Gears OS におけるコード記述

宮城光希[†1]　河野真治[†2]

Gears OS は Continuation based C によってアプリケーションと OS そのものを記述する。OS の下ではプログラムの記述は通常の処理の他に、メモリ管理、スレッドの待ち合わせやネットワークの管理、エラーハンドリング等の記述しなければならない処理が存在する。これらの計算をメタ計算と呼ぶ。メタ計算を通常の計算から切り離して記述するために、Code Gear、Data Gear という単位を提案している。Code Gear、Data Gear にはそれぞれメタレベルの単位である Meta Code Gear、Meta Data Gear が存在する。Code Gear 間の接続は次の Code Gear の番号と thread structure に相当する context によって行われる。ユーザーレベルではメタ構造を直接見ることはなく、継続を用いた関数型プログラミングに見える。メタレベルから見た Data Gear をユーザーレベルの Code Gear に接続するには stub という Meta Code Gear を用いる。stub と Meta はユーザーレベル Code Gear と Data Gear からスクリプトにより作成される。変換に必要な情報はプログラムを構成する Code Gear と Data Gear の集まりから得る。この集まりを Interface として定義している。本論文では、Interface を用いたプログラミングと、メタ計算の実例を示す。

Mitsuki MIYAGI [†1] and Shinji KONO [†2]

1. Continuation based C (CbC)

CbC は Code Gear という処理の単位を用いて記述するプログラミング言語である。Code Gear は CbC における最も基本的な処理単位である。Code Gear は入力と出力を持ち、CbC では引数が入出力となっている。CbC では Code Gear は __code という型を持つ関数の構文で定義される。ただし、これは __code 型の戻り値を返すという意味ではなく、Code Gear であることを示すフラグである。Code Gear は戻り値を持たないので、関数とは異なり return 文は存在しない。

Code Gear から次の Code Gear への遷移は goto による継続で処理を行い、次の Code Gear へ引数として出力を与える。図は Code Gear 間の処理の流れを表している。図 1 は Code Gear 間の処理の流れを表している。

goto の後に Code Gear 名と引数を並べて、次の Code Gear への遷移を記述する。この goto の行き先を継続と呼ぶ。

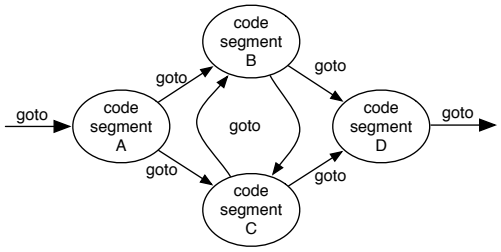

図 1　goto による code gear 間の継続

Scheme の継続と異なり CbC には呼び出し元の環境がないので、この継続は単なる行き先である。したがってこれを軽量継続と呼ぶ。軽量継続により、並列化、ループ制御、関数コールとスタックの操作を意識した最適化がソースコードレベルで行えるようにする。

2. Gears OS

Gears OS は Code Gear、 Data Gear の単位を用いて開発されており、CbC で記述されている。Gears OS では並列実行するための Task を、実行する Code Gear 、実行に必要な Input Data Gear 、Output Data Gear の組で表現する。Gears OS は Input/Output Data Gear の依存関係が解決された Task を並列実行する。Data Gear はデータの単位であり、int

†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.

や文字列などの Primitive Type を持っている。Code Gear は任意の数の Input Data Gear を参照して処理を行い、Output Data Gear を出力し処理を終える。また、接続された Data Gear 以外には参照を行わない。処理やデータの構造が Code Gear、Data Gear に閉じているため、これにより実行時間、メモリ使用量などを予測可能なものにすることが可能になる。

Gears OS では メタ計算 を Meta Code Gear、Meta Data Gear で表現する。Meta Code Gear は通常の Code Gear の直後に遷移され、メタ計算を実行する。Meta Code Gear で OS の機能であるメモリ管理やスレッド管理を行う。

CbC は Code Gear を処理の単位として用いたプログラミング言語であるため、Gears OS の Code Gear を記述するのに適している。

3. Gears OS の構成

Gears OS は以下の要素で構成される。

- Context
- TaskQueue
- TaskManager
- Worker

図 2 に Gears OS の構成図を示す。

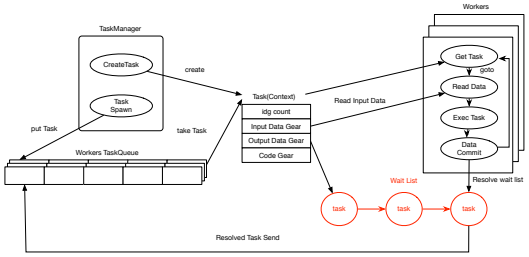

図 2 Gears OS の構成図

Gears OS には Context と呼ばれる接続可能な Code Gear、Data Gear のリスト、Temporal Data Gear のためのメモリ空間等を持っている Meta Data Gear がある。Gears OS は必要な Code Gear、Data Gear に参照したい場合、この Context を通す必要がある。

