Gears OS の CodeGear Management

仲吉菜々子, 河野真治
琉球大学工学部
Nakako Nakayoshi, Shinji KONO
Faculty of Engineering, University of the Ryukyus

Abstract

GearsOS ではすべてのプログラムは Kernel や driver を含めて CodeGear で書かれている。これらは、CodeGear の System DB に格納されるべきである。Microwre OS9 でも module という形でメモリに展開されていた。CodeGear の集合で一つのアプリやサービスが作られる。この CodeGearn の組み合わせを指定する仕組みが必要である。また、CodeGear を実行時に load する機構を作ることにより、現在、clnag の linker で行っている作業抜きで GearsOS を構成できる。これにより、GearsOS の build を簡単にすることができる。高速化が必要な場合は、複数の CodeGear をまとめて最適化して、一つの CodeGearn にする。しかし、実行時のメタ計算がが必要な場合は、それようの hook を用意する必要がある。これの仕組みについて考察する。

In GearsOS, all programs, including kernels and drivers, are written in CodeGear. These should be stored in the System DB of CodeGear. In Microwre OS9, they were deployed in memory as modules. An application or service is created using a set of CodeGear. A mechanism is needed to specify the combination of CodeGear. Additionally, by creating a mechanism to load CodeGear at runtime, GearsOS can be built without the work currently done by the clnag linker. This simplifies the process of building GearsOS. When optimization is necessary, multiple CodeGear can be combined and optimized to create a single CodeGear. However, if runtime metacalculation is necessary, a hook must be provided for it.

1 Gears OS による信頼できるサービス

サービスやアプリケーションの信頼性は、OS とユーザプログラムのように分離することはできない。プログラムの正しさは基本的にはコードの正しさであり、今の OS のようにコードの管理をユーザ空間の問題として放り出す方法には限界がある。

Gears OS では実行単位として codeGear、データ単位として dataGear を使う。さらに、これらは Monad のようにメタ codeGeear やメタ dataGear を持っている。

Gears OS での検証は、Agda[3, 1] を使った GearsAgda を用いる。これらの実行と並列実行は、GearsAgda に対して定義される。当然、すべての GearsAgda の codeGear は、Gears OS に登録されることになる。

OS での証明よる検証はいろいろ行なわれているが、Haskell に近い形式に変換することが多い [4, 2, 7]。Gears OS では、GearsAgda で直接に記述できる点が新しい。

この論文では、Gears OS の codeGear の管理方法について考察する。

2 Normal and Meta computation

Gears OS の基本は、CbC (Continuation based C)[5] であり、input interface と output interface を有限な処理で結ぶものになっている。これは、コンパイラの基本ブロックに大体相当する。これはさらに、GearsAgda の Agda で記述された invariant あるいは、表示的意味論と直結している。

CbC の記述は以下のようなものである。

```
__code startTimer(struct TimerImpl* timer, __code next(...)) {
  struct timeval tv;
  gettimeofday(&tv, NULL);
```

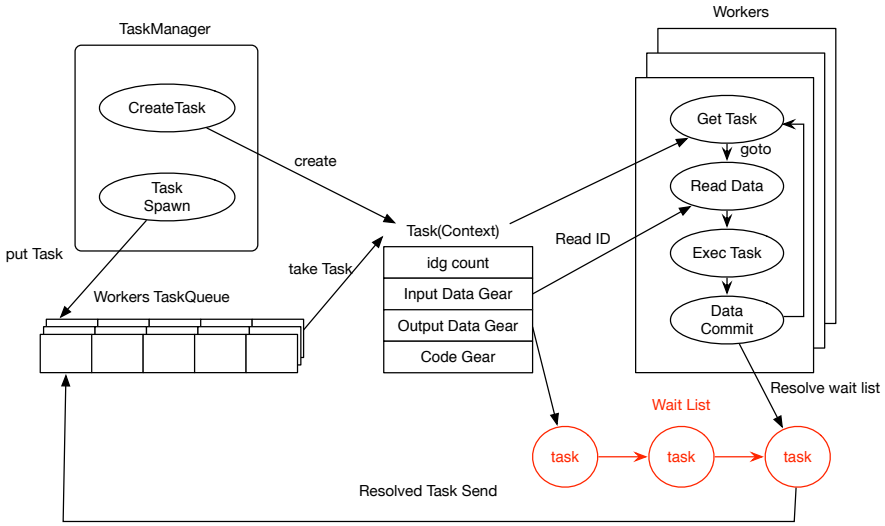

Figure 1: Gears OS

```
    timer->time = tv.tv_sec + (double)tv.tv_usec*1e-6;
    goto next(...);
}
```

next が軽量継続を表している。この codeGear の実行は論理的に割り込まれない。つまり、並行実行はこの単位で行われ定義される。ハードウェアでの並行実行、割り込み処理などは、それにそって実装される必要がある。これらは、GearsAgda では、外部環境へのアクセスがある。この場合は時刻に対するアクセスである。

