Liquid Haskell 上での並列プログラムの時間計算量解析

大喜 優，西崎真也

本研究では，純粋関数型言語 Haskell での並列プログラミングに着目して，静的な時間計算量の解析を試みる．Liquid Haskell は，Haskell に篩型を追加して，SMT ソルバによる検証を行うプログラム検証器である．本研究では，Liquid Haskell を用いた既存の逐次時間計算量解析手法を，並列プログラムへと拡張する．具体的には，時間計算量を解析する 2 つの関数を実装した．1 つは，2 要素のタプルをプロセッサ数に応じて逐次または並列に実行する関数である．もう 1 つは，リストの各要素を並列に実行する関数である．提案した実装を，いくつかの例に適用することで，時間計算量の解析が適切に行われていることを確認した．また，実際に並列プログラムを実行して経過時間の計測を行い，解析した時間計算量との比較を行った．

1 はじめに

1.1 並列プログラムと時間計算量

近年，シングルコアプロセッサの性能が頭打ちを迎えたことで，マルチコアプロセッサを用いた並列計算による高速化が重要性を増してきている．しかし，並列プログラムを意図した性能となるよう記述することは，容易ではない．適切な並列化を行えないと，逐次プログラムと実行時間がさほど変わらなかったり，むしろより遅くなってしまう場合もある．

実行時間の測定は，並列プログラムの場合においても，効率良く動作していることの確認に有効な手段である．しかし，並列化の実行時間は，入力に加えて実行する環境（プロセッサの個数など）によっても変わってくるため，あらゆる場合に対して想定通りに動くことを保証するのは困難である．また，実行時間の測定は，ソースコードの一部箇所を変えただけで測り直しとなってしまう場合もある．時間計算量は，プログラムの性能を見積もる上で重要な指標であり，並列プログラムにおいても同様である．プロセッサ数を変数とする時間計算量により，プロセッサ数を増加させることでどのように実際のステップ数を減らすことができるかを判断することができる．

1.2 Haskell と並列プログラミング

Haskell [2, 9, 12] は，純粋関数型のプログラミング言語である．以下 Haskell に関して述べる際は，最も一般的に用いられているコンパイラである Glasgow Haskell Compiler（以下 GHC）[1] を用いているとする．

Haskell [10, 11, 13, 16] は，並列プログラミングに対応した言語でもある．Haskell で記述した並列プログラムから得られる結果は決定的であり，競合条件やデッドロックを引き起こすことはない．また，同期や通信という概念は暗黙のうちに扱われる．プログラマは並列化を期待する場所をソースコード中に示し，実行時システムによって並列化するかの判断が行われる．

1.2.1 型クラス

型クラスは，多相型において，ある特定の計算を行える型を指定するための概念である．ある型クラスの指定した計算，すなわち関数を実装した型は，その型クラスのインスタンスとなることができる．多相型を引数に取る関数において，型変数が型クラスのインスタンスであることを要求することで，受け取る型が特定の計算を行えることを強制できる．

例えば，型クラス Num のインスタンスとなる型は，(+)や(−)，(∗)などといった数値なら行えるべき演算を扱える型である．具体的には，Int，Float，Integer などが該当する．

また，図 1 関数 sum において，型クラス Num を満たす型 a のリスト [a] を引数として指定すると，リストの値を全て(+)で足し合わせた型 a の値を返す関数を実装することができる．

Time complexity analysis of parallel programs on Liquid Haskell.

Yu Daiki, 東京工業大学情報理工学院情報工学系, Department of Computer Science, Tokyo Institute of Technology.

Shin-ya Nishizaki, 東京工業大学 学術国際情報センター, Global Scientific Information and Computing Center, Tokyo Institute of Technology.

```
sum :: Num a => [a] -> a
sum [] = 0
sum (x:xs) = x + sum xs
```

図 1　関数 sum

```
return :: a -> m a

(>>=) :: forall a b. m a -> (a -> m b) -> m b
```

図 2　関数 return と(>>=)の型

```
data  Maybe a  =  Nothing | Just a
```

図 3　Maybe モナド

1.2.2　モナド

モナド [15] とは，計算の状態や入出力等を表現するための汎用的な構造である．あるモナドは，計算の対象であるデータ型と，計算値の生成関数，計算を合成する関数から構成される．

Haskell では，モナドは return 関数（生成）と(>>=)関数（合成）を持つ型クラスとして実装される．対象のデータ型にこれらの関数を定義し，モナドのインスタンスとすることで，モナドとしてデータ型に計算を行うことができる．図 2 に示すのが，それぞれの型である．

return 関数は，値からモナドの値を生成する関数である．(>>=)関数は，モナドに包んだ型 a の値と，型 a の値を取ってモナドに包んだ型 b の値を返すような関数を受け取り，それらを合成してモナドに包んだ型 b の値を返す関数である．

モナドの例としては，Maybe モナドや IO モナドが挙げられる．

Maybe モナドは，値に，存在しない場合を加えて計算結果として保持するモナドである．Maybe モナドを返すことで，計算結果の値か，その計算が失敗したという情報のいずれかを返す関数を表現できる．(>>=)関数を用いて，失敗するかもしれない計算を連続で行い，全て成功した場合のみ最終的に Just の値を持つ，などの処理を書くことができる．

IO モナドは，副作用を保持するモナドである．Haskell の関数は本来純粋であるが，IO モナドを用いることで，入出力のような純粋でない処理を表現することができる．

1.3　Liquid Haskell における時間計算量解析

1.3.1　篩型

篩型 (Refinement Types) [17] とは，特定の述語を満たすような値を分類する型のことである．次の例は，0 でないという述語を満たす整数の篩型である．

$$\{v \in \mathbb{Z} \mid v \neq 0\}$$

```
(declare-const x Int)
(declare-const y Int)
(assert (= (+ x y) 10))
(assert (= (+ x (* 2 y)) 20))
(check-sat)
(get-model)
```

図 4　SMT ソルバで連立方程式を解く入力

```
{-@ safeDiv :: Int -> { v:Int | v /= 0 } -> Int @-}
safeDiv x y = x `div` y

main = print $ safeDiv 3 0
```

図 5　安全な div 関数の例

1.3.2　SMT ソルバ

SMT(Satisfiability Modulo Theories) [4,21] とは，ある理論に関する一階述語論理式の充足可能性を決定する問題である．SMT ソルバとは，SMT 問題を解くようなソルバのことを言う．SMT ソルバは，命題論理の部分は SAT ソルバ，整数については整数の理論に関するソルバが処理するように作られている．代表的な SMT ソルバとしては，z3 [5] が挙げられる．

例えば $x+y=10$ と $x+2y=20$ の 2 つの等式からなる整数変数 x,y の連立方程式は，SMT ソルバに図 4 のような入力を与えることで解くことができる．

1.3.3　Liquid Haskell

Liquid Haskell [3] は，Haskell に篩型を追加し，SMT ソルバによる静的な検証を可能にした拡張である．Liquid Haskell の検査を通過したプログラムは，GHC でもコンパイル可能であることを保証されている．

Liquid Haskell の検査では，まず篩型から部分型の関係を取り出す．次に部分型の関係を論理式に変換し，その論理式が真であるかを SMT ソルバによって確かめる．

Liquid Haskell では，Haskell のソースコード中にコメントとして篩型を記述する．例えば，0 除算を発生させない安全な除算関数 safeDiv は，図 5 のように定義し，利用できる．

Liquid Haskell は，第二引数が 0 でないという篩型の制約から，このプログラムが安全でない（unsafe である）ことを報告する．

Liquid Haskell では，関数の引数の型に述語を加えることで，関数の事前条件を表現できる．また，関数の返り値の型に述語を加えることで，関数の事後条件を表現

```
{-@ data Tick a = Tick { tcost :: Int, tval :: a } @-}
data Tick a = Tick { tcost :: Int, tval :: a }
```

図 6　Tick モナドのデータ型

```
{-@ reflect return @-}
{-@ return :: x:a
    -> { t:Tick a | x == tval t && 0 == tcost t } @-}
return :: a -> Tick a
return x = Tick 0 x

{-@ reflect >>= @-}
{-@ (>>=) :: t1:Tick a -> f:(a -> Tick b)
      -> { t:Tick b |
         tval (f (tval t1))  == tval  t &&
         tcost t1 + tcost (f (tval t1)) == tcost t }
@-}
infixl 4 >>=
(>>=) :: Tick a -> (a -> Tick b) -> Tick b
Tick m x >>= f = let Tick n y = f x in Tick (m + n) y
```

