Vol.2015-ARC-215 No.16
Vol.2015-OS-133 No.16
2015/5/27

Monad に基づくメタ計算を基本とする Gears OS の設計

小久保 翔平[†1]　伊 波 立 樹[†2]　河 野 真 治[†2]

本研究室では Code Gear, Data Gear を用いた並列フレームワークの開発を行なっている。Code Gear, Data Gear は処理とデータの単位である。並列実行に必要な Meta な機能を関数型言語における Monad の原理に基づいて、実現する。今回設計した Gears OS では Code Gear, Data Gear それぞれに Meta Code Gear と Meta Data Gear を対応させる。Code Gear が実行されるとそれに対応する Meta Code Gear が実行され、Meta Computation が行われる。Meta Computation は OS が行うネットワーク管理、メモリ管理等の資源制御を行う。本論文では基本的な機能を設計し、CbC(Continuation based C) で実装する。

Design of Gears OS with Meta Computation based on Monad

Shohei KOKUBO ,[†1] Tatsuki Iha [†2] and Shinji KONO [†2]

We are developing parallel framework using a Code/Data Gear. Code/Data Gear are unit of processing and data. Meta function for parallel execution based on a Monad in Functional Language is used in Geas OS. A Meta Code/Data Gear attached to a Code/Data Gear as a Monad. Meta Computation performs Network Management, Memory Management and more. We show same implemetation of Gears OS using CbC(Continuation based C).

1. Cerium と Alice

本研究室では並列プログラミングフレームワーク Cerium[1] と分散ネットフレームワーク Alice[2] の開発を行なってきた。

Cerium と Alice を開発して得られた知見から Inherent Parallel, Distributed Open Computation をキーワードとして並列分散フレームワーク Gears OS の設計・開発を行う。

Cerium では Task と呼ばれる分割されたプログラムを依存関係に沿って実行することで並列実行を実現する。依存関係はプログラマ自身が意識して記述する必要がある。Task の種類が増えると記述が煩雑になり、プログラマの負担が大きくなる。Task の依存関係がデータの依存関係を正しく保証しない場合があるという問題がある。また、Task の取り扱うデータには型情報がない。汎用ポインタをキャストして利用するしかなく、型の検査が行われていない。Cerium は C++ で実装されているが、オブジェクトと並列処理が直接対応していないのでオブジェクト指向で記述する利点が少ない。Cell[3], Many Core CPU, GPU といった様々なプロセッサをサポートしている。しかし、それぞれの環境でプログラムを高速に動作させるためにはそれぞれに合わせた実行機構を必要としている。

Alice は本研究室で開発を行なっている分散管理フレームワークである。Alice では処理の単位である Code Segment, データの単位である Data Segment を用いてプログラムを記述[4] する。Code Segment 使用する Input Data Segment, Output Data Segment を指定することで処理とデータの関係を記述する。Alice は Java で実装されており、実行速度が遅いという問題がある。また、Data Segment にアクセスする API のシンタックスが特殊で Alice を用いてプログラムを作成するためには慣れが必要になる。

2. Gears OS

Cerium と Alice の例題から Code の単位だけでなく、Data の単位も必要であることがわかった。Gears OS では Gear という単位を用いてプログラムを Code Gear, Data Gear に細かく分割する。Code Gear は Input Data Gear から Output Data Gear を生成する。Input と Output の関係から Code Gear 同士の依存関係を解決し、並列実行するフレームワークの開発を行う。Code Gear はそれに接続された Data Gear のみを扱う。Code/Data Gear 同士の関係は Meta

†1 琉球大学大学院理工学研究科情報工学専攻
Interdisciplinary Information Engineering, Graduate School of Engineering and Science, University of the Ryukyus.
†2 琉球大学工学部情報工学科
Information Engineering, University of the Ryukyus.

