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

Abstract

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、既存の実装に対する検証は行なえない。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することなくソフトウェアの検証を行なうことができる。

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。また、証明的な検証として CbC における型システムを部分型と

して定義する。部分型を利用して CbC のプログラムが証明支援系言語 Agda 上で正しく証明可能な形で定義できることを示す。

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 3 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

```
1 __code cs0(int a, int b){
2   goto cs1(a+b);
3 }
4
5 __code cs1(int c){
6   goto cs2(c);
7 }
```

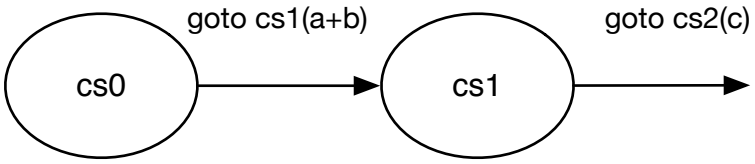

図 2.1: CodeSegment の軽量継続

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

```
1 __code print_factorial(int prod)
2 {
3   printf("factorial = %d\n", prod);
4   exit(0);
5 }
6
7 __code factorial0(int prod, int x)
8 {
```

```
9   if (x >= 1) {
10    goto factorial0(prod*x, x-1);
11  } else {
12    goto print_factorial(prod);
13  }
14
15 }
16
17 __code factorial(int x)
18 {
19   goto factorial0(1, x);
20 }
21
22 int main(int argc, char **argv)
23 {
24   int i;
25   i = atoi(argv[1]);
26
27   goto factorial(i);
28 }
```

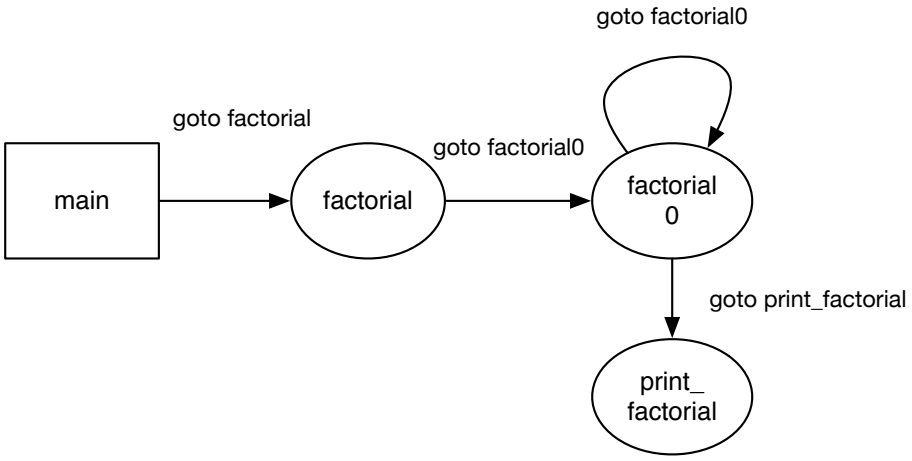

図 2.2: 階乗を求める CbC プログラム

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)。

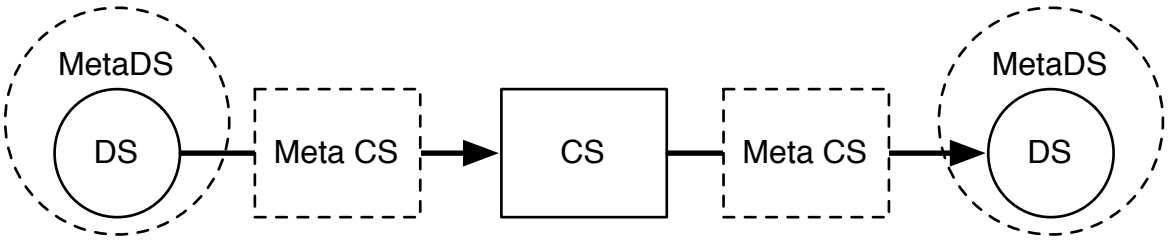

図 2.3: Meta CodeSegment と Meta DataSegment

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

CbC におけるメタ計算は軽量継続を行なう際に Meta CodeSegment を挟むことで実現できる。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 とはマルチコア CPU や GPU 環境での動作を対象とした OS であり、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 によって形式化されている [11]。現在存在するメタ計算としてメモリの確保と割り当て、並列に書き込むことが可能な Synchronized Queue、データの保存に用いる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/CUDA 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ものである。通常 GPU でデータの処理を行なう場合はデータの転送、転送終了を同期で確認、 kernel 実行、kernel の終了を同期で確認する、という手順が必要である。CPU/GPU での処理をメタ計算で行なうことにより、ノーマルレベルでは CodeGear が実行されるデバイスや DataGear の位置を意識する必要が無いというメリットがある。

GearsOS においては軽量継続の呼び出し部分もメタ計算として実現されている。ある CodeGear から次の CodeGear へと継続する際には、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。これは従来の OS の Dynamic Loading Libary や Command の呼び出しに相当する。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。

