# Gears OS のモジュール化と並列 API

宮城光希[†1]　河野真治[†2]

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 によってアプリケーションと OS そのものを記述する。CbC はこの Code Gear と Data Gear の単位でプログラムを記述する。システムやアプリケーションを記述するために Code Gear と Data Gear を柔軟に再利用する必要がある。このときに機能を接続する API と実装の分離が可能であることが望ましい。Gears OS の信頼性を保証するために、形式化されたモジュールシステムを提供する必要がある。本論文では、Interface を用いたモジュールシステムの説明とその応用としての並列 API について考察する。並列 API は継続を基本とした関数型プログラミングと両立する必要がある。ここでは、CbC の goto 文を拡張した par goto 文を導入する。par goto では並列実行のための実行 Context を生成し、TaskScheduler に登録する。Gears OS での同期機構は Data Gear の待ち合わせとして定義する。メタレベルではこれらの同期機能は CAS とそれを用いて実装した Synchronized Queue になる。これらの Queue も Interface を用いてモジュール化されている。モジュール化の詳細と、有効性について考察する。

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

## 1. OS の拡張性と信頼性の両立

さまざまなコンピュータの信頼性の基本はメモリなどの資源管理を行う OS である。OS の信頼性を保証する事自体が難しいが、時代とともに進歩するハードウェア、サービスに対応して OS 自体が拡張される必要がある。OS は非決定的な実行を持ち、その信頼性を保証するには、従来のテストとデバッグでは不十分であり、テストしきれない部分が残ってしまう。これに対処するためには、証明を用いる方法とプログラムの可能な実行をすべて数え上げるモデル検査を用いる方法がある。モデル検査は無限の状態ではなくても巨大な状態を調べることになり、状態を有限に制限したり状態を抽象化したりする方法が用いられている図 1。

証明やモデル検査を用いて OS を検証する方法はさまざまなものが検討されている。検証は一度ですむものではなく、アプリケーションやサービス、デバイスが新しくなることに検証をやり直す必要がある。つまり信頼性と拡張性を両立させることが重要である。

†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.

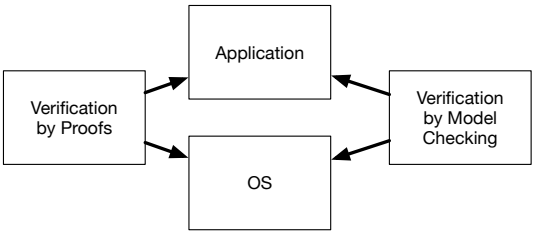


**図 1**: 証明とモデル検査による OS の検証

コンピュータの計算はプログラミング言語で記述されるが、その言語の実行は操作的意味論の定義などで規定される。プログラミング言語で記述される部分をノーマルレベルの計算と呼ぶ。コードが実行される時には、処理系の詳細や使用する資源、あるいは、コードの仕様や型などの言語以外の部分が服属する。これをメタレベルの計算という。この二つのレベルを同じ言語で記述できるようにして、ノーマルレベルの計算の信頼性をメタレベルから保証できるようにしたい。ノーマルレベルでの正当性を保証しつつ、新しく付け加えられたデバイスやプログラムを含む正当性を検証したい。

本論文では、ノーマルレベルとメタレベルを共通して表現できる言語として Continuation based C(以下 CbC)[1] を用いる。CbC は関数呼出時の暗黙の環境 (Environment) を使わずに、コードの単位を行き来で

きる引数付き goto 文 (parametarized goto) を持つ C と互換性のある言語である。これを用いて、Code Gear と Data Gear、さらにそのメタ構造を導入する。これらを用いて、検証された Gears OS を構築したい。図 2。

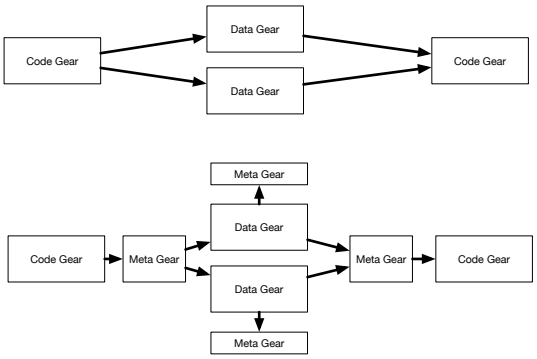


**図 2**: Gears のメタ計算

本論文では、Gears OS の要素である Code Gear、Data Gear、そして、Meta Code Gear 、Meta Data Gear の構成を示す。これらは、CbC に変換されて実行される。Gears OS は、Task Scheduler を CPU や GPU 毎に持ち、一つの Task に対応する Context という Meta Data Gear を使用しながら計算を実行する。