図 7　Tick モナドの return 関数，(>>=) 関数

できる．さらに，データ構造の型に述語を加えることで，そのデータ構造で常に成り立つ不変条件を表現できる．

このように，Liquid Haskell で検査されたプログラムは，指定した事前条件や事後条件，不変条件や満たしていることが保証される．一方で，SMT ソルバの推論の限界から，篩型の条件を満たしていても推論されずに安全でないと判定される場合もある．

1.3.4　Liquid Haskell における時間計算量解析

論文 [6] で Handley らは，Liquid Haskell を用いて時間計算量を静的に解析する手法を提案した．この手法では，リソース使用量を記録するデータ型として，Tick モナドを定義する．

Tick モナドは，計算結果の値とともに，動作ステップ数も保持するするようなモナドである．Tick モナドでは，tcost が動作ステップ数，tval が計算結果の値を表す．操作を行う場合に動作ステップ数を表す tcost の値も適切に加算することで，操作に必要な動作ステップ数を記録することができる．

Tick モナドを構成する return 関数と (>>=) 関数の定義は，図 7 のようになる．

return 関数は，値を取って tcost が 0 の Tick モナドを返す．

(>>=) 関数は，第一引数の tcost である m と，第二引数 f に第一引数の値 x を適用した結果の tcost である n を加算した結果の (m+n) を，返される Tick モナドの tcost とする．

Liquid Haskell を用いた時間計算量解析には，2 つの

```
{-@ eqBind :: n:Int -> t1:Tick a
      -> f:(a -> { tf:Tick b | n == tcost tf })
      -> { t:Tick b |
            tval (f (tval t1))  == tval  t &&
            tcost t1 + n == tcost t }
@-}
eqBind :: Int -> Tick a -> (a -> Tick b) -> Tick b
```

図 8　eqBind 関数

```
{-@ (++) :: xs:[a] -> ys:[a]
  -> { t:Tick {
            zs:[a] | length zs == length xs + length ys
         } | tcost t == length xs }
@-}
```

図 9　(++) 関数

手法がある．Liquid Haskell を用いた自動証明と，Proof Combinators を用いた外部での手動の定理証明である．自動証明では，手動で証明をする必要がないという利点がある．一方で，自動証明では型レベルの計算が必要なことから，(>>=) 関数を用いただけでは推論できないという問題がある．そこで，tcost (f (tval t1)) を引数の整数に置き換えた，eqBind 関数（図 8）を用いて推論を行う．

tcost は動作ステップ数以外のリソース使用量，例えば Thunk の個数などを記録することもできる．

1.4　研究目的

本研究では，Handley らが提唱した Liquid Haskell を用いた時間計算量解析を，並列プログラムに拡張した実装を提案する．提案した実装を用いて，実際にこれを適用したいくつかの並列プログラムを紹介し，その時間計算量を解析する．また，実際に並列プログラムを実行して経過時間の計測を行い，解析したステップ数との比較を行う．

1.5　関連研究

白水ら [20] は，定理証明システム Coq を用いた計算量解析手法 [14] の並列プログラムへの拡張手法を提案した．図 10 のように定義されるモナド C は，値の型 A と，型 A の値とプロセッサ数から成り立つ計算量の性質を表す命題を返す述語 P からなる．並列に拡張するにあたって，並列度を表すプロセッサ数の 2 の対数の引数，並列呼び出しを指定する書式，並列度に応じた計算量算出を追加している．

計算量に関する操作を型の内部に記述することで，Coq から OCaml に計算量の性質を保ってコード抽出を行える．一方で，Coq での証明は，手動で行う必要がある．

```
Definition C (A:Set) (P:A -> nat -> Prop) : Set :=
  {a : A | exists (an:nat), (P a an)}.
```

図 10　モナド C の定義

Hoffmann ら [7] は，並列プログラムのコストを静的に自動解析する手法を提案した．全体の計算時間の合計である work と最も長い計算の時間である depth から並列プログラムのコストを評価できることに着目している．また，提案した解析手法を Resource Aware ML(RaML) に実装することで，その実用性を確認している．

2　並列プログラムの時間計算量解析

本章では，まず並列プログラムの計算機モデルを考える．次に，本研究で実装したライブラリ関数を紹介する．その後，時間計算量の解析対象となる具体的な並列処理を，実装した関数を用いて実装する．

2.1　PRAM モデル

時間計算量や空間計算量について考える場合は，前提となる計算機モデルを考える必要がある．逐次プログラムでは，チューリング機械やランダムアクセス機械（random access machine，以下 RAM と略す）が採用されることが多い．

並列ランダムアクセス機械（parallel random access machine, 以下 PRAM と略す）[18] [8] [19] は，並列計算のモデルとして最もよく用いられているものの 1 つである．PRAM は，局所メモリを持つ RAM であるプロセッサの列と，全てのプロセッサがアクセスできる共有メモリから構成される．PRAM では，各プロセッサは各命令（ステップ）を 1 単位時間で実行し，命令実行サイクルは同期しているものとする．PRAM が停止したとは，全てのプロセッサが停止したことを言う．

並列プログラムの時間計算量は，PRAM が停止するまでに経過した時間で定義される．各ステップは 1 単位時間で実行されることから，時間計算量は全体を 1 つのプロセッサとして見た場合のステップ数としても解釈できる．

一方で，並列プログラムの仕事量は，あるプログラムで各プロセッサが実行する命令数の合計である．仕事量と時間の関係を work-time パラダイムと呼ぶ．

以降，ある入力 n を取る並列アルゴリズムに関して，プロセッサ数を p としたときステップ数が $T(p,n)$，仕事量が $W(n)$ として表記する．

一般に $W(n) \leq pT(p,n)$ が成り立つ．等号は，全てのプロセッサが PRAM が停止するまで命令を実行した場合である．$p = 1$ の場合，すなわち逐次プログラムとして実行した場合は $W(n) = T(1,n)$ である．

```
{-@ evalPair :: t12:(Tick a, Tick b) -> lbp:Nat
  -> { t:Tick (a, b) |
    tcost t ==  if lbp == 0
                then (tcost (fst t12) + tcost (snd t12))
                else tcost (fst t12) &&&
    tval t == (tval (fst t12), tval (snd t12))}
@-}
evalPair :: (NFData a, NFData b) =>
    (Tick a, Tick b) -> Int -> Tick (a, b)
```

図 11　evalPair 関数

2.2　並列処理関数の紹介

本研究では Handley ら [6] の手法を拡張したため，Tick モナドの定義も同一である．tval は型の値であり，本研究では tcost が先ほど定義した並列プログラムの時間計算量を示す．ステップ数と時間計算量は同じものを指すが，加算処理を行う場合はステップ数として解釈した方が自然なため，主に途中過程ではステップ数として扱う．

本研究では，大きく分けて 2 つのライブラリ関数を設計・実装した．ここでは簡単な紹介に留め，詳細に関しては実装の章で説明する．

2.2.1　evalPair 関数

evalPair 関数は，Tick 型 2 要素から構成されるタプル (Tick a, Tick b) とプロセッサ数に関する情報を受け取り，タプルを値とする Tick 型である Tick (a, b) を返すような関数である．

この際，引数のタプルには存在する 2 つの Tick は，それぞれが tcost を持つ．これらは，Tick の値を単体で実行（評価）した場合のステップ数を示している．

evalPair 関数では，プロセッサ数の 2 の対数の値を引数で渡すことで，プロセッサ数が 1 ならば逐次計算，1 より多いならば並列計算が行われる．

逐次計算では，図 12 のように 1 つ目の要素，2 つ目の要素と同じプロセッサで順に評価される．そのため，2 つの要素のステップ数である tcost が加算される．

一方，並列計算では，図 13 のように並列に評価を行う．つまり，空いている別のプロセッサに 1 つ目の処理を任せ，2 つ目の処理を元のプロセッサで行う．両方の処理が終わってから，次の処理が行われる．並列に評価して同期する場合，全体としてのステップ数は 1 つ目と 2 つ目のうちステップ数が多い方を取る．(ただし，ここでは 1 つ目の要素の方が tcost が大きくなるように渡すことを仮定している．詳細は実装の章にて)

また，補助関数として (>>=/) 関数（図 14）を定義しておく．(>>=/) 関数は，次のように定義される関数である．引数は(>>=)と同一だが，篩型における制約が異な

