修士 (工学) 学位論文

Master's Thesis of Engineering

メタ計算を用いた Continuation based C の検証手法

Verification Methods of Continuation based C using Meta Computations

2017 年 3 月

March 2017

比嘉 健太

Yasutaka HIGA

琉球大学
大学院理工学研究科
情報工学専攻

Infomation Engineering Course
Graduate School of Engineering and Science
University of the Ryukyus

指導教員：教授　和田 知久

Supervisor: Prof. Tomohisa WADA

本論文は、修士 (工学) の学位論文として適切であると認める。

論　文　審　査　会

印

(主　査)　　　和田　知久

印

(副　査)　　　高良　富夫

印

(副　査)　　　長田　智和

印

(副　査)　　　河野　真治

要旨

ソフトウェアが期待される仕様を満たすか検査することは重要である。特に実際に動作するソフトウェアを検証できるとなお良い。

本論文では Continuation based C(CbC) 言語で記述されたプログラムを検証用に変更せず信頼性を確保する手法を二つ提案する。一つはプログラムが持つ状態を数え挙げ、常に仕様を満たすことを保証するモデル検査的手法である。プログラムの実行を網羅的に行なうよう変更するメタ計算ライブラリ akasha を用いて赤黒木の仕様を検証する。

もう一つの信頼性向上手法としてデータ構造の持つ性質を証明する手法を提案する。プログラムにおける証明は Curry-Howard Isomorphism により型付き λ 計算に対応する。プログラムを型付けできるよう、CbC の型システムを部分型を用いて定義する。加えて、型の定義を用いて証明支援系言語 Agda 上で CbC のプログラムを記述し、データ構造の性質を証明する。

Abstract

Checking desirable specifications of software are important. If it checks actual implementations, much better.

In this papaer, We propose two verification methods using meta computations which save original implementations. On the hand method verify specification by enumerate possible states on programs. We checked red-black tree specification using our meta computation library named Akasha, which override program executions exhaustively.

On the other hand method veriy programs with proofs. Proposition and proofs have isomorphic relation to typed λ calculus by Curry-Howard Isomorphism. We define the CbC type system with subtype for proving CbC itself. Agda proves properties of translated CbC programs using proposed subtype definition.

目 次

図 目 次

表 目 次

リスト目次

第1章　CbC とメタ計算としての検証手法

ソフトウェアの規模が大きくなるにつれてバグは発生しやすくなる。バグとはソフトウェアが期待される動作以外の動作をすることである。ここで期待された動作は仕様と呼ばれ、自然言語や論理によって記述される。検証とは定められた環境下においてソフトウェアが仕様を満たすことを保証することである。

ソフトウェアの検証手法にはモデル検査と定理証明がある。

モデル検査とはソフトウェアの全ての状態を数え上げ、その状態について仕様が常に真となることを確認する。モデル検査器には Promela と呼ばれる言語でモデルを記述する Spin [1] や、モデルを状態遷移系で記述する NuSMV [2]、C 言語/C++ を記号実行する CBMC [3] などが存在する。定理証明はソフトウェアが満たすべき仕様を論理式で記述し、その論理式が恒真であることを証明する。定理証明を行なうことができる言語には、依存型で証明を行なう Agda [4] や Coq [5] 、 ATS2 [6] などが存在する。

モデル検査器や証明でソフトウェアを検証する際、検証を行なう言語と実装に使われる言語が異なるという問題がある。言語が異なれば二重で同じソフトウェアを記述する必要がある上、検証に用いるソースコードは状態遷移系でプログラムを記述するなど実装コードに比べて記述が困難である。検証されたコードから実行可能なコードを生成可能な検証系もあるが、生成されたコードは検証のコードとは別の言語であったり、既存の実装に対する検証は行なえないなどの問題がある。そこで、当研究室では検証と実装が同一の言語で行なえる Continuation based C [7] 言語を開発している。

Continuation based C (CbC) は C 言語と似た構文を持つ言語である。CbC では処理の単位は関数ではなく CodeSegment という単位で行なわれる。CodeSegment は値を入力として受け取り出力を行なう処理単位であり、CodeSegment を接続していくことによりソフトウェアを構築していく。CodeSegment の接続処理はメタ計算として定義されており、実装や環境によって切り替えを行なうことができる。検証を行なうメタ計算を定義することにより、CodeSegment の定義を検証用に変更せずソフトウェアの検証を行なう。

本論文では CbC のメタ計算として検証手法の提案と CbC の型システムの定義を行なう。モデル検査的な検証として、状態の数え上げを行なう有限のモデル検査と仕様の定義を CbC 自身で行なう。CbC で記述された GearsOS の非破壊赤黒木に対して、メタ計算

ライブラリ akasha を用いて仕様を検査する。また、定理証明的な検証として、CbC のプログラムが証明支援系言語 Agda 上に証明可能な形で定義できることを示す。Agda 上で CbC のプログラムを記述するために、CbC の型システムを部分型を利用して定義する。

1.1 本論文の構成

本論文ではまず第 2 章で Continuation based C の解説を行なう。CbC を記述するプログラミングスタイルである CodeSegment と DataSegment の解説、メタ計算と状態を数え上げるメタ計算ライブラリ akasha の解説を行なう。次に第**??**章で型システムについて取り上げる。型システムの定義とラムダ計算、単純型付きラムダ計算と部分型について述べる。第 4 章では証明支援系プログラミング言語 Agda についての解説を行なう。Agda の構文や使い方、Curry-Howard Isomorphism や Natural Deduction といった証明に関する解説も行なう。第 5 章では、部分型を用いて CbC のプログラムを Agda で記述し、証明を行なう。CodeSegment や DataSegment の Agda 上での定義や、メタ計算はどのように定義されるかを解説する。

第2章 Continuation based C

Continuation based C (CbC) は当研究室で開発しているプログラミング言語であり、OS や組み込みソフトウェアの開発を主な対象としている。CbC は C 言語の下位の言語であり、構文はほぼ C 言語と同じものを持つが、よりアセンブラに近い形でプログラムを記述する。CbC は CodeSegment と呼ばれる単位で処理を定義し、それらを組み合わせることにでプログラム全体を構成する。データの単位は DataSegment と呼ばれる単位で定義し、それら CodeSegment によって変更していくことでプログラムの実行となる。CbC の処理系には llvm/clang による実装 [8] と gcc [9] による実装などが存在する。

2.1 CodeSegment と DataSegment

本研究室では検証を行ないやすいプログラムの単位として CodeSegment と DataSegment を用いるプラグラミングスタイルを提案している。

CodeSegment は処理の単位である。入力を受け取り、それに対して処理を行なった後を出力を行なう。また、CodeSegment は他の CodeSegment と組み合わせることが可能である。ある CodeSegment A を CodeSegment B に接続した場合、 A の出力は B の入力となる。

DataSegment は CodeSegment が扱うデータの単位であり、処理に必要なデータが全て入っている。CodeSegment の入力となる DataSegment は Input DataSegment と呼ばれ、出力は Output DataSegment と呼ばれる。CodeSegment A と CodeSegment B を接続した時、A の Output DataSegment は B の入力 Input DataSegment となる。

2.2 Continuation based C における CodeSegment と DataSegment

最も基本的な CbC のソースコードをリスト 2.1 に、ソースコードが実行される流れを図 2.1 に示す。Continuation based C における CodeSegment は返り値を持たない関数として表現される。CodeSegment を定義するためには、C 言語の関数を定義する構文の返り値の型部分に `__code` キーワードを指定する。Input DataSegment は関数の引数として定

義される。次の CodeSegment へ処理を移す際には `goto` キーワードの後に CodeSegment 名と Input DataSegment を指定する。処理の移動を軽量継続と呼び、リスト 2.1 内の `goto cs1(a+b);` がこれにあたる。この時の `(a+b)` が次の CodeSegment である cs1 の Input DataSegment となる cs0 の Output DataSegment である。

リスト 2.1: CodeSegment の軽量継続

```
__code cs0(int a, int b){
  goto cs1(a+b);
}

__code cs1(int c){
  goto cs2(c);
}
```

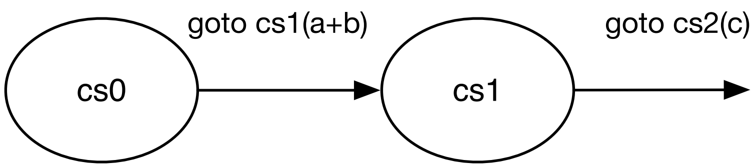

図 2.1: CodeSegment の軽量継続

Scheme などの call/cc といった継続はトップレベルから現在までの位置を環境として保持する。通常環境とは関数の呼び出しスタックの状態である。CbC の軽量継続は呼び出し元の情報を持たないため、スタックを破棄しながら処理を続けていく。よって、リスト 2.1 のプログラムでは cs0 から cs1 へと継続した後に cs0 へ戻ることはできない。

もう少し複雑な CbC のソースコードをリスト 2.2 に、実行される流れを図 2.2 に示す。このソースコードは整数の階乗を求めるプログラムである。CodeSegment factorial0 では自分自身への再帰的な継続を用いて階乗を計算している。軽量継続時には関数呼び出しのスタックは存在しないが、計算中の値を DataSegment で持つことで再帰を含むループ処理も行なうことができる。

リスト 2.2: 階乗を求める CbC プログラム

```
__code print_factorial(int prod)
{
  printf("factorial = %d\n", prod);
  exit(0);
}

__code factorial0(int prod, int x)
{
```

```
  if (x >= 1) {
    goto factorial0(prod*x, x-1);
  } else {
    goto print_factorial(prod);
  }

}

__code factorial(int x)
{
  goto factorial0(1, x);
}

int main(int argc, char **argv)
{
  int i;
  i = atoi(argv[1]);

  goto factorial(i);
}
```

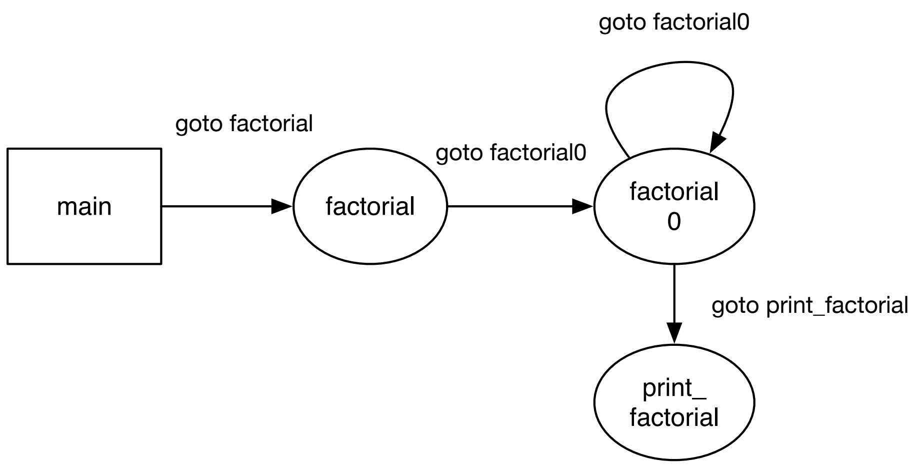

図 2.2: 階乗を求める CbC プログラム

2.3 MetaCodeSegment と MetaDataSegment

プログラムを記述する際、本来行ないたい計算の他にも記述しなければならない部分が存在する。メモリの管理やネットワーク処理、エラーハンドリングや並列処理などがこれ

にあたり、本来行ないたい計算と区別してメタ計算と呼ぶ。プログラムを動作させるためにメタ計算部分は必須であり、しばしば本来の処理よりも複雑度が高い。

CodeSegment を用いたプログラミングスタイルでは計算とメタ計算を分離して記述する。分離した計算は階層構造を持ち、本来行ないたい処理をノーマルレベルとし、メタ計算はメタレベルとしてノーマルレベルよりも上の存在に位置する。複雑なメタ計算部分をライブラリや OS 側が提供することで、ユーザはノーマルレベルの計算の記述に集中することができる。また、ノーマルレベルのプログラムに必要なメタ計算を追加することで、並列処理やネットワーク処理などを含むプログラムに拡張できる。さらに、ノーマルレベルからはメタレベルは隠蔽されているため、メタ計算の実装を切り替えることも可能である。例えば、並列処理のメタ計算用いたプログラムを作成する際、CPU で並列処理を行なうメタ計算と GPU で並列処理メタ計算を環境に応じて作成することができる。

なお、メタ計算を行なう CodeSegment は Meta CodeSegment と呼び、メタ計算に必要な DataSegment は Meta DataSegment と呼ぶ。Meta CodeSegment は CodeSegment の前後にメタ計算を挟むことで実現され、Meta DataSegment は DataSegment を含む上位の DataSegment として実現できる。よって、メタ計算は通常の計算を覆うように計算を拡張するものだと考えられる (図 2.3)。

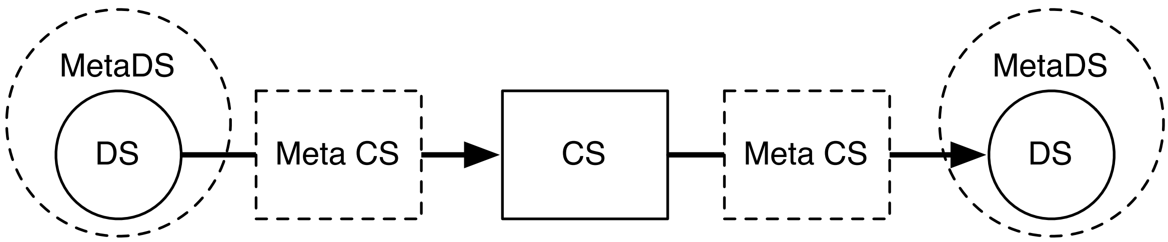

図 2.3: Meta CodeSegment と Meta DataSegment

2.4 Continuation based C におけるメタ計算の例: GearsOS

CbC を用いてメタ計算を実現した例として、GearsOS [10] が存在する。GearsOS は並列に、信頼性高く動作することを目標とした OS であり、 マルチコア CPU や GPU 環境での動作を対象としている。現在 OS の設計と並列処理部分の実装が行なわれている。GearsOS におけるメタ計算は Monad [11] を用いている。 現在実装済みのメタ計算はメモリの管理、並列に書き込むことが可能な Synchronized Queue、データの保存用の非破壊赤黒木がある。

GearsOS では CodeSegment と DataSegment はそれぞれ CodeGear と DataGear と呼ばれている。マルチコア CPU 環境では CodeGear と CodeSegment は同一だが、GPU 環境では CodeGear には OpenCL [12]/CUDA [13] における kernel も含まれる。kernel とは GPU で実行される関数のことであり、GPU 上のメモリに配置されたデータ群に対して並列に実行される。通常 GPU でデータの処理を行なう場合は

- データをメインメモリから GPU のメモリへ転送
- 転送終了を同期で確認
- kernel 起動 (GPU メモリ上のデータに対して並列に処理)
- 処理終了を同期で確認
- 計算結果であるデータを GPU のメモリからメインメモリへ転送
- 転送終了を同期で確認

といった手順が必要であり、ユーザは処理したいデータの位置などを意識しながらプログラミングする必要がある。GearsOS では CPU/GPU での処理をメタ計算としてユーザから隠すことにより、 CodeGear が実行されるデバイスや DataGear の位置を意識する必要がなくなる。

GearsOS で利用する Meta DataGear には以下のものが含まれる。

- DataGear の型情報
- DataGear を格納するメモリの情報
- CodeGear の名前と CodeGear の関数ポインタ との対応表
- CodeGear が参照する DataGear へのポインタ

実際の GearsOS におけるメモリ管理を含むメタ計算用の Meta DataGear の定義例をリスト 2.3 に示す。Meta DataGear は Context という名前の構造体で定義されている。通常レベルの DataGear も構造体で定義されているが、メタ計算側から見た DataGear はそれぞれの構造体の共用体となっており、一様に扱える。

リスト 2.3: GearsOS における Meta DataGear の定義例

```
/* Context definition */

#define ALLOCATE_SIZE 1024

enum Code {
    Code1,
    Code2,
    Allocator,
};

enum UniqueData {
    Allocate,
    Tree,
};

struct Context {
    int codeNum;
    __code (**code) (struct Context *);
    void* heap_start;
    void* heap;
    long dataSize;
    int dataNum;
    union Data **data;
};

union Data {
    struct Tree {
        union Data* root;
        union Data* current;
        union Data* prev;
        int result;
    } tree;
    struct Node {
        int key;
        int value;
        enum Color {
            Red,
            Black,
        } color;
        union Data* left;
        union Data* right;
    } node;
    struct Allocate {
        long size;
        enum Code next;
    } allocate;
```

```
47 };
```

リスト 2.3 のソースコードは以下のように対応している。

- DataGear の型情報

  DataGear は構造体を用いて定義する (リスト 2.3 27-46 行)。Tree や Node、 Allocate 構造体が DataGear に相当する。メタ計算は任意の DataGear 扱うために全ての DataGear を扱える必要がある。全ての DataGear の共用体を定義することで、DataGear を一律に扱うことができる (リスト 2.3 26-47 行)。メモリを確保する場合はこの型情報からサイズを決定する。

- DataGear を格納するメモリの情報

  メモリ領域の管理は、事前に領域を確保した後、必要に応じてその領域を割り当てることで実現する。そのために Context は割り当て済みの領域 heap と、割り当てた DataGear の数 dataNum を持つ。

- CodeGear の名前と CodeGear の関数ポインタ との対応表

  CodeGear の名前と CodeGear の関数ポインタの対応は enum と関数ポインタによって実現されている。CodeGear の名前は enum (リスト 2.3 5-9 行) で定義され、コンパイル後には整数へと変換される。プログラム全体で利用する CodeGear は code フィールドに格納されており、enum を用いてアクセスする。この対応表を動的に変更することにより、実行時に比較ルーチンなどを変更することが可能になる。

- CodeGear が参照する DataGear へのポインタ

  Meta CodeGear は Context を引数に取る CodeGear として定義されている。そのため、Meta CodeGear が DataGear の値を使う為には Context から DataGear を取り出す必要がある。取り出す必要がある DataGear は enum を用いて定義し (リスト 2.3 11-14 行)、 CodeGear を実行する際に data フィールドから取り出す。

Meta CodeGear は定義された Meta DataGear を処理する CodeGear である。メモリ管理や並列処理の待ち合わせといった処理はこのメタレベルにしか表れない。

GearsOS においては軽量継続もメタ計算として実現されている。とある CodeGear から次の CodeGear へと軽量継続する際には、次に実行される CodeGear の名前を指定する。その名前を Meta CodeGear が解釈し、対応する CodeGear へと処理を引き渡す (リスト 3.7 の `meta`)。

リスト 2.4: 通常の CodeSegment の軽量継続

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

CodeGear と名前の対応は Meta DataGear に格納されており、従来の OS の Process や Thread に相当する。名前の対応を動的に切り替えたり、Thread ごとに切り替えることにより、通常レベルのプログラムを変更せず実行を上書きできる。これは従来の OS の Dynamic Loading Libary や Command の呼び出しに相当する。

また、通常レベルの CodeGear から Meta DataGear を操作できてしまうと、ユーザがメタレベル操作を自由に記述できてしまい、メタ計算を分離した意味が無くなってしまう。これを防ぐために、CodeGear を実行する際は Meta DataGear から必要な DataGear だけを渡す。このように、 Meta DataGear から DataGear を取り出す Meta CodeGear を stub と呼ぶ。stub の例をリスト 2.5 に示す。

リスト 2.5: GearsOS における stub Meta CodeSegment

```
__code put(struct Context* context,
           struct Tree* tree,
           struct Node* root,
           struct Allocate* allocate)
{
    /* ... */
}

__code put_stub(struct Context* context)
{
    goto put(context,
             &context->data[Tree]->tree,
             context->data[Tree]->tree.root,
             &context->data[Allocate]->allocate);
}
```

stub は Context が持つ DataGear のポインタ data に対して enum を用いてアクセスしている。なお、現在はメタレベルの計算とノーマルレベルの分離はコンパイラ側がサポートしていないため、引数に Meta DataGear である Context が渡されているが、本来はノーマルレベルではアクセスできない。