ⓒ 2015 Information Processing Society of Japan

Code/Data Gear によって表現される。この Meta Code/Data Gear を用いることで機能やデータ自体を拡張することができる。

Cerium は初め Cell 向けのフレームワークとして設計されたという経緯からプロセッサ毎の実行機構が異なる。Gears OS では Many Core CPU, GPU をはじめとする様々なプロセッサを同等な実行機構でサポートする。

本研究室で開発している CbC(Continuation based C)[5] を用いて、Gears OS を実装する。CbC はプログラムを Code Segment, Data Segment という単位で記述する。CbC において Code Segment 間の処理の移動は function call ではなく、goto を用いた軽量継続を用いる。CbC のコンパイルには LLVM をバックエンドとしたコンパイラ[6] を用いる。

従来の OS が行う排他制御、メモリ管理、並列実行などは Meta Computation に相当する。関数型言語では Meta Computation に Monad を用いる手法[7] がある。Gears OS では、Meta Code/Data Gear を Monad として定義し、Meta Computation を実現する。

Gears OS では並列実行をサポートするだけでなく、信頼性も確保する。そのために Gears OS を用いて作成されたプログラムに対する Model Checking を行う機能[8] を提供する。並列プログラムに Model Checking を行うことでそのプログラムがとり得る状態を列挙する。これにより、並列実行時のデッドロックの検出などを行うことでプログラムの信頼性を確保する。Model Checking も Meta Code/Data Gear を用いて実現する。

Gears OS は Many Core CPU, GPU といった並列実行環境に合わせた設計・実装を行う。また、接続する Gear を変更することでプログラムの振る舞いを変更することを可能にする柔軟性、Monad に基づくメタ計算による並行制御、Model Checking を用いた信頼性の確保を目的とする。Gears OS の構成は図 1 の通りである。

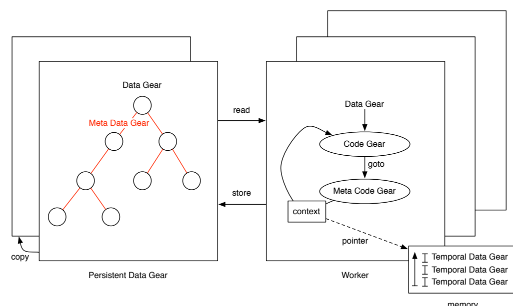

図 1 Gears OS の構成

3. Monad とメタ計算

関数型言語では入力から出力を得る通常の計算の他にメタ計算と呼ばれるものがある。メタ計算の例として、失敗する可能性がある計算、並行処理、入出力などの副作用、メモリ管理などがある。メタ計算の理論的な表現として、Monad を用いることが Moggi らにより提案[7] されている。Gears OS ではメタ計算を表現するのに、Monad と軽量継続を用いる。

Monad は関数が返す通常の値を含むデータ構造であり、メタ計算を表現するのに必要な情報を格納している。失敗する可能性がある計算の場合は、計算が失敗したかどうかのフラグが Monad に含まれる。並行処理の場合は、Monad は可能な計算の interleaving(並び替え) になるが、実際に並び替えを持っているわけではなく、マルチプロセッサで実行する環境そのものが Monad に対応する。

通常の関数を Monad を返すように変更することにより、メタ関数が得られる。逆に Monad の中にある通常の戻り値のみに着目すると通常の関数に戻る。このように、Monad を用いたメタ計算の表現では通常の計算とメタ計算が一対一に対応する。

一般的には複数の Monad の組み合わせが Monad になることを示すのは難しい。Gears OS では Code と Data を分離して、Code から他の Code への呼び出しを継続を用いて行う。Gears OS での Monad は Meta Code と Meta Data になる。

4. Code Gear と Data Gear

Gears OS ではプログラムの実行単位として様々な Gear を使う。Gear が平行実行の単位、データ分割、Gear 間の接続などになる。

Code Gear はプログラムの実行コードそのものであり、OpenCL[9]/CUDA[10] の kernel に相当する。Code Gear は複数の Data Gear を参照し、一つまたは複数の Data Gear に書き込む。Code Gear は接続された Data Gear 以外には触らない。Code Gear はサブルーチンコールではないので、呼び出し元に戻る概念はない。その代わりに、次に実行する Code Gear を指定する機能 (軽量継続) を持つ。

Data Gear には、int や文字列などの Primitive Data Type が入る。自分が持っていない Code Gear, Data Gear は名前で指し示す。

Gear の特徴の一つはその処理が Code Gear, Data Gear に閉じていることにある。これにより、Code Gear の実行時間、メモリ使用量を予測可能なものにする。

Code Gear, Data Gear はポインタを直接には扱わない。これにより、Code と Data の分離性を上げて、ポインタ関連のセキュリティフローを防止する。