具体的には Meta DataGear には以下のようなものが格納される。

- 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。

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

```
1 /* Context definition */
2
```

```
3  #define ALLOCATE_SIZE 1024
4
5  enum Code {
6      Code1,
7      Code2,
8      Allocator,
9  };
10
11 enum UniqueData {
12     Allocate,
13     Tree,
14 };
15
16 struct Context {
17     int codeNum;
18     __code (**code) (struct Context *);
19     void* heap_start;
20     void* heap;
21     long dataSize;
22     int dataNum;
23     union Data **data;
24 };
25
26 union Data {
27     struct Tree {
28         union Data* root;
29         union Data* current;
30         union Data* prev;
31         int result;
32     } tree;
33     struct Node {
34         int key;
35         int value;
36         enum Color {
37             Red,
38             Black,
39         } color;
40         union Data* left;
41         union Data* right;
42     } node;
43     struct Allocate {
44         long size;
45         enum Code next;
46     } allocate;
47 };
```

- 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。

なお、この Context から DataGear を取り出す Meta CodeSegment を stub と呼ぶ。stub の例をリスト 2.4 に示す。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。現在、この stub は全ての CodeGear に対してユーザが 1 つずつ定義する必要がある。この作業は非常に煩雑であり、CodeGear の定義から生成するスクリプトを用いて定義の簡易化を行なっているが、コンパイラ側でのサポートは入っていない。この stub を型情報から自動生成するために Continuation based C における型システムを定義する必要がある。

リスト 2.4: GearsOS における stub Meta CodeSegment

```
1  __code put(struct Context* context,
2             struct Tree* tree,
3             struct Node* root,
4             struct Allocate* allocate)
5  {
6      /* ... */
7  }
8
9  __code put_stub(struct Context* context)
10 {
11     goto put(context,
12              &context->data[Tree]->tree,
13              context->data[Tree]->tree.root,
```

```
14                &context->data[Allocate]->allocate);
15  }
```

2.5　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。メタ計算として定義することにより、ノーマルレベルからは木のバランスを必要なく要素の挿入と探索、削除が行なえる。赤黒木とは二分探索木の一種であり、木の各ノードが赤と黒の色を持っている。木に対して要素の挿入や削除を行なった際、その色を用いて木のバランスを保つ。

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 2.4 に示す。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。

これらの条件より、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。

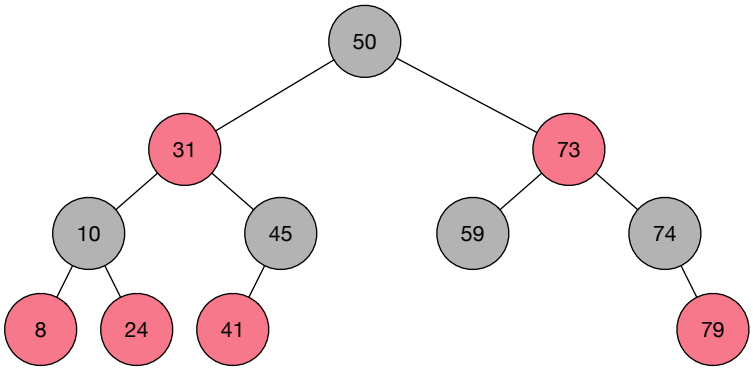

図 2.4: 赤黒木の例

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 2.5)。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。

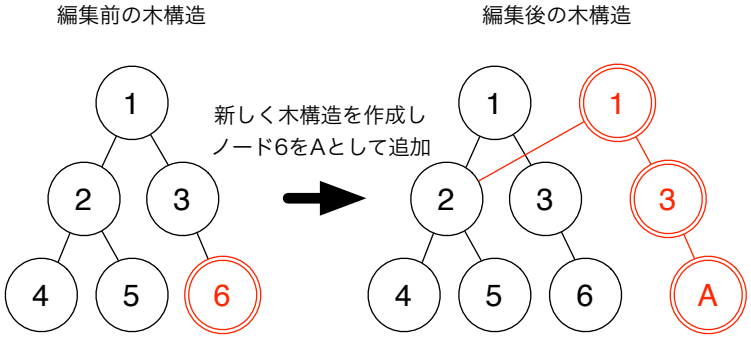

図 2.5: 非破壊赤黒木の編集

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ため、関数の再帰呼び出しによって木が辿れないことがある。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・削除時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、辿った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 に保持させる。

赤黒木を格納する DataSegment と Meta DataSegment の定義をリスト 2.5 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 2.5: 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;
```

```
49      union Data **data;
50  };
```

Meta DataSegment を初期化する Meta CodeSegment initLLRBContext をリスト 2.6 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 2.6 34-48) とスタックの初期化 (リスト 2.6 50-51) を行なう。

リスト 2.6: 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;
25      context->code[InsertCase4]   = insert4_stub;
26      context->code[InsertCase4_1] = insert4_1_stub;
27      context->code[InsertCase4_2] = insert4_2_stub;
28      context->code[InsertCase5]   = insert5_stub;
29      context->code[StackClear]    = stackClear_stub;
30      context->code[Exit]       = exit_code;
31
32      context->heap = context->heapStart;
33
34      context->data[Allocate] = context->heap;
35      context->heap += sizeof(struct Allocate);
36
37      context->data[Tree] = context->heap;
38      context->heap += sizeof(struct Tree);
```

```
39
40      context->data[Node] = context->heap;
41      context->heap += sizeof(struct Node);
42
43      context->dataNum = Node;
44
45      struct Tree* tree = &context->data[Tree]->tree;
46      tree->root = 0;
47      tree->current = 0;
48      tree->deleted = 0;
49
50      context->node_stack = stack_init(sizeof(struct Node*), 100);
51      context->code_stack = stack_init(sizeof(enum Code), 100);
52  }
```