Meta Gear を入れかえることにより、ノーマルレベルの Gear をモデル検査することができるようにしたい。ノーマルレベルでの Code Gear と Data Gear は継続を基本とした関数型プログラミング的な記述に対応する。この記述を定理証明支援系である Agda を用いて直接に証明できるようにしたい。

従来の研究でメタ計算を用いる場合は、関数型言語では Monad を用いる[?]。これは Hakell では実行時の環境を記述する構文として使われている。OS の研究では、メタ計算の記述に型つきアセンブラ (Typed Assembler) を用いることがある[?]。Python や Haskell による記述をノーマルレベルとして採用した OS の検証の研究もある[?,?]。SMIT などのモデル検査を OS の検証に用いた例もある[?]。

本研究で用いる Meta Gear は制限された Monad に相当し、型つきアセンブラよりは大きな表現単位を提供する。Haskell などの関数型プログラミング言語では実行環境が複雑であり、実行時の資源使用を明確にすることができない。CbC はスタック上に隠された環境を持たないので、メタレベルで使用する資源を明確にできるという利点がある。ただし、Gear のプログラミングスタイルは、従来の関数呼出を中心としたものとはかなり異なる。本研究では、Gears の記述をモジュールかするためにインターフェースを導入した。これにより、見通しの良いプログラミングが可能になった。

## 2. メタ計算の重要性

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。

CbC 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。

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、Meta Code Gear、Meta 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[2] 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。

## 3. 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 3 は Code Gear 間の処理の流れを表している。

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。

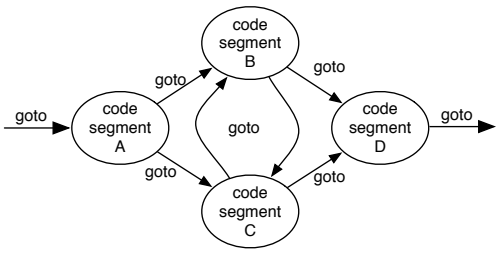


**図 3**: goto による Code Gear 間の継続

## 4. 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 4 である。

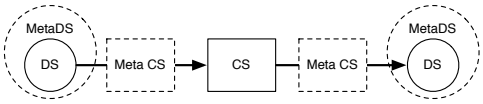


**図 4**: Gears でのメタ計算

Gears OS は Context と呼ばれる使用されるすべての Code Gear、Data Gear 持っている Meta Data Gear を持つ。Gears OS は必要な Code Gear、Data Gear を参照したい場合、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。

## 5. Gears OS の構成

Gears OS は以下の要素で構成される。

- Context
- TaskQueue
- TaskManager
- Worker

図 5 に Gears OS の構成図を示す。

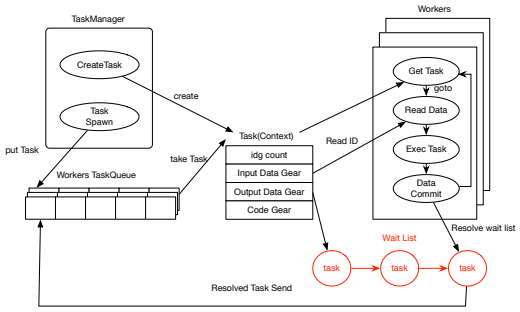


図 5: Gears OS の構成図

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,
    ...
};
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 の定義

```
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: Context の生成

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。

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 は Task を実行する Worker の生成、管理、Task の送信を行う。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、Task の Code Gear を実行し、Output Data Gear の書き出しを行っている。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。

## 6. 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: Stack の 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: SingleLinkedStack の Implement

## 7. 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 を加えたコード

## 8. 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。

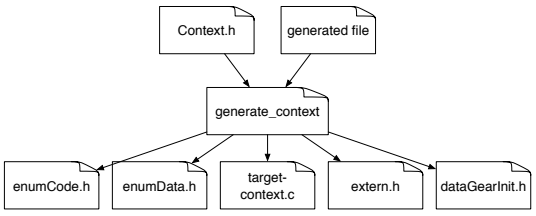


図 6: generate_context による Context の生成

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 (Code**??**) 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。

## 9. Gears OS の並列処理

Gears OS では実行の Task を Code Gear と Input/Output Data Gear の組で表現する。Input/Output Data Gear によって依存関係が決定し、それにそって並列実行を行う。Gears OS では 並列実行する Task を Context で表現する。Context には Task 用の情報として、実行される Code Gear、Input/Output Data Gear の格納場所、待っている Input Data Gear のカウンタ等を持っている Task の Input Data Gear が揃っているかを TaskManager で判断し、 実行可能な Task を Worker に送信する。Worker は送信された Task が指定した Code Gear を実行し、Output Data Gear を書き出す。

## 10. SynchronizedQueue

SynchronizedQueue は Worker の Queue として使用される。Worker の Queue は TaskManager を経由して Task を送信するスレッドと Task を取得する Worker 自身のスレッドで扱われる。そのため SynchronizedQueue はマルチスレッドでもデータの一貫性を保証する Queue を実装する必要がある。

データの一貫性を保証する解決例としての 1 つとしてロックを使った解決方法がある。しかし、ロックを行ってデータを更新した場合、同じ Queue に対して操作を行う際に待ち合わせが発生し、全体の並列度が下がってしまう。そこで、Gears OS ではデータの一貫性を保証するために CAS(Check and Set、Compare and Swap) を利用した Queue を実装している。CAS は値の比較、更新をアトミックに行う命令である。CAS を使う際は更新前の値と更新後の値を渡し、渡された更新前の値を実際に保存されているメモリ番地の値と比較し、同じならデータ競合がないため、データの更新に成功する。異なる場合は他に書き込みがあったとみなされ、値の更新に失敗する。

Gears OS ではこの CAS を行うための Interface を定義している (Code6)。この Interface では、Data Gear 全てを内包している Data 共用体のポインタの値を更新する CAS を定義している (Code6 6 行目)。

