継続を中心とした言語によるOS GearsOS

2019/1/09

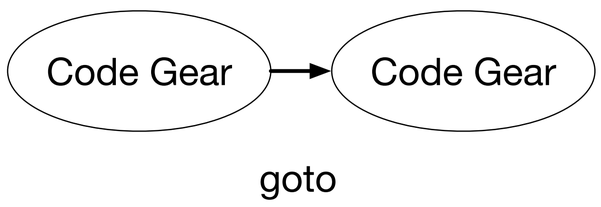
清水 隆博, 河野真治　並列信頼研

メタ計算の重要性

・プログラムの処理には通常の計算の他にメモリ管理などを行うメタ計算が存在する。
・当研究室ではメタ計算を柔軟に記述するために Code Gear、Data Gear という単位を提案している。
・メタ計算を Meta Code/Data Gear を用いて記述する。
・これらを用いることで、検証された Gears OS を構築したい。

Continuation based C (CbC)

・Continuation based C (CbC) は Code Gear を処理の単位としたプログラミング言語として開発している。
・Code Gear は関数呼び出しとは異なり、次の Code Gar へと goto 文によって遷移する。
・この goto 文による遷移を継続と呼ぶ。
・軽量継続はCの関数呼び出しとは異なり, フレームポインタ, スタックポインタの操作によるスタックへの状態保存を行わない
・CbC は C と互換性のある言語なので、C の関数も呼び出せる。




```
__code cg0(struct counter c){
    c->value++;
    goto cg1(c);
}

__code cg1(struct counter c){
    c->value *= 2;
    goto cg2(c);
}
```

Gears OS

・CbCを用いて開発が進められているOS
・現在はCbCにシンタックスシュガーを導入し、いくつかの機能を実装したフレームワークとして実装されている
・Gears OS は Context と呼ばれる全ての Code Gear と Data Gear を持った Data Gear を常に持ち歩いて処理を行う。
・必要な Code Gear、Data Gear は、この Context から取り出して処理を行う。

Meta Code Gear、Meta Data Gear

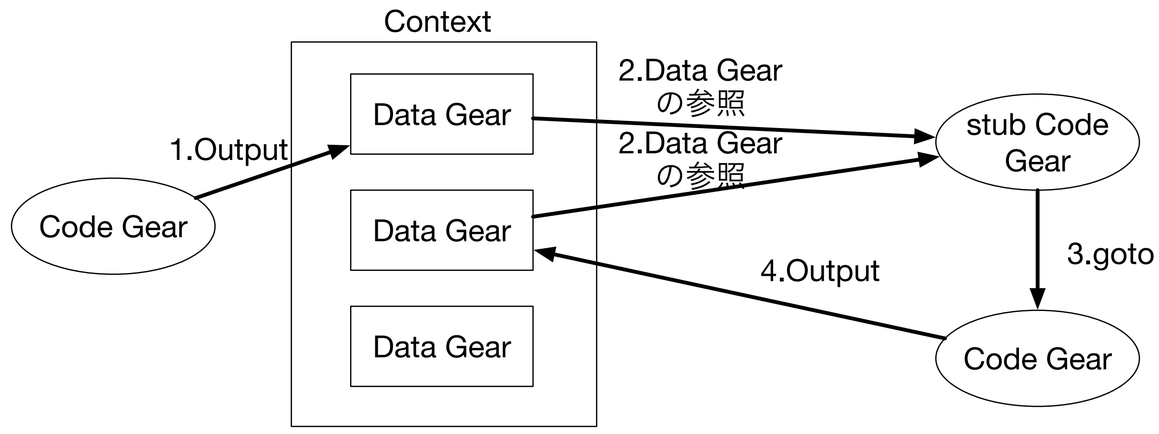
・Gears OS では、通常の計算とメタ計算は切り分けて記述される。
・メタ計算で用いられる Gear を Meta Code/Data Gear と呼ぶ。
・Context は メモリ管理やタスク管理を行う Data Gear であるため Meta Data Gear である。
・Data Gear を扱うためには Context にアクセスする必要がある。
・このアクセスを行う Meta Code Gear を stub Code Gear と呼ぶ。

```
__code code1(Stack* stack) {
    Node* node = new Node();
    goto code2(stack, node);
}
__code code2(Stack* stack, Node* node) {
    Data *data = (union Data*)node;
    Stack->data = data;
    goto code3(stack);
}
```

```
__code code2_stub(struct Context* context) {
    Stack* stack = &context->data[D_Stack]->Stack;
    Node* node = &context->data[D_Node]->Node;
    goto code2(context, stack, node);
}
```

stub Code Gear

・stub Code Gear は Code Gear 間の遷移の間に挿入される。
・これらの Meta Code Gear の記述は煩雑であるため、Meta Code Gear 生成スクリプトを作成した。



Interface

・Interface は Gears OS のモジュール化の仕組みである。
・Interface はある Data Gear と、それに対する操作を行う Code Gear と操作に用いる Data Gear の集合である。
・Java の Interface に対応し、定義することで複数の実装を持つ。

```
typedef struct Stack<Impl>{
        union Data* stack;
        union Data* data;
        __code next(...);
        __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 isEmpty(Impl* stack, __code next(...), __code
whenEmpty(...));
} Stack;
```

Interfaceの実装の型

・Interfaceは具体的な実装を作成する必要がある
・GearsOSでは別のヘッダーファイル形式で実装を書くことで、
　具体的な実装の型を定義できる

```
typedef struct SingleLinkedStack<Type, Isa> impl Stack {
  struct Element* top;
  __code private(__code next(…));
} SingleLinkedStack;
```

・実装側でも独自のCodeGearを定義することができ、プライベートメソッドに該当する概念になる

Interfaceの実装

・Interfaceで定義したCodeGearを実装の型で定義する必要がある
・引数の可変長引数や、次の継続への値渡しはPerlスクリプトでCbCに置換される

```
__code getSingleLinkedStack(struct SingleLinkedStack* stack,
__code next(union Data* data, ...)) {
    if (stack->top) {
        data = stack->top->data;
    } else {
        data = NULL;
    }
    goto next(data, ...);
}
```