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 2.7 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 2.7 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 2.7 14-16)。

リスト 2.7: 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);
12  }
13
14  __code insert2_stub(struct Context* context) {
15      goto insertCase2(context, context->data[Tree]->tree.current);
16  }
```

2.6 メタ計算ライブラリ akasha を用いた赤黒木の実装の検証

GearsOS の赤黒木の仕様の定義とその確認を CbC で行なっていく。赤黒木には以下の性質が求められる。

- 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、木に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、という木がバランスしている際に成り立つ式を定義する (リスト 2.8)。

リスト 2.8: 木の高さに関する仕様記述

```
1 void verifySpecification(struct Context* context, struct Tree* tree) {
2     assert(!(maxHeight(tree->root, 1) > 2*minHeight(tree->root, 1)));
3     return meta(context, EnumerateInputs);
4 }
```

リスト 2.8 で定義した仕様が常に成り立つか、全ての挿入順番を列挙しながら確認していく。まずは最も単純な有限の個数の任意の順の数え上げに対して検証していく。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 2.9)。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 2.9: 検証を行なうための Meta DataSegment

```
1 // Data Segment
2 union Data {
3     struct Tree { /* ... */ } tree;
4     struct Node { /* ... */ } node;
5
6     /* for verification */
7     struct IterElem {
8         unsigned int val;
```

```
9           struct IterElem* next;
10      } iterElem;
11      struct Iterator {
12          struct Tree* tree;
13          struct Iterator* previousDepth;
14          struct IterElem* head;
15          struct IterElem* last;
16          unsigned int  iteratedValue;
17          unsigned long iteratedPointDataNum;
18          void*         iteratedPointHeap;
19      } iterator;
20      struct AkashaInfo {
21          unsigned int minHeight;
22          unsigned int maxHeight;
23          struct AkashaNode* akashaNode;
24      } akashaInfo;
25      struct AkashaNode {
26          unsigned int       height;
27          struct Node*       node;
28          struct AkashaNode* nextAkashaNode;
29      } akashaNode;
30  };
```

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 2.10 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 2.10: 木の最も短かい経路の長さを確認する Meta CodeSegment

```
1   __code getMinHeight_stub(struct Context* context) {
2       goto getMinHeight(context, &context->data[Allocate]->allocate, &
        context->data[AkashaInfo]->akashaInfo);
3   }
4
5   __code getMinHeight(struct Context* context, struct Allocate* allocate,
        struct AkashaInfo* akashaInfo) {
6       const struct AkashaNode* akashaNode = akashaInfo->akashaNode;
7
8       if (akashaNode == NULL) {
9           allocate->size = sizeof(struct AkashaNode);
```

```
10          allocator(context);
11          akashaInfo->akashaNode = (struct AkashaNode*)context->data[
        context->dataNum];
12
13          akashaInfo->akashaNode->height = 1;
14          akashaInfo->akashaNode->node   = context->data[Tree]->tree.root;
15
16          goto getMaxHeight_stub(context);
17      }
18
19      const struct Node* node = akashaInfo->akashaNode->node;
20      if (node->left == NULL && node->right == NULL) {
21          if (akashaInfo->minHeight > akashaNode->height) {
22              akashaInfo->minHeight  = akashaNode->height;
23              akashaInfo->akashaNode = akashaNode->nextAkashaNode;
24              goto getMinHeight_stub(context);
25          }
26      }
27
28      akashaInfo->akashaNode = akashaInfo->akashaNode->nextAkashaNode;
29
30      if (node->left != NULL) {
31          allocate->size = sizeof(struct AkashaNode);
32          allocator(context);
33          struct AkashaNode* left = (struct AkashaNode*)context->data[
        context->dataNum];
34          left->height           = akashaNode->height+1;
35          left->node             = node->left;
36          left->nextAkashaNode   = akashaInfo->akashaNode;
37          akashaInfo->akashaNode = left;
38      }
39
40      if (node->right != NULL) {
41          allocate->size = sizeof(struct AkashaNode);
42          allocator(context);
43          struct AkashaNode* right = (struct AkashaNode*)context->data[
        context->dataNum];
44          right->height            = akashaNode->height+1;
45          right->node              = node->right;
46          right->nextAkashaNode    = akashaInfo->akashaNode;
47          akashaInfo->akashaNode   = right;
48      }
49
50      goto getMinHeight_stub(context);
51  }
```

同様に最も高い高さを取得し、仕様であるリスト 2.8 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 2.7 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 2.11)。

リスト 2.11: 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 2.12 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 2.12: 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);
}
```

第3章　ラムダ計算と型システム

2 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、さらに多くの要素を検証したり無限回の挿入を検証するには状態の抽象化や CbC 側に記号実行の機構を組み込んだり証明を行なう必要がある。CbC は直接自身を証明する機構が存在しない。プログラムの性質を証明するには CbC の形式的な定義が必須となる。3 章では CbC の項の形式的な定義の一つとして、部分型を用いて CbC の CodeSegment と DataSegment が定義できることを示していく。また、型システムの別の利用方法として命題が型で表現できる Curry-Howard 対応を利用した証明が存在するが、その利用方法については 4 章で述べる。