また、GearsOS におけるメタ計算として CodeGear のモデル検査がある。通常レベルの CodeGear を変更することなく、その仕様を検証するものである。個々の CodeGear の仕様を検証することにより、より信頼性の高い OS を目指す。

第3章　メタ計算ライブラリ akasha における検証

第 2 章では Continuation based C 言語の概要と、CbC で記述された GearsOS について述べた。GearsOS の持つメタ計算として、モデル検査的なアプローチで CodeGear の仕様を検証していく。

3.1　モデル検査

モデル検査とは、ソフトウェアの全ての状態において仕様が満たされるかを確認するものである。このモデル検査を行なうソフトウェアをモデル検査器と呼ぶ。モデルは検査器は、仕様の定義と確認ができる。加えて、仕様を満たさない場合にはソフトウェアがどのような状態であったか反例を返す。

モデル検査器には Spin [1] や CBMC [3] などが存在する。

Spin は Promela と呼ばれる言語でモデルを記述し、その中に論理式として仕様を記述する。論理式は assert でモデルの内部に埋め込まれ、並列に実行してもその仕様が満たされるかをチェックする。また、Promela で記述されたモデルから C 言語を生成することができる。しかし、Promela で記述されたモデルは元の C 言語とはかなり異なる構文をしており、ユーザが記述する難易度が高い。

そこで、モデルを個別に記述せずに実装そのものを検査するアプローチがある。例えばモデル検査器 CBMC は C 言語を直接検証できる。CBMC でも仕様は論理式で記述され、 assert と組み合わせる。C 言語の実行は通常の実行とは異なり、記号実行という形で実行される。プログラム上の変数は記号として処理され、`a < b` といった条件式により分岐が行なわれたのなら、その条件を持つ場合の経路、持たない場合の経路、と分岐していくのである。

GearsOS におけるモデル検査的なアプローチは CBMC のように実装言語をそのまま検証できるようにしたい。そのために、assert を利用した仕様の定義と、その検査、必要なら反例を提出するようなメタ計算を定義する。このメタ計算をメタ計算ライブラリ akasha として実装した。

この章では、メタ計算ライブラリ akasha を用いて GearsOS のデータ構造を検証していく。

3.2 GearsOS における非破壊赤黒木

現状の GearsOS に実装されているメタ計算として、非破壊赤黒木が存在する。非破壊赤黒木はユーザがデータを保存する際に利用することを想定している。メタ計算として定義することで、ノーマルレベルからは木のバランスを考慮せず木への操作が行なえる。

なお、赤黒木とは二分探索木の一種であり、木のバランスを取るための情報として各ノードは赤か黒の色を持っている。

二分探索木の条件は以下である。

- 左の子孫の値は親の値より小さい
- 右の子孫の値は親の値より大きい

加えて、赤黒木が持つ具体的な条件は以下のものである。

- 各ノードは赤か黒の色を持つ。
- ルートノードの色は黒である。
- 葉ノードの色は黒である。
- 赤ノードは 2 つの黒ノードを子として持つ (よって赤ノードが続くことは無い)。
- ルートから最下位ノードへの経路に含まれる黒ノードの数はどの最下位ノードでも一定である。

数値を要素に持つ赤黒木の例を図 3.1 に示す。条件に示されている通り、ルートノードは黒であり、赤ノードは連続していない。加えて各最下位ノードへの経路に含まれる黒ノードの個数は全て 2 である。

赤黒木の持つ条件を言い変えるのなら、「木をルートから辿った際に最も長い経路は最も短い経路の高々二倍に収まる」とも言える。この言い換えは「赤が続くことはない」という条件と「ルートから最下位への経路の黒ノードはどの最下位ノードでも同じ」であることから導ける。具体的には、最短経路は「黒のみの経路」であり、最長経路は「黒と赤が交互に続く経路」となる。この条件を言い変えた性質を仕様とし、検証していく。

GearsOS で実装されている赤黒木は特に非破壊赤黒木であり、一度構築した木構造は破壊される操作ごとに新しい木構造が生成される。非破壊の性質を付与した理由として、

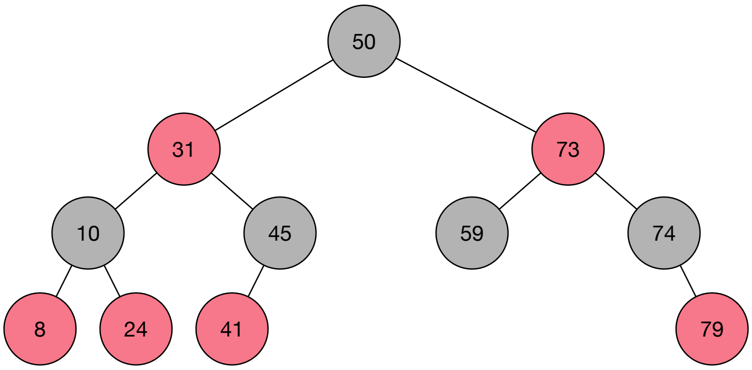

図 3.1: 赤黒木の例

並列実行時のデータの保存がある。同じ赤黒木をロックせずに同時に更新した場合、ノードの値は実行順に依存したり、競合したりする。しかし、ロックを行なって更新した場合は同じ木に対する処理に待ち合わせが発生し、全体の並列度が下がる。この問題に対し GearsOS では、各スレッドは処理を行なう際には非破壊の木を利用することで並列度は保ち、値の更新が発生する時のみ木をアトミックな操作で置き換えることで競合を回避する。具体的には木の操作を行なった後はルートのノードを元に CAS で置き換え、失敗した時は木を読み込み直して処理を再実行する。CAS(Check and Set) とは、アトミックに値を置き換える操作であり、使う際は更新前の値と更新後の値を渡す。CAS で渡された更新前の値が、保存している値と同じであれば競合していないために値の更新に成功し、異なる場合は他に書き込みがあったことを示すために値の更新が失敗する操作のことである。

非破壊赤黒木の実装の基本的な戦略は、変更したいノードへのルートノードからの経路を全て複製し、変更後に新たなルートノードとする。この際に変更が行なわれていない部分は変更前の木と共有する (図 3.2)。これは一度構築された木構造は破壊されないという非破壊の性質を用いたメモリ使用量の最適化である。

CbC を用いて赤黒木を実装する際の問題として、関数の呼び出しスタックが存在しないことがある。C における実装では関数の再帰呼び出しによって木が辿るが、それが行なえない。経路を辿るためには、ノードに親への参照を持たせるか、挿入や削除の際に辿った経路を記憶する必要がある。ノードが親への参照を持つ非破壊木構造は共通部分の共有が行なえないため、経路を記憶する方法を使う。経路の記憶にはスタックを用い、スタックは Meta DataSegment に保持する。

赤黒木を格納する DataSegment と Meta DataSegment の定義をリスト 3.1 に示す。経路の記憶に用いるスタックは Meta DataSegment である Context 内部の `node_stack` である。DataSegment は各ノード情報を持つ `Node` 構造体と、赤黒木を格納する `Tree` 構造

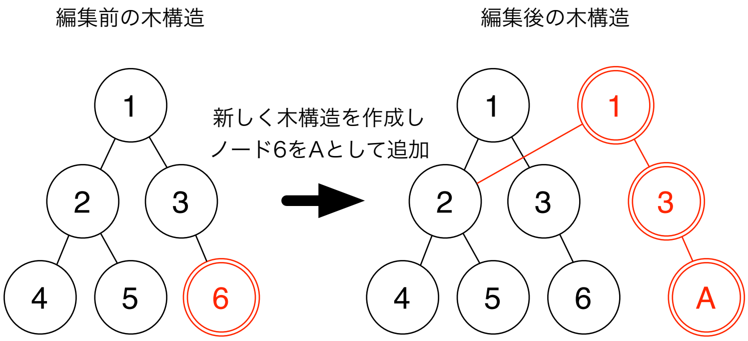

図 3.2: 非破壊赤黒木の編集

体、挿入などで操作中の一時的な木を格納する `Traverse` 共用体などがある。

リスト 3.1: 赤黒木の DataSegment と Meta DataSegment

```
// DataSegments for Red-Black Tree
union Data {
    struct Comparable { // interface
        enum Code compare;
        union Data* data;
    } compare;
    struct Count {
        enum Code next;
        long i;
    } count;
    struct Tree {
        enum Code next;
        struct Node* root;
        struct Node* current;
        struct Node* deleted;
        int result;
    } tree;
    struct Node {
        // need to tree
        enum Code next;
        int key; // comparable data segment
        int value;
        struct Node* left;
        struct Node* right;
        // need to balancing
        enum Color {
            Red,
            Black,
        } color;
    } node;
    struct Allocate {
        enum Code next;
        long size;
    } allocate;
```

```
};


// Meta DataSegment
struct Context {
    enum Code next;
    int codeNum;
    __code (**code) (struct Context*);
    void* heapStart;
    void* heap;
    long heapLimit;
    int dataNum;
    stack_ptr code_stack;
    stack_ptr node_stack;
    union Data **data;
};
```

Meta DataSegment を初期化する Meta CodeSegment initLLRBContext をリスト 3.2 に示す。この Meta CodeSegment ではメモリ領域の確保、CodeSegment 名と CodeSegment の実体の対応表の作成などを行なう。メモリ領域はプログラムの起動時に一定数のメモリを確保し、ヒープとして `heap` フィールドに保持させる。CodeSegment 名と CodeSegment の実体との対応は、enum で定義された CodeSegment 名の添字へと CodeSegment の関数ポインタを代入することにより持つ。例えば `Put` の実体は `put_stub` である。他にも DataSegment の初期化 (リスト 3.2 34-48) とスタックの初期化 (リスト 3.2 50-51) を行なう。

リスト 3.2: 赤黒木の Meta DataSegment の初期化を行なう Meta CodeSegment

```
1  __code initLLRBContext(struct Context* context, int num) {
2      context->heapLimit = sizeof(union Data)*ALLOCATE_SIZE;
3      context->code = malloc(sizeof(__code*)*ALLOCATE_SIZE);
4      context->data = malloc(sizeof(union Data*)*ALLOCATE_SIZE);
5      context->heapStart = malloc(context->heapLimit);
6
7      context->codeNum = Exit;
8
9      context->code[Code1]      = code1_stub;
10     context->code[Code2]      = code2_stub;
11     context->code[Code3]      = code3_stub;
12     context->code[Code4]      = code4;
13     context->code[Code5]      = code5;
14     context->code[Find]       = find;
15     context->code[Not_find]   = not_find;
16     context->code[Code6]      = code6;
17     context->code[Put]        = put_stub;
18     context->code[Replace]    = replaceNode_stub;
19     context->code[Insert]     = insertNode_stub;
20     context->code[RotateL]    = rotateLeft_stub;
21     context->code[RotateR]    = rotateRight_stub;
22     context->code[InsertCase1]   = insert1_stub;
23     context->code[InsertCase2]   = insert2_stub;
24     context->code[InsertCase3]   = insert3_stub;
```

```
    context->code[InsertCase4]   = insert4_stub;
    context->code[InsertCase4_1] = insert4_1_stub;
    context->code[InsertCase4_2] = insert4_2_stub;
    context->code[InsertCase5]   = insert5_stub;
    context->code[StackClear]    = stackClear_stub;
    context->code[Exit]       = exit_code;

    context->heap = context->heapStart;

    context->data[Allocate] = context->heap;
    context->heap += sizeof(struct Allocate);

    context->data[Tree] = context->heap;
    context->heap += sizeof(struct Tree);

    context->data[Node] = context->heap;
    context->heap += sizeof(struct Node);

    context->dataNum = Node;

    struct Tree* tree = &context->data[Tree]->tree;
    tree->root = 0;
    tree->current = 0;
    tree->deleted = 0;

    context->node_stack = stack_init(sizeof(struct Node*), 100);
    context->code_stack = stack_init(sizeof(enum Code), 100);
}
```

実際の赤黒木の実装に用いられている Meta CodeSegment の一例をリスト 3.3 に示す。Meta CodeSegment `insertCase2` は要素を挿入した場合に呼ばれる Meta CodeSegment の一つであり、親ノードの色によって処理を変える。まず、色を確認するために経路を記憶しているスタックから親の情報を取り出す。親の色が黒ならば処理を終了し、次の CodeSegment へと軽量継続する (リスト 3.3 5-8)。親の色が赤であるならばさらに処理を続行して `InsertCase3` へと軽量継続する。ここで、経路情報を再現するためにスタックへと親を再代入してから軽量継続を行なっている。なお、Meta CodeSegment でも Context から DataSegment を展開する処理は stub によって行なわれる (リスト 3.3 14-16)。

リスト 3.3: 赤黒木の実装に用いられている Meta CodeSegment 例

```
1  __code insertCase2(struct Context* context, struct Node* current) {
2      struct Node* parent;
3      stack_pop(context->node_stack, &parent);
4  
5      if (parent->color == Black) {
6          stack_pop(context->code_stack, &context->next);
7          goto meta(context, context->next);
8      }
9  
10     stack_push(context->node_stack, &parent);
11     goto meta(context, InsertCase3);
```

```
}

__code insert2_stub(struct Context* context) {
    goto insertCase2(context, context->data[Tree]->tree.current);
}
```

3.3 メタ計算ライブラリ akasha を用いた赤黒木の実装の検証

赤黒木の仕様の定義とその確認を CbC で行なっていく。仕様には赤黒木の利用方法などによっていくつかのものが考えられる。赤黒木に対する操作の仕様と、その操作によって保証されるべき赤黒木の状態を示すと以下のようになる。

- 挿入したデータは参照できること
- 削除したデータは参照できないこと
- 値を更新した後は更新された値が参照されること
- 操作を行なった後の木はバランスしていること

今回はバランスに関する仕様を確認する。操作を挿入に限定し、どのような順番で要素を挿入しても木がバランスすることを検証する。検証には当研究室で開発しているメタ計算ライブラリ akasha を用いる。

akasha では仕様は常に成り立つべき CbC の条件式として定義される。具体的には Meta CodeSegment に定義した assert が仕様に相当する。仕様の例として「木をルートから辿った際に最も長い経路は最も短い経路の高々 2 倍に収まる」という式を定義する (リスト 3.4)。

リスト 3.4: 木の高さに関する仕様記述

```
void verifySpecification(struct Context* context, struct Tree* tree) {
    assert(!(maxHeight(tree->root, 1) > 2*minHeight(tree->root, 1)));
    return meta(context, EnumerateInputs);
}
```

リスト 3.4 で定義した仕様がプログラムの持つ全ての状態に成り立つかを確認する。また、成り立たない場合には仕様に反する状態を反例として提出する。

まずは最も単純な検証として要素数を有限に固定し、その挿入順番を数え上げる。最初に、検証の対象となる赤黒木と、検証に必要な DataSegment を含む Meta DataSegment を定義する (リスト 3.5)。これが akasha のレベルで利用する Meta DataSegment である。赤黒木自体はユーザから見るとメタレベル計算であるが、今回はその実装の検証するた

め、赤黒木がノーマルレベルとなる。よって akasha はメタメタレベルの計算とも考えられる。

akasha が使う DataSegment は データの挿入順を数え上げるためには使う環状リスト `Iterator` とその要素 `IterElem`、検証に使う情報を保持する `AkashaInfo`、木をなぞる際に使う `AkashaNode` がある。

リスト 3.5: 検証を行なうための Meta DataSegment

```
// Data Segment
union Data {
    struct Tree { /* ... */ } tree;
    struct Node { /* ... */ } node;

    /* for verification */
    struct IterElem {
        unsigned int val;
        struct IterElem* next;
    } iterElem;
    struct Iterator {
        struct Tree* tree;
        struct Iterator* previousDepth;
        struct IterElem* head;
        struct IterElem* last;
        unsigned int  iteratedValue;
        unsigned long iteratedPointDataNum;
        void*         iteratedPointHeap;
    } iterator;
    struct AkashaInfo {
        unsigned int minHeight;
        unsigned int maxHeight;
        struct AkashaNode* akashaNode;
    } akashaInfo;
    struct AkashaNode {
        unsigned int       height;
        struct Node*       node;
        struct AkashaNode* nextAkashaNode;
    } akashaNode;
};
```

挿入順番の数え上げには環状リストを用いた深さ優先探索を用いる。最初に検証する要素を全て持つ環状リストを作成し、木に挿入した要素を除きながら環状リストを複製していく。環状リストが空になった時が組み合わせを一つ列挙し終えた状態となる。列挙し終えた後、前の深さの環状リストを再現してリストの先頭を進めることで異なる組み合わせを列挙する。

仕様には木の高さが含まれるので、高さを取得する Meta CodeSegment が必要となる。リスト 3.6 に木の最も低い経路の長さを取得する Meta CodeSegment を示す。

木を辿るためのスタックに相当する `AkshaNode` を用いて経路を保持しつつ、高さを確認している。スタックが空であれば全てのノードを確認したので次の CodeSegment へと

軽量継続を行なう。空でなければ今辿っているノードが葉であるか確認し、葉ならば高さを更新して次のノードを確認するため自身へと軽量継続する。葉でなければ高さを 1 増やして左右の子をスタックに積み、自身へと軽量継続を行なう。

リスト 3.6: 木の最も短かい経路の長さを確認する Meta CodeSegment

```
__code getMinHeight_stub(struct Context* context) {
    goto getMinHeight(context, &context->data[Allocate]->allocate, &
   context->data[AkashaInfo]->akashaInfo);
}

__code getMinHeight(struct Context* context, struct Allocate* allocate,
   struct AkashaInfo* akashaInfo) {
    const struct AkashaNode* akashaNode = akashaInfo->akashaNode;

    if (akashaNode == NULL) {
        allocate->size = sizeof(struct AkashaNode);
        allocator(context);
        akashaInfo->akashaNode = (struct AkashaNode*)context->data[
   context->dataNum];

        akashaInfo->akashaNode->height = 1;
        akashaInfo->akashaNode->node   = context->data[Tree]->tree.root;

        goto getMaxHeight_stub(context);
    }

    const struct Node* node = akashaInfo->akashaNode->node;
    if (node->left == NULL && node->right == NULL) {
        if (akashaInfo->minHeight > akashaNode->height) {
            akashaInfo->minHeight  = akashaNode->height;
            akashaInfo->akashaNode = akashaNode->nextAkashaNode;
            goto getMinHeight_stub(context);
        }
    }

    akashaInfo->akashaNode = akashaInfo->akashaNode->nextAkashaNode;

    if (node->left != NULL) {
        allocate->size = sizeof(struct AkashaNode);
        allocator(context);
        struct AkashaNode* left = (struct AkashaNode*)context->data[
   context->dataNum];
        left->height           = akashaNode->height+1;
        left->node             = node->left;
        left->nextAkashaNode   = akashaInfo->akashaNode;
        akashaInfo->akashaNode = left;
    }

    if (node->right != NULL) {
        allocate->size = sizeof(struct AkashaNode);
        allocator(context);
        struct AkashaNode* right = (struct AkashaNode*)context->data[
```

```
    context->dataNum];
        right->height            = akashaNode->height+1;
        right->node              = node->right;
        right->nextAkashaNode    = akashaInfo->akashaNode;
        akashaInfo->akashaNode   = right;
    }

    goto getMinHeight_stub(context);
}
```

同様に最も高い高さを取得し、仕様であるリスト 3.4 の assert を挿入の度に実行する。assert は CodeSegment の結合を行なうメタ計算である `meta` を上書きすることにより実現する。

`meta` はリスト 3.3 の `insertCase2` のように軽量継続を行なう際に CodeSegment 名と DataSegment を指定するものである。検証を行なわない通常の `meta` の実装は CodeSegment 名から対応する実体への軽量継続である (リスト 3.7)。