OS 側からみると、これは詳細のない単なるコードに見える。実行の詳細、つまり、実行に関係するすべての dataGear 、すべての codeGear は、プロセスに相当する Context metaDataGear(図 1) に格納される。

つまり、OS からみた codeGear の実行は、

meta codeGear の番号から、**codeGear** を選ぶ
それを実行し、**continuation** として **shceduler** を指定する
これを **CPU worker** 毎に実行する

ということになる。CPU worker 毎に Context は一つなので、Context は single thread で実行される。これにより、並行実行の単位は codeGear となる。ただし、Context 関の共有データがある場合は意味的なずれがでる場合がある。それは、Context の実行をそうなるように実装することになる。

その実装の正しさは、実装を GearsAgda で記述することより可能になるが、その実装が物理的に一致するかどうかの保証はハードウェアの性質に依存する。

外界との対応もメタ計算になるが、それは GearsAgda によるシミュレーションに対する正しさということになる。

scheduler から codeGear への移行は、以下の meta codeGear で実装される。

```
__code meta(struct Context* context, enum Code next) {
  goto (context->code[next])(context);
}
```

context->code[next] が codeGear の table の呼び出しになるが、これは表がコンパイル時に確定し、直接の呼び出しでは、コンパイラが最適化するので、overhead とはならない。meta が書き換えられている場合は、ここで Context switch が起きることになる。

この Context switch でメモリ空間の切り替えが必要かどうかは application に依存する。もし、codeGear の実行の正しさが証明されているなら、メモリ空間を切り替える必然性はない。実際、Kernel 内や Realtime Monitor では切り替えない方が普通である。

3 証明付きのコード

GearsAgda で記述されていれば、codeGear や dataGear は証明を持つ。これらは単なる型なので実行時には実態を持たない。ただし、それを実行時にチェックすることもできる。assert などと同じ扱いである。

GearsAgda の証明に閉じていれば、その範囲内での信頼性がある。しかし、動的に code が読み込まれる場合はそうはならない。その時には、証明しなおす、簡易あるいは詳細なモデル検査を実施するなどが可能だが、それらが実用的とは限らない。

code は Context の表に登録されるが、システム全体の code は Database で管理される。その管理はファイルシステム上でも良い。code は証明が付属している場合もあるが、それは何らかの形で codeGear Databse に格納される。

現状では、証明は Agda で記述されたデータ構造でしかない。それを codeGear DB に入れても利用する方法はないがいくつかの利用法は考えられる。

証明と **codeGear** の一致を確認する
型整合に使う
code の認証に使う

この大域の codeGear DB は、kernel を含めたすべての codeGear に対するもので全部で共有される。つまり、あらゆる version を含む共有ライブラリとなる。これは全世界で unique な DB としても良い。

4 CbC の codeGear と GearsAgda の違い

CbC の codeGear と dataGear は、普通の C の関数と構造体であり、その意味で不思議なところはない。しかし、normal level では pointer が出てこないのが望ましい。なぜなら、メモリ配置は meta level の問題でプログラムの正しさとは直接関係しないからである。つまり、normal level での構造体は List や 木など以外の再帰的構造でも直接的なポインタ操作は行わない。meta level では、メモリの直接操作例えば malloc や free、あるいは共有データの扱いなどをポインタを通して行う。

meta level でのポインタ操作は、データ構造に対する操作であり、その level に閉じる限り、プログラムの正しさはポインタの実装に依存しない。その意味で、meta codeGear も単なる noraml level の codeGear に過ぎない。meta codeGear の証明あるいは、実装をさらに meta level で行うこともできる。

GearsAgda では、すべては Agda の構造体で表現される。これらは変数と項であり、基本的に複製可能な項である。その意味で、ポインタを持たないと言ってよい。List や木の実装を配列と番号で行うことはまったく実用的ではないので、普通に再帰的なデータ構造を使ってよい。ただし、それを自分自身の codeGear の再帰呼び出しで行うと、codeGear の実行が有界な時間に閉じなくなる。なので、loop は軽量継続を用いて、一旦、外にでることになる。これは、この段階で meta level 的に処理が割り込まれることも意味している。

GearsAgda で処理された項は、Context の dataGear に格納される。また、GearsAgda の codeGear は、Context の code table(図 2) に格納される。どちらも、メモリ的には番号で管理されることになる。GearsAgda の Context は、その意味で、プロセスのメモリ空間そのものを抽象化したものになっている。

