Gears Agda 上のモデル検査の形式化

Uechi Yuto, Shinji Kono 琉球大学

Gears Agda によるプログラムの改善

- 思い思いにプログラムを書くと、冗長なコードができてしまい、実行時間も遅い場合がある。この場合にコードに対してアルゴリズムを適応すると、一般的に良いコードが作成できる。
- しかし、世の中にはすでに大量のアルゴリズムが存在するため、これを一人のプログラマーが全て覚え、適応できる場面を思いつくというのは初学者には難しい。そのため、人が書いたコードに対してアルゴリズムを使用するコードに自動的に変換できるようにしたい。
- この際、アルゴリズム適応前後で処理が変わっていないか検証するのは普通のプログラミング言語では難しい。このアルゴリズム適応前後の処理の恒等性を検証するため、Gears Agda を用いる。

Gears Agda と CbC

- Gears Agda とは当研究室で開発している Continuation based C (CbC) の概念を取り入れた記法で書かれた Agda のこと
- 通常のプログラミング言語では関数を実行する際にはメインルーチンからサブルーチンに遷移する。この際メインルーチンで使用していた変数などの環境はスタックされ、サブルーチンが終了した際にメインルーチンに戻り、スタックしていた変数をもとに戻す流れとなる。
- CbCの場合はサブルーチンコールの際にアセンブラで言うjmpで関数遷移を行うことができ、スタックを持たず環境を保持しない。更に遷移後にメインルーチンに戻ることもない。つまり関数の実行では暗黙な環境が存在せず、関数が受け取った引数のみを使用する。
- これにより限定的な実行条件を作成でき、検証がしやすくなる。

Gears Agda でのアルゴリズム入れ替え判定

- 現在、アルゴリズムの適応可否は以下の方法を考えている。
  - あらかじめ、アルゴリズムの実装と検証をおこなったアルゴリズムSetsを用意しておく
  - 書いたコードが事前に定義していたアルゴリズムの仕様を満たしているかを検証していき、満たしているコードがあった場合にそのコードを事前に定義してあるアルゴリズムに入れ替える
- この際、実装が仕様を満たしているか検証する手法には、定理証明やモデル検査などが挙げられる。

Gears Agda でのモデル検査

- アルゴリズムの入れ替え可否判定には Gears Agda でモデル検査を行い、アルゴリズムの仕様がコードに適応できるか検証するのが妥当だと考えている
- 思い思いに書いたコードに対して定理証明を行うのはコストが高く、適応するものの内部動作が一致しない場合定理証明を行っても使えないためである
- アルゴリズムの入れ替え可否をモデル検査で判定し、入れ替えたあとに適応前後で同じ処理をしていることを定理証明で検証することを目標としている
- この Gears Agda でのモデル検査の先駆けとして今回は Dining Philosophers Program のモデル検査を行った

CbC について

- CbCとは当研究室で開発しているC言語の下位言語
  - 継続呼び出しは引数付き goto 文で表現される。
    - 関数呼び出し時のスタックの操作を行わずjmp命令で次の処理に移動する
  - 処理の単位を Code Gear, データの単位を Data Gear として記述するプログラミング言語
- CbC のプログラミングでは Data Gear を Code Gear で変更し、その変更を次の Code Gear に渡して処理を行う。

Agda の基本

- Agdaとは定理証明支援器であり、関数型言語
- Agdaでの関数は、最初に型について定義した後に、関数を定義する事で記述する
- 型の定義部分で、入力と出力の型を定義できる
  - input → output のようになる
- 関数の定義は = を用いて行う
  - 関数名の後、 = の前で受け取る引数を記述します

Gears Agda の記法

Gears Agda では CbC と対応させるためにすべてLoopで記述する

loopは `→ t` の形式で表現する

再帰呼び出しは使わない(構文的には禁止していないので注意が必要)

```
{-# TERMINATING #-}
whileLoop : {l : Level} {t : Set l} → Env → (Code : Env → t) → t
whileLoop env next with lt 0 (varn env)
whileLoop env next | false = next env
whileLoop env next | true = whileLoop (record {varn = (varn env) - 1 ; vari = (vari env) + 1}) next
```

- t を返すことで継続を表す(tは呼び出し時に任意に指定してもよい. testに使える)
- tail call により light weight continuation を定義している

