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

- コードの改良の繰り返しで開発していきたい
- 改良はアルゴリズムの変更を含む
- 変更前と変更後の動作が一致することを検証したい
- プログラムをGears Agdaで記述し、Agdaによるモデル検査により検証する
- Agda Curry-Howard対応に基づく関数型言語、定理証明支援系
- CbC Cの下位言語で軽量継続gotoで記述する
- Gears Agda Agdaにより記述されたCbCプログラム

Gears Agda でのアルゴリズム入れ替え判定

- gears Agdaでアルゴリズムの記述とアルゴリズムの証明(invariant)を同時に記述する
- invariantを変換できればアルゴリズムを置換できる
- 変換はAgdaでの証明で行う
- 非決定的な実行が存在する場合は可能な実行を網羅する(モデル検査)
- AgdaでGears Agdaのモデル検査の定式化を行う

Gears Agda でのモデル検査

- 非決定的な実行の例として並列実行がある
- 5人の哲学者が共有されたフォークを取り合う例題 Dining Philosophers Program
- Live Lock と Dead Lockの検出を行う
- Gears Agda の実行を複数のContext(プロセス)でemurationする
- 5人のtep実行のshuffring
- 可能な状態の網羅のためのDataBase
- とりあえずboundedで実装した

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
    - 関数呼び出し時に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
  - 関数名の後ろの行で、 = の前で受け取る引数を記述、=の後ろで実装を記述する

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)

```
init-table : {n : Level} {t : Set n} → ℕ → (exit : Env → t) → t
init-table n exit = init-table-loop n 0 (record {table = [] ; ph = []}) exit where
  init-table-loop : {n : Level} {t : Set n} → (redu inc : ℕ) → Env → (exit : Env → t) → t
  init-table-loop zero ind env exit = exit env
  init-table-loop (suc redu) ind env exit = init-table-loop redu (suc ind) record env{
    table = 0 ∷ (table env)
    ; ph = record {pid = redu ; left-hand = false ; right-hand = false ; next-code = C_thinking } ∷ (ph env) } exit
```