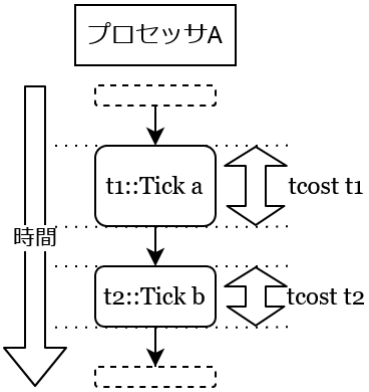

図 12　ペアの逐次計算

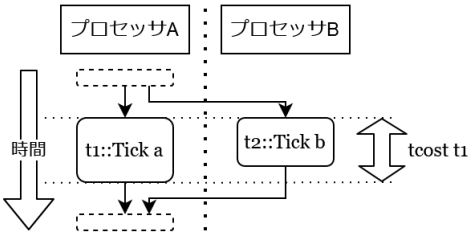

図 13　ペアの並列計算

```
{-@ (>>=/) :: t1:Tick a
      -> f:(a -> { tf:Tick b | tcost tf == 1 })
      -> { t:Tick b |
            tval (f (tval t1))  == tval t &&
            tcost t1 + 1 == tcost t }
@-}
infixl 4 >>=/
(>>=/) :: Tick a -> (a -> Tick b) -> Tick b
```

図 14　(>>=/) 関数

```
{-@ parList :: c:Nat
  -> { xs:[{ t1:Tick a | tcost t1 <= c }] | xs /= [] }
  -> { p:Nat | p >= length xs }
  -> { t:Tick {
       os:[a] | length xs == length os
    } | tcost t == c }
@-}
parList :: (NFData a) =>
  Int -> [Tick a] -> Int -> Tick [a]
```

図 15　parList 関数

る．(>>=/) 関数では，引数 f の返り値の Tick の tcost が 1 であることを制約とする．これにより，tcost (f (tval t1)) の推論を 1 で置き換えることができ，SMT ソルバでの推論が可能である．

2.2.2　parList 関数

parList 関数は，Tick 型で構成されるリスト [Tick a]

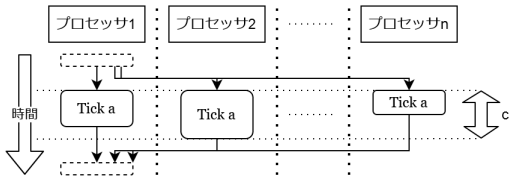

図 16　リストの並列計算

```
{-@ chunk :: { n:Nat | n /= 0 } -> { m:Nat | m /= 0 }
  -> { xs:[a] | length xs == n * m }
  -> { yss:[ { ys:[a] | length ys == n } ] |
    length yss == m }
  / [m]
@-}
chunk :: Int -> Int -> [a] -> [[a]]
```

図 17　chunk 関数

```
{-@ mapParList :: { n:Nat | n > 0 }
  -> { xs:[a] | length xs == n }
  -> c:Nat
  -> (a -> { t:Tick b | tcost t <= c })
  -> { t:Tick { os:[b] | length xs == length os } |
    tcost t == c }
@-}
mapParList :: (NFData b) =>
  Int -> [a] -> Int -> (a -> Tick b) -> Tick [b]
```

図 18　mapParList 関数

とコスト，プロセッサ数の情報を受け取り，リストを値とする Tick 型である Tick [a] を返すような関数である．

parList 関数では，図 16 のように配列を構成する各要素の処理を並列に評価する．

parList 関数の引数として受け取る Tick 型のリストは，引数で受け取る cost 以下の tcost であることを篩型の制約としている．また，引数で受け取ったプロセッサ数は配列の長さ以上であることを篩型の制約としている．よって受け取ったプロセッサを使って配列の各要素を並列に評価することができる．そのため，全体の tcost も引数として受け取った cost とすることができる．

parList 関数を使用する際の補助として，chunk 関数や mapParList 関数を実装した．それらの簡単な紹介も行う．

chunk 関数は，リストの分割数を受け取り，受け取ったリストを分割してリストのリストとして返すような関数である．

mapParList 関数は，リストに対して渡した関数を並列に実行する関数である．先ほどの parList 関数を内部で呼び出し，並列化を行う．

以降は，これらの関数を用いて並列プログラムの時間計算量解析を行う．

```
{-@ fib :: n:Nat -> { v:Nat | v >= n } @-}
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n -1) + fib (n -2)
```

図 19　fib 関数

```
{-@ fibPar :: n:Nat -> lbp:Nat
  -> { t:Tick Int | (tcost t == fibCost n lbp) } @-}
fibPar :: Int -> Int -> Tick Int
fibPar 0 _ = wait 1
fibPar 1 _ = wait 1
fibPar n 0 =
  ((x, y) `evalPair` 0)
    >>=/ (\(a, b) -> wait (a + b))
  where
    x = fibPar (n -1) 0
    y = fibPar (n -2) 0
fibPar n lbp =
  ((x, y) `evalPair` lbp)
    >>=/ (\(a, b) -> wait (a + b))
  where
    x = fibPar (n -1) (lbp -1)
    y = fibPar (n -2) (lbp -1)
```

図 20　fibPar 関数

2.3　並列フィボナッチ関数

2.3.1　フィボナッチ関数

フィボナッチ数列とは，以下の漸化式で定義される数列である．

$$F_0 = 1$$
$$F_1 = 1$$
$$F_{n+2} = F_{n+1} + F_n \ (n \geq 0)$$

まず，入力 n を受け取り，F_n を返す関数 fib の実装を以下に提示する．

入力 n が 0 か 1 の場合は，1 を返す．

入力 $n > 1$ の場合，$fib(n-1)$ と $fib(n-2)$ の和を返す．

2.3.2　並列フィボナッチ関数

fib 関数（図 19）は，再帰的に fib 関数を呼び出す部分を並列に行うことで，容易に並列化が可能である．時間計算量を tcost として保持し，並列化を施した fibPar 関数について考える．

fibPar は，入力 n と，プロセッサ数の 2 の対数を表す lbp を受け取り，F_n を値とするような Tick 型を返す関数である．fibPar 関数の実装は図 33 のようになっている．

入力が 0 または 1 の場合，値は 1 である．wait 関数は，値を受け取り，tcost に 1 を割り当てた Tick を返す関数である．

入力 $n > 1$ の場合，値としては $fib(n-1)$ と $fib(n-2)$

```
{-@ fibCost :: n:Nat -> lbp:Nat -> Nat @-}
fibCost :: Int -> Int -> Int
fibCost 0 _ = 1
fibCost 1 _ = 1
fibCost n lbp
  | n <= lbp + 1 = n
  | otherwise = 2 * fib (n - lbp) -1 + lbp
```

図 21　fibCost 関数

を加算したい．そのため，x には $fib(n-1)$ を計算するように，y には $fib(n-2)$ を計算するように fibPar 関数を呼び出した結果を代入する．このとき，プロセッサ数が 2 以上（すなわち，プロセッサ数の 2 の対数が 1 以上）の場合は，プロセッサ数の 2 の対数から 1 を引いた値を fibPar 関数の引数として渡す．この時点では，x と y はまだ評価されていない．

その上で，タプル (x, y) とプロセッサ数の 2 の対数を evalPair 関数に渡す．evalPair 関数では，プロセッサに空きがある，すなわちプロセッサ数が 2 以上の場合（この処理を実行中のプロセッサに加えて 1 つある場合）は並列に処理が行われ，プロセッサ数が 1 の場合（実行中のプロセッサ以外に空いていない場合）は逐次処理が行われる．x と y の処理内容は fibPar 関数の呼び出しなので，再帰的に処理が繰り返される．

逐次処理でも並列処理でも，evalPair の結果は eqBind 関数の第 2 引数となる．eqBind 関数によって，ペアの和を取って wait 関数に渡して返す関数に x と y の値のペアが渡され，加算された結果の Tick 型が返ってくる．wait 関数では tcost として 1 が指定されるので，全体の tcost は，evalPair 関数の結果の Tick 型の tcost に，さらに 1 が加算された値となる．

ただし，Liquid Haskell の推論の仕様上，$lbp = 0$ の場合とそうでない場合とで明示的に場合分けが必要になってしまっている．

2.3.3　並列フィボナッチ関数の時間計算量

fibCost は fibPar の時間計算量を表す関数であり，図 21 のように定義される．

入力 n と lbp を用いて，fibCost が返す値を $T(n, lbp)$ として以下のように表せる．

$$\begin{aligned}
&T(0, lbp) = 1, \\
&T(1, lbp) = 1, \\
&T(n, lbp) = n, && (n \leq lbp + 1) \\
&T(n, lbp) = 2 \cdot fib(n - lbp) - 1 + lbp, \\
& && (\text{otherwise}).
\end{aligned}$$

したがって，十分大きな n において lbp が $lbp < n$ とな

る場合は，

$$T(n, lbp) = O\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n-lbp}\right).$$

となり，十分大きな n において lbp が，$n \leq lbp + 1$ となる場合は，

$$T(n, lbp) = O(n)$$

となる．

O 記法は，$f(n) = O(g(n))$ と表記したとき，ある正数 k と n_0 に対して任意の $n > n_0$ に対し，$|f(n)| \leq k \cdot g(n)$ が成り立つことを表す．すなわち，f は漸近的に g によって上から抑えられることを意味する．