Code Gear, Data Gear はそれぞれ関係を持っている。例えば、ある Code Gear の次に実行される Code Gear、全体で木構造を持つ Data Gear などである。Gear の関連付けは Meta Gear を通して行う。Meta Gear は、いままでの OS におけるライブラリや内部のデータ構造に相当する。なので、Meta Gear は Code Gear, Data Gear へのポインタを持っている。

5. 継　　続

ある Code Gear から継続するときには、次に実行する Code Gear を名前で指定する。Meta Code Gear が名前を解釈して、処理を対応する Code Gear に引き渡す。これらは、従来の OS の Dynamic Loading Library や Command 呼び出しに対応する。名前と Code Gear へのポインタの対応は Meta Data Gear に格納される。この Meta Data Gear を Context と呼ぶことにする。これは従来の OS の Process や Thread に対応する。

Context には以下のようなものが格納される。

- Code Gear の名前とポインタの対応表
- Data Gear の Allocation 用の情報
- Code Gear が参照する Data Gear へのポインタ
- Data Gear に格納される Data Type の情報

```
/* Context definition */

#define ALLOCATE_SIZE 1024

enum Code {
    Code1,
    Code2,
    Allocator,
};

enum UniqueData {
    Allocate,
    Tree,
};

struct Context {
    int codeNum;
    __code (**code) (struct Context *);
    void* heap_start;
    void* heap;
    long dataSize;
    int dataNum;
    union Data **data;
};

union Data {
    struct Tree {
        union Data* root;
        union Data* current;
        union Data* prev;
        int result;
    } tree;
    struct Node {
        int key;
        int value;
        enum Color {
            Red,
            Black,
        } color;
        union Data* left;
        union Data* right;
    } node;
    struct Allocate {
        long size;
        enum Code next;
    } allocate;
};
```

ソースコード 1　Context

Code Gear の名前とポインタの対応表

Code Gear の名前とポインタの対応は enum と関数ポインタによって表現される。これにより、実行時に比較ルーチンなどを動的に変更することが可能になる。

Data Gear の Allocation 用の情報

Context の生成時にある程度の領域を確保する。Context にはその領域へのポインタとサイズが格納されている。そのポインタを必要な Data Gear のサイズに応じて、インクリメントすることによって Data Gear の Allocation を実現する。

Code Gear が参照する Data Gear へのポインタ

Context には Data Gear へのポインタが格納されている。Code Gear は Context を通して Data Gear へアクセスする。

Data Gear に格納される Data Type の情報

Data Gear は union と struct によって表現される。Context には Data Gear の Data Type の情報が格納されている。この情報から確保する Data Gear のサイズなどを決定する。

```
#include <stdlib.h>

#include "context.h"

extern __code code1(struct Context*);
extern __code code2(struct Context*);
extern __code allocate(struct Context*);

__code initContext(struct Context* context) {
    context->dataSize = sizeof(union Data)*
          ALLOCATE_SIZE;
    context->code = malloc(sizeof(__code*)*
          ALLOCATE_SIZE);
    context->data = malloc(sizeof(union Data*)*
          ALLOCATE_SIZE);
    context->heap_start = malloc(context->dataSize);
    context->heap = context->heap_start;

    context->codeNum = 3;
    context->code[Code1] = code1;
    context->code[Code2] = code2;
    context->code[Allocator] = allocate;

    context->dataNum = 2;
    context->data[Allocate] = context->heap;
    context->heap += sizeof(struct Allocate);
    context->data[Tree] = context->heap;
    context->heap += sizeof(struct Tree);

    context->root = 0;
```

ⓒ 2015 Information Processing Society of Japan

```
    context->current = 0;
}
```
ソースコード 2　initContext

6. Persistent Data Gear

Data Gear の管理には木構造を用いる。この木構造は非破壊で構築される。非破壊的木構造では、図 2 のように編集元の木構造を破壊することなく新しい木構造を構成する。破壊的木構造と異なりロックの必要がなく、平行して読み書き、参照を行うことが可能である。また、変更前の木構造をすべて保持しているので過去のデータにアクセスすることができる。