リスト 3.7: 通常の CodeSegment の軽量継続

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

これを、検証を行なうように変更することで `insertCase2` といった赤黒木の実装のコードを修正することなく検証を行なうことができる。検証を行ないながら軽量継続する `meta` はリスト 3.8 のように定義される。実際の検証部分は `PutAndGoToNextDepth` の後に行なわれるため、直接は記述されていない。この `meta` が行なうのは検証用にメモリの管理である。状態の数え上げを行なう際に状態を保存したり、元の状態に戻す処理が行なわれる。このメタ計算を用いた検証では、要素数 13 個までの任意の順で挿入の際に仕様が満たされることを確認できた。また、赤黒木の処理内部に恣意的なバグを追加した際には反例を返した。

リスト 3.8: 検証を行なう CodeSegment の軽量継続

```
__code meta(struct Context* context, enum Code next) {
    struct Iterator* iter = &context->data[Iter]->iterator;

    switch (context->prev) {
        case GoToPreviousDepth:
            if (iter->iteratedPointDataNum == 0) break;
            if (iter->iteratedPointHeap == NULL) break;

            unsigned int diff =(unsigned long)context->heap - (unsigned
    long)iter->iteratedPointHeap;
            memset(iter->iteratedPointHeap, 0, diff);
            context->dataNum = iter->iteratedPointDataNum;
            context->heap    = iter->iteratedPointHeap;
            break;
        default:
```

```
            break;
    }
    switch (next) {
        case PutAndGoToNextDepth:   // with assert check
            if (context->prev == GoToPreviousDepth) break;
            if (iter->previousDepth == NULL)        break;
            iter->previousDepth->iteratedPointDataNum = context->dataNum;
            iter->previousDepth->iteratedPointHeap    = context->heap;
            break;
        default:
            break;
    }

    context->prev = next;
    goto (context->code[next])(context);
}
```

3.4 モデル検査器 CBMC との比較

akasha の比較対象として、 C 言語の有限モデルチェッカ CBMC [?] を用いて赤黒木を検証した。CBMC は ANSI-C を記号実行し、仕様の否定となるような実行パターンが無いかを検証するツールである。

比較のために全く同じ赤黒木のソースコードを用いたいが、CbC の構文は厳密には C とは異なるために変換が必要である。具体的には、`__code` を `void` に、 `goto` を `return` に置換することで機械的に C 言語に変換できる。

CBMC における仕様は bool を返す式として記述するため、akasha と同様の仕様定義が利用できる (リスト 3.9。assert が true になるような実行パターンを CBMC が見付けると、その実行パターンが反例として出力される。

リスト 3.9: CBMC における仕様記述

```
void verifySpecification(struct Context* context,
                         struct Tree* tree) {
    assert(!(maxHeight(tree->root, 1) >
            2*minHeight(tree->root, 1)));
    return meta(context, EnumerateInputs);
}
```

挿入順の数え上げには CBMC の機能に存在する非決定的な値 `nondet_int()` を用いた (リスト 3.10)。この `nondet_int()` 関数は int の持ちうる値の内から非決定的に値を取得する関数である。akasha では有限の要素個分の組み合わせを用いて挿入順の数え上げとしたが、CBMC では要素数回分だけランダムな値を入力させることで数え上げとする。

リスト 3.10: CBMC における挿入順の数え上げ

```
void enumerateInputs(struct Context* context,
                     struct Node* node) {
    if (context->loopCount > LIMIT_OF_VERIFICATION_SIZE) {
        return meta(context, Exit);
    }

    node->key     = nondet_int();
    node->value   = node->key;
    context->next = VerifySpecification;
    context->loopCount++;

    return meta(context, Put);
}
```

CBMC では有限のステップ数だけ C 言語を記号実行し、その範囲内で仕様が満たされるかを確認する。条件分岐や繰り返しなどは展開されて実行される。基本的にはメモリの許す限り展開を行なうことができるが、今回の赤黒木の検証では 411 回まで展開することができた。この 411 回のうちの実行パスでは赤黒木の仕様は常に満たされる。しかし、この展開された回数は挿入された回数とは無関係であり、実際どの程度検証することができたか確認できない。実際、赤黒木に恣意的なバグを追加した際にも仕様の反例は得られず、CBMC で扱える範囲内では赤黒木の性質は検証できなかった。

よって、CBMC では検証できない範囲の検証を akasha で行なえることが確認できた。

第4章　証明支援系言語 Agda による証明手法

3 章では CbC のモデル検査的検証アプローチとして、akasha を用いた有限の要素数の挿入時の仕様の検証を行なった。しかし、要素数 13 個分の挿入を検証しても赤黒木の挿入が必ずバランスするとは断言しづらい。

そこで、 CbC の性質をより厳密に定義し、その上で証明を行なうことを考えた。CbC のプログラムを証明できる形に変換し、任意の回数の挿入に対しても性質が保証できるよう証明するのである。証明を行なう機構として注目したのが型システムである。

型システムは Curry-Howard 同型対応により命題と型付きラムダ計算が一対一に対応する。依存型という型を持つ証明支援系言語 Agda を用いて型システムで証明が行なえることを示す。

4.1　型システムとは

型システムとは、計算する値を分類することにでプログラムがある種の振舞いを行なわないことを保証する機構の事である [14] [15]。ある種の振舞いとはプログラム中の評価不可能な式や、言語として未定義な式などが当て嵌まる。例えば、gcc や clang といったコンパイラは関数定義時に指定された引数の型と呼び出し時の値の型が異なる時に警告を出す。この警告は関数が受けつける範囲以外の値をプログラマが渡してしまった場合などに有効に働く。加えて、関数を定義する側も受け付ける値の範囲を限定できるため関数内部の処理を記述しやすい。

型システムで行なえることには以下のようなものが存在する。

- エラーの検出

  文字列演算を行なう関数に整数を渡してしまったり、複雑な場合分けで境界条件を見落すなど、プログラマの不注意が型の不整合として表れる。

- 抽象化

  型は大規模プログラムの抽象化の単位にもなる。例えば特定のデータ構造に対する処理をモジュール化し、パッケージングすることができる。

- ドキュメント化

  関数やモジュールの型を確認することにより、プログラムの理解の助けになる。また、型はコンパイラが実行されるたびに検査されるため、常に最新の正しい情報を提供する。

- 言語の安全性

  例えばポインタを直接扱わないようメモリアクセスを抽象化し、データを破壊する可能性をプログラマに提供しないようにできる。

- 効率性

  そもそも、科学計算機における最初の型システムは Fortran **??** などにおける整数と実数の算術式の区別だった。型の導入により、ソースからコンパイラがより最適化されたコードを生成できる。

型システムには多くの定義が存在する。型の表現能力には単純型や総称型、部分型などが存在する。型付けにも動的型付けや静的型付けが存在し、どの型システムを採用するかは言語の設計に依存する。例えば C 言語では数値と文字を二項演算子 `+` で加算できるが、Haskell では加算することができない。これは Haskell が C 言語よりも厳密な型システムを採用しているからである。具体的には Haskell は暗黙的な型変換を許さず、 C 言語は言語仕様として暗黙の型変換を持っている。

型システムを定義することはプログラミング言語がどのような特徴を持つかを決めることにも繋がる。

4.2　依存型を持つ証明支援系言語 Agda

型システムを用いて証明を行なうことができる言語に Agda [4] が存在する。Agda は依存型という強力な型システムを持っている。依存型とは型も第一級オブジェクトとする型システムであり、型の型や型を引数に取る関数、値を取って型を返す関数などが記述できる。この節では Agda の文法的な紹介を行なう [16] [17]。

Agda はインデントに意味を持つ言語であるため、インデントはきちんと揃える必要がある。また、非常に多くの記号を利用できる言語であり、スペースの有無は厳格にチェックされる。なお、 `--` の後はコメントである。

まず、Agda のプログラムを記述するファイルを作成する。Agda のプログラムは全てモジュール内部に記述されるため、まずはトップレベルにモジュールを定義する必要がある。トップレベルのモジュールはファイル名と同一となる。例えば `AgdaBasics.agda` を作成する時のモジュール名はリスト 4.1 のように定義する。

リスト 4.1: Agda のモジュールの定義する

```
module AgdaBasics where
```

Agda における型指定は : を用いて行なう。例えば、変数 x が型 A を持つ、ということを表すには `x : A` と記述する。

データ型は Haskell や ML に似た代数的なデータ構造である。データ型の定義は `data` キーワードを用いる。`data` キーワードの後に `where` 句を書きインデントを深くした後、値にコンストラクタとその型を列挙する。例えば Bool 型を定義するとリスト 4.2 のようになる。Bool はコンストラクタ `true` か `false` を持つデータ型である。Bool 自身の型は `Set` であり、これは Agda が組み込みで持つ「型の型」である。Set は階層構造を持ち、型の型の型を指定するには Set1 と書く。

リスト 4.2: Agda におけるデータ型 Bool の定義

```
data Bool : Set where
  true  : Bool
  false : Bool
```

関数の定義は Haskell に近い。関数名と型を記述した後に関数の本体を `=` の後に指定する。関数の型は $\rightarrow$ を用いる。なお、$\rightarrow$ に対しては糖衣構文 `->` も用意されている。例えば引数が型 `A` で返り値が型 `B` の関数は `A -> B` のように書ける。Bool 変数 `x` を取って true を返す関数 `f` はリスト ??のようになる。

リスト 4.3: Agda における関数定義

```
f : Bool -> Bool
f x = true
```

引数は変数名で受けることもできるが、具体的なコンストラクタを指定することでそのコンストラクタが渡された時の挙動を定義できる。これはパターンマッチと呼ばれ、コンストラクタで case 文を行なっているようなものである。例えば Bool 型の値を反転する `not` 関数を書くとリスト 4.4 のようになる。

リスト 4.4: Agda における関数 not の定義

```
not : Bool -> Bool
not  true = false
not false = true
```

パターンマッチは全てのコンストラクタのパターンを含まなくてはならない。例えば、Bool 型を受け取る関数で `true` の時の挙動のみを書くことはできない。なお、コンストラクタをいくつか指定した後に変数で受けると、変数が持ちうる値は指定した以外のコンストラクタとなる。例えばリスト 4.5 の not は x には true しか入ることは無い。なお、マッチした値を変数として利用しない場合は `_` を用いて捨てることもできる。

リスト 4.5: Agda におけるパターンマッチ

```
not : Bool -> Bool
not false = true
not x     = false
```

関数にはリテラルが存在し、関数名を定義せずともその場で生成することができる。これをラムダ式と呼び、`\arg1 arg2 -> function body` のように書く。例えば Bool 型の引数 `b` を取って not を適用する `not-apply` をラムダ式で書くとリスト 4.6 のようになる。関数 `not-apply` をラムダ式を使わずに定義すると `not-apply-2` になるが、この二つの関数は同一の動作をする。

リスト 4.6: Agda におけるラムダ式

```
not-apply : Bool -> Bool
not-apply = (\b -> not b)   -- use lambda

not-apply : Bool -> Bool
not-appy b = not b          -- not use lambda
```

Agda では特定の関数内のみで利用する関数を `where` 句で記述できる。これは関数内部の冗長な記述を省略するのに活用できる。スコープは `where` 句が存在する関数内部のみであるため、名前空間が汚染させることも無い。例えば自然数 3 つを取ってそれぞれ 3 倍して加算する関数 `f` を定義するとき、 `where` を使うとリスト 4.7 のように書ける。これは `f'` と同様の動作をする。`where` 句は利用したい関数の末尾にインデント付きで `where` キーワードを記述し、改行の後インデントをして関数内部で利用する関数を定義する。

リスト 4.7: Agda における where 句

```
f : Int -> Int -> Int
f a b c = (t a) + (t b) + (t c)
  where
    t x = x + x + x

f' : Int -> Int -> Int
f' a b c = (a + a + a) + (b + b + b) + (c + c + c)
```

データ型のコンストラクタには自分自身の型を引数に取ることもできる (リスト 4.8)。自然数のコンストラクタは 2 つあり、片方は自然数ゼロ、片方は自然数を取って後続数を返すものである。例えば 0 は `zero` であり、1 は `suc zero` に、3 は `suc (suc (suc zero))` に対応する。

リスト 4.8: Agda における自然数の定義

```
data Nat : Set where
  zero : Nat
  suc  : Nat -> Nat
```

自然数に対する演算は再帰関数として定義できる。例えば自然数どうしの加算は二項演算子+としてリスト 4.9 のように書ける。

この二項演算子は正確には中置関数である。前置や後置で定義できる部分を関数名に `_` として埋め込んでおくことで、関数を呼ぶ時にあたかも前置や後置演算子のように振る舞う。例えば `!_` 関数を定義すると `! true` のように利用でき、`_~` 関数を定義すると `false ~` のように利用できる。

また、Agda は再帰関数が停止するかを判定できる。この加算の二項演算子は左側がゼロに対しては明らかに停止する。左側が 1 以上の時の再帰時には `suc n` から `n` へと減っているため、再帰で繰り返し減らすことでいつかは停止する。もし `suc n` のまま自分自身へと再帰した場合、Agda は警告を出す。

リスト 4.9: Agda における自然数の加算の定義

```
_+_ : Nat -> Nat -> Nat
zero  + m = m
suc n + m = suc (n + m)
```

次に依存型について見ていく。依存型で最も基本的なものは関数型である。依存型を利用した関数は引数の型に依存して返す型を決定できる。

Agda で `(x : A) -> B` と書くと関数は型 A を持つ x を受け取り、B を返す。ここで B の中で x を扱っても良い。例えば任意の型に対する恒等関数はリスト 4.10 のように書ける。

リスト 4.10: 依存型を持つ関数の定義

```
identity : (A : Set) -> A -> A
identity A x = x

identity-zero : Nat
identity-zero = identity Nat zero
```

この恒等関数 `identitiy` は任意の型に適用可能である。実際に関数 `identitiy` を Nat へ適用した例が `identitiy-zero` である。

多相の恒等関数では型を明示的に指定せずとも `zero` に適用した場合の型は自明に `Nat -> Nat` である。Agda はこのような推論をサポートしており、推論可能な引数は省略できる。推論によって解決される引数を暗黙的な引数 (implicit arguments) と言い、記号 `{}` でくくる。

例えば、`identitiy` の対象とする型 `A` を暗黙的な引数として省略するとリスト 4.11 のようになる。この恒等関数を利用する際は特定の型に属する値を渡すだけでその型が自動的に推論される。よって関数を利用する際は `id-zero` のように型を省略して良い。なお、関数の本体で暗黙的な引数を利用したい場合は `{variableName}` で束縛することもできる (id' 関数)。適用する場合も `{}`でくくり、`id-true` のように使用する。

リスト 4.11: Agda における暗黙的な引数を持つ関数

```
id : {A : Set} -> A -> A
id x = x
```

```
id-zero : Nat
id-zero = id zero

id' : {A : Set} -> A -> A
id' {A} x = x

id-true : Bool
id-true = id {Bool} true
```

Agda には C における構造体に相当するレコード型も存在する。定義を行なう際は `record` キーワードの後にレコード名、型、`where` の後に `field` キーワードを入れた後、フィールド名と型名を列挙する。例えば x と y の二つの自然数からなるレコード `Point` を定義するとリスト 4.12 のようになる。レコードを構築する際は `record` キーワードの後の `{}` の内部に `fieldName = value` の形で値を列挙していく。複数の値を列挙する際は `;` で区切る。

リスト 4.12: Agda におけるレコード型の定義

```
record Point : Set where
  field
    x : Nat
    y : Nat

makePoint : Nat -> Nat -> Point
makePoint a b = record { x = a ; y = b }
```

構築されたレコードから値を取得する際には `RecordName.fieldName` という名前の関数を適用する (リスト 4.13 内2行目) 。なお、レコードにもパターンマッチが利用できる (リスト 4.13 内5行目)。また、値を更新する際は `record oldRecord {field = value ; ... }` という構文を利用する。Point の中の x の値を 5 増やす関数 `xPlus5` はリスト 4.13 の 7,8 行目のように書ける。

リスト 4.13: Agda におけるレコードの射影、パターンマッチ、値の更新

```
1 getX : Point -> Nat
2 getX p = Point.x p
3
4 getY : Point -> Nat
5 getY record { x = a ; y = b} = b
6
7 xPlus5 : Point -> Point
8 xPlus5 p = record p { x = (Point.x p) + 5}
```

Agda における部分型のように振る舞う機能として Instance Arguments が存在する。これはとあるデータ型が、ある型と名前を持つ関数を持つことを保証する機能であり、Haskell における型クラスや Java におけるインターフェースに相当する。Agda における部分型の制約は、必要な関数を定義した record に相当し、その制約を保証するにはそ

の record を instance として登録することになる。例えば、同じ型と比較することができる、という性質を表すとリスト 4.14 のようになる。具体的にはとある型 A における中置関数 `_==_` を定義することに相当する。

リスト 4.14: Agda における部分型制約

```
record Eq (A : Set) : Set where
  field
    _==_ : A -> A -> Bool
```

ある型 T がこの部分型制約を満たすことを示すには、型 T でこのレコードを作成できることを示し、それを instance 構文で登録する。型 Nat が Eq の上位型であることを記述するとリスト 4.15 のようになる。

リスト 4.15: Agda における部分型関係の構築

```
_==Nat_ : Nat -> Nat -> Bool
zero    ==Nat zero    = true
(suc n) ==Nat zero    = false
zero    ==Nat (suc m) = false
(suc n) ==Nat (suc m) = n ==Nat m

instance
  natHas== : Eq Nat
  natHas== = record { _==_ = _==Nat_}
```

これで `Eq` が要求される関数に対して Nat が適用できるようになる。例えば型 A の要素を持つ List A から要素を探してくる elem を定義する。部分型のインスタンスは `{{}}` 内部に名前と型名で記述する。なお、名前部分は必須である。仮に変数として受けても利用しない場合は `_` で捨てると良い。部分型として登録した record は関数本体において `{{variableName}}` という構文で変数に束縛できる。

リスト 4.16: Agda における部分型を使う関数の定義

```
elem : {A : Set} {{eqA : Eq A}} → A → List A → Bool
elem {{eqA}} x (y :: xs) = (Eq._==_ eqA x y) || (elem {{eqA}} x xs)
elem         x []        = false
```

この `elem` 関数はリスト 4.17 のように利用できる。Nat 型の要素を持つリストの内部に 4 が含まれるか確認している。この `listHas4` は `true` に評価される。

リスト 4.17: 部分型を持つ関数の適用

```
listHas4 : Bool
listHas4 = elem 4 (3 :: 2 :: 5 :: 4 :: []) -- true
```

最後にモジュールについて述べる。モジュールはほとんど名前空間として作用する。なお、依存型の解決はモジュールのインポート時に行なわれる。モジュールをインポートする時は `import` キーワードを指定する。また、インポートを行なう際に名前を別名に変更

することもでき、その際は `as` キーワードを用いる。モジュールから特定の関数のみをインポートする場合は `using` キーワードを、関数の名前を変える時は `renaming` キーワードを、特定の関数のみを隠す場合は `hiding` キーワードを用いる。なお、モジュールに存在する関数をトップレベルで用いる場合は `open` キーワードを使うことで展開できる。モジュールをインポートする例をリスト 4.18 に示す。

リスト 4.18: Agda におけるモジュールのインポート

```
import Data.Nat                    -- import module
import Data.Bool as B              -- renamed module
import Data.List using (head)      -- import Data.head function
import Level renaming (suc to S) -- import module with rename suc to S
import Data.String hiding (_++_) -- import module without _++_
open import Data.List              -- import and expand Data.List
```

また、モジュールには値を渡すことができる。そのようなモジュールは Parameterized Module と呼ばれ、渡された値はそのモジュール内部で一貫して扱える。例えば要素の型と比較する二項演算子を使って並べ替えをするモジュール `Sort` を考える。そのモジュールは引数に型 A と二項演算子 `<`を取り、ソートする関数を提供する。`Sort` モジュールを `Nat` と `Bool` で利用した例がリスト 4.19 である。

リスト 4.19: Agda における Parameterized Module

```
module Sort (A : Set) (_<_ : A -> A -> Bool) where
  sort : List A -> List A
  sort = ...

open import Sort Nat  Nat._<_  as N
open import Sort Bool Bool._<_ as B
```

