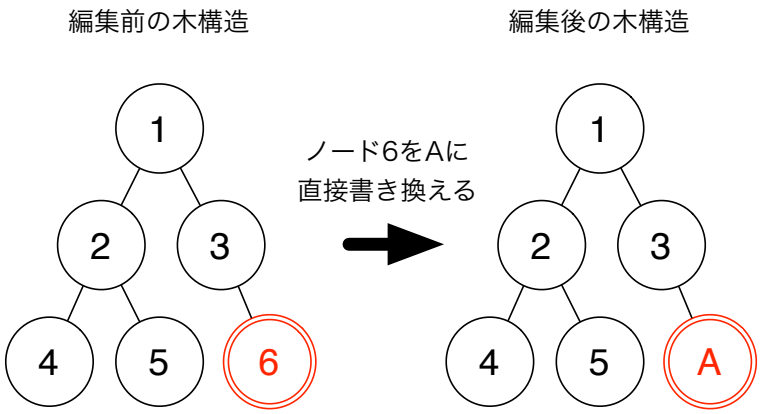

図 3.1: 木構造の破壊的編集

破壊的に編集する方法では、並列環境において問題が発生する。閲覧者が木構造を参照している間に編集者が木構造を書き換えると、閲覧者が参照を開始した時点での木構造ではなくなり整合性が崩れてしまう。整合性が崩れた状態では、正しい状態のコンテンツを参照することは出来ない（図 3.2）

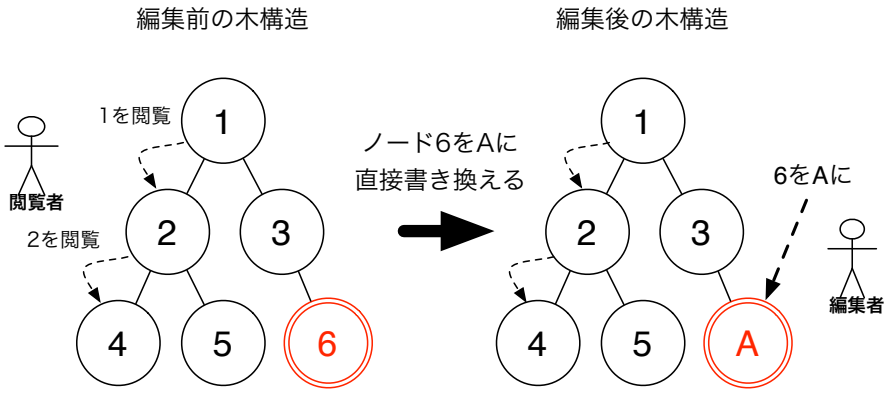

図 3.2: 競合状態に陥る木構造の破壊的編集

この状態を回避するためには、木構造にアクセスする際に排他制御を行う。その場合、木構造に 1 つのスレッドしかアクセスできないため並列度が下がりスケーラビリティを損なってしまう。破壊的木構造では、スケーラビリティのあるデータベースの開発はできない。

3.2.2　非破壊的木構造

非破壊的木構造は、木構造を書き換えることなく編集を行うため、読み書きを並列に行うことが可能である。図 3.3 では、ノード 6 をノード A へ書き換える処理を行なっている。