3.1　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 [12] [13]。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、データの単位を間違えてしまったり、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なって早期に指摘できる。この指摘できる詳細さは、型システムの表現力とプログラムの内容に依存する。多用なデータ構造を扱うプログラム (コンパイラのような記号処理アプリケーションなど) は数値計算のような数種類の単純な型しか使わないプログラムよりも型検査器から受けられる恩恵が大きい。他にも、ある種のプログラムにとっては型は保守のためのツールともなる。複雑なデータ構造を変更する時、その構造に関連するソースコードを型検査器は明らかにしてくれる。

- 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。モジュール化されたデータ構造は厳格に定義されたインターフェースを経由して呼び出すことになる。このインターフェースは利用する側に取っては呼び出しの規約となり、実装する側にとってはモジュールの要約となる。

- ドキュメント化

  型はプログラムを理解する際にも有用である。関数やモジュールの型を確認することにより、どのデータを対象としているのかといった情報が手に入る。また、型はコンパイラが実行されるために検査されるため、コメントに埋め込まれた情報と異なり常に正しい情報を提供する。

- 言語の安全性

  安全性のの定義は言語によって異なるが、型はデータの抽象化によってある種の安全性を確保できる。例えば、プログラマは配列をソートする関数があった場合、与えられた配列のみがソートされ、他のデータには影響が無いことを期待するだろう。しかし、低水準言語ではメモリを直接扱えるため、予想された処理の範囲を越えてデータを破壊する可能性がある。より安全な言語ではメモリアクセスが抽象化し、データを破壊する可能性をプログラマに提供しないという選択肢がある。

- 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 などにおける式の区別であった。整数の算術式と実数の算術式を区別し、数値計算の効率化を測るために導入されたのである。型の導入により、コンパイラはプリミティブな演算とは異なる表現を用い、実行コードを生成する時に適切な機械語表現を行なえるようになった。昨今の高性能コンパイラでは最適化とコード生成のフェーズにおいて型検査器が収集する情報を多く利用している。

型システムの定義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、部分型などが存在し、動的型付けや静的型付けなど、言語によって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決めることにも繋がる。

3.2　型無し算術式

まず、型システムやその性質について述べるためにプログラミング言語そのものの基本的な性質について述べる。プログラムの構文と意味論、推論について考えるために自然数とブール値のみで構成される小さな言語を扱いながら考察する。この言語は二種類の値しか持たないが、項の帰納的定義や証明、評価、実行時エラーのモデル化を表現することができる。

この言語はブール定数 $true$ と $false$ 、条件式、数値定数 0 、算術演算子 $succ$ と $pred$ 、判定演算子 $iszero$ のみからなる。算術演算子 $succ$ は与えられた数の次の数を返し、$pred$ はその前の数を返す。判定演算子 $iszero$ は与えられた項が 0 なら $true$ を返し、それ以外は $false$ を返す。これらを文法として定義すると以下のリスト 3.1 のようになる。

リスト 3.1: 算術式の項定義

```
1 t ::=
2       true
3       false
4       if t then t else t
5       0
6       succ t
7       pred t
8       iszero t
```

この定義では算術式の項 t を定義している。::= は項の集合の定義を表であり、t は項の変数のようなものである。それに続くすべての行は、構文の選択肢である。構文の選択肢内に存在する記号 t は任意の項を代入できることを表現している。このように再帰的に定義することにより、 `if (ifzero (succ 0)) then true else (pred (succ 0))` といった項もこの定義に含まれる。例において、 $succ$ 、 $pred$ 、 $iszero$ に複合的な引数を渡す場合は読みやすさのために括弧でくくっている。括弧の定義は項の定義には含んでいない。コンパイラなど具体的な字句をパースする必要がある場合、曖昧な構文を排除するために括弧の定義は必須である。しかし、今回は型システムに言及するために曖昧な構文は明示的に括弧で指示することで排除し、抽象的な構文のみを取り扱うこととする。

現在、項と式という用語は同一である。型のような別の構文表現を持つ計算体系においては式はあらゆる種類の構文を表す。項は計算の構文的表現という意味である。

この言語におけるプログラムとは上述の文法で与えられた形からなる項である。評価の結果は常にブール定数か自然数のどちらかになる。これら項は値と呼ばれ、項の評価順序の形式化において区別が必要となる。

なお、この項の定義においては `succ true` といった怪しい項の形成を許してしまう。実際、これらのプログラムは無意味なものであり、このような項表現を排除するために型システムを利用する。

ある言語の構文を定義する際に、他の表現かいくつか存在する。先程の定義は次の帰納的な定義のためのコンパクトな記法である。

定義 **3.1** 項の集合とは以下の条件を満たす最小の集合 T である。

$$\{true, false, 0\} \subseteq T$$
$$t_1 \in T \text{ ならば } \{succ\, t_1, pred\, t_1, iszero\, t_1\} \subseteq T$$
$$t_1 \in T \text{ かつ } t_2 \in T \text{ かつ } t_3 \in T \text{ ならば } if\ t_1\ then\ t_2 else\ t_3 \subseteq T$$

