Code Gear と Data Gear を持つ Gears OS の設計

宮城 光希[a)]　河野 真治[1,b)]

概要：現代の OS では拡張性と信頼性を両立させることが要求されている。信頼性をノーマルレベルの計算に対して保証し、拡張性をメタレベルの計算で実現することを目標に Gears OS を設計中である。Gears OS は Continuation based C(CbC) 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。CbC はこの Code Gear と Data Gear の単位でプログラムを記述する。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。

キーワード：OS, プログラミング言語, コンパイラ, CbC, Gears OS

1. メタ計算の重要性

プログラムを記述する際、ノーマルレベルの処理の他に、メモリ管理、スレッド管理、CPU や GPU の資源管理等、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。

従来の OS では、メタ計算はシステムコールやライブラリーコールの単位で行われる。実行時にメタ計算の変更を行う場合には、OS 内部のパラメータの変更を使用し、実行されるユーザープログラム自体への変更は限定的である。しかし、メタ計算は性能測定あるいはプログラム検証、さらに並列分散計算のチューニングなど細かい処理が必要で実際のシステムコール単位では不十分である。例えば、モデル検査ではアセンブラあるいはバイトコード、インタプリタレベルでのメタ計算が必要になる。しかし、バイトコードレベルでは粒度が細かすぎて扱いが困難になっている。具体的にはメタ計算の実行時間が大きくなってしまう。

メタ計算を通常の計算から切り離して記述する

[1] 情報処理学会
[a)] mir3636@cr.ie.u-ryukyu.ac.jp
[b)] kono@ie.u-ryukyu.ac.jp

ためには処理を細かく分割する必要がある。しかし、関数やクラスなどの単位は容易に分割できない。そこで当研究室ではメタ計算を柔軟に記述するためのプログラミング言語の単位として Code Gear、Data Gear という単位を提案している。これによりシステムコードよりも細かくバイトコードよりも大きなメタ計算の単位を提供できる。

Code Gear は処理の単位である。関数に比べて細かく分割されているのでメタ計算をより柔軟に記述できる。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 が存在し、これらを用いてメタ計算を実現する。

Continuation based C (CbC)[2] はこの Code Gear 単位を用いたプログラミング言語として開発している。

CbC は軽量継続による遷移を行うので、継続前の Code Gear に戻ることはなく、状態遷移ベースのプログラミングに適している。

また、当研究室で開発している Gears OS[3] 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。CbC での記述はメタ計算を含まないノーマルレベルでの記述と、Code Gear、Data Gear の記述を含むメタレベルの記述の 2 種類がある。メタレベルでもさらに、メタ計算を用いることが可能になっている。この 2 つのレベルはプログラミング言語レベルでの変換として実現される。CbC は LLVM[1] 上で実装されており、メタレベルでの変換系は本論文では、Perl による変換スクリプトにより実装されている。

Code Gear と Data Gear は Interface と呼ばれるまとまりとして記述される。Interface は使用される Data Gear の定義と、それに対する操作を行う Code Gear の集合である。Interface は複数の実装を持つことができ、Meta Data Gear によって定義される。Interface の操作に対応する Code Gear の引数は Interface に定義されている Data Gear を通して行われる。

従来の関数呼び出しでは引数をスタック上に構成し、関数の実装アドレスを Call する。Gears OS では引数は Context 上に用意された Interface の Data Gear に格納され、操作に対応する Code Gear に goto する。Context とは使用される Code Gear と Data Gear を全て格納している Meta Data Gear である。これは従来のスレッド構造体に対応する。つまり Gears OS では従来はコンパイラが定義する ABI(Application Binary Interface) を Meta Data Gear として CbC で表現し、メタ計算として操作することができる。

ノーマルレベルでは Context を直接見ることはできず、引数は Code Gear の引数を明示する必要がある。この時に呼び出し側の引数を不定長引数として追加する構文を CbC に追加した。これにより Interface 間の呼び出しを簡潔に記述することが出来るようになった。メタレベルでは Code Gear の引数は単一または複数の Data Gear として見ることができる。これは Context を直接操作することができることを意味する。この部分はノーマルレベルの Code Gear を呼び出す stub として生成される。ノーマルレベルでの goto 文はメタ計算への goto で置き換えられる。Gears OS でのメタ計算は stub と goto のメタ計算の 2 箇所で実現される。

メタ計算の例としては並列処理があり、Context を切り替えることによって複数のスレッドを実現している。Context を複数の CPU に割り当てることにより並列実行を可能にしている。

本研究では CbC を用いての Gears OS の実装と Gears OS におけるメタ計算 (Context と stub) の自動生成の実装について述べる。

2. 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、C の関数とは異なり 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 へ引数として入出力を与える。図 1 は Code Gear 間の処理の流れを表している。

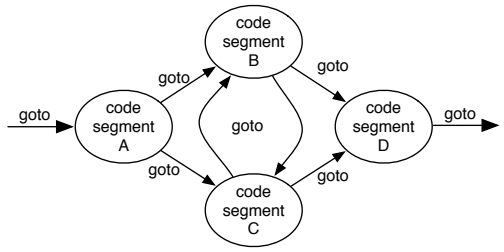

図 1: goto による Code Gear 間の継続

CbC の goto による継続は Scheme の継続と異なり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。

3. Gears OS

Gears OS は Code Gear とデータの単位である Data Gear 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 と、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 CodeGear を並列実行する。Data Gear はデータの単位であり、int 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できる。

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。これを図示したものが図 2 である。

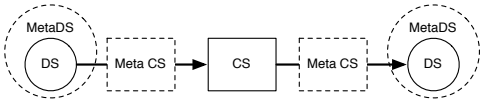

図 2: Gears でのメタ計算

Gears OS 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 を持つ。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。