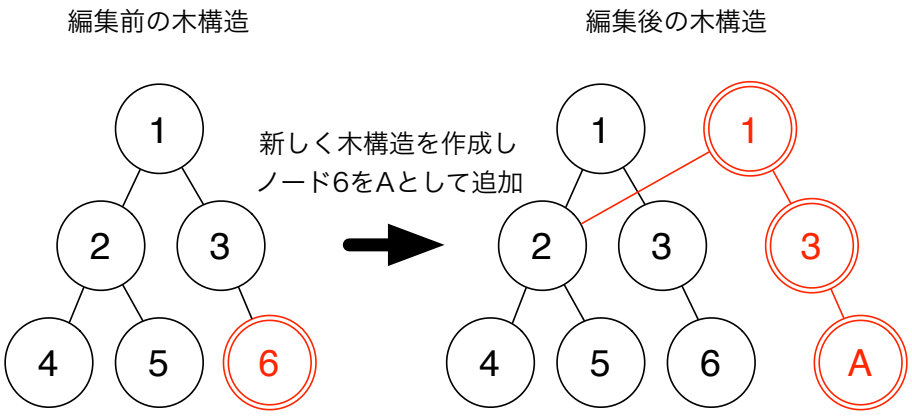

図 3.3: 木構造の非破壊的編集

非破壊的木構造の基本的な戦略は、変更したいノードへのルートノードからのパスを全てコピーする。そして、パス上に存在しない (編集に関係のない) ノードはコピー元の木構造と共有することである。

編集は以下の手順で行われる。

1. 変更したいノードまでのパスを求める。
2. 変更したいノードをコピーし、コピーしたノードの内容を変更する。
3. 求めたパス上に存在するノードをルートノードに向かって、コピーする。 コピーしたノードに一つ前にコピーしたノードを子供として追加する。
4. 影響のないノードをコピー元の木構造と共有する。

この編集方法を用いた場合、閲覧者が木構造を参照してる間に、木の変更を行っても問題がない。閲覧者は木が変更されたとしても、保持しているルートノードから整合性を崩さずに参照が可能である。ロックをせずに並列に読み書きが可能であるため、スケーラブルなシステムに有用であると考えられる。元の木構造は破壊されることがないため、自由にコピーを作成しても構わない。したがってアクセスの負荷の分散も可能である。

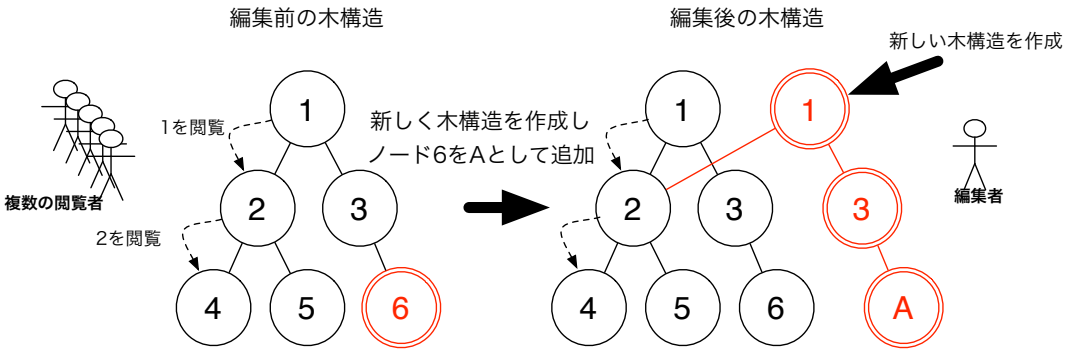

図 3.4: 並列に読み書きが可能な非破壊的木構造

3.3　Haskell の並列処理

純粋関数型言語 Haskell は並列処理に向いていると言われる。しかしながら、安直にそう言い切ることはできない。参照透過性があるため、各々の関数の処理は独立している。そのため、並列で走らせても問題ないように思われるが、Haskell は遅延評価を行うため問題が発生する。遅延評価では、結果が必要になるまで評価せず、同じ呼び出しがあった場合メモ化を行うことで最適化を行う。並列で実行する際は、遅延評価を行っていては並列度を高めることができない。また、メモ化は結果をキャッシュするため各スレッド間で同期するコストが発生する。Haskell においても並列化によるコストが実際の処理とは別に発生するため、並列化箇所をよく見極める必要がある。