上の結果からは，並列数が少なければ指数時間がかかるが，充分並列性が確保されたならば，線形時間で計算が終わることがわかる．一方で，lbp が n より大きい場合というのは，プロセッサ数の 2 の対数が n より大きい場合であり，およそ 2^n のプロセッサ数が必要であるので，現実的ではない．

それぞれの場合について，上の時間計算量が正しいことを簡単に説明する．

$n = 0$ または $n = 1$ の場合

wait 関数の tcost が 1 であるため，lbp の値に関わらずステップ数は 1 となる．

$1 < n$ かつ $n \leq lbp + 1$ の場合

$n \leq lbp + 1$ の場合とは，プロセッサ数が充分多く，並列化したいだけ並列化できる場合である．fibPar 関数は引数 n を受け取ると，fibPar 関数に $n-1$，$n-2$ を渡して並列に実行する．このうち，よりステップ数が多いのは $n-1$ を渡した場合である．ステップ数が多い方を追うと，最終的に $n = 1$ で停止する．n から 2 までは加算の際，1 では値を返す際に wait 関数により 1 つずつステップ数が増えるので，tcost の値 n と一致する．

一方，$1 < n$ かつ $lbp \leq n$ の場合に関しては，$lbp = 0$ の場合とそうでない場合とで場合分けして考える．

$1 < n$ かつ $lbp = 0$ の場合

この場合，プロセッサ数は 1 である．$n = 0$ と $n = 1$ の場合を除くと $n > 1$ であり，また $n \leq lbp + 1$ を満たさないので，4 つ目の場合となる．$lbp = 0$ を代入すると，$c = 2 \times fib(n) - 1$ となる．

この値になることの直感的な説明を行う．$lbp = 0$ かつ $n > 1$ の場合，$fib(n)$ の計算結果は，以下のように $fib(0)$ または $fib(1)$ の和となる．

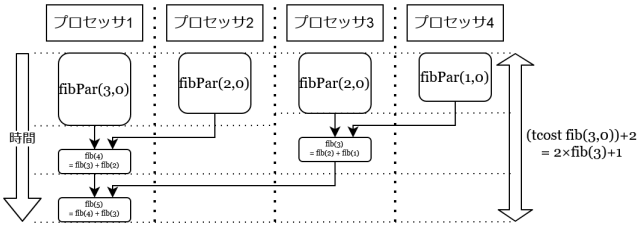

図 22　fib(5,2) の実行例

$$\begin{aligned}
fib(2) &= fib(1) + fib(0) = 2 \\
fib(3) &= fib(2) + fib(1) \\
&= fib(1) + fib(0) + fib(1) = 3 \\
fib(4) &= fib(3) + fib(2) \\
&= fib(2) + fib(1) + fib(1) + fib(0) + fib(1) \\
fib(n) &= \overbrace{fib(1) + \ldots + fib(1)}^{fib(n)}
\end{aligned}$$

$fib(0) = fib(1) = 1$ のため，$fib(n)$ を構成する $fib(0)$ と $fib(1)$ の和は $fib(n)$ となる．fibPar の計算では，$fib(0)$ または $fib(1)$ を返す際に tcost に 1 を指定して返す．また，$fib(n-1)$ と $fib(n-2)$ を加算する際にも tcost に 1 を追加する．これら以外で tcost が加算されることはない．したがって，値を返す際に wait 関数でステップ数 $fib(n)$，加算の際にステップ数 $(fib(n) - 1)$ が追加されるため，合計のステップ数は $2 \times fib(n) - 1$ である．

$1 < n$ かつ $0 < lbp \leq n$ の場合

この場合は，プロセッサが余っていて並列化できるところまでは並列化して，各プロセッサにタスクが行き渡ったら逐次処理を行う．

例として，図 33 は fibPar(5,2) を実行した場合の処理の流れを表している．fibPar(5,2) は，2^2 個のプロセッサ上で fib(5) の計算を行う．fibPar(5,2) では並列に fibPar(4,1) と fibPar(3,1) を呼び出す．fibPar(4,1) では fibPar(3,0) と fibPar(2,0) を並列に呼び出し，これら fib(3) と fib(2) の計算が各プロセッサで並列に行われる．fibPar(3,1) も同様で，fib(2) と fib(1) の計算をそれぞれのプロセッサで並列に行う．ステップ数が最も多いのは fib(3) であり，上述したように $2 \times fib(3) - 1$ である．よって，fibPar(4,1) のステップ数は加算処理でステップ数に 1 を加えた $2 \times fib(3)$ となる．したがって fibPar(5,2) のステップ数は加算処理でさらにステップ数に 1 を加えた $2 \times fib(3) + 1 = 2 \times fib(5-2) - 1 + 2$ となる．

以上が，並列 fib 関数の時間計算量の解析と，その直感的な説明である．

```
{-@ pow2 :: n:Nat -> { v:Nat | v >= n } @-}
pow2 :: Int -> Int
pow2 0 = 1
pow2 n = pow2 (n -1) + pow2 (n -1)
```

図 23　pow2 関数

```
{-@ pow2Par :: n:Nat -> lbp:Nat
  -> { t:Tick Int | (tcost t == pow2Cost n lbp) } @-}
pow2Par :: Int -> Int -> Tick Int
```

図 24　pow2Par 関数

```
{-@ reflect pow2Cost @-}
{-@ pow2Cost :: Nat -> lbp:Nat -> {v:Nat | v >= 1} @-}
pow2Cost :: Int -> Int -> Int
pow2Cost 0 _ = 1
pow2Cost n lbp
  | n <= lbp = n + 1
  | otherwise = pow2 (n + 1 - lbp) -1 + lbp
```

図 25　pow2Cost 関数

2.4　並列に 2 の冪乗を計算する関数

pow2 関数とは，与えられた n に対して 2^n を計算する関数のことを意味する．以下が，pow2 関数の実装の一例である．より効率のよい実装は存在するが，今回は並列化の紹介のため図 23 のような実装とする．

並列で pow2 を計算する pow2Par 関数も，fibPar 関数と同様に，再帰的に呼び出す処理を並列化することで図 24 のように定義できる．

pow2Cost 関数は pow2Par 関数の時間計算量を表す関数であり，図 25 のように定義される．

pow2Cost が返す値は，入力 n と lbp を用いて以下の式で表せる．

$$T(n, lbp) = \begin{cases} 1 & = O(1) \\ & (n = 0) \\ n + 1 & = O(n) \\ & (0 < n \wedge n <= lbp) \\ 2^{n+1-lbp} - 1 + lbp & = O\left(2^{n-lbp}\right) \\ & (\text{otherwise}) \end{cases} \tag{1}$$

これが pow2Par 関数の時間計算量として正しいことは，fibPar 関数と同様に考えることでわかる．

この結果は，論文 [20] における並列 pow2 関数の時間計算量解析の結果とも一致する．ただし，pow2 関数の計算の際には加算する 2 つの値が同じ式なので，GHC の最適化によってこの通りの時間計算量にはならないことに注意が必要である．

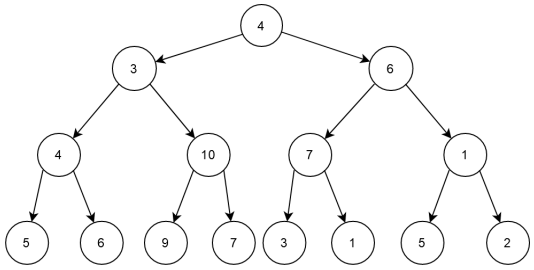

図 26　完全二分木の例

```
{-@ measure getHeight @-}
{-@ getHeight :: PBT a -> Nat @-}
getHeight :: PBT a -> Int
getHeight Leaf = 0
getHeight (Node _ h _ _) = h

{-@
  data PBT a
  =  Leaf
  |  Node
      { key :: a,
        h :: Nat,
        l :: { v:PBT a | getHeight v == h - 1 },
        r :: { v:PBT a | getHeight v == h - 1 }
      }
@-}
data PBT a
  = Leaf
  | Node
      { key :: a,
        h :: Int,
        l :: PBT a,
        r :: PBT a
      }
  deriving (Show)
```