4.3 Natural Deduction

まず始めに証明を行なうために Natural Deduction(自然演繹) を示す。

Natural Deduction は Gentzen によって作られた論理と、その証明システムである [18]。命題変数と記号を用いた論理式で論理を記述し、推論規則により変形することで求める論理式を導く。

natural deduction において

$$\begin{array}{c} \vdots \\ A \end{array} \tag{4.1}$$

と書いた時、最終的に命題 A を証明したことを意味する。証明は木構造で表わされ、葉の命題は仮定となる。仮定には dead か alive の 2 つの状態が存在する。

$$\begin{array}{c} A \\ \vdots \\ B \end{array} \tag{4.2}$$

式 4.2 のように A を仮定して B を導いたとする。この時 A は alive な仮定であり、証明された B は A の仮定に依存していることを意味する。

ここで、推論規則により記号 $\Rightarrow$ を導入する。

$$\frac{\begin{array}{c}[A] \\ \vdots \\ B\end{array}}{A \Rightarrow B} \Rightarrow \mathcal{I}$$

$\Rightarrow \mathcal{I}$ を適用することで仮定 A は dead となり、新たな命題 $A \Rightarrow B$ を導くことができる。A という仮定に依存して B を導く証明から、「A が存在すれば B が存在する」という証明を導いたこととなる。このように、仮定から始めて最終的に全ての仮定を dead とすることで、仮定に依存しない証明を導ける。なお、dead な仮定は `[A]` のように `[]` で囲んで書く。

alive な仮定を dead にすることができるのは $\Rightarrow \mathcal{I}$ 規則のみである。それを踏まえ、natural deduction には以下のような規則が存在する。

- Hypothesis

  仮定。葉にある式が仮定となるため、論理式 A を仮定する場合に以下のように書く。

  A

- Introductions

  導入。証明された論理式に対して記号を導入することで新たな証明を導く。

$$\frac{\begin{array}{c}\vdots \\ A\end{array} \quad \begin{array}{c}\vdots \\ B\end{array}}{A \wedge B} \wedge\mathcal{I}$$

$$\cfrac{\begin{array}{c}\vdots\\ A\end{array}}{A \lor B}\ \lor 1\mathcal{I}$$

$$\cfrac{\begin{array}{c}\vdots\\ B\end{array}}{A \lor B}\ \lor 2\mathcal{I}$$

$$\cfrac{\begin{array}{c}[A]\\ \vdots\\ B\end{array}}{A \Rightarrow B} \Rightarrow \mathcal{I}$$

- Eliminations

  除去。ある論理記号で構成された証明から別の証明を導く。

$$\cfrac{\begin{array}{c}\vdots\\ A \land B\end{array}}{A}\ \land 1\mathcal{E}$$

$$\cfrac{\begin{array}{c}\vdots\\ A \land B\end{array}}{B}\ \land 2\mathcal{E}$$

$$\cfrac{\begin{array}{c}\\ \vdots\\ A \lor B\end{array}\quad \begin{array}{c}[A]\\ \vdots\\ C\end{array}\quad \begin{array}{c}[B]\\ \vdots\\ C\end{array}}{C}\ \lor\mathcal{E}$$

$$\cfrac{\begin{array}{c}\vdots\\ A\end{array}\quad \begin{array}{c}\vdots\\ A \Rightarrow B\end{array}}{B} \Rightarrow \mathcal{E}$$

記号 $\lor, \land, \Rightarrow$ の導入の除去規則について述べた。natural deduction には他にも $\forall, \exists, \bot$ といった記号が存在するが、ここでは解説を省略する。

それぞれの記号は以下のような意味を持つ

- $\land$ conjunction。2 つの命題が成り立つことを示す。$A \land B$ と記述すると A かつ B と考えることができる。

- $\lor$ disjunction。2 つの命題のうちどちらかが成り立つことを示す。$A \lor B$ と記述すると A または B と考えることができる。

- $\Rightarrow$ implication。左側の命題が成り立つ時､右側の命題が成り立つことを示す。$A \Rightarrow B$ と記述すると A ならば B と考えることができる。

例として、natural deduction で三段論法を証明する。なお、三段論法とは「A は B であり、 B は C である。よって A は C である」といった文を示す。

$$\cfrac{\cfrac{\cfrac{\cfrac{[A]_{(1)} \qquad \cfrac{[(A \Rightarrow B) \land (B \Rightarrow C)]_{(2)}}{(A \Rightarrow B)} \land 1\mathcal{E}}{B} \qquad \cfrac{[(A \Rightarrow B) \land (B \Rightarrow C)]_{(2)}}{(B \Rightarrow C)} \land 2\mathcal{E}}{C}}{A \Rightarrow C} \Rightarrow \mathcal{I}_{(1)}}{((A \Rightarrow B) \land (B \Rightarrow C)) \Rightarrow (A \Rightarrow C)} \Rightarrow \mathcal{I}_{(2)}$$

まず、三段論法を論理式で表す。

「A は B であり、B は C である。よって A は C である」が証明するべき命題である。まず､「A は B であり」から､A から性質 B が導けることが分かる。これが $A \Rightarrow B$ となる。次に､「B は C である」から､B から性質 C が導けることが分かる。これが $B \Rightarrow C$ となる。そしてこの 2 つは同時に成り立つ。よって $(A \Rightarrow B) \land (B \Rightarrow C)$ が仮定となる。この仮定が成り立つ時に「A は C である」を示せば良い。仮定と同じように「A は C である」は､$A \Rightarrow C$ と書けるため､証明するべき論理式は $((A \Rightarrow B) \land (B \Rightarrow C)) \Rightarrow (A \Rightarrow C)$ となる。

証明の手順はこうである。まず条件 $(A \Rightarrow B) \land (B \Rightarrow C)$ と A の 2 つを仮定する。条件を $\land 1\mathcal{E}$ $\land 2\mathcal{E}$ により分解する。A と $A \Rightarrow B$ から B を、 B と $B \Rightarrow C$ から C を導く。ここで $\Rightarrow \mathcal{I}$ により $A \Rightarrow C$ を導く。この際に dead にする仮定は A である。数回仮定を dead にする際は $_{(1)}$ のように対応する [] の記号に数値を付ける。これで残る alive な仮定は $(A \Rightarrow B) \land (B \Rightarrow C)$ となり、これから $A \Rightarrow C$ を導くことができたためにさらに $\Rightarrow \mathcal{I}$ を適用する。結果、証明すべき論理式 $((A \Rightarrow B) \land (B \Rightarrow C)) \Rightarrow (A \Rightarrow C)$ が導けたために証明終了となる。

4.4 Curry-Howard Isomorphism

4.3 節では natural deduction における証明手法について述べた。natural deduction における証明はほとんど型付き λ 計算のような形をしている。実際、Curry-Howard Isomorphism により Natural Deduction と型付き λ 計算は対応している。Curry-Howard Isomorphism の概要を 4.4 節に述べる。

関数型 $\rightarrow$ のみに注目した時

1. 導入規則 (T-ABS) は、その型の要素がどのように作られるかを記述する
2. 除去規則 (T-APP) は、その型の要素がどのように作られるかを記述する

例えば命題 A が成り立つためには A という型を持つ値が存在すれば良い。しかしこの命題は A という alive な仮定に依存している。natural deduction では A の仮定を dead にするために $\Rightarrow \mathcal{I}$ により $\Rightarrow$ を導入する。これが λ による抽象化 (T-ABS) に対応している。

$$\begin{array}{c} x : A \\ \lambda x.x : A \rightarrow A \end{array}$$

プログラムにおいて、変数 x は内部の値により型が決定される。特に、x の値が未定である場合は未定義の変数としてエラーが発生する。しかし、 x を取って x を返す関数は定義することはできる。これは natural deduction の $\Rightarrow \mathcal{I}$ により仮定を discharge することに相当する。

また、仮定 A が成り立つ時に結論 B を得ることは、関数適用 (T-APP) に相当している。

$$\frac{A \qquad A \rightarrow B}{B}\ \text{T-APP}$$

このように、 natural deduction における証明はそのまま 型付き λ 計算に変換することができる。

それぞれの詳細な対応は省略するが、表 4.1 のような対応が存在する。

	natural deduction	型付き λ 計算
hypothesis	A	型 A を持つ変数 x
conjunction	$A \wedge B$	型 A と型 B の直積型 を持つ変数 x
disjunction	$A \vee B$	型 A と型 B の直和型 を持つ変数 x
implication	$A \Rightarrow B$	型 A を取り型 B の変数を返す関数 f

表 4.1: natural deuction と 型付き λ 計算との対応 (Curry-Howard Isomorphism)

4.5 Reasoning

次に Agda における証明を記述していく。例題として、自然数の加法の可換法則を示す。

証明を行なうためにまずは自然数を定義する。今回用いる自然数の定義は以下のようなものである。

- 0 は自然数である
- 任意の自然数には後続数が存在する
- 0 はいかなる自然数の後続数でもない
- 異なる自然数どうしの後続数は異なる ($n \neq m \to Sn \neq Sm$)
- 0 がある性質を満たし、a がある性質を満たせばその後続数 S(n) も自然数である

この定義は peano arithmetic における自然数の定義である。

Agda で自然数型 Nat を定義するとリスト 4.20 のようになる。

リスト 4.20: Agda における自然数型 Nat の定義

```
module nat where

data Nat : Set where
  O : Nat
  S : Nat -> Nat
```

自然数型 Nat は 2 つのコンストラクタを持つ。

- O

  引数を持たないコンストラクタ。これが 0 に相当する。

- S

  Nat を引数に取るコンストラクタ。これが後続数に相当する。

よって、数値の 3 は `S (S (S O))` のように表現される。S の個数が数値に対応する。

次に加算を定義する (リスト 4.21)。

リスト 4.21: Agda における自然数型に対する加算の定義

```
open import nat
module nat_add where

_+_ : Nat -> Nat -> Nat
O + m     = m
(S n) + m = S (n + m)
```

加算は中置関数 `_+_` として定義する。2 つの Nat を取り、Nat を返す。関数 `_+_` はパターンマッチにより処理を変える。0 に対して m 加算する場合は m であり、 n の後続数に対して m 加算する場合は n に m 加算した数の後続数とする。S を左の数から右の数へ 1 つずつ再帰的に移していくような加算である。

例えば 3 + 1 といった計算は (S (S (S O))) + (S O) のように記述される。ここで 3 + 1 が 4 と等しいことの証明を行なう。

等式の証明には agda の standard library における Relation.Binary.Core の定義を用いる。

リスト 4.22: Relation.Binary.Core による等式を示す型 ≡

```
data _≡_ {a} {A : Set a} (x : A) : A → Set a where
  refl : x ≡ x
```

Agda において等式は、等式を示すデータ型 ≡ により定義される。≡ は同じ両辺が同じ項に簡約される時にコンストラクタ refl で構築できる。

実際に 3 + 1 = 4 の証明は refl で構成できる (リスト 4.23)。

リスト 4.23: Agda における 3 + 1 の結果が 4 と等しい証明

```
open import Relation.Binary.PropositionalEquality
open import nat
open import nat_add

module three_plus_one where

3+1 : (S (S (S O)))  +  (S O) ≡  (S (S (S (S O))))
3+1 = refl
```

3+1 という関数を定義し、その型として証明すべき式を記述し、証明を関数の定義として定義する。証明する式は `(S (S (S O))) + (S O)≡(S (S (S (S O))))` である。今回は `_+_` 関数の定義により `(S (S (S (S O))))` に簡約されるためにコンストラクタ refl が証明となる。

≡ によって証明する際、必ず同じ式に簡約されるとは限らないため、いくつかの操作が Relation.Binary.PropositionalEquality に定義されている。

- sym : $x \equiv y \rightarrow y \equiv x$

  等式が証明できればその等式の左辺と右辺を反転しても等しい。

- cong : $f \rightarrow x \equiv y \rightarrow fx \equiv fy$

  証明した等式に同じ関数を与えても等式は保たれる。

- trans : $x \equiv y \rightarrow y \equiv z \rightarrow x \equiv z$

  2 つの等式に表れた同じ項を用いて 2 つの等式を繋げた等式は等しい。

ではこれから nat の加法の交換法則を証明していく (リスト 4.24)。

リスト 4.24: Agda における加法の交換法則の証明

```
open import Relation.Binary.PropositionalEquality
open import nat
open import nat_add
open ≡-Reasoning

module nat_add_sym where

addSym : (n m : Nat) -> n + m ≡  m + n
addSym O       O   = refl
addSym O    (S m)  = cong S (addSym O m)
addSym (S n)   O   = cong S (addSym n O)
addSym (S n) (S m) = {!!} -- 後述
```

証明する式は $n + m \equiv m + n$ である。n と m は Nat であるため、それぞれがコンストラクタ O か S により構成される。そのためにパターンは 4 通りある。

- n = O, m = O

  `_+_` の定義により、 O に簡約されるため refl で証明できる。

- n = O, m = S m

  $O + (Sm) \equiv (Sm) + O$ を証明することになる。この等式は `_+_` の定義により $O + (Sm) \equiv S(m + O)$ と変形できる。$S(m + O)$ は $m + O$ に S を加えたものであるため、 cong を用いて再帰的に `addSym` を実行することで証明できる。

  この 2 つの証明はこのような意味を持つ。n が 0 であるとき、m も 0 なら簡約により等式が成立する。n が 0 であり、 m が 0 でないとき、 m は後続数である。よって m が (S x) と書かれる時、 x は m の前の値である。前の値による交換法則を用いてからその結果の後続数も `_+_` の定義により等しい。

  ここで、 `addSym` に渡される m は 1 つ値が減っているため、最終的には n = 0, m = 0 である refl にまで簡約され、等式が得られる。

- n = S n, m = O

  $(Sn) + O \equiv O + (Sn)$ を証明する。この等式は `_+_` の定義により $S(n + O) \equiv (Sn)$ と変形できる。さらに変形すれば $S(n + O) \equiv S(O + n)$ となる。よって `addSym` により O と n を変換した後に cong で S を加えることで証明ができる。

  ここで、 $O + n \equiv n$ は `_+_` の定義により自明であるが、$n + O \equiv n$ をそのまま導出できないことに注意して欲しい。`_+_` の定義は左側の項から S を右側の項への移すだけであるため、右側の項への演算はしない。

- n = S n, m = S m

  3 つのパターンは証明したが、このパターンは少々長くなるため別に解説することとする。

3 つのパターンにおいては refl や cong といった単純な項で証明を行なうことができた。しかし長い証明になると refl や cong といった式を trans で大量に繋げていく必要性がある。長い証明を分かりやすく記述するために ≡-Reasoning を用いる。

≡-Reasoning では等式の左辺を begin の後に記述し、等式の変形を $\equiv \langle \mathit{expression} \rangle$ に記述することで変形していく。最終的に等式の左辺を ■ の項へと変形することで等式の証明が得られる。

自然数の加法の交換法則を ≡-Reasoning を用いて証明した例がリスト 4.25 である。特に n と m が 1 以上である時の証明に注目する。

リスト 4.25: ≡ - Reasoning を用いた証明の例

```
open import Relation.Binary.PropositionalEquality
open import nat
open import nat_add
open ≡-Reasoning

module nat_add_sym_reasoning where

addToRight : (n m : Nat) → S (n + m) ≡ n + (S m)
addToRight O m     = refl
addToRight (S n) m = cong S (addToRight n m)

addSym : (n m : Nat) → n + m ≡ m + n
addSym O       O   = refl
addSym O    (S m)  = cong S (addSym O m)
addSym (S n)   O   = cong S (addSym n O)
addSym (S n) (S m) = begin
  (S n) + (S m)  ≡⟨ refl ⟩
  S (n + S m)    ≡⟨ cong S (addSym n (S m)) ⟩
  S ((S m) + n)  ≡⟨ addToRight (S m) n ⟩
  S (m + S n)    ≡⟨ refl ⟩
  (S m) + (S n)  ■
```

まず `(S n) + (S m)` は `_+_` の定義により `S (n + (S m))` に等しい。よって refl で導かれる。なお、基本的に定義などで同じ項に簡約される時は refl によって記述することが多い。

次に `S (n + (S m))` に対して addSym を用いて交換し、 cong によって S を追加することで `S ((S m) + n)` を得る。これは、前 3 パターンにおいて + の右辺の項が 1 以上であっても上手く交換法則が定義できたことを利用して項を変形している。このように同じ法則の中でも、違うパターンで証明できた部分を用いて別パターンの証明を行なうこともある。

最後に `S ((S m) + n)` から `(S m) + (S n)` を得なくてはならない。しかし、 `_+_` の定義には右辺に対して S を移動する演算が含まれていない。よってこのままでは証明することができない。そのため、等式 $S(m+n) \equiv m+(Sn)$ を addToRight として定義する。addToRight はコンストラクタによる分岐を用いて証明できる。値が 0 であれば自明に成り立ち、1 以上であれば再帰的に addToRight を適用することで任意の数に対して成り立つ。addToRight を用いることで `S ((S m) + n)` から `(S m) + (S n)` を得られた。これで等式 $(Sm)+(Sn) \equiv (Sn)+(Sm)$ の証明が完了した。

自然数に対する $+$ の演算を考えた時にありえるコンストラクタの組み合せ 4 パターンのいずれかでも交換法則の等式が成り立つことが分かった。このように、Agda における等式の証明は、定義や等式を用いて右辺と左辺を同じ項に変形することで行なわれる。

第5章　Agda における Continuation based C の表現

4 章では Curry-Howard 同型対応により、型付きラムダ計算を用いて命題が証明できることを示した。加えて、証明支援系言語 Agda を用いてデータの定義とそれを扱う関数の性質の証明が行なえることを確認した。

この章では CbC の型システムを部分型を利用して定義する。定義した型システムがきちんと検査されるかを確認するため、Agda 上に DataSegment と CodeSegment の定義、CodeSegment の接続と実行、メタ計算を定義し、型付けされることを確認する。また、Agda 上で定義した DataSegment とそれに付随する CodeSegment の持つ性質を Agda で証明する。

5.1　部分型付け

単純型付きラムダ計算では、ラムダ計算の項が型付けされることを確認した。ここで、単純型の拡張として、レコードを導入する。

レコードとは名前の付いた複数の値を保持するデータである。C 言語における構造体などがレコードに相当する。値を保持する各フィールド t_1 はあらかじめ定められた集合 L からラベル l_i を名前として持つ。例えば $x : Nat$ や $no : 100, point33$ などがこれに相当する。なお、あるレコードの項や値に表れるラベルはすべて相異なるとする。レコードから値を取り出す際にはラベルを指定して値を射影する。

レコードの拡張の定義は以下である。

定義 **5.1** レコードの拡張に用いる新しい構文形式

$$
\begin{array}{rr}
t ::= ... & 項 : \\
\{l_i = t_i^{i\in 1..n}\} & レコード \\
t.l & 射影
\end{array}
$$

$$
\begin{array}{rr}
v ::= ... & 値 : \\
l_i : v_i^{i\in 1..n} & レコードの値
\end{array}
$$

$$
\begin{array}{rr}
T ::= ... & 型 : \\
\{l_i : T_i^{i\in 1..n}\} & レコードの型
\end{array}
$$

定義 **5.2** レコードの拡張に用いる新しい評価規則

$$
\{l_i = v_i^{i\in 1..n}.l_j \rightarrow v_j\} \quad \text{E-PROJRCD}
$$

$$
\frac{t_1 \rightarrow t'_1}{t_1.l \rightarrow t'_1.l} \quad \text{E-PROJ}
$$

$$
\frac{t_j \rightarrow t'_j}{\{l_i = v_i^{i\in 1..j-1}, l_j = t_j, l_k = t_k^{k\in j+1..n}\} \rightarrow \{l_i = v_i^{i\in 1..j-1}, l_j = t'_j, l_k = t_k^{k\in j+1..n}\}} \quad \text{E-RCD}
$$