Normal level と Meta Level を用いた信頼性の向上

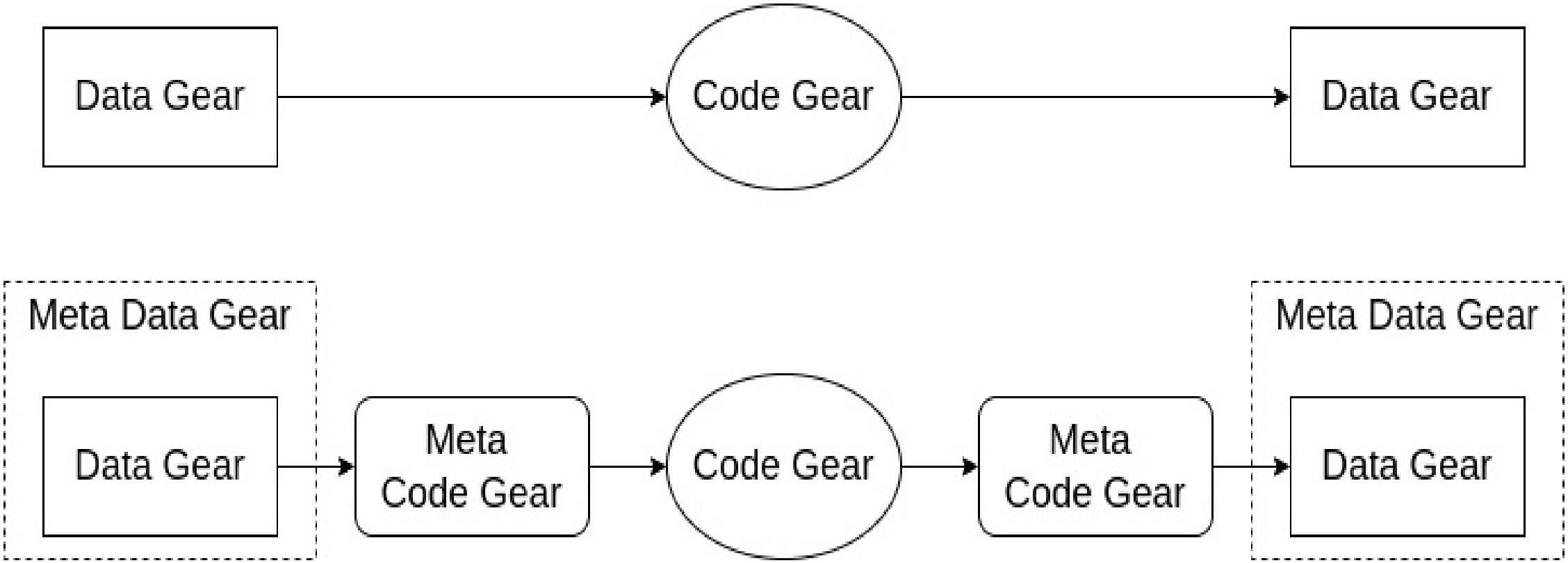
Normal Level

- 軽量継続
- Code Gear 単位で関数型プログラミングとなる
- atomic(Code Gear 自体の実行は割り込まれない)
- ポインタの操作は含まれない