図 27　完全二分探索木の定義

2.5　完全二分木の並列走査

ある木の高さとは，その木の根から葉までの経路のうち，最も長いもののことを言う．完全二分木とは，全てのノードの左右部分木の高さが等しいような二分木のことである．

Liquid Haskell では，図 27 のように完全二分木を定義できる．

PBT は，完全二分木を表すデータ型である．Haskell 上では単なる高さと key を保持した二分木となるが，篩型の制約によって完全二分木であることを保証している．getHeight 関数は，与えられた二分木の高さを得る関数である．左右の部分木の高さが両方とも二分木の高さ h より 1 小さい値であることを制約としているので，PBT は完全二分木となる．ただし，この定義では葉は値を持たないことに注意が必要である．

```
{-@ sumPBT :: { p:PBT Nat | getHeight p /= 0 }
  -> lbp:Nat
  -> { t:Tick Int |
    (tcost t == pow2Cost (getHeight p - 1) lbp) }
@-}
sumPBT :: PBT Int -> Int -> Tick Int
sumPBT (Node key 1 _ _) _ = wait key
sumPBT (Node key _ l r) 0 =
  ((x, y) `evalPair` 0)
    >>=/ (\(a, b) -> wait (a + b + key))
  where
    x = sumPBT l 0
    y = sumPBT r 0
sumPBT (Node key _ l r) lbp =
  ((x, y) `evalPair` lbp)
    >>=/ (\(a, b) -> wait (a + b + key))
  where
    x = sumPBT l (lbp -1)
    y = sumPBT r (lbp -1)
```

図 28　sumPBT 関数

2.5.1　並列に全てのノードの和を計算

全てのノードの和を計算する場合，全てのノードを走査する必要がある．このような処理は，並列化により高速化が可能である．

並列に全てのノードの和を計算する sumPBT 関数の定義は，図 28 のようになる．基本的には並列 fib 関数や並列 pow2 関数と同様，再帰的に呼び出す部分を並列化して別のプロセッサに任せる．

sumPBT 関数の並列時間計算量は，pow2Par 関数の並列時間計算量を表現する関数 pow2Cost によって表現できる．これは，1 つ小さな値を引数として 2 つ並列に呼び出す pow2Par 関数と，1 つ高さが小さい完全二分木を引数として 2 つ並列に呼び出す sumPBT 関数とが，同じ構造をしているためである．pow2Cost 関数の引数としては，getHeight p-1 を渡す．この定義では完全二分木は高さ 1 の時点で再帰を終えるので，完全二分木の高さから 1 を引いた値を渡す．

2.5.2　並列に全てのノードの中から最大値を計算

並列に全てのノードの和を計算する maxPBT 関数も，sumPBT 関数と同様に，図 2.5.1 のように定義できる．同じように全てのノードを走査する必要があるが，並列化によって左右の部分木を別のプロセッサに担当させることができる．時間計算量も，sumPBT 関数と同様である．

2.6　リスト処理の並列分割

次に，ある程度長いリストに対して処理を行う場合に，その処理を分割して並列化することで高速化を図るような場合を考える．

countSolve 関数は，文字列からなる数独の問題のリスト

```
{-@ maxPBT :: { p:PBT Nat | getHeight p /= 0 }
  -> lbp:Nat
  -> { t:Tick Int |
    (tcost t == pow2Cost (getHeight p - 1) lbp) }
@-}
maxPBT :: PBT Int -> Int -> Tick Int
```

図 29　maxPBT 関数

```
{-@ countSolve :: { xs:[String] | length xs /= 0 }
  -> { t:Tick Int | tcost t == calcCost * length xs }
  / [length xs]
@-}
countSolve :: [String] -> Tick Int

{-@ countSolveList :: { n:Nat | n > 0 }
  -> { m:Nat | m > 0 }
  -> { xs:[String] | length xs == n * m }
  -> { t:Tick [Int] | tcost t == calcCost * n }
@-}
countSolveList :: Int -> Int
  -> [String] -> Tick [Int]
countSolveList n m xs =
  mapParList m ys (calcCost * n) countSolve
  where
    ys = chunk n m xs
    c = calcCost * n
```

図 30　countSolve 関数，countSolveList 関数

トを受け取り，解を得られた個数を記録する関数である．1 つの数独の問題を解くのにかかるステップ数を calcCost として，全体のステップ数は calcCost にリストの長さを掛けたものとする．ここでは，countSolve 関数の実装には踏み込まない．

countSolve 関数をリストに適用すれば，逐次処理の結果 Tick Int が得られる．これを並列化したのが countSolveList 関数である．並列化においては，mapParList 関数と chunk 関数を用いることができる．まず，chunk 関数を用いて，与えられたリストを長さ n の m 個のリストに分割する．この時，m としてプロセッサ数（またはそれ以下）の値を渡す．得た結果 ys は文字列のリストのリストの型 [[String]] をもつので，mapParList 関数に渡して，並列に各要素 [String] を countSolve 関数に渡して評価させる．

分割したリストに対して並列に処理が行われるので，結果のステップ数は 1 つのプロセッサで行われる countSolve 関数のステップ数，すなわち calcCost に n を掛けた値となる．

このように，配列に対して処理を行い，Tick を返すような関数は，プロセッサ数などを渡して配列を分割することで，比較的容易に並列化することが可能である．

```
par :: a -> b -> b
pseq :: a -> b -> b
```

図 31　par 関数，pseq 関数

3　並列時間計算量解析ライブラリの実装

本研究では，Liquid Haskell における計算量解析ライブラリを，並列処理に拡張した実装を提案する．並列処理関数としては，要素 2 つのタプル（ペア），およびリストの形で渡された関数への処理を行う．

3.1　Haskell における正格評価

Haskell において式を正確に評価する方法として，deepseq 関数がある．deepseq 関数は，式を正規形まで評価する関数である．正規形とは，それ以上評価する部分式のない値のことである．deepseq 関数は，引数の型が NFData 型クラスを満たしていることを制約とする．

3.2　並列 Haskell

Haskell における並列化の手法には，いくつか方法がある．その中でも，parallel パッケージを用いた手法について説明する．

もっとも単純な並列性の実現として，Control.Parallel の par 関数と pseq 関数を用いる方法がある．

par 関数は，第 1 引数と第 2 引数を並列に評価することが有用であることを示し，第 2 引数を返す関数である．第 1 引数に対応したスパークが生成され，スパークプールに追加される．ここで，スパークとは，実行待ちの計算処理のことであり，それらをためておく場所がスパークプールである．スパークプールに追加されたスパークの評価は，ワークスチール (work stealing) という手法を用いて余ったプロセッサに割り当てられる．

pseq 関数は，第 1 引数を正格評価し，第 2 引数を返す関数である．ただし，正規形まで評価するわけではないので，正規形まで評価したい場合は deepseq 関数を用いる．

上記の 2 つの関数を用いても並列性は記述できる．しかし，par/pseq 関数は式の途中に書く必要があり，並列性がアルゴリズムに依存してしまう．そこで，同じパッケージに含まれる Control.Parallel.Strategies を紹介する．(図 32)

Strategy 型は，どのような評価戦略を用いるかを表す型である．using 関数は，値と評価戦略を受け取り，その評価戦略で値を評価する関数である．using 型は中置演算子として用いることができる．例えば`(f x) `using` rpar`などと書くことで，値やそれを計算するアルゴリズムと，

```
infixl 0 `using`
using :: a -> Strategy a -> a

rdeepseq :: NFData a => Strategy a
rpar :: Strategy a

rparWith :: Strategy a -> Strategy a

evalTuple2 :: Strategy a -> Strategy b
  -> Strategy (a, b)
parList :: Strategy a -> Strategy [a]
```

図 32　Control.Parallel.Strategies

評価戦略を分けて記述できる．

rdeepseq 関数は正規形までの評価，rpar 関数は並列に評価を行う戦略を返す関数である．rparWith 関数は，並列評価に加えて評価を追加する．例えば，rparWith 関数に rdeepseq 関数を渡すことで，並列に正規形まで評価できる．

evalTuple2 関数は，2 つの評価戦略を受け取り，1 要素目と 2 要素目にその戦略を適用するような評価戦略を返す関数である．parList 関数は，評価戦略を受け取り，リストに対してその戦略を適用するような評価戦略を返す関数である．

以降では，これらの関数を組み合わせて並列処理の実装を行う．

3.3　並列計算量解析関数の素朴な実装

Liquid Haskell における時間計算量解析を並列処理に拡張することを考えてみる．まず具体的な状況として，並列なフィボナッチ関数の時間計算量解析を試みる．

素朴な実装として，Tick モナドから tval と tcost を直接取り出して適切に操作を行う方法を選択する．並列なフィボナッチ関数として実装した関数 fibPar の定義は図 33 のようになる．

得られる結果自体は 2.3 節で紹介した並列フィボナッチ関数と同一である．$lbp = 0$ の場合は逐次処理，$lbp > 0$ の場合は並列処理を行うのも同一である．

一方で，2.3 節の場合と比べて tval に関する推論が可能になっている．`tval t == fib n`は，返された値が逐次処理のフィボナッチ関数に同じ入力 n を与えた場合と等しいことを篩型の制約として保証する．