しかし Context を通常の計算から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 Meta Code Gear である 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 へと継続する前に挿入される。goto による継続を行うと、実際には次の Code Gear の stub Code Gear を呼び出す。

4. Gears OS の構成

Gears OS は以下の要素で構成される。

- Context
- TaskQueue
- TaskManager
- Worker

図 3 に Gears OS の構成図を示す。

Code1 は Context の定義で Code2 は Context の生成である。

```
enum Code {
    C_cs1,
    C_cs2,
};
enum DataType {
    D_Meta,
    D_TaskManager,
```

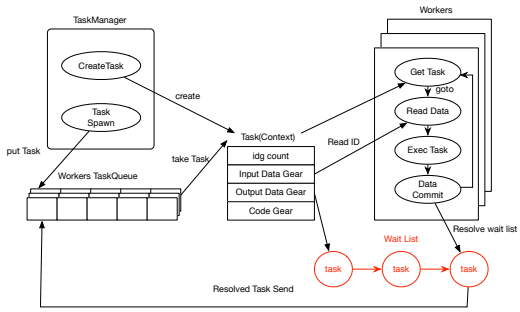

図 3: Gears OS の構成図

```
    ...
};
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;
    int idg;
    int maxIdg;
    int odg;
    int maxOdg;
    int workerId;
    int gpu;
    struct Context* task;
    struct Queue* tasks;
    union Data **data;
};
union Data {
    struct Meta {
        enum DataType type;
        long size;
        struct Queue* wait;
    } meta;
    struct Task {
        enum Code code;
        struct Queue* dataGears;
        int idsCount;
    } Task;
    ...
};
```

Code 1: Context

```
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_cs1] = cs1_stub;
    context->code[C_cs2] = cs2_stub;
    context->code[C_exit_code] =
        exit_code_stub;
    context->code[C_start_code] =
        start_code_stub;

    ALLOC_DATA(context, Context);
    ...
}
```

Code 2: initContext

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。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、Task は通常の OS のスレッドに対応する。Task は実行する Code Gear と Data Gear をすべて持っている。TaskManager によって Context が生成され Task Queue へ挿入する。Gears OS における Task Queue は Synchronized Queue で実現される。Worker は TaskQueue から Task である Context を取得し、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。

5. Interface

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。呼び出される Code Gear の引数となる Data Gear はここで全て定義される。

Code3 は stack の Interface 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る。

```
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 3: Interface

Code4 は stack の Implement の例である。createImpl は関数呼び出しで呼び出され、Implement の初期化と Code Gear のスロットに対応する Code Gear の番号を入れる。

```
Stack* createSingleLinkedStack(struct Context
     * context) {
    struct Stack* stack = new Stack();
    struct SingleLinkedStack*
        singleLinkedStack = new
        SingleLinkedStack();
    stack->stack = (union Data*)
        singleLinkedStack;
    singleLinkedStack->top = NULL;
    stack->push = C_pushSingleLinkedStack;
    stack->pop = C_popSingleLinkedStack;
    stack->pop2 = C_pop2SingleLinkedStack;
    stack->get = C_getSingleLinkedStack;
    stack->get2 = C_get2SingleLinkedStack;
    stack->isEmpty =
        C_isEmptySingleLinkedStack;
    stack->clear = C_clearSingleLinkedStack;
    return stack;
}

__code clearSingleLinkedStack(struct
     SingleLinkedStack* stack,__code next(...)
     ) {
    stack->top = NULL;
    goto next(...);
}

__code pushSingleLinkedStack(struct
     SingleLinkedStack* stack,union Data* data
     , __code next(...)) {
    Element* element = new Element();
    element->next = stack->top;
    element->data = data;
    stack->top = element;
    goto next(...);
}
```

Code 4: Implement

6. stub Code Gear の生成

Gears OS を CbC で実装する上でメタ計算の記述が煩雑であることがわかった。これらのメタ計算を自動生成することにより Gears OS を記述する上においてより良い構文をユーザーに提供することにした。

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 generate stub を Perl スクリプトで作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。また、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。(Code5)

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

```
    goto meta(context, next);
}

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

Code 5: stub Code Gear

7. Context の生成

generate_context は Context.h、Interface.cbc、generate_stub で生成された Impl.cbc を見て Context を生成する Perl スクリプトである。

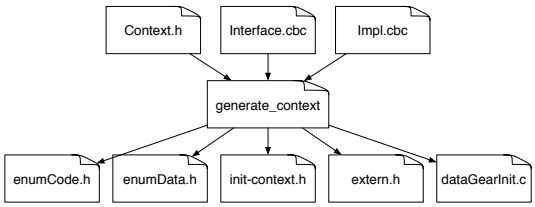

図 4: generate_context による Context の生成

Context は Meta Data Gear に相当し、Code Gear や Data Gear を管理している。

generate_context は context の定義 (Code1) を読み宣言されている Data Gear を取得する。Code Gear の取得は指定された generate_stub で生成されたコードから __code 型を見て行う。取得した Code Gear、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 (Code6) 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_cs1] = cs1_stub;
    context->code[C_cs2] = cs2_stub;
    context->code[C_exit_code] =
        exit_code_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);
}
```

Code 6: 生成された initContext

8. 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] : LLVM documentation.

[2] TOKKMORI, K. and KONO, S.: Implementing Continuation based language in LLVM and Clang, *LOLA 2015* (2015).

[3] 河野真治，伊波立樹，東恩納琢偉：Code Gear、Data Gear に基づく OS のプロトタイプ，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) (2016).