Haskell では、様々な並列化手法が提供されている。もちろんスレッドを直接操作することも可能である。本研究では、抽象度の高い Eval モナドを利用した。Eval モナドを利用することで、並列処理のために必要な処理の流れを分かりやすく記述することができる。しかしながら実行順序を細かく制御することはできない。Par モナドを利用すれば、並列処理の流れを細かく記述できるが、Eval モナドのように処理と並列処理の流れを分けて記述し、後からプログラムに並列処理を組み込むというようなことはできない。

Haskell で並列処理を実装する場合は、どの並列化手法を採用するかということをよく考察する必要がある。

第4章　Haskellによる並列データベースの実装

本章では、並列データベースの実装について述べる。まず、実装した木構造データベースの利用方法について述べ、次に詳細な設計と実装について述べる。

4.1　木構造データベース Jungle

木構造データベース Jungle は、Haskell で実装された並列データベースである。非破壊的木構造の方法に則った API を提供する。本研究では、HTTP サーバ Warp と組み合わせて掲示板システムとして利用しているが、他のシステムに組み込むことも可能である。

木の作成

Jungle は複数の木構造を保持する事ができる。木構造は、名前を付けて管理する。名前を利用することで他の木構造と識別し、作成・編集を行う。

createJungle で、データベースを作成できる。木を作成するには、createTree を利用する。createTree には、createJungle で作成したデータベースと新しい木の名前を渡す。

```
jungle <- createJungle
createTree jungle "name␣of␣new␣tree␣here"
```

ソースコード 4.1: データベースと木の作成

ルートノード

Jungle は参照および編集に木構造のルートノードを活用する。ルートノードに関する API を説明する。

getRootNode は、最新のルートノードを取得できる。データベースと木の名前を渡すことで利用する。

```
node <- getRootNode jungle "your␣tree␣name␣here"
```

ソースコード 4.2: 最新のルートノードの取得

木構造を編集する API は全て Node を受け取って Node を返す。その返ってきた Node をルートノードとして新たに登録することで木構造が更新される。updateRootNode は、データベースと木の名前、変更して返ってきた木構造を渡す。updateRootNode をした後は、getRootNode で取得できるルートノードが更新された状態になっている。

```
updateRootNode jungle "your␣tree␣name␣here" node
```

ソースコード 4.3: ルートノードの更新

updateRootNodeWith は、ノードを更新する関数とデータベース、木の名前を渡して利用する。ノードを更新する関数とは、ノードを受け取ってノードを返す関数である。この updateRootNodeWith を利用することで、getRootNode をした後、編集し updateRootNode を行う一連の操作が atomically に行われることが保証される。

```
updateRootNodeWith func jungle "your␣tree␣name␣here"
```

ソースコード 4.4: 関数を渡してルートノードの更新

木の編集

木の編集には、Node を使う。木の編集に用いる API は全て Node を受け取って Node を返す。非破壊的木構造を利用しているため、getRootNode などで取得してきた Node は他のスレッドと干渉することなく自由に参照、編集できる。これらの編集のための API は、編集後 updateRootNode するか、ひとつの関数にまとめて updateRootNodeWith をすることで木構造に反映させることができる。

編集対象のノードを指定するには、NodePath を利用する。NodePath は、ルートノードからスタートし、ノードの子どもの場所を次々に指定したものである。Haskell の基本データ構造であるリストを利用している。

addNewChildAt で、ノードに新しい子を追加できる。addNewChildAt には、Node と NodePath を渡す。子には Position という場所の情報があるが、インクリメントしながら自動的に指定される。

```
new_node = addNewChildAt node [1,2]
```

ソースコード 4.5: 子の追加

deleteChildAt で、ノードの子を削除できる。deleteChildAt には、Node と NodePath、削除したい子の Position を指定する。

```
new_node = deleteChildAt node [1,2] 0
```