しかし，fibPar 関数の返す型を，Tick Nat であるとすると推論に失敗する．フィボナッチ関数の値は Nat，すなわち非負の整数であることは推論できるべきであると予想される．しかし，ここで問題となるのは，rdeepseq 関数の引数には，型クラス NFData を満たしていることが要求されることである．Nat 型はあくまで Liquid Haskell

```
{-@ fibPar :: n:Nat -> lbp:Nat ->
  { t:Tick Int |
    (tcost t == fibCost n lbp) && (tval t == fib n) }
@-}
fibPar :: Int -> Int -> Tick Int
fibPar 0 _ = wait 1
fibPar 1 _ = wait 1
fibPar n 0 = Tick (c1 + c2 + c3) v3
  where
    x0 = fibPar (n -1) 0
    y0 = fibPar (n -2) 0
    Tick c1 v1 = x0 `usingStrategy` rdeepseq
    Tick c2 v2 = y0 `usingStrategy` rdeepseq
    Tick c3 v3 = wait (v1 + v2)
fibPar n lbp = Tick (c1 + c3) v3
  where
    x0 = fibPar (n -1) (lbp-1)
    y0 = fibPar (n -2) (lbp-1)
    Tick c1 v1 = x0 `usingStrategy` rparWith rdeepseq
    Tick c2 v2 = y0 `usingStrategy` rdeepseq
    Tick c3 v3 = wait (v1 + v2)
```

図 33　fibPar 関数

```
{-@ usingStrategy :: x:a
  -> Strategy a -> {v:a | v == x }  @-}
usingStrategy :: a -> Strategy a -> a
```

図 34　usingStrategy 関数

のアノテーション上の型であり，型クラス NFData を満たしているわけではないので，rdeepseq 関数の返り値の型は，引数が Nat ではなく Int として解釈されるため Strategy Int が返り，usingStrategy を通して Int を返す．

usingStrategy 関数は，第一引数を第二引数の評価戦略に基づいて評価し，引数と同じ値を返す関数である．

3.4　一般化の検討

先ほどの並列フィボナッチ関数の例では，Tick の中の tcost を直接操作していた．しかし，tcost を直接操作すると，誤ってステップ数を増やしたり減らしたりしてしまう場合が起こり得る．ここでは，ステップ数を適切に処理してくれる並列処理関数を定義して，それを呼び出すことを考える．

まずは，Tick を返す関数を引数に取った形での一般化を試みる．具体的には，値を取って Tick を返す関数を 2 つと，それらへ渡す引数を受け取り，それらを実際に逐次または並列に適用して，tcost を適切に加えた Tick を返すような形である．しかし，Tick を返す関数に適用した場合の tcost を得るには更に高階な推論が必要となり，bind 関数の推論ができないのと同様の理由から上手く行かなかった．

そこで，関数の代わりに Tick 自体を渡すことにして

```
{-@ evalPair :: t12:(Tick a, Tick b) -> lbp:Nat
  -> { t:Tick (a, b) |
    tcost t ==
        if lbp == 0
        then (tcost (fst t12) + tcost (snd t12))
        else tcost (fst t12) &&
    tval t == (tval (fst t12), tval (snd t12))}
@-}
evalPair :: (NFData a, NFData b)
  => (Tick a, Tick b)
  -> Int -> Tick (a, b)
t12 `evalPair` 0 = seqPair t12
t12 `evalPair` lbp = parPair t12 lbp
```

図 35　evalPair 関数

みる．具体的には，Tick から構成されるペアを受け取り，ペアを要素として持つ Tick を返すような関数を考えてみる．それが，本研究で実装した関数の 1 つである，evalPair 関数である．

3.5　並列にペアを処理する関数

3.5.1　evalPair 関数

evalPair 関数の紹介は，2.2.1 節で行っている．ここでは，内部の実装に関しての話を行う．evalPair 関数は，引数として取ったプロセッサの 2 の対数 lbp が 0 ならば逐次処理関数 seqPair を，正ならば並列処理関数 parPair を呼び出す．

並列 Haskell の性質から，並列化した場合も結果が変わらないことは保証したいが，SMT ソルバの推論の限界から，evalPair 関数では篩型の制約には含まれているものの実際に使う際には推論できていない．

3.5.2　seqPair 関数

seqPair 関数の定義は図 36 のようになる．seqPair 関数は，引数として Tick から構成されるペア xy0 を受け取り，それを逐次的に評価した結果を返す関数である．evalTuple2 関数は，2 要素のタプルの 1 つ目と 2 つ目の評価戦略をそれぞれ受け取り，2 要素のタプルの評価戦略を返す関数である．1 つ目と 2 つ目を rdeepseq で正規形まで評価するとして，ペアの要素を逐次的に評価する．評価した結果の t1 と t2 はそれぞれ Tick 型なので，分解してそれぞれの tcost を加算し，tval はペアとして返す．

3.5.3　parPair 関数

parPair 関数の定義は図 37 のようになる．parPair 関数は，引数として Tick から構成されるペア xyl とプロセッサ数の 2 の対数 lbp を受け取り，それを並列的に評価した結果を返す関数である．プロセッサ数の 2 の対数は，正であること，つまり空いたプロセッサが存在することを確認するために用いており，処理には使用しない．seqPair 関数と共通する部分が多いが，1 つ目の要素の評

```
{-@ seqPair :: t12:(Tick a, Tick b)
  -> { t:Tick (a, b) |
    tcost t == tcost (fst t12) + tcost (snd t12) &&
    tval t == (tval (fst t12), tval (snd t12))}
@-}
seqPair :: (NFData a, NFData b)
  => (Tick a, Tick b) -> Tick (a, b)
seqPair xy0 =
  Tick (tcost t1 + tcost t2) (tval t1, tval t2)
  where
    (t1, t2) =
      xy0 `usingStrategy`
      evalTuple2 rdeepseq rdeepseq
```

図 36　seqPair 関数

```
{-@ parPair :: t12:(Tick a, Tick b)
  -> { lbp:Nat | lbp > 0 }
  -> { t:Tick (a, b) |
    tcost t == tcost (fst t12) &&
    tval t == (tval (fst t12), tval (snd t12))}
@-}
parPair :: (NFData a, NFData b)
  => (Tick a, Tick b) -> Int -> Tick (a, b)
parPair xyl _ = Tick (tcost t1) (tval t1, tval t2)
  where
    (t1, t2) =
        xyl `usingStrategy`
        evalTuple2 (rparWith rdeepseq) rdeepseq
```

図 37　parPair 関数

価戦略として rdeeepseq の前に rparWith を加えて並列処理を行っている点，tcost の計算に t1 のみを使用している点が異なる．

parPair 関数は，篩型の制約ではない暗黙の制約として，ペアの 1 つ目の要素の tcost が 2 つ目の要素より大きいことを仮定している．これは，結果の tcost として，1 つ目の tcost を割り当てることで，推論を簡略化するためである．

より tcost が大きい方を採用する，といった実装にした場合について検討する．このような実装の場合でも，pow2Par 関数のように，左右の tcost が一致するような場合は推論が行える．一方で fibPar 関数は左右で tcost が一致せず，さらに左右のどちらがより tcost が大きいかを SMT ソルバでは推論できない．そのため，暫定的な処置として，暗黙の制約を設けることを選択した．

3.6　並列にリストを処理する関数

ペアに対して並列に処理を行う parPair と同様，リストに対して並列に処理を行う関数を考える．

3.6.1　parList 関数

parList 関数の紹介は，2.2.2 節で行っている．ここでは，内部の実装に関しての話を行う．

```
{-@ parList :: c:Nat
  -> { xs:[{ t1:Tick a | tcost t1 <= c }] | xs /= [] }
  -> { p:Nat | p >= length xs }
  -> { t:Tick { os:[a] | length xs == length os } |
    tcost t == c }
@-}
parList :: (NFData a)
  => Int -> [Tick a] -> Int -> Tick [a]
parList c xs p =
  Tick c (go xs `usingStrategy` parList rdeepseq)
  where
    {-@ go :: xs:[Tick a]
      -> { ys:[a] | length xs == length ys } @-}
    go :: [Tick a] -> [a]
    go [] = []
    go (x : xs) = cons (tval x) (go xs)
```

図 38　parList 関数

parList 関数は，Tick のコストとして c，値としてはリスト xs を並列に実行した結果を指定する．，usingStrategy 関数と parList 関数，rdeepseq 関数を用いて，並列にリスト xs を正規形まで評価している．go 関数は，Tick のリストからリストを取り出す補助関数である．