まず 1 つめの条件は、T に属する 3 つの式を挙げている。2 つめと 3 つめの条件は、ある種の複合的な式が T に属することを判断するための規則を表している。最後の「最小」という単語は T がこの 3 つの条件によって要求される要素以外の要素を持たないことを表している。

また、項の帰納的表現の略記法として、二次元の推論規則形式を用いる方法もある。これは論理体系を自然演繹スタイルで表現するためによく使われる。自然演繹による証明は **??**章内で触れるが、今回は項表現として導入する。

定義 **3.2** 項の集合は次の規則によって定義される。

$$true \in T$$

$$false \in T$$

$$0 \in T$$

$$\frac{t_1 \in T}{succ\, t_1 \in T}$$

$$\frac{t_1 \in T}{pred\, t_1 \in T}$$

$$\frac{t_1 \in T}{iszero\, t_1 \in T}$$

$$\frac{t_1 \in T \qquad t_2 \in T \qquad t_3 \in T}{if\ t_1\ then\ t_2\ else\ t_3 \in T}$$

最初の $true,\ false,\ 0$ の 3 つ規則は再帰的定義の 1 つめの条件と同じである。それ以外の 4 つの規則は再帰的定義の 2 つめと 3 つめの条件と同じである。それぞれの規則は「も

し線の上に列挙して前提が成立するのならば、線の下の結論を導出できる」と読む。T がこれらの規則を満たす最小の集合である事実は明示的に述べられない。

言語の構文は定義できたので、次は項がどう評価されるかの意味論について触れていく。意味論の形式化には操作的意味論や表示的意味論、公理的意味論やゲーム意味論などがあるが、ここでは操作的意味論について述べる。操作的意味論とは、言語の抽象機械を定義することにより言語の振舞いを規程する。この抽象機械が示す抽象とは、扱う命令がプロセッサの命令セットなどの具体的なものでないことを表している。単純な言語の抽象機械における状態は単なる項であり、機械の振舞いは遷移関数で定義される。この関数は各状態において項の単純化ステップを実行して次の状態を与えるか、機械を停止させる。ここで項 t の意味は、t を初期状態として動き始めた機械が達する最終状態である。

なお、一つの言語に複数の操作的意味論を与えることもある。例えば、プログラマが扱う項に似た機械状態を持つ意味論の他に、コンパイラの内部表現やインタプリタが扱う意味論を定義する。これらの振舞いが同じプログラムを実行した時に何かしらの意味であれば、結果としてその言語の実装の正しさを証明することに繋がる。

まずはブール式のみの操作的意味論を定義する。

定義 **3.3** ブール値 (B)

項

$t ::=$	項
$true$	定数真
$false$	定数偽
$if\ t\ then\ t\ else\ t$	条件式

値

$v ::=$	値
$true$	真
$false$	偽

評価

$$if\ true\ then\ t_2\ elset_3 \rightarrow t_2 \quad \text{(E-IFTRUE)}$$

$$if\ false\ then\ t_2\ elset_3 \rightarrow t_3 \quad \text{(E-IFFALSE)}$$

$$\frac{t_1 \rightarrow t_1'}{if\ t_1\ then\ t_2\ else\ t_3 \rightarrow if\ t_1'\ thent_2\ else\ t_3} \quad \text{(E-IF)}$$

評価の最終結果になりえる項である値は定数 $true$ と $false$ のみである。評価の定義は評価関係の定義である。評価関係 $t \to t'$ は「t が 1 ステップで t' に評価される」と読む。直感的には抽象機械の状態が t ならば t' が手に入るという意味である。

評価関係は 3 つあるが、2 つは前提を持たないため、2 つの公理と 1 つの規則から成る。1 つめの規則 E-IFTRUE の意味は、評価の対象となる項の条件式が定数 $true$ である時に、then 節にある t_2 を残して他の全ての項を捨てるという意味である。E-EIFFALSE も同様に条件式が $false$ の時に t_3 のみを残す。3 つ目の規則 E-IF は条件式の評価である。条件式 t が t' に評価されうるのならば then 節と else 節を変えずに条件部のみを評価する。

評価の定義から分かることの中に、if の中の then 節 と else 節は条件部より先に評価されないことがある。よって、この言語は条件式の評価に対し条件部から評価が優先されるという評価戦略を持つことが分かる。

定義 **3.4** 推論規則のインスタンスとは、規則の結論や前提に対し、一貫して同じ項による書き換えを行なったものである。

例えば、$if\ true\ then\ true\ else\ (\ if\ false\ then\ false\ else\ false)$ は E-IFTRUE のインスタンスであり、 E-IFTRUE の t_2 が `true` かつ、t_3 が $if\ false\ then\ false\ else\ false$ の時に相当する。

定義 **3.5** 1 ステップ評価関係 $\to$ とは、3 つの評価の規則を満たす、項に関する最小の二項関係である。$(t,\ t')$ がこの関係の元である時、「評価関係式 $t \to t'$ は導出可能である」と言う。

ここで「最小」という言葉が表れるため、評価関係式 $t \to t'$ が導出可能である時かつその時に限り、その関係式は規則によって正当化される。すなわち評価関係式は公理 E-IFTRUE か E-IFFALSE 、前提が成り立つ時の E-IF のインスタンスとなる。与えられた評価関係式が導出可能であることを証明するには、葉が E-IFTRUE か E-IFFALSE であり、内部ノードのラベルが E-IF のインスタンスである導出木が示せれば良い。例えば以下の略記の元 $if\ t\ then\ false\ then\ false \to if\ u\ then\ false\ else\ false$ の導出可能性は以下のような導出木によって示せる。