ソースコード 4.6: 子の削除

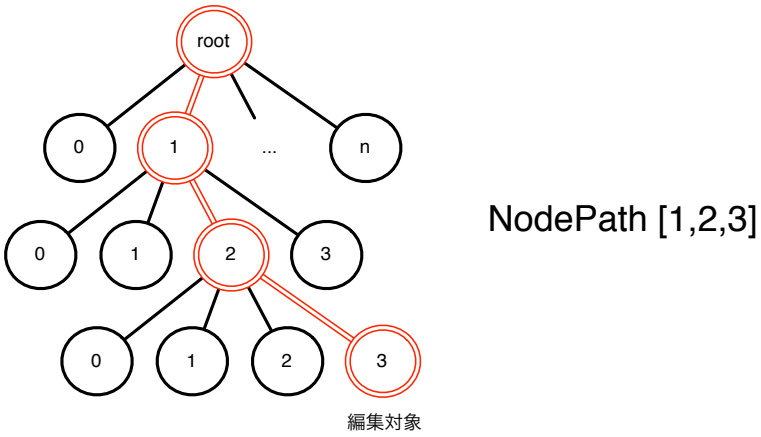

図 4.1: NodePath

putAttribute で、ノードに属性を追加できる。putAttribute には、Node と NodePath、Key、Value を渡す。Key は String、 Value は、ByteString である。

```
new_node = putAttribute node [1,2] "key" "value"
```

ソースコード 4.7: 属性の追加

deleteAttribute で、ノードの属性を削除できる。deleteAttribute には、Node と NodePath、Key を渡す。

```
new_node = deleteAttribute node [1,2] "key"
```

ソースコード 4.8: 属性の削除

木の参照

木の参照にも Node を用いる。様々な参照の API があるため、ひとつずつ紹介していく。

getAttributes は、対象の Path に存在する属性を Key を用いて参照できる。

```
bytestring = getAttributes node [1,2] "key"
```

ソースコード 4.9: 属性の取得

ある Node に存在する全ての子に対して、参照を行いたい場合に利用する。getChildren は、対象の Node が持つ全ての子を Node のリストとして返す

```
nodelist = getChildren node [1,2]
```

ソースコード 4.10: 対象の Node の全ての子を取得

ある Node に存在する全ての子に対して、参照を行いたい場合に利用する。getChildren と違い、子の Position も取得できる。assocsChildren は、対象の Node が持つ全ての子を Position とのタプルにし、そのタプルのリストを返す。

```
nodelist = assocsChildren node [1,2]
```

ソースコード 4.11: 対象の Node の全ての子と Position を取得

ある Node に存在する全ての属性に対して、参照を行いたい場合に利用する。assocsAttribute は、対象の Node が持つ全ての属性を、Key と Value のタプルとし、そのタプルのリストを返す。

```
attrlist = assocsAttribute node [1,2]
```

ソースコード 4.12: 対象の Node の全ての Attribute の Key と Value を取得

numOfChild では、対象の Node が持つ子どもの数を取得できる。

```
num = numOfChild node [1,2]
```

ソースコード 4.13: 対象の Node の子どもの数を取得

currentChild では、対象の Node が持つ最新の子を取得できる。

```
node = currentChild node [1,2]
```

ソースコード 4.14: 対象の Node の最新の子を取得

4.2　木構造データベース Jungle の実装

開発環境

実装には、Haskell を用いる。

全体の構造

木構造の集まりを表現する Jungle、単体の木構造を表現する Node から構成される。Jungle は複数の Node の集まりである。Jungle を利用して最新のルートノードを取得することができる。Node は子と属性を任意の数持てる。

表 4.1: 全体の構造

名前	概要
Jungle	木の作成・取得を担当する。
Node	基本的なデータ構造、子と属性を任意の数持てる。
RootNode	木構造のルートを表す。Jungle から最新のルートノードを取得できる。
children	子となるノードを任意の数持つことができる。
attributes	Key と Value の組み合わせを任意の数持つことができる。