コスト c の値は，Tick 型からなるリスト xs のうち，tcost が最大のものを取ってくれば本来必要がないものである．しかし，Liquid Haskell の推論において，そのような値を取り出して制約とすることは難しい．また，本来なら xs を第一引数とした方が自然ではあるが，xs の全要素が条件を満たしていることを明記するには，コスト c を先に渡さざるを得ないため，このような順にしている．

例えば，受け取るリストを，特定の値以下の Tick 型からなるリストを表すデータ構造とすれば，この問題は解決できる可能性がある．しかし，単にコストと Tick 型のリストを受け取る現在の形の方が，他の処理と組み合わせる際には便利である．

3.6.2　chunk 関数

chunk 関数は，入力として正の数 n と m，長さ n × m のリスト xs を受け取り，長さ n の m 個のリストからなるリストを返す関数である．篩型を用いて，指定した長さとなるように指定している．

ここでは Tick に包まない単なるリストのリストとして返している．しかし，並列に分割する部分も処理の一部なので，tcost を記録して Tick に包んで返すべきである可能性はある．

3.6.3　map 関数

map 関数は，コスト c と，そのコストとなる Tick を返す関数 f と，リスト xs を受け取り，f を xs に適用した結果として Tick 型からなる配列を返す関数である．この関数では，単にリスト xs の各要素に関数 f の処理を適用するだけで，並列処理に基づいた挙動などは行わない点

```
{-@ chunk :: { n:Nat | n /= 0 }
  -> { m:Nat | m /= 0 }
  -> { xs:[a] | length xs == n * m }
  -> { yss:[ { ys:[a] | length ys == n } ] |
    length yss == m }
  / [m]
@-}
chunk :: Int -> Int -> [a] -> [[a]]
chunk _ 1 xs = [xs]
chunk n m xs = cons as (chunk n (m -1) bs)
  where
    as = take n xs
    bs = drop n xs
```

図 39　chunk 関数

```
{-@ map :: c:Nat
  -> (a -> { t:Tick b | tcost t <= c })
  -> xs:[a]
  -> { ys:[{ t:Tick b | tcost t <= c }] |
    length ys == length xs }
@-}
map :: Int -> (a -> Tick b) -> [a] -> [Tick b]
map c f [] = []
map c f (x : xs) = cons (f x) (map c f xs)
```

図 40　map 関数

に注意が必要である．コスト c を渡すことで，篩型の制約として処理結果の Tick 型の tcost が全て c 以下であることを保証している．

3.6.4　mapParList 関数

mapParList 関数は，parList 関数と map 関数を内部で利用する関数である．正の整数 n と，長さ n のリスト xs，コスト c，コスト c の Tick 型を返す関数 f を受け取り，xs の各要素に並列に関数 f を適用した結果として，リストを tval の型とするような Tick を返す関数である．返り値の Tick 型の tcost は，c と一致することを保証している．

parList 関数の第二引数としては，map 関数を適用した後の Tick のリストを渡す．つまり，map 関数に渡した段階ではどのように評価されるかは決まらず，それを parList 関数の引数として渡すことで，並列に評価されることが決まる．map 関数の返り値としての tcost は，捨てられることとなる．

4　並列時間計算量解析の妥当性に関する議論

4.1　実行時間測定結果との比較

本研究では並列プログラムのステップ数を解析する手法を提案した．その妥当性の検証のため，実際に並列プログラムの実行時間を計測して，解析によって得たステップ数との比較を行う．本節での実行時間は，測定対象の

```
{-@ mapParList :: { n:Nat | n > 0 }
  -> { xs:[a] | length xs == n }
  -> c:Nat
  -> (a -> { t:Tick b | tcost t <= c })
  -> { t:Tick { os:[b] | length xs == length os } |
    tcost t == c }
@-}
mapParList :: (NFData b)
  => Int -> [a] -> Int -> (a -> Tick b) -> Tick [b]
mapParList n xs c f = parList c (map c f xs) n
```

図 41　mapParList 関数

計算が始まる時刻から，計算が終了するまでの経過時間のことを指す．実行環境は，次の通りである．

CPU	AMD Ryzen Threadripper 3970X 32-Core Processor
RAM	32.0GB
OS	Ubuntu 20.04.2 LTS

GHC では，同時に並列で走らせることができるスレッド数を，実行時オプション`-N` で指定できる．この値は CPU のコア数と等しくすることが推奨されている．今回は 32 コアの CPU を用いるため，`-N` で指定する値，すなわち想定するプロセッサ数を 1 から 32 まで変化させ，それぞれ測定を行う．

4.1.1　完全二分木の走査

表 1 は，完全二分木の走査における実行時間 3 回分とその平均，解析したステップ数の比較である．

ここでは，引数 $n = 27$，引数 lbp をプロセッサ数の 2 の対数として 2.5.1 節の maxPBT 関数の計算を測定対象とした．ただし，完全二分木の生成は予め行い，実行時間には含んでいない．

プロセッサ数としては 1,2,4,8,16,32 を指定し，それぞれ引数 lbp として 0,1,2,3,4,5 を与えた．

また，横軸をプロセッサ数，縦軸を実行時間および解析したステップ数としたグラフが図 42 である．

グラフを見ると，プロセッサ数 (`numbers of processors`) を増やすごとに，解析したステップ数 (`step`) とともに実行時間 (`time`) も減少傾向にあることがわかる．一方で，プロセッサ数が 32 個の場合の実行時間は，プロセッサ数が 16 個の場合より短くなっていない．

さらに，横軸を解析したステップ数，縦軸を実行時間として，回帰直線を引いた散布図が図 45 である．

実行時間とステップ数の相関係数は，0.9939287316 だった．ただし，データ数が 6 つと非常に少ないこと，相関があると判断できるからと言って有用性にすぐ結び付けられるわけではないことに注意が必要である．

プロセッサ数	1 回目の実行時間 (s)	2 回目 (s)	3 回目 (s)	平均実行時間 (s)	ステップ数
$1 = 2^0$	10.987433	10.996685	11.023951	11.00269	134,217,727
$2 = 2^1$	7.094920	7.064267	7.036276	7.065154	67,108,864
$4 = 2^2$	4.063124	3.993696	4.005592	4.020804	33,554,433
$8 = 2^3$	2.457884	2.472770	2.487182	2.472612	16,777,218
$16 = 2^4$	1.869392	1.882695	1.882147	1.878078	8,388,611
$32 = 2^5$	2.216256	2.229249	2.196343	2.213949	4,194,308

表 1　完全二分木の走査における実行時間とステップ数

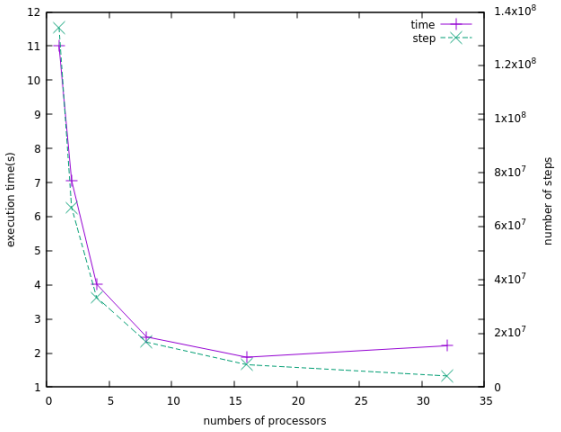

図 42　完全二分木の走査における実行時間とステップ数の比較

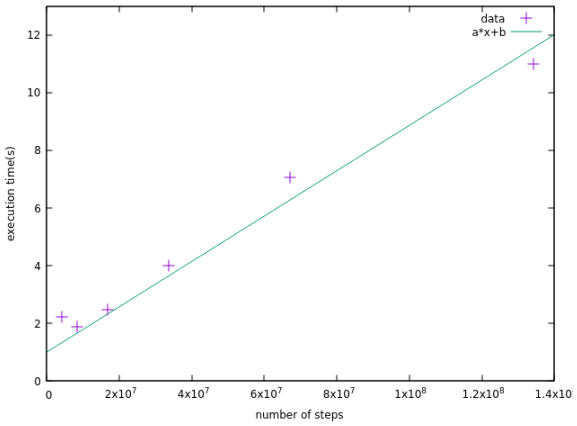

図 43　完全二分木の走査における実行時間とステップ数による散布図

4.1.2　リスト処理の並列分割

表 2 は，リスト処理の並列分割における実行時間と，解析したステップ数の比較である．

ここでは，16,000 個の数独の問題を，2.6 節の mapParList 関数を用いてプロセッサ数で分割してそれらを解く計算を測定対象とした．プロセッサごとに均等に割り振れるよう，プロセッサ数として 1,2,4,5,8,10,16,20,32 をそれぞれ指定し，計測を行った．