つまり、CbC の codeGear と GearsAgda は、normal level ではポインタを使わないのだが、その意味は、CbC と GearsAgda で若干異なる。CbC での実装は、GearsAgda での実装の持つ性質を保存する必要がある。

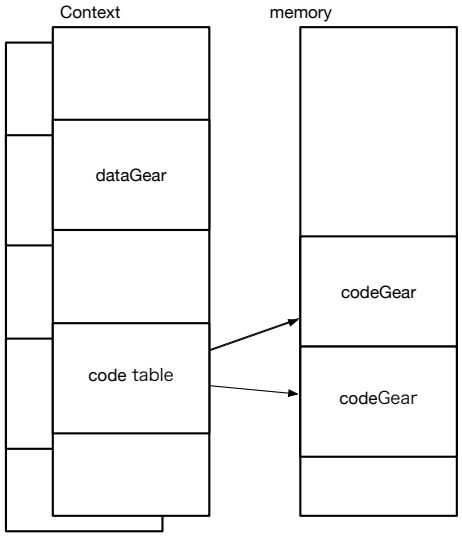

Figure 2: code table

5 Context を通した dataGear と codeGear の連携

Gears OS では、Context は C の構造体であり、一つの Context で使用する dataGear と codeGear は、そこに格納される。複数の Context から、dataGear も codeGear も共有されることがある。その排他制御は、Gears OS の meta codeGear によって行われる。

Gears OS は、interface という構造体でオブジェクト表現していて、これらには、メソッドを格納する配列がある。この配列の添字を他の interface が使用する。なので、interface 間の呼び出しは、interface を表す dataGear とそのメソッドの番号で決定される。

引数は、interface 毎に Context に決まった場所が確保される。これは、通常では stack 上に場所を確保する。しかし、CbC では関数呼び出しは stack を使わないので、このようにする必要がある。これは、もちろん、マルチスレッドな実行では破壊されるおそれがあるが、Context はシングルスレッドで実行することになっているので問題ない。

Context の切り替えは、codeGear の境目で行われるので、Context と実行している codeGear の code table の番号で状態が決定する。つまり、Gears OS では、Task Switch にはレジスタは関係しない。割り込みなどは、codeGear の境界と独立に起きるが、それを meta codeGear が境界で Context switch が起きたのと同じようにすることを保証する。これは一種の遅延処理となる。もちろん、影響がないなら、割り込み中に meta な処理を行ってもよい。これは、本来は CPU などのハードウェアでサポートされるべきかもしれない。

dataGear は、meta な情報として、dataGear の構造定義を番号として持っている。これを使うことにより任意の dataGear の表示を正しく行うことができる。この情報を取り扱うことは meta level からでしか許されないが、CbC 的には特に制限はない。

引数として呼び出される dataGear は、Context 上に前もって確保されているが、実行時に allocate される dataGear も Context から参照される pool に確保される。これの管理は meta dataGear (Gears OS の kernel) が行う。GearsAgda は、この部分は Agda が管理するので meta 的な管理は行われない。しかし、Context を用いた並行実行の場合は、Context 上での管理の問題が生じる。

6 code table

context の初期化の中で、code table に実際の codeGear stub へのポインタが初期化される。

```
context->code[C_add]    = add_stub;
context->code[C_checkAndSetAtomicReference]
  = checkAndSetAtomicReference_stub;
context->code[C_clearSingleLinkedQueue]    = clearSingleLinkedQueue_stub;
context->code[C_clearSynchronizedQueue]    = clearSynchronizedQueue_stub;
context->code[C_code1]    = code1_stub;
context->code[C_createTask1]    = createTask1_stub;
context->code[C_createTask2]    = createTask2_stub;
context->code[C_decrementTaskCountTaskManagerImpl]
  = decrementTaskCountTaskManagerImpl_stub;
context->code[C_exit_code]    = exit_code_stub;
context->code[C_getTaskCPUWorker]    = getTaskCPUWorker_stub;
context->code[C_incrementTaskCountTaskManagerImpl]
  = incrementTaskCountTaskManagerImpl_stub;
```

interface の初期化の中で、この番号が interface を表す dataGear に格納される。

```
Tree* createRedBlackTree(struct Context* context) {
  struct Tree* tree = &ALLOCATE(context, Tree)->Tree;
  struct RedBlackTree* redBlackTree
    = &ALLOCATE(context, RedBlackTree)->RedBlackTree;
  tree->tree = (union Data*)redBlackTree;
  redBlackTree->root = NULL;
  redBlackTree->nodeStack = createSingleLinkedStack(context);
  tree->put = C_putRedBlackTree;
  tree->get = C_getRedBlackTree;
  tree->remove = C_removeRedBlackTree;
  return tree;
}
```