定義 **5.3** レコードの拡張に用いる新しい型付け規則

$$\frac{\text{各 i に対して}\ \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i = t_i{}^{i \in 1..n} : \{l_i : T_i{}^{i \in 1..n}\}} \qquad \text{T-RCD}$$

$$\frac{\Gamma \vdash t_1 : \{l_i : T_i{}^{i \in 1..n}\}}{\Gamma \vdash t_1.lj : T_j} \qquad \text{T-PROJ}$$

レコードを用いることで複数の値を一つの値としてまとめて扱うことができる。しかし、引数にレコードを取る場合、その型と完全に一致させる必要がある。例えば $\{x : Nat\}$ を引数に取る関数に対して $\{x : Nat, y : Bool\}$ といった値を適用することができない。しかし、直感的には関数の要求はフィールド x が型 Nat を持つことのみであり、その部分にのみ注目すると $\{x : Nat, y : Bool\}$ も要求に沿っている。

部分型付けの目標は上記のような場合の項を許すように型付けを改良することにある。つまり型 S の任意の項が、型 T の項が期待される文脈において安全に利用できることを示す。この時、S を T の部分型と呼び、 $S <: T$ と書く。これは型 S が型 T よりも情報を多く持っていることを示しており、S は T の部分型である、と読む。$S <: T$ の別の読み方として、型 T は型 S の上位型である、という読み方も存在する。

型付け関係と部分型関係をつなぐための新しい型付け規則を考える。

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T} \qquad \text{T-SUB}$$

この規則は $S <: T$ ならば S 型の要素 t はすべて T の要素であると言っている。例えば、先程の $\{x : Nat\}$ と $\{x : Nat, y : Bool\}$ が $\{x : Nat, y : Bool\} <: \{x : Nat\}$ となるように部分型関係を定義した時に $\Gamma \vdash \{x = 0, y = 1\} : \{x : Nat\}$ を導ける。

部分型関係は $S <: T$ という形式の部分型付け式を導入するための推論規則の集まりとして定式化される。始めに、部分型付けの一般的な規定から考える。部分型は反射的であり、推移的である。

$$S <: S \qquad \text{S-REFL}$$

$$\frac{S <: U \qquad U <: T}{S <: T} \qquad \text{S-TRANS}$$

これらの規則は安全代入に対する直感より正しい。次に、基本型や関数型、レコード型などの型それぞれに対して、その形の型の要素が期待される部分に対して上位型の要素を利用しても安全である、という規則を導入していく。

レコード型については型 $T = \{l_i : T_1...l_n : T_n\}$ が持つフィールドが型 $S = \{k_1 : S_1...k_n : T_n\}$ のものよりも少なければ S を T の部分型とする。つまりレコードの終端フィールドのいくつかを忘れてしまっても安全である、ということを意味する。この直感は幅部分型付け規則となる。

$$\{l_i : T_i^{i \in 1..n+k}\} <: \{l_i : T_i^{i \in 1..n}\} \qquad \text{S-RCDWIDTH}$$

フィールドの多い方が部分型となるのは名前に反するように思える。しかし、フィールドが多いレコードほど制約が多くなり表すことのできる集合の範囲は小さくなる。集合の大きさで見ると明かにフィールドの多い方が小さいのである。

幅部分型付け規則が適用可能なのは、共通のフィールドの型が全く同じな場合のみである。しかし、その場合フィールドの型に部分型を導入できず、フィールドの型の扱いで同じ問題を抱えることとなる。そのために導入するのが深さ部分型付けである。二つのレコードの中で対応するフィールドの型が部分型付け関係にある時に個々のフィールドの型が異なることを許す。これは具体的には以下の規則となる。

$$\frac{\text{各 i に対して } S_i <: T_i}{\{l_i : S_i^{i \in 1..n}\} <: \{l_i : T_i^{i \in 1..n}\}} \qquad \text{S-RCDDEPTH}$$

これらを用いて $\{x : \{a : Nat, b : Nat\}, y : \{m : Nat\}\}$ が $\{x : \{a : Nat\}, y : \{\}\}$ の部分型であることは以下のように導出できる。

$$\dfrac{\dfrac{}{\{a : Nat, b : Nat\} <: \{a : Nat\}}\,\text{S-RCDWIDTH} \qquad \dfrac{}{\{m : Nat\} <: \{\}}\,\text{S-RCDWIDTH}}{\{x : \{a : Nat, b : Nat\}, y : \{m : Nat\}\} <: \{x : \{a : Nat\}, y : \{\}\}}\,\text{S-RCDDEPTH}$$

最後に、レコードを利用する際はフィールドさえ揃っていれば順序は異なっても良いという規則を示す。

$$\frac{\{k_j : S_j^{i \in 1..n}\} \text{ は } \{l_i : T_i^{i \in 1..n}\} \text{ の並べ替えである}}{\{k_j : S_j^{j \in 1..n}\} <: \{l_i : T_i^{i \in 1..n}\}} \qquad \text{S-RCDPERM}$$

S-RCDPERM を用いることで、終端フィールドだけではなく任意の位置のフィールドを削ることができる。

レコードの部分型は定義できたので、次は関数の部分型を定義していく。関数の部分型は以下 S-ARROW として定義できる。

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2} \qquad \text{S-ARROW}$$

前提の条件二つを見ると部分型の関係が逆転している。左側の条件は関数型自身の型と逆になっているため反変と言い、返り値の型は同じ向きであるため共変と言う。引数について考えた時、求める型よりも大きい型であれば明らかに安全に呼ぶことができるために関数の引数の型の方が上位型になる。返り値については関数が返す型が部分型であれば上位型を返すことができるため、関数の返り値の方が部分型になる。

別の見方をすると、型 $S_1 \rightarrow S_2$ の関数を別の型 $T_1 \rightarrow T_2$ が期待される文脈で用いることが安全な時とは

- 関数に渡される引数がその関数にとって想定外でない ($T_1 <: S_1$)
- 関数が返す値も文脈にとって想定外でない ($S_2 <: T_2$)

という場合に限る。

5.2 部分型と Continuation based C

部分型を用いて Conituation based C の型システムを定義していく。

まず、DataSegment の型から定義してく。DataSegment 自体は C の構造体によって定義されているため、レコード型として考えることができる。例えばリスト 3.5 に示していた DataSegment の一つに注目する (リスト 5.1)。

リスト 5.1: akashaContext の DataSegment である AkashaInfo

```
struct AkashaInfo {
    unsigned int minHeight;
    unsigned int maxHeight;
    struct AkashaNode* akashaNode;
};
```

この AkashaInfo は $\{minHeight : unsigned\ int, maxHeight : unsigned\ int, akashaNode : AkashaNode*\}$ であると見なせる。CodeSegment は DataSegment を引数に取るため、DataSegment の型は CodeSegment が要求する最低限の制約をまとめたものと言える。

次に Meta DataSegment について考える。Meta DataSegment はプログラムに出現する DataSegment の共用体であった。これを DataSegment の構造体に変更する。こうすることにより、Meta DataSegment はプログラム中に出現する DataSegment を必ず持つため、Meta DataSegment は任意の DataSegment の部分型となる。もしくは各 DataSegment の全てのフィールドを含むような 1 つの構造体でも良い。第 5 章における Meta DataSegment はそのように定義している。なお、GearsOS では DataSegment の共用体をプログラムで必要な数だか持つ実装になっている。

具体的な CbC における Meta DataSegment である Context (リスト 5.2) は、DataSegment の集合を値として持っているために明らかに DataSegment よりも多くの情報を持っている。

リスト 5.2: CbC の Meta DataSegment である Context

```
struct Data { /* data segments as types */
    struct Tree { /* ... */ } tree;
    struct Node { /* ... */ } node;

    struct IterElem { /* ..  */ } iterElem;
    struct Iterator { /* ... */ } iterator;
    struct AkashaInfo { /* ... */} akashaInfo;
    struct AkashaNode { /* ... */} akashaNode;
};


struct Context { /* meta data segment as subtype */
    /* ... */
    struct Data **data;
};
```

部分型として定義するなら以下のような定義となる。

定義 **5.4** Meta DataSegment の定義

$$Meta\ DataSegment <: DataSegment_i^{i \in N} \qquad \text{S-MDS}$$

なお、N はあるプログラムに出現するデータセグメントの名前の集合であるとする。

次に CodeSegment の型について考える。CodeSegment は DataSegment を DataSegment へと移す関数型とする。

定義 **5.5** CodeSegment の定義

$$DataSegment \rightarrow DataSegment \qquad \text{T-CS}$$

そして Meta CodeSegment は Meta DataSegment を Meta DataSegment へと移す関数となる。

定義 **5.6** Meta CodeSegment の定義

$$Meta\ DataSegment \rightarrow Meta\ DataSegment \qquad \text{T-MCS}$$

ここで具体的なコード (リスト 5.3) と比較してみる。

リスト 5.3: 具体的な CbC における CodeSegment

```
__code getMinHeight_stub(struct Context* context) {
    goto getMinHeight(context, &context->data[Allocate]->allocate, &
   context->data[AkashaInfo]->akashaInfo);
}

__code getMinHeight(struct Context* context, struct Allocate* allocate,
   struct AkashaInfo* akashaInfo) {
    /* ... */
    goto getMinHeight_stub(context);
}
```

CodeSegment `getMinHeight` は DataSegment `Allocate` と `AkashaInfo` を引数に取っている。現状は `Context` も継続のために渡しているが、本来ノーマルレベルからはアクセスできないために隠れているとする。その場合、引数の型は $\{allocate : Allocate, akashaInfo : AkahsaInfo\}$ となる。また、返り値は構文的には存在していないが、軽量継続で渡す値は $Context$ である。よって `getMinHeight` の型は $\{allocate : Allocate, akashaInfo : AkahsaInfo\} \rightarrow Context$ となる。$Context$ の型は Meta DataSegment なので、subtype の S-ARROW より $Context$ の上位型への変更ができる。

$\{allocat; : Allocate, akashaInfo : AkahsaInfo\}$ を X と置いて、`getMinHeignt` を $X \rightarrow X$ とする際の導出木は以下である。

$$\dfrac{\dfrac{}{X <: X}\,\text{S-REFL} \qquad \dfrac{}{Conttext <: X}\,\text{S-MDS}}{X \rightarrow Context <: X \rightarrow X}\,\text{S-ARROW}$$

返り値部分を部分型として定義することにより、軽量継続先が上位型であればどの CodeSegment へと遷移しても良い。プログラムによっては遷移先は確定しているために部分型にする必要性は無いが、メタ計算やライブラリなどの遷移先が不定の場合は一度部分型として確定し、その後コンパイル時やランタイム時に包摂関係から具体型を導けば良い。例えばコンパイル時に解決すればライブラリの静的リンク時実行コード生成が行なえ、ランタイム時に解決すればネットワークを経由するプログラムとの接続初期化に利用できる。例えば、プロトコルがバージョンを持つ場合に接続先のプログラムが利用しているプロトコルと互換性があるかの判断を Context どうしの部分型関係で判断できる。

また、stub のみに注目すると、stub は Context から具体的な DataSegment X を取り出す操作に相当し、S-ARROW の左側の前提のような振舞いをする。加えて、軽量継続する際に X の計算結果を Context に格納してから goto する部分を別の Meta CodeSegment として分離すると、S-ARROW の右側の前提のような振舞いを行なう。このようにノーマルレベルの CodeSegment の先頭と末尾にメタ計算を接続することによってノーマルレベルの CodeSegment が型付けできる。型付けに DataSegment の集合としての Meta DataSegment が必須になるが、これは構造体として定義可能なためコンパイル時に生成することで CbC に組み込むことができる。

なお、メタ計算に利用する Meta DataSegment と Meta DataSegment も同様に型付けできる。ここで興味深いのはあるレベルの CodeSegment は同レベルの DataSegment において型付けされるが、一つ上の階層から見ると、下の階層の DataSegment として一貫して扱えることにある。このようにメタ計算を階層化することにより、メタ計算で拡張された計算に対しても他のメタ計算が容易に適用できる。

5.3 DataSegment の定義

まず DataSegment から定義していく。DataSegment はレコード型で表現できるため、Agda のレコードをそのまま利用できる。例えは 2.1 に示していた a と b を加算して c を出力するプログラムに必要な DataSegment を記述すると 5.4 のようになる。cs0 は a と b の二つの Int 型の変数を利用するため、対応する ds0 は a と b のフィールドを持つ。cs1 は計算結果を格納する c という名前の変数のみを持つので、同様に ds1 も c のみを持つ。

リスト 5.4: Agda における DataSegment の定義

```
record ds0 : Set where
  field
    a : Int
    b : Int

record ds1 : Set where
  field
    c : Int
```

5.4 CodeSegment の定義

次に CodeSegment を定義する。CodeSegment は DataSegment を取って DataSegment を返すものである。よって $I \rightarrow O$ を内包するデータ型を定義する。

レコード型の型は Set なので、Set 型を持つ変数 I と O を型変数に持ったデータ型 CodeSegment を定義する。I は Input DataSegment の型であり、 O は Output DataSegment である。

CodeSegment 型のコンストラクタには `cs` があり、Input DataSegment を取って Output DataSegment を返す関数を取る。具体的なデータ型の定義はリスト 5.5 のようになる。

リスト 5.5: Agda における CodeSegment 型の定義

```
data CodeSegment {l1 l2 : Level} (I : Set l1) (O : Set l2) : Set (l ⊔ l1 ⊔ l2) where
  cs : (I → O) → CodeSegment I O
```

この CodeSegment 型を用いて CodeSegment の処理本体を記述する。

まず計算の本体となる cs0 に注目する。cs0 は二つの Int 型変数を持つ ds0 を取り、一つの Int 型変数を作った上で cs1 に軽量継続を行なう。DataSegment はレコードなので、a と b のフィールドから値を取り出した上で加算を行ない、c を持つレコードを生成する。そのレコードを引き連れたまま cs1 へと goto する。

次に cs1 に注目する。cs1 は値に触れず cs2 へと goto するだけである。よって何もせずにそのまま goto する関数をコンストラクタ `cs` に渡すだけで良い。

最後に cs2 である。cs2 はリスト 2.1 では省略していたが、今回は計算を終了させる CodeSegment として定義する。どの CodeSegment にも軽量継続せずに値を持ったまま計算を終了させる。コンストラクタ `cs` には関数を与えなくては値を構成できないため、何もしない関数である id を渡している。

最後に計算をする cs0 へと軽量継続する main を定義する。例として、 a の値を 100 とし、 b の値を 50 としている。

cs0, cs1, cs2, main を Agda で定義するとリスト 5.6 のようになる。

リスト 5.6: Agda における CodeSegment の定義

```
cs2 : CodeSegment ds1 ds1
cs2 = cs id

cs1 : CodeSegment ds1 ds1
cs1 = cs (\d -> goto cs2 d)

cs0 : CodeSegment ds0 ds1
cs0 = cs (\d -> goto cs1 (record {c = (ds0.a d) + (ds0.b d)}))

main : ds1
main = goto cs0 (record {a = 100 ; b = 50})
```

正しく計算が行なえたなら値 150 が得られるはずである。

5.5 ノーマルレベル計算の実行

プログラムを実行することは `goto` を定義することと同義である。軽量継続 `goto` の性質としては

- 次に実行する CodeSegment を指定する
- CodeSegment に渡すべき DataSegment を指定する
- 現在実行している CodeSegment から制御を指定された CodeSegment へと移動させる

がある。Agda における CodeSegment の本体は関数である。関数をそのまま使用すると再帰を許してしまうために CbC との対応が失われてしまう。よって、goto を利用できるのは関数の末尾のみである、という制約を関数に付け加える必要がある。

この制約さえ満たせば、CodeSegment の実行は CodeSegment 型から関数本体を取り出し、レコード型を持つ値を適用することに相当する。具体的に goto を関数として適用するとリスト 5.7 のようになる。

リスト 5.7: Agda における goto の定義

```
goto : {l1 l2 : Level} {I : Set l1} {O : Set l2}
   -> CodeSegment I O -> I -> O
goto (cs b) i = b i
```

この goto の定義を用いることで main などの関数が評価できるようになり、値 150 が得られる。本文中での CodeSegment の定義は一部を抜粋している。実行可能な Agda のソースコードは付録に載せる。

5.6 Meta DataSegment の定義

ノーマルレベルの CbC を Agda 上で記述し、実行することができた。次にメタレベルの計算を Agda 上で記述していく。

Meta DataSegment はノーマルレベルの DataSegment の集合として定義できるものであり、全ての DataSegment の部分型であった。ノーマルレベルの DataSegment はプログラムによって変更されるので、事前に定義できるものではない。ここで、Agda の Parameterized Module を利用して、「Meta DataSegment の上位型は DataSegment である」のように DataSegment を定義する。こうすることにより、全てのプログラムは一つ以上の Meta DataSegment を持ち、任意の個数の DataSegment を持つ。また、Meta DataSegment をメタレベルの DataSegment として扱うことにより、「Meta DataSegment の部分型である Meta Meta DataSegment」を定義できるようになる。階層構造でメタレベルを表現することにより、計算の拡張を自在に行なうことができる。

具体的な Meta DataSegment の定義はリスト 5.8 のようになる。型システム subtype は、Meta DataSegment である Context を受けとることにより構築される。Context を Meta DataSegment とするプログラム上では DataSegment は Meta CodeSegment の上位型となる。その制約を DataSegment 型は表わしている。

リスト 5.8: Agda における Meta DataSegment の定義

```
module subtype {l : Level} (Context : Set l) where

record DataSegment {ll : Level} (A : Set ll) : Set (l ⊔ ll) where
  field
    get : Context → A
```

```
  set : Context → A → Context
```

ここで、関数を部分型に拡張する S-ARROW をもう一度示す。

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2} \qquad \text{S-ARROW}$$

S-ARROW は、前提である部分型関係 $T_1 <: S_1$ と $S_2 <: T_2$ が成り立つ時に、 上位型 $S_1 \rightarrow S_2$ の関数を、部分型 $T_1 \rightarrow T_2$ に拡張できた。ここでの上位型は DataSegment であり、部分型は Meta DataSegment である。制約 `DataSegment` の `get` は、 Meta DataSegment から DataSegment が生成できることを表す。これは前提 $T_1 <: S_1$ に相当する。そして、`set` は $S_2 <: T_2$ に相当する。しかし、任意の DataSegment が Meta DataSegment の部分型となるには、 DataSegment が Meta DataSegment よりも多くの情報を必ず持たなくてはならないが、これは通常では成り立たない。だが、メタ計算を行なう際には常に Meta DataSegment を一つ以上持っていると仮定するならば成り立つ。実際、GearsOS における赤黒木では Meta DataSegment に相当する `Context` を常に持ち歩いている。GearsOS における計算結果はその持ち歩いている Meta DataSegment の更新に相当するため、常に Meta DataSegment を引き連れていることを無視すれば DataSegment から Meta DataSegment を導出できる。よって $S_2 <: T_2$ が成り立つ。

なお、 $S_2 <: T_2$ は Output DataSegment を Meta DataSegment を格納する作業に相当し、 $T_1 <: S_1$ は Meta DataSegment から Input DataSegment を取り出す作業であるため、これは明らかに `stub` である。

5.7 Meta CodeSegment の定義

Meta DataSegment が定義できたので Meta CodeSegment を定義する。実際、DataSegment が Meta DataSegment に拡張できたため、Meta CodeSegment の定義には比較的変更は無い。ノーマルレベルの `CodeSegment` 型に、DataSegment を取って DataSegment を返す、という制約を明示的に付けるだけである (リスト 5.9)

リスト 5.9: Agda における Meta CodeSegment の定義

```
data CodeSegment {l1 l2 : Level} (A : Set l1) (B : Set l2) : Set (l ⊔ l1
   ⊔ l2) where
  cs : {{_ : DataSegment A}} {{_ : DataSegment B}}
        → (A → B) → CodeSegment A B
```