Temporal Data Gear のためのメモリ空間は Context 毎に異なり、互いに干渉することはできない。Context は Task でもあり、TaskManager によって Context が生成され Worker へ送られる。Worker に渡された Task である Context の Input/Output Data Gear の依存関係が解決されたものから並列実行される。

Gears OS における Task Queue は Synchronized Queue で実現される。Worker は TaskQueue から Task を取得し、実行することで並列処理を行う。

4. interface の記述

interface は呼び出しの引数になる Data Gear の集合であり、そこで呼び出される Code Gear のエントリである。Code Gear、Data Gear に参照するために Context を通す必要があるが、interface を記述することでデータ構造の api と Data Gear を結びつけることが出来、呼び出しが容易になった。

```
typedef struct Stack<Impl>{
    union Data* stack;
    union Data* data;
    union Data* data1;
    __code whenEmpty(...);
    __code clear(Impl* stack,__code next(...));
    __code push(Impl* stack,union Data* data, __code
         next(...));
    __code pop(Impl* stack, __code next(union Data*,
         ...));
    __code pop2(Impl* stack, union Data** data, union
          Data** data1, __code next(union Data**,
         union Data**, ...));
    __code isEmpty(Impl* stack, __code next(...),
         __code whenEmpty(...));
    __code get(Impl* stack, union Data** data, __code
          next(...));
    __code get2(Impl* stack,..., __code next(...));
    __code next(...);
} Stack;
```

Code 1 interface

5. Gearef、GearImpl

Context には Allocation 等で生成した Data Gear へのポインタが格納されている。Code Gear が Context にアクセスする際、ポインタを使用してデータを取り出すため、リスト 2 のようにコードが煩雑になってしまう。そこで Code Gear がデータを参照するための Gearef というマクロを定義した。Gearef に Context と型を渡すことでデータの参照が行える。また impliment のデータを参照する際も、ポインタでの記述が複雑になってしまうため 同様に GearImpl を定義した。GearImpl は Context と interface 名、interface の変数名を指定して参照する。Gearef と GearImpl を用いたコードがリスト 3 である。

```
__code pushSingleLinkedStack_stub(struct Context*
     context) {
    SingleLinkedStack* stack = (SingleLinkedStack*)
         context->data[D_Stack]->Stack.stack->Stack.
         stack;
    Data* data = context->data[D_Stack]->Stack.data;
    enum Code next = context->data[D_Stack]->Stack.
         next;
    goto pushSingleLinkedStack(context, stack, data,
         next);
```

Code 2 Gearef1

```
__code pushSingleLinkedStack_stub(struct Context*
     context) {
    SingleLinkedStack* stack = (SingleLinkedStack*)
```

```
        GearImpl(context, Stack, stack);
    Data* data = Gearef(context, Stack)->data;
    enum Code next = Gearef(context, Stack)->next;
    goto pushSingleLinkedStack(context, stack, data,
        next);
}
```

Code 3　Gearef2

```
    SingleLinkedStack* stack = (SingleLinkedStack*)
        GearImpl(context, Stack, stack);
    enum Code next = Gearef(context, Stack)->next;
    goto clearSingleLinkedStack(context, stack, next
        );
}
```

Code 4　stub

6. stub Code Gear

Code Gear が必要とする Data Gear を取り出す際に Context を通す必要があるが、Context を直接扱うのはセキュリティ上好ましくない。そこで Context から必要なデータを取り出して Code Gear に接続する stub Code Gear を定義し、これを介して間接的に必要な Data Gear にアクセスする。stub Code Gear は Code Gear 毎に生成され、次の Code Gear へと継続する間に挟まれる。

7. Context、stub Code Segment の自動生成

Gears OS では 通常の計算の他に Context や stub などのメタ計算を記述する必要があが、Gears OS を現在の CbC の機能のみを用いて記述するとこのメタ計算の記述を行わなくてはならず、これには多くの労力を要する。この記述を助けるために Context を生成する generate_context と stub Code Gear を生成する generate_stub を perl スクリプトで作成した。

8. stub Code Segment の生成

stub Code Gear は Code Gear 間の継続に挟まれる Code Gear が必要な Data Gear を Context から取り出す処理を行うものである。Code Gear 毎に記述する必要があり、その Code Gear の引数を見て取り出す Data Gear を選択する。stub Code Gear を自動生成することによって Code Gear の記述量を約半分にすることができる。

stub を生成するために generate_stub は指定された cbc ファイルの __code 型である Code Gear を取得し、引数から必要な Data Gear を選択する。generate_stub は引数と interface を照らし合わせ、Gearef または GearImpl を決定する。この時既に stub Code Gear が記述されている Code Gear は無視される。

cbc ファイルから、生成した stub Code Gear を加えて stub を加えたコードに変換を行う。(**??**)

```
__code clearSingleLinkedStack(struct Context *context
     ,struct SingleLinkedStack* stack,enum Code next
     ) {
    stack->top = NULL;
    goto meta(context, next);
}

__code clearSingleLinkedStack_stub(struct Context*
     context) {
```

