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。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、実現することにした。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 を付属させる。Code Gear が実行されるとそれに属する 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 CbC(Continuation based C) で実装し、評価する。

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

1. Gears OS

並列実行は Code の並列実行だけでなく、Data の単位が重要である。本研究では Code Gear, Data Gear という単位で細かく分割し、依存関係を記述することで並列実行するフレームワーク Gears OS の開発を行なっている。Code Gear, Data Gear はそれぞれ他の Code Gear, Data Gear に接続することで機能や Data 自体を拡張することが可能である。

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。Gears OS では、Meta な機能を関数型言語における Monad に基づいて実現する。

Gears OS を用いて作成されたプログラムに対する Model Checking を行う機能を提供する。これにより、生成されたプログラムの信頼性を保証する。

Gears OS は Gear を継続することによる柔軟性、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。

2. 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。

3. 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、Cuda 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

†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.

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。

4. 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;
};
```

Code 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;
    context->current = 0;
}
```

Code 2　initContext

5. 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、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。

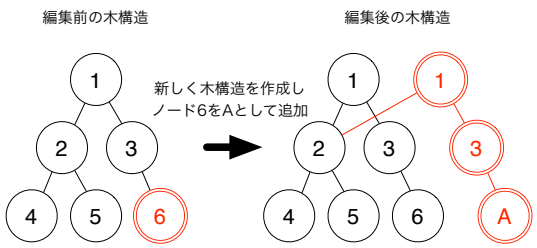

図 1　木構造の非破壊的編集

6. 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 code1(struct Context* context) {
    context->data[Allocate]->allocate.size = sizeof(
        struct Node);
    context->data[Allocate]->allocate.next = Code2;
    goto Allocate(context);
}

__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 code2(struct Context* context) {
    // processing content
}
```

Code 3　Allocator

7.

参　考　文　献

1) Yasuhiko OGATA, Toshio Endo, Naoya MARUYAMA, Satoshi MATSUOKA: 性能モデルに基づく CPU 及び GPU を併用する効率的な FFT ライブラリ, 情報処理学会論文誌コンピューティングシステム (2008).
2) Sony Corporation: Cell broadband engine architecture (2005).
3) : SourceForge.JP: Cerium Rendering Engine, https//sourceforge.jp/projects/cerium/.
4) Cédric Augonnet, Samuel Thibault, Raymond Namyst, Pierre-André Wacrenier: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures, *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, Vol.23, pp. 187–198 (2011).
5) 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 (2008).
6) Chiaki SUGIYAMA: SceneGraph と StatePattern を用いたゲームフレームワークの設計と実装 (2008).
7)　金城裕, 河野真治, 多賀野海人, 小林佑亮 (琉球大学)：ゲームフレームワーク Cerium TaskManager の改良, 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) (2011).
8) 當眞大千, 河野真治：Cerium Task Manager におけるマルチコア上での並列実行機構の実装, 第 53 回プログラミング・シンポジウム (2012).
9) Aaftab Munshi, Khronos OpenCL Working Group: *The OpenCL Specification Version 1.0* (2007).
10) Khronos OpenCL Working Group: *OpenCL 1.2 Reference Pages* (2012).
11) : CUDA, https://developer.nvidia.com/category/zone/cuda-

zone/.