5.8 メタレベル計算の実行

Meta DataSegment と Meta CodeSegment の定義を行なったので、残るは実行である。

実行はノーマルレベルにおいては軽量継続 `goto` を定義することによって表せた。メタレベル実行ではそれを Meta CodeSegment と Meta DataSegment を扱えるように拡張する。Meta DataSegment は Parameterized Module の引数 `Context` に相当するため、Meta CodeSegment は Context を取って Context を返す CodeSegment となる。軽量継続 `goto` と区別するために名前を `exec` とするリスト 5.10 のように定義できる。行なっていることは Meta CodeSegment の本体部分に Meta DataSegment を渡す、という `goto` と変わらないが、`set` と `get` を用いることで上位型である任意の DataSegment を実行する CodeSegment も Meta CodeSegment として一様に実行できる。

リスト 5.10: Agda におけるメタレベル実行の定義

```
exec : {l1 l2 : Level} {I : Set l1} {O : Set l2}
       {{_ : DataSegment I}} {{_ : DataSegment O}}
       → CodeSegment I O → Context → Context
exec {l} {{i}} {{o}}  (cs b) c = set o c (b (get i c))
```

実行例として、リスト 2.1 に示していた a と b の値を加算して c に代入するプログラムを考える。実行する際に c の値を `c'` に保存してから加算ようなメタ計算を考える。c の値を `c'` に保存するタイミングは軽量継続時にユーザが指定する。よって軽量継続を行なうのと同等の情報を保持してなくてはならない。そのために Meta Meta DataSegment `Meta` には制御を移す対象であるノーマルレベル CodeSegment を持つ。値を格納する `c'` の位置は Meta DataSegment でも Meta Meta DataSegment でも構わないが、今回は Meta Meta DataSegemnt に格納するものとする。それらを踏まえた上での Meta Meta DataSegment の Agda 上での定義は 5.11 のようになる。なお、`goto` などの名前の衝突を避けるためにノーマルレベルの定義は `N` に、メタレベルの定義は `M` へと名前を付けかえている。

リスト 5.11: Agda における Meta Meta DataSegment の定義例

```
...
open import subtype Context as N

record Meta : Set where
  field
    context : Context
    c'      : Int
    next    : N.CodeSegment Context Context

open import subtype Meta as M
...
```

定義した `Meta` を利用して、c を `c'` に保存するメタ計算 `push` を定義する。より構文が CbC に似るように `gotoMeta` を糖衣構文的に定義する。`gotoMeta` や `push` で利用し

ている `liftContext` や `liftMeta` はノーマルレベル計算をメタ計算レベルとするように型を明示的に変更するものである。結果的に `main` の `goto` を `gotoMeta` に置き換えることにより、c の値を計算しながら保存できる。リスト 5.12 に示したプログラムでは、通常レベルのコードセグメントを全く変更せずにメタ計算を含む形に拡張している。加算を行なう前の `c` の値が `70` であったとした時、計算結果 `150` は `c` に格納されるが、`c'` には `70` に保存されている。

リスト 5.12: Agda における Meta Meta CodeSegment の定義と実行例

```
...
-- meta level
liftContext : {X Y : Set} {{_ : N.DataSegment X}} {{_ : N.DataSegment Y}} -> N.CodeSegment X Y -> N.CodeSegment Context Context
liftContext {{x}} {{y}} (N.cs f) = N.cs (\c -> N.DataSegment.set y c (f (N.DataSegment.get x c)))

liftMeta : {X Y : Set} {{_ : M.DataSegment X}} {{_ : M.DataSegment Y}} -> N.CodeSegment X Y -> M.CodeSegment X Y
liftMeta (N.cs f) = M.cs f

gotoMeta : {I O : Set} {{_ : N.DataSegment I}} {{_ : N.DataSegment O}} -> M.CodeSegment Meta Meta -> N.CodeSegment I O -> Meta -> Meta
gotoMeta mCode code m = M.exec mCode (record m {next = (liftContext code) })

push : M.CodeSegment Meta Meta
push = M.cs (\m -> M.exec (liftMeta (Meta.next m)) (record m {c' = Context.c (Meta.context m)}))

-- normal level

cs2 : N.CodeSegment ds1 ds1
cs2 = N.cs id

cs1 : N.CodeSegment ds1 ds1
cs1 = N.cs (\d -> N.goto cs2 d)

cs0 : N.CodeSegment ds0 ds1
cs0 = N.cs (\d -> N.goto cs1 (record {c = (ds0.a d) + (ds0.b d)}))

-- meta level (with extended normal)
main : Meta
main = gotoMeta push cs0 (record {context = (record {a = 100 ; b = 50 ; c = 70}) ; c' = 0 ; next = (N.cs id)})
-- record {context = record {a = 100 ; b = 50 ; c = 150} ; c' = 70 ; next = (N.cs id)}
```

なお、CbC におけるメタ計算を含む軽量継続 `goto meta` と Agda におけるメタ計算実行の比較はリスト ??のようになる

CodeSegment や Meta CodeSegment などの定義が多かったため、どの処理やデータがどのレベルに属するか複雑になったため、一度図にてまとめる。Meta DataSegment を含む

任意の DataSegment は Meta DataSegment になりえるので、この階層構造は任意の段数定義することが可能である。

また、この節で取り扱ったソースコードは付録に付す。

5.9 Agda を用いた Continuation based C の検証

Agda において CbC の CodeSegment と DataSegment を定義することができた。実際の CbC のコードを Agda に変換し、それらの性質を証明していく。

ここでは GearsOS が持つ DataSegment `SingleLinkedStack` を証明していく。この `SingleLinkedStack` はポインタを利用した片方向リストを用いて実装されている。

CbC における DataSegment `SingleLinkedStack` の定義はリスト 5.13 のようになっている。

リスト 5.13: CbC における構造体 stack の定義

```
struct SingleLinkedStack {
    struct Element* top;
} SingleLinkedStack;
struct Element {
    union Data* data;
    struct Element* next;
} Element;
```

次に Agda における `SingleLinkedStack` の定義について触れるが、Agda にはポインタが無いため、メモリ確保や `NULL` の定義は存在しない。CbC におけるメモリ確保部分はノーマルレベルには出現しないと仮定し、`NULL` の表現には Agda の標準ライブラリに存在する `Data.Maybe` を用いる。

`Data.Maybe` の `maybe` 型は、コンストラクタを二つ持つ。片方のコンストラクタ `just` は値を持ったデータであり、ポインタの先に値があることに対応している。一方のコンストラクタ `nothing` は値を持たない。これは値が存在せず、ポインタの先が確保されていない (`NULL` ポインタである) ことを表現できる。

リスト 5.14: Agda における Maybe の定義

```
data Maybe {a} (A : Set a) : Set a where
  just    : (x : A) → Maybe A
  nothing : Maybe A
```

Maybe を用いて片方向リストを Agda 上に定義するとリスト 5.15 のようになる。CbC とほぼ同様の定義ができている。CbC、Agda 共に `SingleLinkedStack` は `Element` 型の `top` を持っている。`Element` 型は値と次の `Element` を持つ。CbC ではポインタで表現していた部分が Agda では Maybe で表現されているが、`Element` 型の持つ情報は変わっていない。

リスト 5.15: Agda における片方向リストを用いたスタックの定義

```
data Element (a : Set) : Set where
  cons : a -> Maybe (Element a) -> Element a

datum : {a : Set} -> Element a -> a
datum (cons a _) = a

next : {a : Set} -> Element a -> Maybe (Element a)
next (cons _ n) = n

record SingleLinkedStack (a : Set) : Set where
  field
    top : Maybe (Element a)
```

Agda で片方向リストを利用する DataSegment の定義をリスト 5.16 に示す。ノーマルレベルからアクセス可能な場所として `Context` に `element` フィールドを追加する。そしてノーマルレベルからアクセスできないよう分離した Meta Meta DataSegment である `Meta` にスタックの本体を格納する。CbC における実装では ... で不定であった `next` も、agda ではメタレベルのコードセグメント `nextCS` となり、きちんと型付けされている。

リスト 5.16: スタックを利用するための DataSegment の定義

```
record Context : Set where
  field
    -- fields for stack
    element : Maybe A


open import subtype Context as N

record Meta  : Set^^e2^^82^^81 where
  field
    -- context as set of data segments
    context : Context
    stack   : SingleLinkedStack A
    nextCS  : N.CodeSegment Context Context

open import subtype Meta as M
```

次にスタックへの操作に注目する。スタックへと値を積む `pushSingleLinkedStack` と値を取り出す `popSingleLinkedStack` の CbC 実装をリスト 5.17 に示す。`SingleLinkedStack` は Meta CodeSegment であり、メタ計算を実行した後には通常の CodeSegment へと操作を移す。そのために `next` という名前で次のコードセグメントを保持している。具体的な `next` はコンパイル時にしか分からないため、... 構文を用いて不定としている。

`pushSingleLinkedStack` は `element` を新しく確保し、値を入れた後に `next` へと繋げ、 `top` を更新して軽量継続する。`popSingleLinkedStack` は先頭が空でなければ先頭の値を `top` から取得し、`element` を一つ進める。値が空であれば `data` を `NULL` にしたまま軽量継続を行なう。

リスト 5.17: CbC における SingleLinkedStack を操作する Meta CodeSegment

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

次に Agda における定義をリスト 5.18 に示す。同様に `pushSingleLinkedStack` と `popSingleLinkedStack` を定義している。`pushsinglelinkedstack` では、スタックの値を更新したのちにノーマルレベルの CodeSegment である `n` を `exec` している。なお、`liftMeta` はノーマルレベルの計算をメタレベルとする関数である。

実際に値を追加する部分は where 句に定義された関数 `push` である。これはスタックへと積む値が空であれば追加を行なわず、値がある時は新たに element を作成して top を更新している。本来の CbC の実装では空かチェックはしていないが、値が空であるかに関わらずにスタックに積んでいるために挙動は同じである。

次に `popSingleLinkedStack` に注目する。こちらも操作後に `nextCS` へと継続を移すようになっている。

実際に値を取り出す操作はノーマルレベルからアクセスできる `element` の値の確定と、アクセスできない `stack` の更新である。

`element` については、 `top` が空なら取り出した後の値は無いので `element` は `nothing` である。`top` が空でなければ `element` は `top` の値となる。

`stack` は空なら空のままであり、`top` に値があればその先頭を捨てる。ここで、`pop` の実装はスタックが空であっても、例外を送出したり停止したりせずに処理を続行できることが分かる。

リスト 5.18: Agda における片方向リストを用いたスタックの定義

```
pushSingleLinkedStack : Meta -> Meta
pushSingleLinkedStack m = M.exec (liftMeta n) (record m {stack = (push s e) })
  where
    n = Meta.nextCS m
```

```
    s = Meta.stack  m
    e = Context.element (Meta.context m)
    push : SingleLinkedStack A -> Maybe A -> SingleLinkedStack A
    push s nothing  = s
    push s (just x) = record {top = just (cons x (top s))}

popSingleLinkedStack : Meta -> Meta
popSingleLinkedStack m = M.exec (liftMeta n) (record m {stack = (st m) ;
   context = record con {element = (elem m)}})
  where
    n = Meta.nextCS m
    con  = Meta.context m
    elem : Meta -> Maybe A
    elem record {stack = record { top = (just (cons x _)) }} = just x
    elem record {stack = record { top = nothing           }} = nothing
    st : Meta -> SingleLinkedStack A
    st record {stack = record { top = (just (cons _ s)) }} = record {top
   = s}
    st record {stack = record { top = nothing           }} = record {top
   = nothing}


pushSingleLinkedStackCS : M.CodeSegment Meta Meta
pushSingleLinkedStackCS = M.cs pushSingleLinkedStack

popSingleLinkedStackCS : M.CodeSegment Meta Meta
popSingleLinkedStackCS = M.cs popSingleLinkedStack
```

また、この章で取り上げた CbC と Agda の動作するソースコードは付録に載せる。

5.10 スタックの実装の検証

定義した SingleLinkedStack に対して証明を行なっていく。ここでの証明は SingleLinked-Stack の処理が特定の性質を持つことを保証することである。

例えば

- スタックに積んだ値は取り出せる
- 値は複数積むことができる
- スタックから値を取り出すことができる
- スタックから取り出す値は積んだ値である
- スタックから値を取り出したらその値は無くなる
- スタックに値を積んで取り出すとスタックの内容は変わらない

といった多くの性質がある。

ここでは、最後に示した「スタックに値を積んで取り出すとスタックの内容は変わらない」ことについて示していく。この性質を具体的に書き下すと以下のようになる。

定義 5.7 任意のスタック s に対して

- 任意の値 n
- 値 x を積む操作 push(x, s)
- 値を一つスタックから取り出す操作 pop(s)

がある時、

pop . push(n) s = s

である。

これを Agda 上で定義するとリスト 5.19 のようになる。Agda 上の定義ではスタックそのものではなく、スタックを含む任意の `Meta` に対してこの性質を証明する。この定義が `Meta` の値によらず成り立つことを、自然数の加算の交換法則と同様に等式変形を用いて証明していく。

リスト 5.19: Agda におけるスタックの性質の定義 (1)

```
pushOnce : Meta → Meta
pushOnce m = M.exec pushSingleLinkedStackCS m

popOnce : Meta → Meta
popOnce m = M.exec popSingleLinkedStackCS m

push-pop-type : Meta → Set1
push-pop-type meta =
    M.exec (M.csComp  (M.cs popOnce) (M.cs pushOnce)) meta ≡ meta
```

今回注目する条件分けは、スタック本体である `stack` と、push や pop を行なうための値を格納する `element` である。それぞれが持ち得る値を場合分けして考えていく。

- スタックが空である場合
  - element が存在する場合
    値が存在するため、 push は実行される。push が実行されたためスタックに値があるため、pop が成功する。pop が成功した結果スタックは空となるため元のスタックと同一となり成り立つ。
  - element が存在しない場合

値が存在しないため、 push が実行されない。push が実行されなかったため、スタックは空のままであり、pop も実行されない。結果スタックは空のままであり、元のスタックと一致する。

- スタックが空でない場合
  - element が存在する場合

    element に設定された値 n が push され、スタックに一つ値が積まれる。スタックの先頭は n であるため、pop が実行されて n は無くなる。結果、スタックは実行する前の状態に戻る。
  - element が存在しない場合

    element に値が存在しないため、push は実行されない。スタックは空ではないため、pop が実行され、先頭の値が無くなる。実行後、スタックは一つ値を失なっているため、これは成りたたない。

スタックが空でなく、push する値が存在しないときにこの性質は成り立たないことが分かった。また、`element` が空でない制約を仮定に加えることでこの命題は成り立つようになる。

push 操作と pop 操作を連続して行なうとスタックが元に復元されることは分かった。ここで SingleLinkedStack よりも範囲を広げて `Meta` も復元されるかを考える。一見これも自明に成り立ちそうだが、push 操作と pop 操作は操作後に実行される CodeSegment を持っている。この CodeSegment は任意に設定できるため、`Meta` 内部の DataSegement が書き換えられる可能性がある。よってこれも制約無しでは成り立たない。

逆にいえば、CodeSegment を指定してしまえば `Meta` に関しても push/pop の影響を受けないことを保証できる。全く値を変更しない CodeSegment `id` を指定した際には自明にこの性質が導ける。実際、 Agda 上でも等式変形を明示的に指定せず、定義からの推論でこの証明を導ける (リスト 5.20)。

なお、今回 SingleLinkedStack に積むことができる値は Agda の標準ライブラリ (`Data.Nat`) における自然数型 ℕ としている。push/pop 操作の後の継続が `Meta` に影響を与えない制約は `id-meta` に表れている。これは `Meta` を構成する要素を受け取り、継続先の CodeSegment に恒等関数 `id` を指定している。加えて、スタックが空で無い制約 where 句の `meta` に表れている。必ずスタックの先頭 `top` には値 x が入っており、それ以降の値は任意としている。よってスタックは必ず一つ以上の値を持ち、空でないという制約を表わせる。証明は refl によって与えられる。つまり定義から自明に推論可能となっている。

リスト 5.20: Agda におけるスタックの性質の証明 (1)

```
id-meta : ℕ → ℕ → SingleLinkedStack ℕ → Meta
id-meta n e s = record { context = record {n = n ; element = just e}
```

```
                            ; nextCS = (N.cs id) ; stack = s}

push-pop-type : ℕ → ℕ  → ℕ → Element ℕ → Set1
push-pop-type n e x s = M.exec (M.csComp {meta} (M.cs popOnce) (M.cs
    pushOnce)) meta ≡ meta
  where
    meta = id-meta n e record {top = just (cons x (just s))}

push-pop : (n e x : ℕ) → (s : Element ℕ) →  push-pop-type n e x s
push-pop n e x s = refl
```

ここで興味深い点は、 SingleLinkedStack の実装から証明に仮定が必要なことが証明途中でに分かった点にある。例えば、CbC の SingleLinkedStack 実装の push/pop 操作は失敗しても成功しても指定された CodeSegment に軽量継続する。この性質により、指定された CodeSegment によっては、スタックの操作に関係なく `Meta` の内部の DataSegemnt が書き換えられる可能性があることが分かった。スタックの操作の際に行なわれる軽量継続の利用方法は複数考えられる。例えば、スタックが空である際に pop を行なった時はエラー処理用の継続を行なう、といった実装も可能である。実装が異なれば、同様の性質でも証明は異なるものとなる。このように、実装そのものを適切に型システムで定義できれば、明示されていない実装依存の仕様も証明時に確定させることができる。

証明した定理をより一般的な「任意の自然数回だけスタックへ値を積み、その後同じ回数スタックから値を取り出すとスタックは操作前と変わらない」という形に拡張する。この性質を Agda で定義するとリスト 5.21 のようになる。自然数 n 回だけ push/pop することを記述するために Agda 上に `n-push` 関数と `n-pop` 関数を定義している。それぞれ一度操作を行なった後に再帰的に自身を呼び出す再帰関数である。

リスト 5.21: Agda におけるスタックの性質の定義 (2)

```
n-push : {m : Meta} {{_ : M.DataSegment Meta}} (n : ℕ) → M.CodeSegment
    Meta Meta
n-push {{mm}} (zero)      = M.cs {{mm}} {{mm}} id
n-push {m} {{mm}} (suc n) = M.cs {{mm}} {{mm}} (\m → M.exec {{mm}} {{mm
    }} (n-push {m} {{mm}} n) (pushOnce m))

n-pop : {m : Meta} {{_ : M.DataSegment Meta}} (n : ℕ) → M.CodeSegment
    Meta Meta
n-pop {{mm}} (zero)      = M.cs {{mm}} {{mm}} id
n-pop {m} {{mm}} (suc n) = M.cs {{mm}} {{mm}} (\m → M.exec {{mm}} {{mm}}
     (n-pop {m} {{mm}} n) (popOnce m))

pop-n-push-type : ℕ → ℕ → ℕ → SingleLinkedStack ℕ → Set1
pop-n-push-type n cn ce s = M.exec (M.csComp {meta} (M.cs popOnce) (n-
    push {meta} (suc n))) meta
                            ≡ M.exec (n-push {meta} n) meta
  where
    meta = id-meta cn ce s
```