9. Context の生成

Context は Meta Data Gear に相当し、Code Gear や Data Gear を管理している。Data Gear を取得するために generate context は context の定義を読み宣言されている Data Gear を取得する。

```
struct Context {
    enum Code next;
    struct Worker* worker;
    struct TaskManager* taskManager;
    int codeNum;
    __code (**code) (struct Context*);
    void* heapStart;
    void* heap;
    long heapLimit;
    int dataNum;
    int idgCount; //number of waiting dataGear
    int odg;
    int maxOdg;
    int workerId;
    union Data **data;
};

union Data {
    struct Meta {
        enum DataType type;
        long size;
        struct Queue* wait; // tasks waiting this
            dataGear
    } meta;
    struct Task {
        enum Code code;
        struct Queue* dataGears;
        int idsCount;
    } Task;
    // Stack Interface
    struct Stack {
        union Data* stack;
        union Data* data;
        union Data* data1;
        enum Code whenEmpty;
        enum Code clear;
        enum Code push;
        enum Code pop;
        enum Code isEmpty;
        enum Code get;
        enum Code next;
    } Stack;
    // Stack implementations
    struct SingleLinkedStack {
        struct Element* top;
    } SingleLinkedStack;
    struct Element {
        union Data* data;
        struct Element* next;
    } Element;
    struct Node {
        int key; // comparable data segment
        union Data* value;
        struct Node* left;
```

```
        struct Node* right;
        // need to balancing
        enum Color {
            Red,
            Black,
        } color;
    } Node;
}; // union Data end this is necessary for context
     generator
```

Code 5　context

Code Gear の取得は指定された stub を加えたコードから __code 型を見て行う。取得した Code/Data Gear の enum の定義は enumCode.h、enumData.h に生成される。

Code/Data Gear の名前とポインタの対応は generate_context によって生成される enum Code、enum Data を指定することで接続を行う。また、generate context は取得した Code/Data Gear から Context の生成を行うコードも生成する。

Context には Allocation 等で生成した 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 の Allocation を行うコードは dataGearInit.c に生成される。

Data Gear は union Data とその中の 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。この情報から確保される Data Gear のサイズなどを決定する。

```
#include <stdlib.h>

#include "../context.h"

void initContext(struct Context* context) {
    context->heapLimit = sizeof(union Data)*
        ALLOCATE_SIZE;
    context->code = (__code(**) (struct Context*))
        NEWN(ALLOCATE_SIZE, void*);
    context->data = NEWN(ALLOCATE_SIZE, union Data*);
    context->heapStart = NEWN(context->heapLimit,
        char);
    context->heap = context->heapStart;

    context->code[C_clearSingleLinkedStack] =
        clearSingleLinkedStack_stub;
    context->code[C_exit_code] = exit_code_stub;
    context->code[C_getSingleLinkedStack] =
        getSingleLinkedStack_stub;
    context->code[C_isEmptySingleLinkedStack] =
        isEmptySingleLinkedStack_stub;
    context->code[C_popSingleLinkedStack] =
        popSingleLinkedStack_stub;
    context->code[C_pushSingleLinkedStack] =
        pushSingleLinkedStack_stub;
    context->code[C_stack_test1] = stack_test1_stub;
    context->code[C_stack_test2] = stack_test2_stub;
    context->code[C_stack_test3] = stack_test3_stub;
    context->code[C_stack_test4] = stack_test4_stub;
    context->code[C_start_code] = start_code_stub;

#include "dataGearInit.c"

}
```

```
__code meta(struct Context* context, enum Code next)
     {
    // printf("meta %d\n",next);
    goto (context->code[next])(context);
}

__code start_code(struct Context* context) {
    goto meta(context, context->next);
}

__code start_code_stub(struct Context* context) {
    goto start_code(context);
}

__code exit_code(struct Context* context) {
    free(context->code);
    free(context->data);
    free(context->heapStart);
    goto exit(0);
}

__code exit_code_stub(struct Context* context) {
    goto exit_code(context);
}

// end context_c
```

Code 6　initcontext

10. 今後の課題

本研究では LLVM/Clang のデバッグ、interface の記述、CbC ファイルから Gears OS の記述に必要な Context と stub の生成を行う perl スクリプトの生成を行なった。これにより Gears OS のコードの煩雑さは改善され、ユーザーは Context への接続を意識する必要がなくなった。

今後の課題は Code Gear からメタ計算を行う meta Code Gear を生成できるようにし、ユーザーがメタレベルの処理を意識せずにコードを記述できるようにする。また、今回 perl スクリプトによって Context や stub の生成を行なったが、LLVM/clang 上で実装しコンパイラで直接 CbC を実行できるようにすることを目的とする。

参　考　文　献

1) 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).
2) 小久保翔平，伊波立樹，河野真治：Monad に基づくメタ計算を基本とする Gears OS の設計，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) (2015).
3) TOKKMORI, K. and KONO, S.: Implementing Continuation based language in LLVM and Clang, *LOLA 2015* (2015).