```
typedef struct Atomic<Impl>{
    union Data* atomic;
    union Data** ptr;
    union Data* oldData;
    union Data* newData;
    __code checkAndSet(Impl* atomic, union Data** ptr
          , union Data* oldData, union Data* newData,
          __code next(...), __code fail(...));
    __code next(...);
    __code fail(...);
} Atomic;
```

Code 6: AtomicInterface

AtomicInterface での CAS の実際の実装を Code7 に示す。実際の実装では __sync_bool_compare_and_swap 関数を呼び出すことで CAS を行う (Code7 2 行目)。この関数は第一引数に渡されたアドレスに対して第二引数の値から第三引数の値へ CAS を行う。CAS に成功した場合、true を返し、失敗した場合は false を返す。Code7 では CAS に成功した場合と失敗した場合それぞれに対応した Code Gear へ継続する。

```
__code checkAndSetAtomicReference(struct
      AtomicReference* atomic, union Data** ptr,
      union Data* oldData, union Data* newData,
      __code next(...), __code fail(...)) {
    if (__sync_bool_compare_and_swap(ptr, oldData,
          newData)) {
        goto next(...);
    }
    goto fail(...);
}
```

Code 7: CAS の実装

SynchronizedQueue の Data Gear の定義を Code8 に示す。SynchronizedQueue はデータのリストの先頭と、終端のポインタを持っている。Queue を操作する際はこのポインタに対して CAS をすることでデータの挿入と取り出しを行う。

```
struct SynchronizedQueue {
    struct Element* top;
    struct Element* last;
    struct Atomic* atomic;
};

// Singly Linked List element
struct Element {
    union Data* top;
    struct Element* next;
};
```

Code 8: SynchronizedQueue の定義

## 11. 並 列 構 文

Gears OS の並列構文は par goto 文で用意されている。par goto の引数には Input/Output Data Gear と 実行後に継続する __exit を渡す。par goto で生成された Task は __exit に継続することで終了する。par goto 文でも 通常の goto 分と同様にメタへの goto 文へ置き換えられるが、par goto 文では通常の goto 文とは異なるメタへと継続する。Gears OS の Task は Output Data Gear を生成した時点で終了するので、par goto では直接 __exit に継続するのではなく、Output Data Gear への書き出し処理 (Commit) に継続される。

Code Gear は Input に指定した Data Gear が全て書き込まれると実行され、実行した結果を Output に指定した Data Gear に書き出しを行う。Code Gear の引数である__next が持つ引数の Data Gear が Output Data Gear となる。

書き出し処理は Data Gear の Queue から、依存関係にある Task を参照する。参照した Task が持つ実行に必要な Input Data Gear カウンタのデクリメントを行う。カウンタが 0 になると Task が待っている Input Data Gear が揃ったことになり、その Task を TaskManager 経由で 実行される Worker に送信する。

この par goto 文は通常のプログラミングの関数呼び出しのように扱うことができる。

## 12. 比　　較

従来のプログラミングスタイルとの比較。Gears のプログラミングは関数呼出を中心とするプログラミングとはかなり異なる。Gears は関数呼出を禁止しているわけではなく、使用する資源の制御に問題がないなら普通に関数呼出して良い。Linux kernel などでは関数呼出の大半はインライン展開されることを期待してプログラミングされており、関数呼出で予測できないスタックの爆発や CPU 資源の浪費が起きないようにプログラミングされている。Gears では Gears 間のプログラミングは戻り先や使用する資源を明示する必要がある。

goto 文での引数は通常の関数呼出と異なり、スタック (環境) に積むことができない。引数に必要な情報を含む Data Gear を持ち歩くスタイルとなる。一つのインタフェース内部では、これらは共通している。実際、これらはメタレベルでは、Context という Meta Data Gear にすべて格納されている。メタレベルは、Data Gear の詳細な型は使用されない。ノーマルレベルに移行する際に stub Code Gear を通して詳細な型が接続される。

インタフェースを再利用する際には、呼び出すインタフェースが持つ引数は保存される必要がある。これらは、実際には Context 内にあるので自動的に保存されている。ノーマルレベルの記述では、... の部分にその意味が込めれている。これは、可変長引数の... と同じ意味だと考えても良い。ただ、LLVM/GCC レベルでそれを実装するのは比較的難しい。なので、今回は Script による変換を採用している。

ノーマルレベルの記述と関数型プログラミングの記述の比較。Gear は必ず継続を渡す必要がある。これは一段の関数呼出を許しているのと同等である。70 年代の Fortan の関数呼出は決まった場所に戻り先を格納するので再起呼出ができなかったのと同じである。例えば Code Gears 以下のような型を持つ。ここで t は継続の型である。Stack は Stack を受けとる Stack -> t という Code Gear を継続として引数で受けとる。popStack はこの引数を呼び出す。