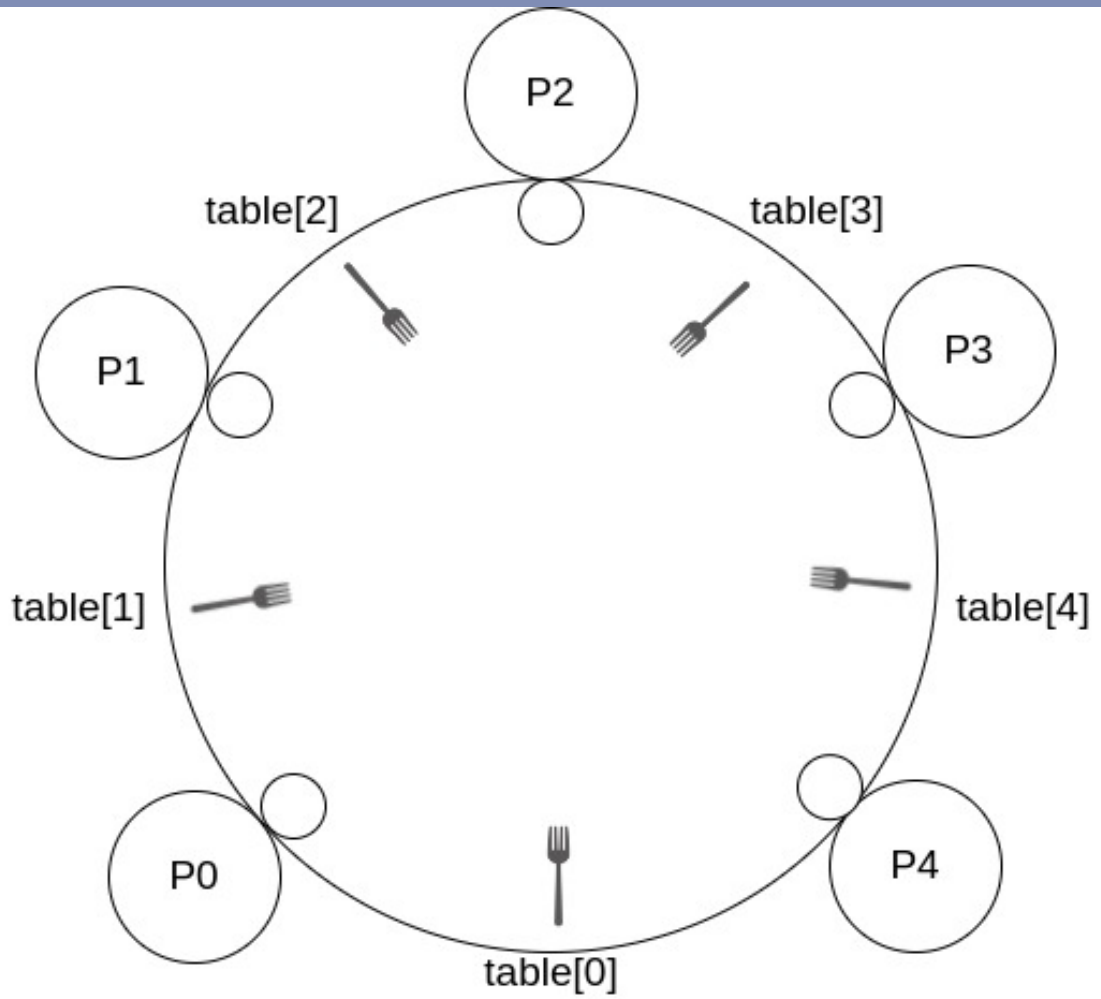
Meta Level

- メモリやCPUなどの資源管理, ポインタ操作
- Hoare Condition と証明
- Contextによる並列実行
- monadに相当するData構造

Normal level と Meta Level の対応



DPPの説明



モデル検査と定理証明の説明

モデル検査はコストが安いが完全な検証にはならない
定理証明はコスト高いが完全な検証になる

DPPの記述

- DPPはマルチプロセスの同期問題である
  - しかし、Agdaでは並列実行をサポートしていないため、step実行毎に一つづつ処理することで並列実行を表現している
- 加えて、哲学者が思考中に食事をし始めるのか、食事中に思考に戻ろうとするのかで分岐が発生する
  - これを網羅するために今回はその状態に対してビット全探索を行った

モデル検査でのデッドロック検知

- デッドロックとは全てのプロセスが他のプロセスの実行待ちの状態となり実行が進まなくなること
- 今回Gears Agda でのデッドロックの定義として、以下2つを満たすものとした
  - ビット全探索で分岐が作れない
  - 実行時に状態に変動がない
- これでプログラムがデッドロックしているのか検知するところまではできるようになった

まとめと今後の研究方針

- 今回は Gears Agda にてDPPを記述し、モデル検査にてデッドロックを検知できるようになった
  - 今回のモデル検査を一般化し、自動でモデル検査を実行できるようにしたい
  - さらにプログラムの仕様を渡し、これを満たしているかモデル検査で検証したい
    - 仕様にLTLを使えるようにもしたい
- アルゴリズムの自動適応にて、モデル検査で仕様を満たしている場合に入れ替えて同じ動作をしているかを定理証明で証明できるようにしたい