- 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

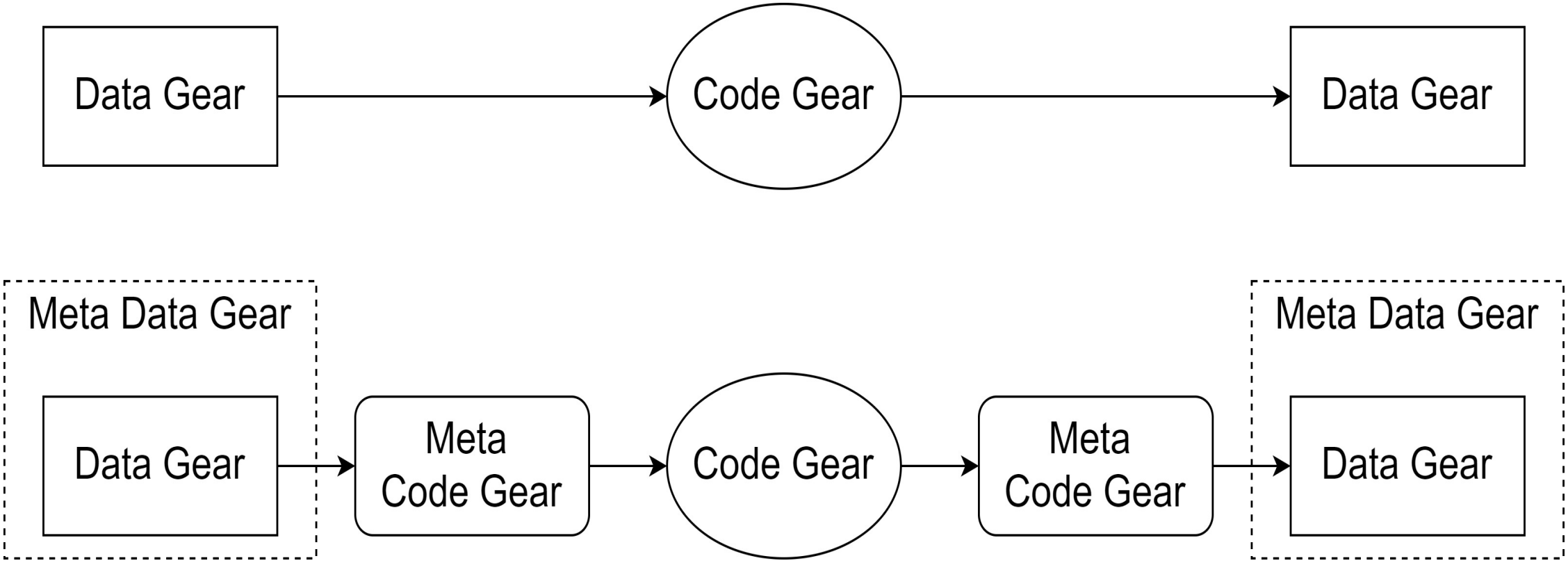
Normal Level

- 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

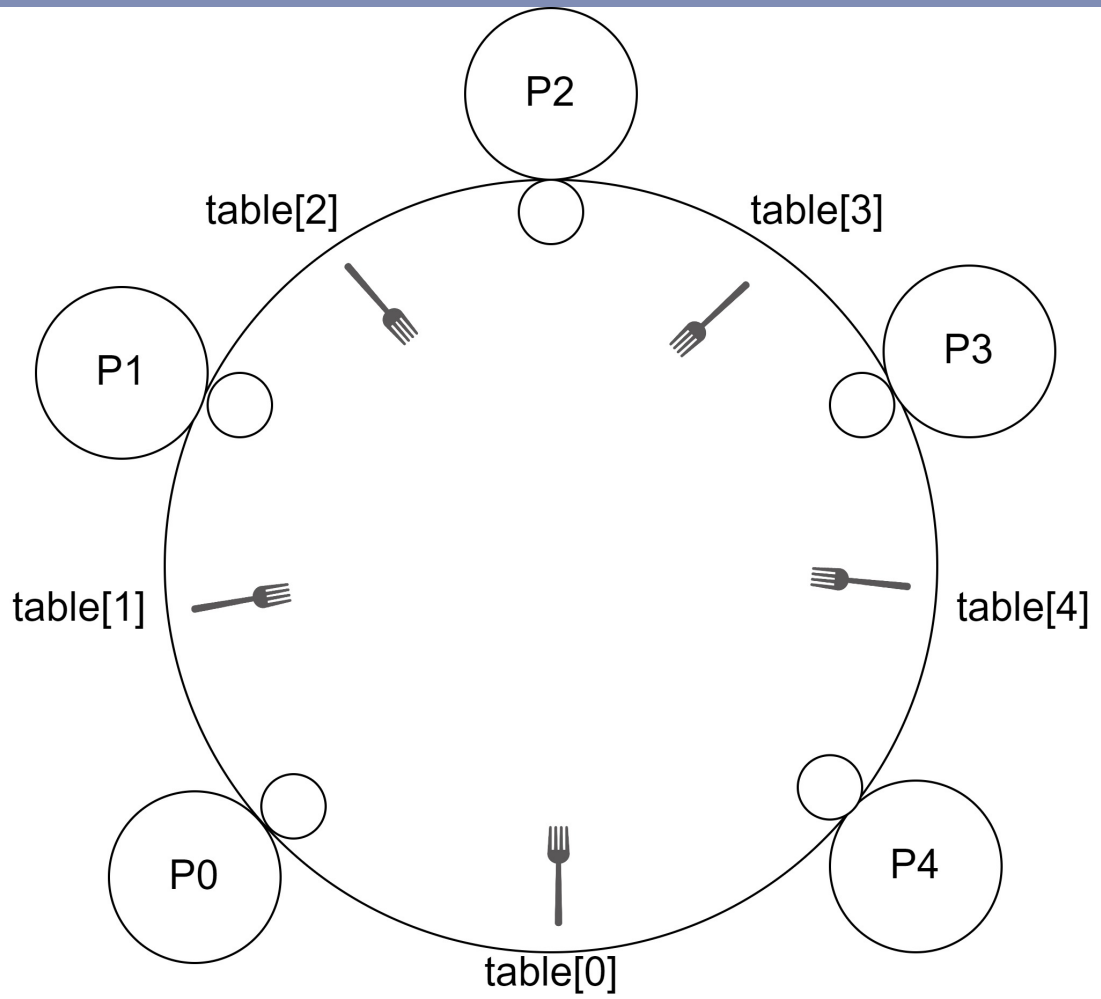
Meta Level

- 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



Dining Philosophers Program



モデル検査と定理証明の説明

- モデル検査は入力を網羅的に仕様を満たしているか検証する
- 入力を無限に検証することはできないため、定理証明と比べると完全な検証にならない
- 定理証明に比べてコストが低い