```
popStack : {a t : Set} -> Stack
             -> (Stack -> t) -> t
```

つまり、Code Gear は制限された関数の形式を持っている。Data Gear は、関数型言語の直積や排他的論理和 (Sum) を含むデータ型に対応する。しかし、一つの Context で実行される Data Gear は、一つの巨大な Sum に含まれるようになっている。これをメタレベルでは、中の型の詳細に立ち入ることなく実行する。

Context はノーマルレベル の Data Gear の他に様々なメタ情報を持つ。例えば、メモリ管理情報や実行される CPU 、あるいは、Task の状態、待ち合わせている Data Gear などである。これらの情報は C やアセンブラのレベルで実装されるのと同時に、通常の Gear のプログラミングにも対応する。例えば、CPU をかそうてきに Gears で記述すればソフトウェア的な並列実行を実現し、実際の GPU を用いれば GPU による並列実行となる。この実行をモデル検査的な状態数え上げに対応させればモデル検査を実行できる。

Haskell などを実行可能仕様記述として用いる OS の検証[?),?)] と、Code Gear を用いる手法は類似¿しているが、Code Gear の場合は、記述を制限し、Code と仕様の対応、さらに Code と資源の対応が明確になる利点がある。

型つきアセンブラ[?)] は、より低レベルの関数型の記述であると言える。アセンブラの記述自体は小さく扱いやすいが、OS レベルあるいはアプリケーションレベルからの距離が大きい。型の整合性を保証するだけでは OS の検証としては不十分なので、証明やモデル検査を用いることになるが、記述量が多いのが、その際に欠点となる。Code Gear は、より大きな単位であり、プログラミングレベルの抽象化が可能になっているので、これらの記述の大きさの欠点をカバーできる可能性がある。

証明手法は、従来では Hoare Logic[?)] のような Post

Condition / Pre Condition を用いる方法が使われている。現在の Gears は、Agda への変換は考えているが、その上の具体的な証明方法はまだ用意されていない。

## 13. 今後の課題

本論文では Code Gear、 Data Gear によって構成される Gears OS のプロトタイプの設計、実装、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。Code Gear 、Data Gear を処理とデータの単位として用いて Gears OS を設計した。Code Gear、Data Gear にはメタ計算を記述するための Meta Code Gear、Meta Data Gear が存在する。メタ計算を Meta Code Gear、によって行うことでメタ計算を階層化して行うことができる。Code Gear は関数より細かく分割されてるためメタ計算を柔軟に記述できる。

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 は複数の実装をもち、Meta Data Gear として定義される。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 するが、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 は使用する Code Gear、Data Gear をすべて格納している Meta Data Gear である。通常の計算から Context を直接扱うことはセキュリティ上好ましくない。このため 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 を定義した。stub Code Gear は Code Gear 毎に記述され、Code Gear 間の遷移に挿入される。

これらのメタ計算の記述は煩雑であるため Perl スクリプトによる自動生成を行なった。これにより Gears OS のコードの煩雑さは改善され、ユーザーレベルではメタ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) TOKKMORI, K. and KONO, S.: Implementing Continuation based language in LLVM and Clang, *LOLA 2015* (2015).
2) : LLVM documentation.
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).