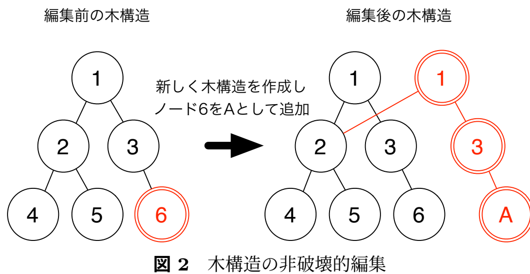

図 2　木構造の非破壊的編集

7. Allocator

Gears OS では Context の生成時にある程度の領域を確保し、その領域を指すポインタをインクリメントすることで Allocation を実現する。

Context には Allocation 用の Data Gear が格納されている。この Data Gear に確保するサイズと確保後に接続する Code Gear の名前を書き込み、Allocation を行う Code Gear に接続することで必要な領域を確保する。

```
// Code Gear
__code code1(struct Context* context) {
    context->data[Allocate]->allocate.size = sizeof(
        struct Node);
    context->data[Allocate]->allocate.next = Code2;
    goto meta(context, Allocate);
}

// Meta Code Gear
__code meta(struct Context* context, enum Code next)
    {
    // meta computation
    goto (context->code[next])(context);
}

// Meta Code Gear
__code allocate(struct Context* context) {
    context->data[++context->dataNum] = context->heap
        ;
    context->heap += context->data[Allocate]->
        allocate.size;
    goto (context->code[context->data[Allocate]->
        allocate.next])(context);
}

// Code Gear
__code code2(struct Context* context) {
    // processing content
}
```
ソースコード 3　Allocator

ソースコード 3 では Code Gear である code1 でポインタを扱っており、Code Gear でポインタを扱わないという設計思想に合っていない。そこで、ソースコード 4 をソースコード 3 として解釈するようにコンパイラを改良する。

```
// Code Gear
__code code1(Allocate allocate) {
    allocate.size = sizeof(long);
    allocate.next = Code2;
    goto Allocate(allocate); // goes through meta
}

// Meta Code Gear
__code meta(struct Context* context, enum Code next)
    {
    // meta computation
    goto (context->code[next])(context);
}

// Meta Code Gear
__code allocate(struct Context* context) {
    context->data[++context->dataNum] = context->heap
        ;
    context->heap += context->data[Allocate]->
        allocate.size;
    goto (context->code[context->data[Allocate]->
        allocate.next])(context);
}

// Code Gear
__code code2(Allocate allocate, Count count) {
    // processing
}
```
ソースコード 4　SyntaxSugar

8. List

通常 List は要素と次へのポインタを持つ構造体で表現される。Gears OS の場合、Meta レベル以外でポインタは扱わないので図 3 のように任意の要素を持つ Data Gear と次へのポインタを持つ Meta Data Gear の組によって List は表現される。

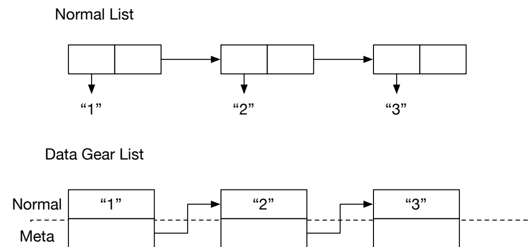

図 3　List の表現

9. Synchronized Queue

Gears OS では List を表現する Code/Data Gear に CAS(Compare and Swap) を行う Meta Code/-Data Gear を接続することで Synchronized Queue を実現する。Gears OS の機能は状態遷移図とクラスダイアグラムを組み合わせた図で表現する。この図を GearBox と呼ぶことにする。図 4 は Synchronized Queue の GearBox である。M:receiver/sender が CAS を行う Meta Code Gear となる。