この辺の構造は、GearsAgda 側では簡素化されている。実際、pointer を直接操作しないとかも、厳密に守る必要はなく、GearsAgda と CbC での実装が平行していれば問題はない。

7 codeGear の linkage

Gears OS のあらゆるコードは、codeGear DB のコードの組み合わせになる。しかし、そのためには、Context から詳細なデータを取り出して、実行し、次の codeGear を呼び出す時に Context の正しい場所に dataGear を格納する必要がある。これは codeGear 全体に対して必要になる。

現状では、これは stub としてコンパイル時に生成される。

```
__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(...);
}
```

では、Context 上には以下の構造と、それを呼び出す stub がある。

```
struct Atomic {
   union Data* atomic;
   union Data** ptr;
   union Data* oldData;
   union Data* newData;
   enum Code checkAndSet;
   enum Code next;
   enum Code fail;
} Atomic;

__code checkAndSetAtomicReference_stub(struct Context* context) {
      AtomicReference* atomic = (AtomicReference*)
        GearImpl(context, Atomic, atomic);
      Data** ptr = Gearef(context, Atomic)->ptr;
      Data* oldData = Gearef(context, Atomic)->oldData;
      Data* newData = Gearef(context, Atomic)->newData;
      enum Code next = Gearef(context, Atomic)->next;
      enum Code fail = Gearef(context, Atomic)->fail;
      goto checkAndSetAtomicReference(context, atomic,
         ptr, oldData, newData, next, fail);
}
```

これらは単一の Gears OS 内で整合する必要がある。実際には、これらは、stub が参照する Context 内の offset にすぎない。Atomic は dataGear の巨大な Union になっている。これは、GearsAgda でも状況は同じで、巨大な Sum type になっている。

この Context 内での offset と呼び出す codeGear の番号が一致すれば、Kernel 側で整合性の問題はない。

8 codeGear のコンパイル方法

現在の Gears OS の codeGear は、interface を含む記述を .cbc に書き、それを CbC に変換している。この時に、meta codeGear として stub そして、meta dataGear として Context の定義が生成される。

Context の定義は Application 毎に異なっているが、全部をそろえることも可能である。この変換は Perl script で記述されていて、煩雑になっている。これを codeGear / interface 単位で .o と meta dataGear にできれば、全体の構成と、interface のコンパイルが簡単になると期待される。

ただし、この場合は、Context の中での引数領域の offset 管理、code table の初期化、code 間の遷移を扱う codeGear の番号の指定の書き換えなどが必要となる。

9 static linkage

コンパイル時に codeGear の結合が明らかならば、それは一つの codeGear としてまとめてよい。Context への書き込みもさぼることが可能になる。ただし、時分割実行される場合は、codeGear の切れ目で分割するのと同じ実行が要求される。つまり、割り込み処理などで途中で実行が中断するならば、それはその単位まで実行してから、Context を切り替える必要がある。

これを実現する手法は flag を参照する方法や、コードを書き換える方法あるいは、poling を埋め込むなどの方法が考えられる。整合性が不要であれば途中で止めても問題ないが、その場合は register などの状態を従来の OS のように保持する必要がある。

codeGear 単位で正直に分割コンパイルすると最適化の余地がない。しかし、その場合でも JIT などは可能である。

10 boot codeGear

今の実装では、UEFI から Gears OS が読み出される。Gears OS 自体は、x.v6[6] と同じで、ファイルシステムを含む一体の binary になっている。

codeGear DB を実装すれば、かなりの部分を後からロードすることが可能になる。

11 まとめ

Gears OS と GearsAgda における codeGear の管理方法について考察した。将来的には GearsAgda で記述された証明付き interface のコードを CbC に変換し、それを Gears OS で正しさを確認しながら組み合わせて、meta 計算の変更を可能にしながら実行するシステムを作りたい。

References

[1] AgdaWiki. https://wiki.portal.chalmers.se/agda/main/homepage.

[2] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the fscq file system. In Proceedings of the 25th Symposium on Operating Systems Principles, SOSP '15, pages 18–37, New York, NY, USA, 2015. ACM.

[3] Yoshiki Kinoshita. Agda language. Technical Report PS-2008-014, 独立行政法人産業技術総合研究所, 2008.

[4] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an os microkernel. ACM Trans. Comput. Syst., 32(1):2:1–2:70, February 2014.

[5] Shinji KONO. CbC, March 2020.

[6] Zhiyi Wang. xv6-rpi. https://code.google.com/archive/p/xv6-rpi/, 2013.

[7] Jean Yang and Chris Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '10, pages 99–110, New York, NY, USA, 2010. ACM.