4.2.1　Jungle

Jungle は木構造の集まりを表現する。木には名前がついており、ルートノードの情報も一緒に保持している。

木の取り扱い

Jungle の木の取り扱いには、Haskell の Data.Map を利用している。Haskell で連想配列を扱いたい場合、平衡木によって実装された Data.Map を一般的に用いる。Haskell のライブラリには配列や、ハッシュ・テーブルといったものも存在するがあまり使われない。配列は参照に適しているが、データを追加する際に配列を再作成するためコストが大きい。ハッシュ・テーブルは更新操作に副作用を伴うため、IO モナドの中でしか使うことが出来ず、扱いにくい。Data.Map は、挿入や参照が O(log n) で済む。

また、木の取り扱いには Haskell のソフトウェア・トランザクショナル・メモリ (STM) を利用して状態を持たせ、スレッド間で共有できるようにしてある。これは、木構造の各スレッドから作成できるようにするためである。STM は、スレッド間でデータを共有するためのツールである。STM を利用することでロック忘れによる競合状態や、デッドロックといった問題から解放される。STM は、アクションのブロックを atomically コンビネータを使ってトランザクションとして実行する。いったんブロック内に入るとそこから出るまでは、そのブロック内の変更は他のスレッドから見ることはできない。こちら側のスレッドからも他のスレッドによる変更はみることはできず、実行は完全に孤立して行われる。トランザクションから出る時に、以下のことが 1 つだけ起こる。

- 同じデータを平行して変更したスレッドが他になければ、加えた変更が他のスレッドから見えるようになる。
- そうでなければ、変更を実際に実行せずに破棄し、アクションのブロックを再度実行する。

STM は簡単に使え、また同時に安全である。

4.2.2　Node

Node は木構造を表現するデータ構造である。再帰的に定義されている。各ノードは Children としてノードを複数持つことができる（図 4.2)。

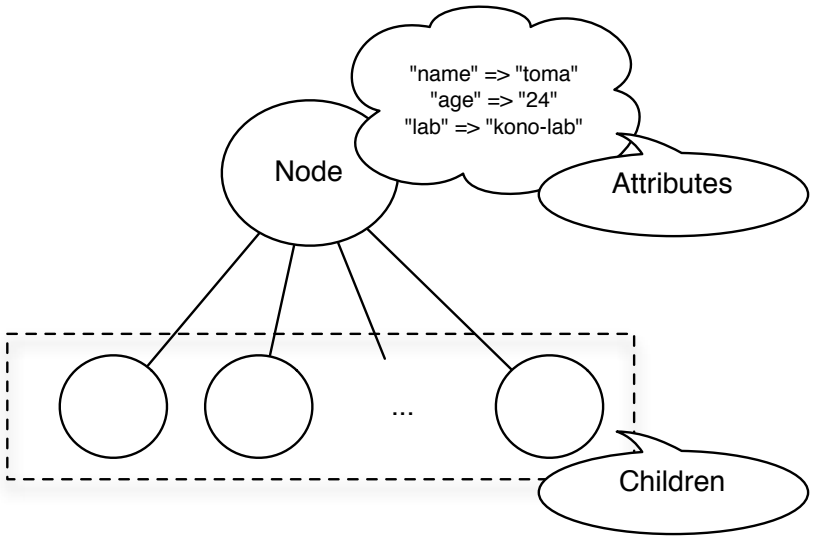

図 4.2: Node の構成要素

Children および Attributes も Data.Map を用いて定義されている（ソースコード 4.15)。

```
data Node = Node
          { children   :: (Map Int Node)
          , attributes :: (Map String ByteString) }
```

ソースコード 4.15: Node のデータ型の定義

ルートノード

非破壊的木構造ではノードは破壊されない。そのため、どのノードが最新のルートノードなのかという情報が必要である。この情報もスレッドセーフに取り扱う必要があるため、Haskell の ソフトウェア・トランザクショナル・メモリ (STM) を用いて管理している。

4.3　並列実行

木構造データベース Jungle は、並列に実行することができる。アプリケーション側で、データベースを参照や変更する際に各スレッドから呼び出しても問題ない。利用方法も、シングルスレッドで実行する場合と同じである。