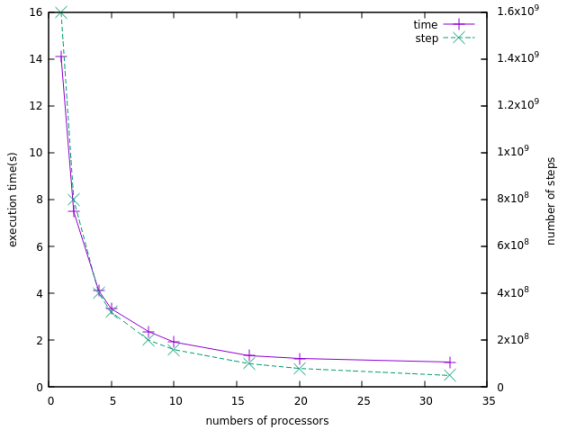

図 44　リスト処理の並列分割における実行時間とステップ数の比較

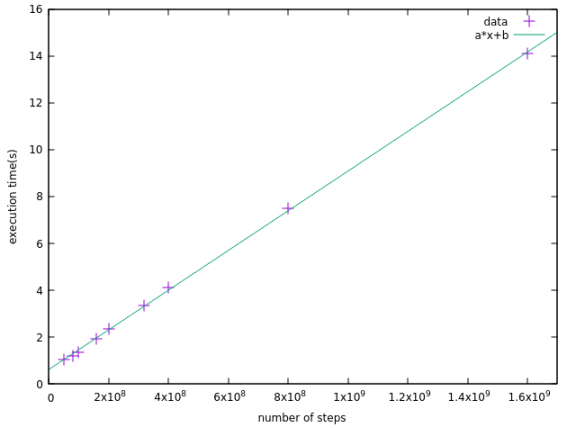

図 45　リスト処理の並列分割における実行時間とステップ数による散布図

また，横軸をプロセッサ数，縦軸を実行時間および解析したステップ数としたグラフが図 44 である．

さらに，横軸を解析したステップ数，縦軸を実行時間として，回帰直線を引いた散布図が図 45 である．

実行時間とステップ数の相関係数は，0.9998209612 だった．こちらも同様にデータ数が少なく，有用性に直接結びつけるには注意が必要である．

プロセッサ数	1 回目の実行時間 (s)	2 回目 (s)	3 回目 (s)	平均実行時間 (s)	ステップ数
1	13.727178	14.910259	13.647531	14.09499	1.6×10^9
2	7.563414	7.460836	7.429231	7.484494	8.0×10^8
4	4.087452	4.191272	4.068193	4.115639	4.0×10^8
5	3.335491	3.307790	3.386718	3.343333	3.2×10^8
8	2.330277	2.402992	2.358420	2.363896	2.0×10^8
10	1.926612	1.933136	1.911500	1.923749	1.6×10^8
16	1.356350	1.326441	1.350727	1.344506	1.0×10^8
20	1.289420	1.195688	1.184411	1.223173	8.0×10^7
32	1.223408	0.965334	1.026178	1.071640	5.0×10^7

表 2　リスト処理の並列分割における実行時間とステップ数

4.2　PRAM model 採用の妥当性

本研究では，並列計算のモデルとして PRAM を採用した．PRAM は，並列アルゴリズムの時間計算量や空間計算量を考えるための抽象的な計算モデルで，複数台の RAM を並列に動かすモデルである．RAM は，メモリのどこにでも一定時間でアクセスが可能とする計算モデルである．しかし，Haskell のメモリモデルでは，ランダムアクセスが一定時間で行えるわけではない．

並列計算のモデルとしては，有向非巡回グラフモデル（以下 DAG モデルと呼ぶ）がある．DAG モデルでは，並列 Haskell と同様，タスクをプロセッサに割り振るスケジューリングの問題を抽象化している．一方で，それによりアルゴリズムの時間計算量の解析が困難である．

本研究では，時間計算量の解析を容易にするために PRAM を採用したが，他の計算モデルの採用は検討の余地がある．

5　おわりに

5.1　結論

本研究では，Liquid Haskell において，並列プログラムの時間計算量を静的に解析する手法を提案し，これを実装した．また，提案した実装を適用して，いくつかの並列プログラムにおける時間計算量の解析を行った．実際に並列プログラムを実行して経過時間の計測を行い，解析したステップ数と比較してある程度妥当であることを確認した．

5.2　今後の課題

今後の課題としては，本研究で提案した手法をより多くの並列プログラム例に対して適用することが挙げられる．クイックソートやマージソートのような典型的なアルゴリズムから，より実用的なプログラムまで様々な適用先を考えることができる．

また，Liquid Haskell では，Proof Combinators を用いて，自動推論の制約に囚われず手動で証明を行うこともできる．Proof Combinators を利用した場合の並列プログラムの時間計算量の解析についても，今後の課題とする．

さらに，議論の章で述べた通り，PRAM 以外の計算モデルの採用も検討の余地があると思われる．

謝辞 本研究においてご意見いただいた西崎研究室の皆様に感謝します．

参 考 文 献

[1] : The Glasgow Haskell Compiler. `https://www.haskell.org/ghc/` (visited on 08/01/2021).

[2] : Haskell Language. `https://www.haskell.org/` (visited on 08/01/2021).

[3] : LiquidHaskell. `https://ucsd-progsys.github.io/liquidhaskell-blog/` (visited on 08/01/2021).

[4] Biere, A., Biere, A., Heule, M., van Maaren, H., and Walsh, T.: *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*, IOS Press, NLD, 2009.

[5] de Moura, L. and Bjørner, N.: Z3: An Efficient SMT Solver, *Tools and Algorithms for the Construction and Analysis of Systems*, Ramakrishnan, C. R. and Rehof, J.(eds.), Berlin, Heidelberg, Springer Berlin Heidelberg, 2008, pp. 337–340.

[6] Handley, M. A. T., Vazou, N., and Hutton, G.: Liquidate Your Assets: Reasoning about Resource Usage in Liquid Haskell, *Proc. ACM Program. Lang.*, Vol. 4, No. POPL(2019).

[7] Hoffmann, J. and Shao, Z.: Automatic Static Cost Analysis for Parallel Programs, *Programming Languages and Systems*, Vitek, J.(ed.), Berlin, Heidelberg, Springer Berlin Heidelberg, 2015, pp. 132–157.

[8] JéJé, J.: *An Introduction to Parallel Algorithms*, Reading, MA: Addison-Wesley, 1992.

[9] Jones, S. P.: *Haskell 98 language and libraries: the revised report*, Cambridge University Press, 2003.

[10] Loidl, H. W.: *Granularity in large-scale parallel functional programming*, PhD Thesis, ProQuest Dissertations & Theses,, 1998.

[11] Marlow, S.: *Parallel and Concurrent Programming in Haskell*, O'Reilly Media, Inc., 2013.

[12] Marlow, S. et al.: Haskell 2010 language report, *Available on: https://www.haskell.org/onlinereport/haskell2010*, (2010).

[13] Marlow, S., Maier, P., Loidl, H.-W., Aswad, M. K., and Trinder, P.: Seq No More: Better Strategies for Parallel Haskell, *SIGPLAN Not.*, Vol. 45, No. 11(2010), pp. 91–102.

[14] McCarthy, J., Fetscher, B., New, M. S., Feltey, D., and Findler, R. B.: A Coq library for internal verification of running-times, *Science of Computer Programming*, Vol. 164(2018), pp. 49 – 65. Special issue of selected papers from FLOPS 2016.

[15] Moggi, E.: Notions of computation and monads, *Information and Computation*, Vol. 93, No. 1(1991), pp. 55–92. Selections from 1989 IEEE Symposium on Logic in Computer Science.

[16] Trinder, P. W., Hammond, K., Loidl, H.-W., and Peyton Jones, S. L.: Algorithm + Strategy = Parallelism, *J. Funct. Program.*, Vol. 8, No. 1(1998), pp. 23–60.

[17] Vazou, N., Seidel, E. L., Jhala, R., Vytiniotis, D., and Peyton-Jones, S.: Refinement Types for Haskell, *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, New York, NY, USA, Association for Computing Machinery, 2014, pp. 269–282.

[18] Wyllie, J. C.: The complexity of parallel computations, Technical report, Cornell University, 1979.

[19] 宮野悟: 並列アルゴリズム 理論と設計, 近代科学社, 1993.

[20] 白水駿, 江本健斗: 並列プログラム計算量の系統的機械証明手法の開発, 日本ソフトウェア科学会大会論文集, Vol. 34(2017), pp. 141–151.

[21] 梅村晃広: SAT ソルバ・SMT ソルバの技術と応用, コンピュータ ソフトウェア, Vol. 27, No. 3(2010), pp. 3_24–3_35.