- $s = if\ true\ then\ false\ else\ false$
- $t = if\ s\ then\ true\ else\ true$
- $u = if\ false\ then\ true\ else\ true$

$$\dfrac{\dfrac{\dfrac{}{s \to true}\ \text{E-IFTRUE}}{t \to u}\ \text{E-IF}}{if\ t\ then\ false\ then\ false/ \to if\ u\ then\ false\ else\ false}\ \text{E-IF}$$

1 ステップ評価関係は与えられた項に対して抽象機械の状態遷移を定義する。この時、機械がそれ以上ステップを進められない時にそれが最終結果となる。

定義 **3.6** 正規形

項 t が正規形であるとは、$t \to t'$ となる評価規則が存在しないことである。

この言語において $true$ や $false$ は正規形である。逆に言えば、構文的に正しい if が用いられている場合は評価することが可能なため正規形ではない。極端に言えばこの言語における全ての値は正規形なのである。しかし、他の言語における値は一般的に正規形ではない。実のところ、値でない正規形は実行時エラーとなって表れる。

実際にこの言語に自然数を導入し、値では無い正規形を確認していく。

定義 **3.7** 算術式 BN (B の拡張) の項

$t ::=$	項
$true$	定数真
$false$	定数偽
$if\ t\ then\ t\ else\ t$	条件式
0	定数ゼロ
$succ\ t$	後者値
$pred\ t$	前者値
$iszero\ t$	ゼロ判定

定義 **3.8** 算術式 BN の値

$v ::=$	値
$true$	真
$false$	偽
nv	数値

定義 **3.9** 算術式 BN の数値

$nv ::=$	数値
0	ゼロ
$succnv$	後者値

定義 **3.10** 算術式 BN の評価 $(t \to t')$

$$if\ true\ then\ t_2\ elset_3 \to t_2 \quad \text{(E-IFTRUE)}$$

$$if\ false\ then\ t_2\ elset_3 \to t_3 \quad \text{(E-IFFALSE)}$$

$$\frac{t_1 \to t_1'}{if\ t_1\ then\ t_2\ else\ t_3 \to if\ t_1'\ thent_2\ else\ t_3} \quad \text{(E-IF)}$$

$$pred\ 0 \to 0 \quad \text{(E-PREDZERO)}$$

$$pred\ (succ\ nv_1) \to nv_1 \quad \text{(E-PREDSUCC)}$$

$$\frac{t_1 \to t_1'}{pred\ t_1 \to pred\ t_1'} \quad \text{(E-PRED)}$$

$$iszero\ 0 \to true \quad \text{(E-ISZEROZERO)}$$

$$iszero\ (succ\ nv_1) \to false \quad \text{(E-ISZEROSUCC)}$$

$$\frac{t_1 \to t_1'}{iszero\ t_1 \to iszero\ t_1'} \quad \text{(E-ISZERO)}$$

今回値の定義に数値を表す構文要素が追加されている。数は 0 かある数に後者関数を適用したもののどちらかである。評価規則 E-PREDZERO、E-PREDSUCC、E-ISZEROZERO、E-ISZEROSUCC は演算 `pred` と `iszero` が数に適用された時にどう振る舞うかを定義している。E-SUCC 、 E-PRED 、 E-ISZERO の合同規則も E-IF のように部分項から先に評価することを示している。

数値の構文要素 (nv) はこの定義によって重要な役割をはたす。例えば、E-PREDSUCC 規則が適用できる項は任意の項 t ではなく数値 nv_1 である。これは $pred\ (succ\ (pred\ 0))$ を $pred0$ に評価できないことを意味する。なぜなら $pred\ 0$ は数値に含まれないからである。

ここで言語の操作的意味論について考える時、すべての項に関する振舞いを定義する必要がある。すべての項には $pred\ 0$ や $succfalse$ のような項も含まれる。しかし、 $succ$ を $false$ に適用する評価結果は定義されていないため、 $succ\ false$ は正規形である。このような、正規形であるが値でない項は行き詰まり状態であるという。つまり、実行時エ

ラーとは行き詰まり状態の項を指す。直感的な解釈としてはプログラムが無意味な状態になったこと示しておい、操作的意味論が次に何も行なえないことを特徴付けているのである。プログラング言語において実行時エラーはセグメンテーションフォールトや不正な命令などいくつかのものが挙げられるが、型システムを考える際にはこれらのエラーは行き詰まり状態という単一の概念で表す。

3.3　単純型

先程定義した算術式には $pred\ false$ のようなこれ以上評価できない行き詰まり状態が存在する。項を実際に評価する前に評価が行き詰まり状態にならないことを保証したい。そのために、自然数に評価される項とブール値に評価される項とを区別する必要がある。項を分類するために 2 つの型 Nat と Bool を定義する。

ここで、項 t が型 T を持つ、という表現を用いた場合、t を評価した結果が明らかに適切な形の値になることを意味する。明らかに、という意味は項を実行することなく静的に分かるという意味である。例えば項 $if\ true\ then\ false\ else\ true$ は Bool 型を持ち、$pred\ (succ\ (succ\ 0))$ は Nat 型を持つ。しかし、項の型の分析は保守的であり、$if\ true\ then\ 0\ else\ false$ のような項は実際には行き詰まりにならないが型を持てない。