4.4　Haskell の生産性

Java を用いた Jungle の実装と比較して、コード行数が約 3000 行から約 300 行へと短くなった。

これは Haskell の表現力が高いためである。Haskell では、データ型を簡単に作成することができる。再帰的なデータ構造の定義も容易である。共通の性質を扱うための型クラスという仕組みが存在し、既存のライブラリを作成したデータ型に利用できる。また、Haskell は参照透過性を持つため、コードの再利用が行い易く、関数同士の結合も簡単である。

同じような機能を実装する場合でも、Java と比較してコード行数が短くなり生産性が向上する。

第5章　ベンチマーク

前章では木構造データベース Jungle の利用方法及び詳細な実装について述べた。本章では、まず始めに木構造データベース Jungle の読み込みと書き込みの性能の計測を行う。次に簡単な掲示板システムを作成し、既存の Java の並列データベースの実装との性能比較を行う。また、Web アプリケーションの性能計測の際の諸注意について述べる。

5.1　読み込み

木構造データベース Jungle の並列読み込みの実験を行う。

実験方法

12 コア CPU のマシン上で、プログラムを実行するコア数を変更しながら並列に読み込む実験を行う。木構造データベース Jungle を利用し、約 90 万の子を持つ木構造を作成する。その木構造を読み込むための Task を 1000 個作成し、並列に実行する。

実験環境

実験を行うサーバの仕様について表5.1 に示す。Haskell のコンパイラには、The Glasgow Haskell Compiler (GHC) の 7.63 を使用する。

表 5.1: 実験を行うサーバの仕様

名前	概要
CPU	Intel(R) Xeon(R) CPU X5650@2.67GHz * 2
物理コア数	12
論理コア数	24
Memory	126GB
OS	Fedora 14
GHC	7.63

5.1.1　実験結果

並列読み込みの実験結果を表 5.2 に示す。12 CPU で動かした時に、1 CPU と比べて 10.72 倍の速度向上が見られる。また、Intel のハイパースレッディング機能を利用し 24 CPU で動かした際にも僅かながら性能が向上している。

表 5.2: 並列読み込みの実験結果

CPU 数	実行時間
1 コア	59.72 s
2 コア	33.77 s
4 コア	15.95 s
6 コア	10.72 s
8 コア	8.10 s
10 コア	6.56 s
12 コア	5.57 s
24 コア	4.80 s

第6章　結論

6.1　まとめ

まとめ

6.2　今後の課題

今後の課題

謝辞

ありがとうございました。

参考文献

[1] 玉城将士, 河野真治. Cassandra を使った cms の pc クラスタを使ったスケーラビリティの検証. 日本ソフトウェア科学会, August 2010.

[2] 玉城将士, 河野真治. Cassandra を使ったスケーラビリティのある cms の設計. 情報処理学会, March 2011.

[3] 玉城将士, 河野真治. Cassandra と非破壊的構造を用いた cms のスケーラビリティ検証環境の構築. 日本ソフトウェア科学会, August 2011.

[4] Avinash Lakshman and Prashant Malik. Cassandra - a decentralized structured storage system. *LADIS*, March 2003.

[5] Fay Changand Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable : A distributed storege system for structured data.

[6] Nancy Lynch and Seth Gilbert. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 2002.

[7] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly avaliable key-value store.

[8] The warp package. `http://hackage.haskell.org/package/warp`. [Online; accessed 19-July-2013].

[9] Deos. `http://www.dependable-os.net/osddeos/data.html`. [Online; accessed 19-July-2013].

発表履歴

- Haskell による非破壊的木構造を用いた CMS の実装,
  當眞大千, 河野真治 (琉球大学), 永山辰巳 (株式会社 Symphony)
  日本ソフトウェア科学会 30 回大会 (2013 年度), Sep, 2013