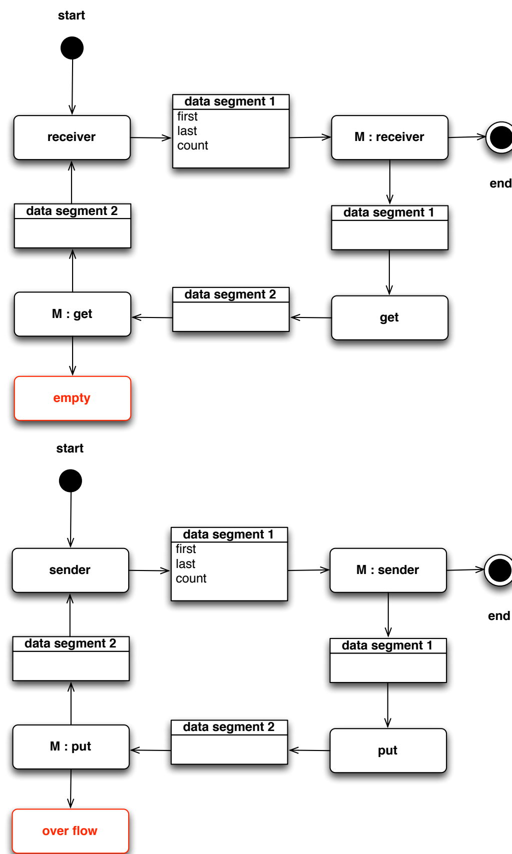

図 4 Synchronized Queue

10. 比　　較

Cerium/Alice, OpenCL/CUDA, 従来の OS との比較を以下に示す。

Cerium/Alice

Gears OS の Code Gear は Cerium の Task, Context は HTask に相当する。Cerium とは異なり、Gears OS は処理とデータが分離している。Gears OS では分離したデータを Data Gear と呼称する。これは Alice の Data Segment と同等のものである。Gears OS では Alice と同様に Code と Data の関係から依存関係を解決する。

Alice は Data Segment を MessagePack[11] を利用して通信することで分散実行を実現する。Gears OS 上での分散実行も Alice に沿って設計・実装する。

OpenCL/CUDA

Code Gear は OpenCL/CUDA の kernel に相当する。OpenCL/CUDA には Data Gear に相当する仕組みはない。接続された複数の Code Gear は接続された順番通りに実行される。これは、OpenCL の CommandQueue, CUDA の Stream と同等のものである。OpenCL/CUDA では kernel の依存関係を複雑に記述する必要があるが、Gears OS では Code と Data の関係から自動的に依存関係を解決する。

従来の OS

従来の OS が行なってきたネットワーク管理、メモリ管理、平行制御などのメタな部分を Gears OS では Meta Code Gear, Meta Data Gear を用いて行う。このメタ計算は Monad に基づいて実現される。

11. ま と め

Gears OS は Inherent Parallel をキーワードとして、Gears OS 上で実行されるプログラムが自動的に並列で処理されるように設計した。Gear を他の Gear に接続することで機能およびデータの拡張を行える柔軟性を持つ。Meta な機能や並行制御を関数型言語における Monad に基づいて実現する。また、機能として Model Checking を持ち、Gears OS 上で実行されるプログラムの信頼性を保証する。本論文では必要な機能の一部である Context, Allocator, List, Non-Destructive Red-Black Tree を CbC を用いて実装した。今後は、Synchronized Queue, Woker を実装する予定である。これにより、Cerium と同等の例題を動かすことが可能となる。例題としては Bitonic Sort, Word Count を予定している。例題が動くことを確認し次第、Gears OS の測定・評価を行う。

参 考 文 献

1) 宮國　渡，河野真治，神里　晃，杉山千秋：Cell 用の Fine-grain Task Manager の実装，情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS) (2008).
2) 赤嶺一樹，河野真治：DataSegment API を用いた分散フレームワークの設計，日本ソフトウェア科学会第 28 回大会論文集 (2011).
3) Sony Corporation: Cell broadband engine architecture (2005).
4) 河野真治，杉本　優：Code Segment と Data Segment によるプログラミング手法，第 54 回プログラミング・シンポジウム (2013).
5) 河野真治，島袋　仁：C with Continuation と、

その PlayStation への応用，情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS) (2000).
6) 徳森海斗，河野真治：Continuation based C の LLVM/clang 3.5 上の実装について，情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS) (2014).
7) Moggi, E.: Computational lambda-calculus and monads, *Proceedings of the Fourth Annual Symposium on Logic in computer science* (1989).
8) 下地篤樹，河野真治：線形時相論理による Continuation based C プログラムの検証，情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS) (2007).
9) Aaftab Munshi, Khronos OpenCL Working Group: *The OpenCL Specification Version 1.0* (2007).
10) : CUDA, https://developer.nvidia.com/category/zone/cuda-zone/.
11) : MessagePack, http://msgpack.org/.

ⓒ 2015 Information Processing Society of Japan