算術式のための型付け関係は $t : T$ と書き、項に型を割り当てる推論規則の集合によって定義される。具体的な数値とブール値に関する拡張は以下である。

定義 **3.11** NB(型付き) の新しい構文形式

$T ::=$	型 :
$Bool$	ブール型
Nat	自然数型

定義 **3.12** NB(型付き) の型付け規則

$$true : Bool \qquad \text{T-TRUE}$$

$$false : Bool \qquad \text{T-FALSE}$$

$$\frac{t_1 : Bool \qquad t_2 : T \qquad t_3 : T}{if\ t_1\ then\ t_2\ else\ t_3 : T} \qquad \text{T-IF}$$

$$0 : Nat \qquad \text{T-ZERO}$$

$$\frac{t_1 : Nat}{succ\ t_1 : Nat} \qquad \text{T-SUCC}$$

$$\frac{t_1 : Nat}{pred\ t_1 : Nat} \qquad \text{T-PRED}$$

$$\frac{t_1 : Nat}{iszero\ t_1 : Bool} \qquad \text{T-BOOL}$$

T-TRUE と T-FALSE はブール定数に Bool 型を割り当てている。T-IF は条件式の部分に Bool 型を、部分式に関しては同じ型を要求している。これは同じ変数 T を二回使用することで制約を表している。

また、数に関しては T-ZERO は Nat 型を 0 に割り当てている。T-SUCC と T-PRED は t_1 が Nat である時に限り Nat 型となる。同様に、 T-ISZERO は t_1 が Nat である時に Bool となる。

定義 **3.13** 算術式のための型付け関係とは、NB における規則のすべてのインスタンスを満たす、項と型の二項関係である。項 t に対してある型 T が存在して $t : T$ である時、t は型付け可能である (または正しく型付けされている) という。

型推論をを行なう時、$succt_1$ という項が何らかの型を持つならばそれは Nat 型である、といった言及を行なう。型付け関係を逆転させた補題を定義することで型推論の基本的なアルゴリズムを考えることができる。なお、逆転補題は型付け関係の定義により直ちに成り立つ。

補題 **3.3.1** 型付け関係の逆転

1. $true : R$ ならば $R = Bool$ である
2. $false : R$ ならば $R = Bool$ である
3. $if\ t_1\ then\ t_2\ else\ t_3 : R$ ならば $t_1 : Bool$ かつ $t_2 : R$ かつ $t_3 : R$ である。
4. $0 : R$ ならば $R = Nat$ である

5. $succ\, t_1 : R$ ならば $R = Nat$ かつ $t_1 : Nat$ である

6. $pred\, t_1 : R$ ならば $R = Nat$ かつ $t_1 : Nat$ である

7. $iszero\, t_1 : R$ ならば $R = Bool$ かつ $t_1 : Nat$ である

逆転補題は型付け関係のための生成補題と呼ばれることもある。なぜならば、与えられた型付け判断式に対してその証明がどのように生成されたかを示すからである。

型無し算術式の評価導出のように型付けも導出可能であり、それも規則のインスタンスの木である。型付け関係に含まれる二つ組 $(t,\, T)$ は $t : T$ を結論とする型付け導出により正当化される。例えば $if\ (iszero\ 0)\ then\ 0\ else\ (pred\ 0) : Nat$ の型付け判断の導出木である。

$$\dfrac{\dfrac{\dfrac{}{0 : Nat}\,\text{T-ZERO}}{iszero\ 0 : Bool}\,\text{T-ISZERO} \quad \dfrac{}{0 : Nat}\,\text{T-ZERO} \quad \dfrac{\dfrac{}{0 : Nat}\,\text{T-ZERO}}{pred\ 0 : Bool}\,\text{T-PRED}}{\text{if}\quad (\text{iszero}\quad 0)\quad \text{then}\quad 0\quad \text{else}\quad (\text{pred}\quad 0) : \text{Nat}}\,\text{T-IF}$$

項その型付けの定義より、型システムが行き詰まり状態にならないことを示す。その証明は指向定理と保存定理によって証明する。

- 進行とは、正しく型付けされた項は行き詰まり状態では無いことである
- 保存とは、評価可能な正しく型付けされた項は評価後も正しく型付けされていることである。

型システムがこれらの性質を持つ時、正しく型付けされた項は行き詰まり状態になりえない。

進行定理の証明の為に Bool 型と Nat 型の標準形 (それらの型を持つ正しく型付けされた値) を示す。

補題 **3.3.2** 標準形

1. v が $Bool$ 型の値ならば v は $true$ または $false$ である。

2. v が Nat 型の値ならば、0 もしくは Nat に対して $succ$ を適用した値である。

標準形の証明に関しては値における構造的帰納法を用いる。この言語における値とは $true$ と $false$ と 0 と $succ\, nv$ のいずれかの形をしている。Bool 型に関して注目した時、$true$ と $false$ は定義によって正しい。0 と $succ\, nv$ に関しては逆転補題より Nat 型を持つため、Bool 型を持つ値は $true$ と $false$ のどちらかとなる。Nat についても同様である。