この性質の証明は少々複雑である。結論から先に示すとリスト 5.22 のように証明で

きる。

リスト 5.22: Agda におけるスタックの性質の証明 (2)

```
pop-n-push-type : ℕ → ℕ → ℕ → SingleLinkedStack ℕ → Set1
pop-n-push-type n cn ce s = M.exec (M.csComp (M.cs popOnce) (n-push (suc n))) meta
                              ≡ M.exec (n-push n) meta
  where
    meta = id-meta cn ce s

pop-n-push : (n cn ce : ℕ) → (s : SingleLinkedStack ℕ) → pop-n-push-type n cn ce s
pop-n-push zero cn ce s    = refl
pop-n-push (suc n) cn ce s = begin
  M.exec (M.csComp  (M.cs popOnce) (n-push  (suc (suc n)))) (id-meta cn ce s)
  ≡⟨ refl ⟩
  M.exec (M.csComp (M.cs popOnce) (M.csComp  (n-push  (suc n)) (M.cs pushOnce))) (id-meta cn ce s)
  ≡⟨ exec-comp (M.cs popOnce) (M.csComp  (n-push  (suc n)) (M.cs pushOnce)) (id-meta cn ce s) ⟩
  M.exec (M.cs popOnce) (M.exec (M.csComp  (n-push  (suc n)) (M.cs pushOnce)) (id-meta cn ce s))
  ≡⟨ cong (\x → M.exec (M.cs popOnce) x) (exec-comp (n-push (suc n)) (M.cs pushOnce) (id-meta cn ce s)) ⟩
  M.exec (M.cs popOnce) (M.exec (n-push  (suc n))(M.exec (M.cs pushOnce) (id-meta cn ce s)))
  ≡⟨ refl ⟩
  M.exec (M.cs popOnce) (M.exec (n-push  (suc n)) (id-meta cn ce (record {top = just (cons ce (SingleLinkedStack.top s))})))
  ≡⟨ sym (exec-comp (M.cs popOnce) (n-push (suc n)) (id-meta cn ce (record {top = just (cons ce (SingleLinkedStack.top s))}))) ⟩
  M.exec (M.csComp  (M.cs popOnce) (n-push  (suc n))) (id-meta cn ce (record {top = just (cons ce (SingleLinkedStack.top s))}))
  ≡⟨ pop-n-push n cn ce (record {top = just (cons ce (SingleLinkedStack.top s))}) ⟩
  M.exec (n-push n) (id-meta cn ce (record {top = just (cons ce (SingleLinkedStack.top s))}))
  ≡⟨ refl ⟩
  M.exec (n-push n) (pushOnce (id-meta cn ce s))
  ≡⟨ refl ⟩
  M.exec (n-push n) (M.exec (M.cs pushOnce) (id-meta cn ce s))
  ≡⟨ refl ⟩
  M.exec (n-push (suc n)) (id-meta cn ce s)
  ■



n-push-pop-type : ℕ →  ℕ  → ℕ → SingleLinkedStack ℕ → Set1
n-push-pop-type n cn ce st = M.exec (M.csComp (n-pop n) (n-push n)) meta ≡ meta
  where
```

```
    meta = id-meta cn ce st

n-push-pop : (n cn ce : ℕ) → (s : SingleLinkedStack ℕ) → n-push-pop-
    type n cn ce s
n-push-pop zero    cn ce s = refl
n-push-pop (suc n) cn ce s = begin
  M.exec (M.csComp (n-pop (suc n)) (n-push (suc n))) (id-meta cn ce s)
  ≡⟨ refl ⟩
  M.exec (M.csComp (M.cs (\m → M.exec (n-pop n) (popOnce m))) (n-push (
    suc n))) (id-meta cn ce s)
  ≡⟨ exec-comp (M.cs (\m → M.exec (n-pop n) (popOnce m))) (n-push (suc n
    ))  (id-meta cn ce s) ⟩
  M.exec (M.cs (\m → M.exec (n-pop  n) (popOnce m))) (M.exec (n-push (
    suc n)) (id-meta cn ce s))
  ≡⟨ refl ⟩
  M.exec (n-pop n) (popOnce (M.exec (n-push (suc n)) (id-meta cn ce s)))
  ≡⟨ refl ⟩
  M.exec (n-pop n) (M.exec (M.cs popOnce) (M.exec (n-push (suc n)) (id-
    meta cn ce s)))
  ≡⟨ cong (\x → M.exec (n-pop n) x) (sym (exec-comp (M.cs popOnce) (n-
    push (suc n)) (id-meta cn ce s))) ⟩
  M.exec (n-pop n) (M.exec (M.csComp (M.cs popOnce) (n-push (suc n))) (id
    -meta cn ce s))
  ≡⟨ cong (\x → M.exec (n-pop n) x) (pop-n-push n cn ce s) ⟩
  M.exec (n-pop n) (M.exec (n-push n) (id-meta cn ce s))
  ≡⟨ sym (exec-comp (n-pop n) (n-push n) (id-meta cn ce s)) ⟩
  M.exec (M.csComp (n-pop n) (n-push n)) (id-meta cn ce s)
  ≡⟨ n-push-pop n cn ce s ⟩
  id-meta cn ce s
  ■
```

これは以下のような形の証明になっている。

- 「n 回 push した後に n 回 pop しても同様になる」という定理を `n-push-pop` とおく。
- `n-push-pop` は自然数 n と特定の `Meta` に対して `exec (n-pop (suc n)) . (n-push (suc n))` が成り立つことである
- 特定の `Meta` とは、 push/pop 操作の後の継続が DataSegment を変更しない `Meta` である。
- また、簡略化のために `csComp` による CodeSegment の合成を二項演算子 `.` とおく
  - 例えば `exec (csComp f g) x` は `exec (f . g) x` となる。
- `n-push-pop` を証明するための補題 `pop-n-push` を定義する
- `n-push-pop` とは「n+1 回 push して 1 回 pop することは、n 回 push することと等しい」という補題である。

- `n-push-pop` は `exec (pop . n-push (suc n)) m = exec (n-push n) m` と表現できる。
- `n-push-pop` の n が zero の時は直ちに成り立つ。
- `n-push-pop` の n が zero でない時 (suc n である時) は以下のように証明できる。
  - `exec (n-push (suc n)) m` を `X` とおく
  - `exec (pop . n-push (suc (suc n))) m = X`
  - n-push の定義より `exec (pop . (n-push (suc n) . push)) m = X`
  - 補題 exec-comp より `exec (pop (exec (n-push (suc n) . push) m)) = X`
  - 補題 exec-comp より `exec (pop (exec (n-push (suc n) (exec push m)))) = X`
  - 一度 push した結果を `m'` とおくと `exec (pop (exec (n-push (suc n) m'))) = X`
  - `n-push-pop` より `exec (exec (n-push n m')) = X`
  - push の定義より `exec (exec (n-push n (exec push m))) = X`
  - n-push の定義より `exec (exec (n-push (suc n) m) = X` となる
  - 全く同一の項に変更できたので証明終了
- 次に `n-push-pop` の証明を示す。
- `n-push-pop` の n が zero の時は、 `suc zero` 回の push/pop が行なわれるため、`push-pop` より成り立つ。
- `n-push-pop` の n が zero でない時は以下により証明できる。
  - `exec ((n-pop (suc n)) . (n-push (suc n))) m = m` を示せれば良い。
  - `X` に注目した時 n-pop の定義より `exec (n-pop n) . pop . (n-push (suc n)) m = m`
  - exec-comp より `exec (n-pop n) (exec pop (n-push (suc n)) m) = m`
  - exec-comp より `exec (n-pop n) (exec pop (exec (n-push (suc n)) m)) = m`
  - exec-comp より `exec (n-pop n) (exec pop . (n-push (suc n)) m) = m`
  - pop-n-push より `exec (n-pop n) (exec (n-push n) m) = m`
  - n-push-pop より `m = m` となり証明終了。
  - なお、n-push-pop は (suc n) が n に減少するため、確実に停止することから自身を自身の証明に適用している。

push-pop を一般化した n-push-pop を証明することができた。n-push-pop は証明の途中で補題 pop-n-push と push-pop を利用した定理である。このように、CbC で記述されたプログラムを Agda 上に記述することで、データ構造の性質を定理として証明することができた。これらの証明機構を CbC のコンパイラやランタイム、モデルチェッカなどに組み込むことにより CbC は CbC で記述されたコードを証明することができるようになる。なお、本論文で取り扱っている Agda のソースコードは視認性の向上のために暗黙的な引数を省略して記述している。動作する完全なコードは付録に付す。

第6章　まとめ

本論文ではメタ計算を用いた Continuation based C プログラムの検証手法を二つ提案した。

一つはモデル検査的なアプローチであり、メタ計算ライブラリ akasha を用いて GearsOS の非破壊赤黒木の仕様を保証した。CbC における仕様の定義は assert に渡す論理式として定義され、状態の数え上げは軽量継続 `meta` を切り替えることで実現できた。CbC で記述された非破壊赤黒木のプログラムを検証用に変更することなく、CbC 自身で検証した。検証できた範囲は有限の要素数のみであるが、有限モデルチェッカ CBMC よりも大きな範囲を検証した。

二つめは定理証明的なアプローチである。akasha を用いた検証では挿入回数は有限の数に限定されていた。データ構造とそれにまつわる処理を直接証明することにより、任意の回数操作を行なっても性質を保証する。部分型を利用して CbC の型システムの定義を行ない、依存型を持つ言語 Agda 上で記述することで CbC の形式的な定義とした。Agda 上で記述された CbC プログラムの性質を証明することで、 CbC が部分型できちんと型付けできること、依存型を CbC コンパイラに組み込むことで CbC 自身を証明できることが分かった。

また、型システムは証明以外にも実用的に利用できることが分かった。akasha を用いて検証を行なう際、全ての CodeSegment に対して stub をユーザが定義する必要があった。CbC の型を定義することにより、stub の自動生成と型チェックが行なえることが分かった。

6.1　今後の課題

今後の課題として、型システムの詳細な性質の解析がある。本論文では部分型の定義を CbC に適用した。CodeSegment は関数呼び出しを末尾でしか許さない制限があるので、関数型の計算規則をより制限できるはずである。その制約の元に生まれた計算体系の持つ性質や表現能力に興味がある。

また、提案した型システムを CbC コンパイラの内部に組み込み、CodeSegment と DataSegment の型チェックを行なえるようにしたい。加えて部分型を組み込むことによ

り、stub の自動生成をする。さらに依存型を加えれば CbC で CbC 自身を証明できるようになる。

モデル検査的アプローチの展望としては、依存型を CbC コンパイラに実装し、型情報を用いた記号実行や状態の列挙を行なうシステムの構築などがある。

また、型システムの拡張としては総称型などを CbC に適用することも挙げられる。総称型は Java におけるジェネリクスや C++ におけるテンプレートに相当し、ユーザが定義できるデータ構造の表現能力が向上する。他にも、CbC における型推論や推論器のコンパイラへの実装などが挙げられる。

謝辞

本研究の遂行、本論文の作成にあたり、御多忙にも関わらず終始懇切なる御指導と御教授を賜わりました河野真治准教授に心より感謝致します。そして、 共に研究を行い暖かな気遣いと励ましをもって支えてくれた並列信頼研究室の全てのメンバーに感謝致します。最後に、 有意義な時間を共に過ごした理工学研究科情報工学専攻の学友、並びに物心両面で支えてくれた家族に深く感謝致します。

2017 年 3 月
比嘉健太

参考文献

[1] Spin - formal verification. http://spinroot.com/spin/whatispin.html. Accessed: 2016/01/20(Fri).

[2] Nusmv home page. http://nusmv.fbk.eu/. Accessed: 2016/01/20(Fri).

[3] The cbmc homepage. http://www.cprover.org/cbmc/. Accessed: 2016/01/20(Fri).

[4] The agda wiki. http://wiki.portal.chalmers.se/agda/pmwiki.php. Accessed: 2016/01/20(Fri).

[5] Welcome! — the coq proof assistant. https://coq.inria.fr/. Accessed: 2016/01/20(Fri).

[6] Ats-pl-sys. http://www.ats-lang.org/. Accessed: 2016/01/20(Fri).

[7] 徳森海斗. Llvm clang 上の continuation based c コンパイラ の改良. Master's thesis, 琉球大学 大学院理工学研究科 情報工学専攻, 2016.

[8] Tokumori Kaito and Kono Shinji. The implementation of continuation based c compiler on llvm/clang 3.5. *IPSJ SIG Notes*, Vol. 2014, No. 10, pp. 1–11, may 2014.

[9] 信康大城, 真治河野. Continuation based c の gcc4.6 上の実装について. 第 53 回プログラミング・シンポジウム予稿集, 第 2012 巻, pp. 69–78, jan 2012.

[10] 翔平小久保, 立樹伊波, 真治河野. Monad に基づくメタ計算を基本とする gears os の設計. Technical Report 16, 琉球大学大学院理工学研究科情報工学専攻, 琉球大学工学部情報工学科, 琉球大学工学部情報工学科, may 2015.

[11] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, Vol. 93, No. 1, pp. 55–92, July 1991.

[12] Opencl — nvidia developer. https://developer.nvidia.com/opencl. Accessed: 2016/02/06(Mon).

[13] Cuda zone — nvidia developer. `https://developer.nvidia.com/cuda-zone`. Accessed: 2016/02/06(Mon).

[14] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[15] B.C. Pierce. 型システム入門プログラミング言語と型の理論:. オーム社, 2013.

[16] Ulf Norell. Dependently typed programming in agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pp. 1–2, New York, NY, USA, 2009. ACM.

[17] Welcome to agda's documentation! — agda 2.6.0 documentation. `http://agda.readthedocs.io/en/latest/index.html`. Accessed: 2016/01/31(Tue).

[18] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, New York, NY, USA, 1989.

[19] Joachim (mathématicien) Lambek and P. J. Scott. *Introduction to higher order categorical logic*. Cambridge studies in advanced mathematics. Cambridge University Press, Cambridge, New York (N. Y.), Melbourne, 1986.

[20] Michael Barr and Charles Wells. *Category Theory for Computing Science*. International Series in Computer Science. Prentice-Hall, 1990. Second edition, 1995.

[21] M. P. Jones and L. Duponcheel. Composing monads. Research Report YALEU/DCS/RR-1004, Yale University, December 1993.

[22] John Backus. The history of fortran i, ii, and iii. *SIGPLAN Not.*, Vol. 13, No. 8, pp. 165–180, August 1978.

[23] Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, Vol. 6, No. 4, pp. 308–320, January 1964.

[24] Alonzo Church. *The Calculi of Lambda-Conversion*. Princeton University Press, Princeton, New Jork, 1941.

[25] P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, Vol. 27, No. 5, May 1992.

[26] N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, Vol. 75, No. 5, pp. 381 – 392, 1972.

発表履歴

- 比嘉健太, 河野真治. Agda 入門. オープンソースカンファレンス 2014 Okinawa, May 2014.
- 比嘉健太, 河野真治. 形式手法を学び始めて思うことと、形式手法を広めるには. 情報処理学会ソフトウェア工学研究会 (IPSJ SIGSE) ウィンターワークショップ 2015・イン・宜野湾 (WWS2015), Jan 2015.
- 比嘉健太, 河野真治. Continuation based C を用いたプログラムの検証手法. 2016 年並列／分散／協調処理に関する『松本』サマー・ワークショップ (SWoPP2016) 情報処理学会・プログラミング研究会 第 110 回プログラミング研究会 (PRO-2016-2) Aug 2016.

付 録A　ソースコード一覧

本論文中に取り上げた Agda の動作するソースコードを示す。

A-1　部分型の定義

リスト A.1 に Agda 上で定義した CbC の部分型の定義を示す。

リスト A.1: Agda 上で定義した CbC の部分型の定義 (subtype.agda)

```
open import Level
open import Relation.Binary.PropositionalEquality

module subtype {l : Level} (Context : Set l) where


record DataSegment {ll : Level} (A : Set ll) : Set (l ^^e2^^8a^^94 ll)
    where
  field
    get : Context -> A
    set : Context -> A -> Context
open DataSegment

data CodeSegment {l1 l2 : Level} (A : Set l1) (B : Set l2) : Set (l ^^e2
    ^^8a^^94 l1 ^^e2^^8a^^94 l2) where
  cs : {{_ : DataSegment A}} {{_ : DataSegment B}} -> (A -> B) ->
    CodeSegment A B

goto : {l1 l2 : Level} {I : Set l1} {O : Set l2} -> CodeSegment I O -> I
    -> O
goto (cs b) i = b i

exec : {l1 l2 : Level} {I : Set l1} {O : Set l2} {{_ : DataSegment I}} {{
    _ : DataSegment O}}
      -> CodeSegment I O -> Context -> Context
exec {l} {{i}} {{o}}  (cs b) c = set o c (b (get i c))


comp : {con : Context} -> {l1 l2 l3 l4 : Level}
       {A : Set l1} {B : Set l2} {C : Set l3} {D : Set l4}
       {{_ : DataSegment A}} {{_ : DataSegment B}} {{_ : DataSegment C}}
    {{_ : DataSegment D}}
       -> (C -> D) -> (A -> B) -> A -> D
```

```
comp {con} {{i}} {{io}} {{oi}} {{o}} g f x = g (get oi (set io con (f x)))

csComp : {con : Context} -> {l1 l2 l3 l4 : Level}
        {A : Set l1} {B : Set l2} {C : Set l3} {D : Set l4}
         {{_ : DataSegment A}} {{_ : DataSegment B}} {{_ : DataSegment C}} {{_ : DataSegment D}}
       -> CodeSegment C D -> CodeSegment A B -> CodeSegment A D
csComp {con} {A} {B} {C} {D} {{da}} {{db}} {{dc}} {{dd}} (cs g) (cs f)
      = cs {{da}} {{dd}} (comp {con} {{da}} {{db}} {{dc}} {{dd}} g f)



comp-associative : {A B C D E F : Set l} {con : Context}
                   {{da : DataSegment A}} {{db : DataSegment B}} {{dc : DataSegment C}}
                   {{dd : DataSegment D}} {{de : DataSegment E}} {{df : DataSegment F}}
                   -> (a : CodeSegment A B) (b : CodeSegment C D) (c : CodeSegment E F)
                   -> csComp {con} c (csComp {con} b a)  ≡  csComp {con} (csComp {con} c b) a
comp-associative (cs _) (cs _) (cs _) = refl
```

A-2 ノーマルレベル計算の実行

5.5 節で取り上げたソースコードをリスト A.2 に示す。CbC のコードにより近づけるように A gda 上の `Data.Nat` を `Int` という名前に変更している。

リスト A.2: ノーマルレベル計算例の完全なソースコード (atton-master-sample.agda)

```
module atton-master-sample where

open import Data.Nat
open import Data.Unit
open import Function
Int = ℕ

record Context : Set where
  field
    a : Int
    b : Int
    c : Int


open import subtype Context



record ds0 : Set where
  field
```

```
    a : Int
    b : Int

record ds1 : Set where
  field
    c : Int

instance
  _ : DataSegment ds0
  _ = record { set = (\c d → record c {a = (ds0.a d) ; b = (ds0.b d)})
             ; get = (\c →   record { a = (Context.a c) ; b = (Context.b c)})}
  _ : DataSegment ds1
  _ = record { set = (\c d → record c {c = (ds1.c d)})
             ; get = (\c →   record { c = (Context.c c)})}

cs2 : CodeSegment ds1 ds1
cs2 = cs id

cs1 : CodeSegment ds1 ds1
cs1 = cs (\d → goto cs2 d)

cs0 : CodeSegment ds0 ds1
cs0 = cs (\d → goto cs1 (record {c = (ds0.a d) + (ds0.b d)}))

main : ds1
main = goto cs0 (record {a = 100 ; b = 50})
```