Dining Philosophers Program の実装

- 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
  - 今回はその状態に対して網羅することでモデル検査をboundedに行っている

Dining Philosophers Program の実装

Gears Agdaで使用するData Gearを以下のように定義した

```
record Phi : Set where
  field
    pid : ℕ
    right-hand : Bool
    left-hand : Bool
    next-code : Code
open Phi
```

```
record Env : Set where
  field
    table : List ℕ
    ph : List Phi
open Env
```

Dining Philosophers Program の実装

```
data Code : Set  where
   C_putdown_rfork : Code
   C_putdown_lfork : Code
   C_thinking : Code
   C_pickup_rfork : Code
   C_pickup_lfork : Code
   C_eating : Code
```

```
code_table : {n : Level} {t : Set n} → Code → ℕ → Phi → Env → (Env → t) → t
code_table C_putdown_rfork = putdown-rfork-c
code_table C_putdown_lfork = putdown-lfork-c
code_table C_thinking = thinking-c
code_table C_pickup_rfork = pickup-rfork-c
code_table C_pickup_lfork = pickup-lfork-c
code_table C_eating = eating-c
```

Dining Philosophers Program の実装

以下が哲学者の動作の実装の一つ

```
pickup-lfork-c : {n : Level} {t : Set n} → ℕ → Phi → Env → (Env → t) → t
pickup-lfork-c ind p env exit = pickup-lfork-p (suc ind) [] (table env) p env exit where
  pickup-lfork-p : {n : Level} {t : Set n} → ℕ → (f b : List ℕ) → Phi → Env → (Env → t) → t
  pickup-lfork-p zero f [] p env exit with table env
  ... | [] = exit env
  ... | 0 ∷ ts = exit record env{ph = ((ph env) ++ (record p{left-hand = true ;
         next-code = C_eating} ∷ [])); table = ((pid p) ∷ ts)}
  ... | (suc x) ∷ ts = exit record env{ph = ((ph env) ++ p ∷ [])}
  pickup-lfork-p zero f (0 ∷ ts) p env exit = exit record env{
    ph = ((ph env) ++ (record p{left-hand = true ;
         next-code = C_eating} ∷ [])); table = (f ++ ((pid p) ∷ ts))}
```

モデル検査でのデッドロック検知

- 現在は探索する深さを指定している(boundedなモデル検査)
- 今回Gears Agdaでのデッドロックの定義として、以下2つを設定した
  - 網羅で分岐が作れない
  - 実行時に状態に変動がない
- これでプログラムがdead Lockしているのか検知するところまではできた

モデル検査でのデッドロック検知

網羅には以下のMetaCodeGearを実装した(一部抜粋)

```
brute-force-search : {n : Level} {t : Set n}
  → Env → (exit : List Env → t) → t
brute-force-search env exit = make-state-list
  1 [] (ph env) env (env ∷ []) exit where
  make-state-list : {n : Level} {t : Set n}
    → ℕ → List Bool → List Phi
    → Env → (List Env) → (exit : List Env → t) → t
  make-state-list redu state (x ∷ pl) env envl exit with next-code x
  ... | C_thinking = make-state-list (redu + redu)
    (state ++ (false ∷ [])) pl env envl exit
  ... | C_eating = make-state-list (redu + redu)
    (state ++ (false ∷ [])) pl env envl exit
  ... | _ = make-state-list redu state pl env envl exit
  make-state-list redu state [] env envl exit =
    bit-force-search redu [] state env envl exit where
```

まとめと今後の研究方針

- 今回は Gears Agda にて Dining Philosophers Program を記述し、モデル検査にてデッドロックをboundedに検知できた
  - unboundedなデッドロック検知
  - プログラムの仕様(invariant)を渡し、これを満たしているかモデル検査で検証
    - 仕様にLTLを使えるように
- 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