A-3　メタレベル計算の実行

5.8 節で取り上げたソースコードをリスト A.3 に示す。

リスト A.3: メタレベル計算例の完全なソースコード (atton-master-meta-sample.agda)

```
module atton-master-meta-sample where

open import Data.Nat
open import Data.Unit
open import Function
Int = ℕ

record Context : Set where
  field
    a  : Int
    b  : Int
    c  : Int

open import subtype Context as N

record Meta : Set where
  field
    context : Context
```

```
    c'      : Int
    next    : N.CodeSegment Context Context

open import subtype Meta as M

instance
  _ : N.DataSegment Context
  _ = record { get = id ; set = (\_ c → c) }
  _ : M.DataSegment Context
  _ = record { get = (\m → Meta.context m)  ;
               set = (\m c → record m {context = c}) }
  _ : M.DataSegment Meta
  _ = record { get = id ; set = (\_ m → m) }


liftContext : {X Y : Set} {{_ : N.DataSegment X}} {{_ : N.DataSegment Y}} → N.CodeSegment X Y → N.CodeSegment Context Context
liftContext {{x}} {{y}} (N.cs f) = N.cs (\c → N.DataSegment.set y c (f (N.DataSegment.get x c)))

liftMeta : {X Y : Set} {{_ : M.DataSegment X}} {{_ : M.DataSegment Y}} → N.CodeSegment X Y → M.CodeSegment X Y
liftMeta (N.cs f) = M.cs f


gotoMeta : {I O : Set} {{_ : N.DataSegment I}} {{_ : N.DataSegment O}} → M.CodeSegment Meta Meta → N.CodeSegment I O → Meta → Meta
gotoMeta mCode code m = M.exec mCode (record m {next = (liftContext code)})

push : M.CodeSegment Meta Meta
push = M.cs (\m → M.exec (liftMeta (Meta.next m)) (record m {c' = Context.c (Meta.context m)}))


record ds0 : Set where
  field
    a : Int
    b : Int

record ds1 : Set where
  field
    c : Int

instance
  _ : N.DataSegment ds0
  _ = record { set = (\c d → record c {a = (ds0.a d) ; b = (ds0.b d)})
             ; get = (\c →   record { a = (Context.a c) ; b = (Context.b c)})}
  _ : N.DataSegment ds1
  _ = record { set = (\c d → record c {c = (ds1.c d)})
             ; get = (\c →   record { c = (Context.c c)})}

cs2 : N.CodeSegment ds1 ds1
cs2 = N.cs id
```

```

cs1 : N.CodeSegment ds1 ds1
cs1 = N.cs (\d → N.goto cs2 d)

cs0 : N.CodeSegment ds0 ds1
cs0 = N.cs (\d → N.goto cs1 (record {c = (ds0.a d) + (ds0.b d)}))


main : Meta
main = gotoMeta push cs0 (record {context = (record {a = 100 ; b = 50 ; c
     = 70}) ; c’ = 0 ; next = (N.cs id)})
-- record {context = record {a = 100 ; b = 50 ; c = 150} ; c’ = 70 ; next
     = (N.cs id)}
```

A-4 Agda を用いた Continuation based C の検証

5.9 節で取り上げたソースコードを以下に示す。

リスト A.4: Agda を用いた Continuation based C の検証コード (SingleLinkedStack.cbc)

```
#include "../context.h"
#include "../origin_cs.h"
#include <stdio.h>

// typedef struct SingleLinkedStack {
//     struct Element* top;
// } SingleLinkedStack;

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;
}

void printStack1(union Data* data) {
    struct Node* node = &data->Element.data->Node;
    if (node == NULL) {
        printf("NULL");
    } else {
        printf("key = %d ,", node->key);
        printStack1((union Data*)data->Element.next);
```

```
    }
}

void printStack(union Data* data) {
    printStack1(data);
    printf("\n");
}

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

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

__code pop2SingleLinkedStack(struct SingleLinkedStack* stack, __code next
    (union Data* data, union Data* data1, ...)) {
    if (stack->top) {
        data = stack->top->data;
        stack->top = stack->top->next;
    } else {
        data = NULL;
    }
    if (stack->top) {
        data1 = stack->top->data;
        stack->top = stack->top->next;
    } else {
        data1 = NULL;
    }
    goto next(data, data1, ...);
}


```

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

__code get2SingleLinkedStack(struct SingleLinkedStack* stack, __code next(union Data* data, union Data* data1, ...)) {
    if (stack->top) {
        data = stack->top->data;
        if (stack->top->next) {
            data1 = stack->top->next->data;
        } else {
            data1 = NULL;
        }
    } else {
        data = NULL;
        data1 = NULL;
    }
    goto next(data, data1, ...);
}

__code isEmptySingleLinkedStack(struct SingleLinkedStack* stack, __code next(...), __code whenEmpty(...)) {
    if (stack->top)
        goto next(...);
    else
        goto whenEmpty(...);
}
```

リスト A.5: Agda を用いた Continuation based C の検証コード (stack-subtype.agda)

```
open import Level hiding (lift)
open import Data.Maybe
open import Data.Product
open import Data.Nat hiding (suc)
open import Function

module stack-subtype (A : Set) where

-- data definitions

data Element (a : Set) : Set where
  cons : a → Maybe (Element a) → Element a

datum : {a : Set} → Element a → a
datum (cons a _) = a

next : {a : Set} → Element a → Maybe (Element a)
next (cons _ n) = n

```

```
record SingleLinkedStack (a : Set) : Set where
  field
    top : Maybe (Element a)
open SingleLinkedStack

record Context : Set where
  field
    -- fields for concrete data segments
    n       : ℕ
    -- fields for stack
    element : Maybe A




open import subtype Context as N

instance
  ContextIsDataSegment : N.DataSegment Context
  ContextIsDataSegment = record {get = (\c → c) ; set = (\_ c → c)}


record Meta  : Set1 where
  field
    -- context as set of data segments
    context : Context
    stack   : SingleLinkedStack A
    nextCS  : N.CodeSegment Context Context




open import subtype Meta as M

instance
  MetaIncludeContext : M.DataSegment Context
  MetaIncludeContext = record { get = Meta.context
                              ; set = (\m c → record m {context = c}) }

  MetaIsMetaDataSegment : M.DataSegment Meta
  MetaIsMetaDataSegment  = record { get = (\m → m) ; set = (\_ m → m) }


liftMeta : {X Y : Set} {{_ : M.DataSegment X}} {{_ : M.DataSegment Y}}
    → N.CodeSegment X Y → M.CodeSegment X Y
liftMeta (N.cs f) = M.cs f

liftContext : {X Y : Set} {{_ : N.DataSegment X}} {{_ : N.DataSegment Y}}
     → N.CodeSegment X Y → N.CodeSegment Context Context
liftContext {{x}} {{y}} (N.cs f) = N.cs (\c → N.DataSegment.set y c (f (
    N.DataSegment.get x c)))

-- definition based from Gears(209:5708390a9d88) src/parallel_execution

```

```
emptySingleLinkedStack : SingleLinkedStack A
emptySingleLinkedStack = record {top = nothing}


pushSingleLinkedStack : Meta → Meta
pushSingleLinkedStack m = M.exec (liftMeta n) (record m {stack = (push s e) })
  where
    n = Meta.nextCS m
    s = Meta.stack  m
    e = Context.element (Meta.context m)
    push : SingleLinkedStack A → Maybe A → SingleLinkedStack A
    push s nothing  = s
    push s (just x) = record {top = just (cons x (top s))}



popSingleLinkedStack : Meta → Meta
popSingleLinkedStack m = M.exec (liftMeta n) (record m {stack = (st m) ; context = record con {element = (elem m)}})
  where
    n = Meta.nextCS m
    con  = Meta.context m
    elem : Meta → Maybe A
    elem record {stack = record { top = (just (cons x _)) }} = just x
    elem record {stack = record { top = nothing           }} = nothing
    st : Meta → SingleLinkedStack A
    st record {stack = record { top = (just (cons _ s)) }} = record {top = s}
    st record {stack = record { top = nothing           }} = record {top = nothing}




pushSingleLinkedStackCS : M.CodeSegment Meta Meta
pushSingleLinkedStackCS = M.cs pushSingleLinkedStack

popSingleLinkedStackCS : M.CodeSegment Meta Meta
popSingleLinkedStackCS = M.cs popSingleLinkedStack


-- for sample

firstContext : Context
firstContext = record {element = nothing ; n = 0}


firstMeta : Meta
firstMeta = record { context = firstContext
                   ; stack = emptySingleLinkedStack
                   ; nextCS = (N.cs (\m → m))
                   }
```

A-5 スタックの実装の検証

5.10 節で取り上げたソースコードをリスト A.6 に示す。

リスト A.6: スタックの実装の検証コード (stack-subtype-sample.agda)

```
module stack-subtype-sample where

open import Level renaming (suc to S ; zero to O)
open import Function
open import Data.Nat
open import Data.Maybe
open import Relation.Binary.PropositionalEquality

open import stack-subtype ℕ
open import subtype Context  as N
open import subtype Meta     as M


record Num : Set where
  field
    num : ℕ

instance
  NumIsNormalDataSegment : N.DataSegment Num
  NumIsNormalDataSegment = record { get = (\c → record { num = Context.n c})
                                  ; set = (\c n → record c {n = Num.num n})}
  NumIsMetaDataSegment : M.DataSegment Num
  NumIsMetaDataSegment = record { get = (\m → record {num = Context.n (Meta.context m)})
                                ; set = (\m n → record m {context = record (Meta.context m) {n = Num.num n}})}


plus3 : Num → Num
plus3 record { num = n } = record {num = n + 3}

plus3CS : N.CodeSegment Num Num
plus3CS = N.cs plus3



plus5AndPushWithPlus3 : {mc : Meta} {{_ : N.DataSegment Num}}
               → M.CodeSegment Num (Meta)
plus5AndPushWithPlus3 {mc} {{nn}} = M.cs (\n → record {context = con n ; nextCS = (liftContext {{nn}} {{nn}} plus3CS) ; stack = st} )
  where
    co    = Meta.context mc
    con : Num → Context
    con record { num = num } = N.DataSegment.set nn co record {num = num + 5}
    st    = Meta.stack mc
```

```
push-sample : {{_ : N.DataSegment Num}} {{_ : M.DataSegment Num}} → Meta
push-sample {{nd}} {{md}} = M.exec {{md}} (plus5AndPushWithPlus3 {mc} {{nd}}) mc
  where
    con  = record { n = 4 ; element = just 0}
    code = N.cs (\c → c)
    mc   = record {context = con ; stack = emptySingleLinkedStack ; nextCS = code}


push-sample-equiv : push-sample ≡ record { nextCS  = liftContext plus3CS
                                         ; stack   = record { top = nothing}
                                         ; context = record { n = 9} }
push-sample-equiv = refl


pushed-sample : {m : Meta} {{_ : N.DataSegment Num}} {{_ : M.DataSegment Num}} →  Meta
pushed-sample {m} {{nd}} {{md}} = M.exec {{md}} (M.csComp {m} {{md}} pushSingleLinkedStackCS (plus5AndPushWithPlus3 {mc} {{nd}})) mc
  where
    con  = record { n = 4 ; element = just 0}
    code = N.cs (\c → c)
    mc   = record {context = con ; stack = emptySingleLinkedStack ; nextCS = code}



pushed-sample-equiv : {m : Meta} →
                      pushed-sample {m} ≡ record { nextCS  = liftContext plus3CS
                                                 ; stack   = record { top = just (cons 0 nothing) }
                                                 ; context = record { n = 12} }
pushed-sample-equiv = refl



pushNum : N.CodeSegment Context Context
pushNum = N.cs pn
  where
    pn : Context → Context
    pn record { n = n } = record { n = pred n  ; element = just n}


pushOnce : Meta → Meta
pushOnce m = M.exec pushSingleLinkedStackCS m
```

```

n-push : {m : Meta} {{_ : M.DataSegment Meta}} (n : ℕ) → M.CodeSegment Meta Meta
n-push {{mm}} (zero)      = M.cs {{mm}} {{mm}} id
n-push {m} {{mm}} (suc n) = M.cs {{mm}} {{mm}} (\m → M.exec {{mm}} {{mm}} (n-push {m} {{mm}} n) (pushOnce m))

popOnce : Meta → Meta
popOnce m = M.exec popSingleLinkedStackCS m

n-pop : {m : Meta} {{_ : M.DataSegment Meta}} (n : ℕ) → M.CodeSegment Meta Meta
n-pop {{mm}} (zero)      = M.cs {{mm}} {{mm}} id
n-pop {m} {{mm}} (suc n) = M.cs {{mm}} {{mm}} (\m → M.exec {{mm}} {{mm}} (n-pop {m} {{mm}} n) (popOnce m))



initMeta : ℕ  → Maybe ℕ → N.CodeSegment Context Context → Meta
initMeta n mn code = record { context = record { n = n ; element = mn}
                            ; stack   = emptySingleLinkedStack
                            ; nextCS  = code
                            }

n-push-cs-exec = M.exec (n-push {meta} 3) meta
  where
    meta = (initMeta 5 (just 9) pushNum)


n-push-cs-exec-equiv : n-push-cs-exec ≡ record { nextCS  = pushNum
                                                ; context = record {n = 2 ; element = just 3}
                                                ; stack   = record {top = just (cons 4 (just (cons 5 (just (cons 9 nothing)))))}}
n-push-cs-exec-equiv = refl


n-pop-cs-exec = M.exec (n-pop {meta} 4) meta
  where
    meta = record { nextCS  = N.cs id
                  ; context = record { n = 0 ; element = nothing}
                  ; stack   = record {top = just (cons 9 (just (cons 8 (just (cons 7 (just (cons 6 (just (cons 5 nothing)))))))))}
                  }

n-pop-cs-exec-equiv : n-pop-cs-exec ≡ record { nextCS  = N.cs id
                                              ; context = record { n = 0 ; element = just 6}
                                              ; stack   = record { top = just (cons 5 nothing)}
                                              }

n-pop-cs-exec-equiv = refl

```

```

open ≡-Reasoning

id-meta : ℕ → ℕ → SingleLinkedStack ℕ → Meta
id-meta n e s = record { context = record {n = n ; element = just e}
                       ; nextCS = (N.cs id) ; stack = s}

exec-comp : (f g : M.CodeSegment Meta Meta) (m : Meta) → M.exec (M.csComp {m} f g) m ≡ M.exec f (M.exec g m)
exec-comp (M.cs x) (M.cs _) m = refl


push-pop-type : ℕ → ℕ  → ℕ → Element ℕ → Set1
push-pop-type n e x s = M.exec (M.csComp {meta} (M.cs popOnce) (M.cs pushOnce)) meta ≡ meta
  where
    meta = id-meta n e record {top = just (cons x (just s))}

push-pop : (n e x : ℕ) → (s : Element ℕ) →  push-pop-type n e x s
push-pop n e x s = refl



pop-n-push-type : ℕ → ℕ → ℕ → SingleLinkedStack ℕ → Set1
pop-n-push-type n cn ce s = M.exec (M.csComp {meta} (M.cs popOnce) (n-push {meta} (suc n))) meta
                         ≡ M.exec (n-push {meta} n) meta
  where
    meta = id-meta cn ce s

pop-n-push : (n cn ce : ℕ) → (s : SingleLinkedStack ℕ) → pop-n-push-type n cn ce s

pop-n-push zero cn ce s    = refl
pop-n-push (suc n) cn ce s = begin
  M.exec (M.csComp {id-meta cn ce s} (M.cs popOnce) (n-push {id-meta cn ce (record {top = just (cons ce (SingleLinkedStack.top s))})} (suc (suc n)))) (id-meta cn ce s)
  ≡⟨ refl ⟩
  M.exec (M.csComp {id-meta cn ce s} (M.cs popOnce) (M.csComp {id-meta cn ce s} (n-push {id-meta cn ce (record {top = just (cons ce (SingleLinkedStack.top s))})} (suc n)) (M.cs pushOnce))) (id-meta cn ce s)
  ≡⟨ exec-comp (M.cs popOnce) (M.csComp {id-meta cn ce s} (n-push {id-meta cn ce (record {top = just (cons ce (SingleLinkedStack.top s))})} (suc n)) (M.cs pushOnce)) (id-meta cn ce s) ⟩
  M.exec (M.cs popOnce) (M.exec (M.csComp {id-meta cn ce s} (n-push {id-meta cn ce (record {top = just (cons ce (SingleLinkedStack.top s))})} (suc n)) (M.cs pushOnce)) (id-meta cn ce s))
  ≡⟨ cong (\x → M.exec (M.cs popOnce) x) (exec-comp (n-push {id-meta cn ce (record {top = just (cons ce (SingleLinkedStack.top s))})} (suc n)) (M.cs pushOnce) (id-meta cn ce s)) ⟩
  M.exec (M.cs popOnce) (M.exec (n-push {id-meta cn ce (record {top =
```

```
    just (cons ce (SingleLinkedStack.top s))})} (suc n))(M.exec (M.cs pushOnce) (id-meta cn ce s)))
  ≡⟨ refl ⟩
  M.exec (M.cs popOnce) (M.exec (n-push {id-meta cn ce (record {top = just (cons ce (SingleLinkedStack.top s))})} (suc n)) (id-meta cn ce (record {top = just (cons ce (SingleLinkedStack.top s))})))
  ≡⟨ sym (exec-comp (M.cs popOnce) (n-push {id-meta cn ce (record {top = just (cons ce (SingleLinkedStack.top s))})} (suc n)) (id-meta cn ce (record {top = just (cons ce (SingleLinkedStack.top s))}))) ⟩
  M.exec (M.csComp {id-meta cn ce s} (M.cs popOnce) (n-push {id-meta cn ce (record {top = just (cons ce (SingleLinkedStack.top s))})} (suc n))) (id-meta cn ce (record {top = just (cons ce (SingleLinkedStack.top s))}))
  ≡⟨ pop-n-push n cn ce (record {top = just (cons ce (SingleLinkedStack.top s))}) ⟩
  M.exec (n-push n) (id-meta cn ce (record {top = just (cons ce (SingleLinkedStack.top s))}))
  ≡⟨ refl ⟩
  M.exec (n-push n) (pushOnce (id-meta cn ce s))
  ≡⟨ refl ⟩
  M.exec (n-push n) (M.exec (M.cs pushOnce) (id-meta cn ce s))
  ≡⟨ refl ⟩
  M.exec (n-push {id-meta cn ce s} (suc n)) (id-meta cn ce s)
  ∎



n-push-pop-type : ℕ →  ℕ  → ℕ → SingleLinkedStack ℕ → Set1
n-push-pop-type n cn ce st = M.exec (M.csComp {meta} (n-pop {meta} n) (n-push {meta} n)) meta ≡ meta
  where
    meta = id-meta cn ce st

n-push-pop : (n cn ce : ℕ) → (s : SingleLinkedStack ℕ) → n-push-pop-type n cn ce s
n-push-pop zero    cn ce s = refl
n-push-pop (suc n) cn ce s = begin
  M.exec (M.csComp {id-meta cn ce s} (n-pop {id-meta cn ce s}  (suc n)) (n-push {id-meta cn ce s} (suc n))) (id-meta cn ce s)
  ≡⟨ refl ⟩
  M.exec (M.csComp  {id-meta cn ce s}  (M.cs (\m → M.exec (n-pop {id-meta cn ce s} n) (popOnce m))) (n-push {id-meta cn ce s} (suc n))) (id-meta cn ce s)
  ≡⟨ exec-comp (M.cs (\m → M.exec (n-pop n) (popOnce m))) (n-push {id-meta cn ce s}  (suc n))  (id-meta cn ce s) ⟩
  M.exec (M.cs (\m → M.exec (n-pop  {id-meta cn ce s}  n) (popOnce m))) (M.exec (n-push {id-meta cn ce s} (suc n)) (id-meta cn ce s))
  ≡⟨ refl ⟩
  M.exec (n-pop n) (popOnce (M.exec (n-push {id-meta cn ce s} (suc n)) (id-meta cn ce s)))
  ≡⟨ refl ⟩
  M.exec (n-pop n) (M.exec (M.cs popOnce) (M.exec (n-push {id-meta cn ce
```

```
  s} (suc n)) (id-meta cn ce s)))
≡⟨ cong (\x → M.exec (n-pop  {id-meta cn ce s}  n) x) (sym (exec-comp
  (M.cs popOnce) (n-push {id-meta cn ce s}  (suc n)) (id-meta cn ce s)))
   ⟩
M.exec (n-pop n) (M.exec (M.csComp {id-meta cn ce s} (M.cs popOnce) (n-
  push {id-meta cn ce s} (suc n))) (id-meta cn ce s))
≡⟨ cong (\x → M.exec (n-pop {id-meta cn ce s}  n) x) (pop-n-push n cn
  ce s) ⟩
M.exec (n-pop n) (M.exec (n-push n) (id-meta cn ce s))
≡⟨ sym (exec-comp (n-pop n) (n-push n) (id-meta cn ce s)) ⟩
M.exec (M.csComp (n-pop n) (n-push n)) (id-meta cn ce s)
≡⟨ n-push-pop n cn ce s ⟩
id-meta cn ce s
∎
```