定理 **3.3.1** 進行

t が正しく型付けされたと仮定すると、t は値であるか、またはある t' が存在して $t \to t'$ となる。

証明は $t : T$ の導出に関する帰納法による。T-TRUE 、 T-FALSE 、 T-ZERO の場合は t が値であることより成立する。

T-IF の場合、帰納法の仮定により $t1$ は値であるか、t'_1 が存在して $t_1 \to t'_1$ を満たす。t_1 が値ならば、標準形補題により $true$ か $false$ であり、その場合は E-IFTRUE か E-IFFALSE が適用可能である。一方 $t_1 \to t'_1$ ならば E-IF が適用できる。

T-SUCC の場合も帰納法の仮定により $t1$ は値であるか、t'_1 が存在して $t_1 \to t'_1$ を満たす。t_1 が値ならば標準形補題により数値でなければならず、その場合 t も数値であるため成り立つ。一方 $t_1 \to t'_1$ ならば E-SUCC が適用できる。

T-SUCC の場合も同様で、 t_1 が値ならば標準形補題により数値でなければならず、その場合 E-PREDZERO か E-PREDSUCC が使える。$t_1 \to t'_1$ ならば E-PRED が適用できる。

T-ISZERO の場合も値ならば標準形補題により t_1 は数値であり、どちらの場合でも E-ISZEROZERO と E-ISZEROSUCC が適用できる。$t_1 \to t'_1$ ならば E-ISZERO が適用できる。

定理 **3.3.2** 保存

$t : T$ かつ $t \to t'$ ならば $t' : T$ となる。

保存定理も $t : T$ の導出に関する帰納法によって導ける。帰納法の各ステップにおいて全ての部分導出に関して所望の性質が成り立つと仮定し、導出の最後の規則についての場合分けで証明を行なう。

導入の最後の規則が T-TRUE の場合、その規則の形から t は定数 $true$ でなければならず、 T は $Bool$ となる。そして t は値であるためにどのような t' も存在せず、定理の要求は満たされる。T-FALSE と T-ZERO の場合も同様である。

導入の最後の規則 T-IF の場合は、t はある t_1, t_2 t_3 に対して $if\ t_1 then t_2 else t_3$ という形となる。さらに $t_1 : Bool$ と $t_2 : T$ と $t_3 : T$ となる部分導出がある。ここで if を持つ評価規則において $t \to t'$ を導入できる規則は E-IFTRUE と E-IFFALSE と E-IF のみである。それぞれの場合について別々に場合分けをして考える。

- E-IFTRUE の場合 (E-IFFALSE も同様)

  $t \to t'$ が E-IFTRUE を使った導出ならば、 t_1 は $true$ であり、結果の項 t' は t_2 となる。このことより $t_2 : T$ であることが分かるため、条件を満たす。

- E-IF の場合

  場合分け T-IF の仮定より $t_1 : Bool$ が結論となる、部分導出が得られる。帰納法の仮定を部分導出に適用して $t'_1 : Bool$ とし、 $t_2 : T$ と $t_3 : T$ を合わせると規則 T-IF が適用できる。T-IF を適用すると $if\ t'_1\ then\ t_2\ else\ t_3$ となり、$t' : T$ が成り立つ。

T-SUCC が導入の最後であれば、$t \to t'$ を導くためには E-SUCC のみであり、この形から $t_1 \to t'_1$ が分かる。$t_1 : Nat$ であることも分かるため、帰納法の仮定より $t'_1 : Nat$ が得られる。この時 T-SUCC が適用できるため $succ\, t_1 : Nat$ となって $t' : T$ が成り立つ。T-PRED も同様である。

3.4　型なしラムダ計算

計算とは何か、エラーとは何か、を算術式を定義することによって示してきた。また、型を導入することにより行き詰まり状態を回避することも示した。ここで、プログラミング言語における計算を形式的に定義していく。プログラミング言語は複雑だが、その計算はある本質的な仕組みからの派生形式として定式化可能であることを Peter Ladin が示した。 この時 Landin が使った本質的な仕組みとしての核計算がラムダ計算であった。ラムダ計算は Alonzo Church が発明した形式的体系の一つである。 ラムダ計算では全ての計算が関数定義と関数適用の基本的な演算に帰着される。ラムダ計算はプログラミング言語の機能の仕様記述や、言語設計と実装、型システムの研究に多く使われている。この計算体系の重要な点は、ラムダ計算内部で計算が記述できるプログラミング言語であると同時に、それ自身について厳格な証明が可能な数学的対象としてみなせる点にある。

ラムダ計算はいろいろな方法で拡張できる。数や組やレコードなどはラムダ計算そのもので模倣することができるが、記述が冗長になってしまう。それらの機能のための具体的な特殊構文を加えることは言語の利用者の視点で便利である。他にも書き換え可能な参照セルや非局所的な例外といった複雑な機能を表現することもできるが、膨大な変換を用いなければモデル化できない。それらを言語として備えた拡張に ML や Haskell といったものがある。

ラムダ計算 (または λ 計算) とは、関数定義と関数適用を純粋な形で表現する。ラムダ計算においてはすべてが関数である。関数によって受け付ける引数も関数であり、関数が返す結果もまた関数である。

ラムダ計算の項は変数と抽象と適用の 3 種類の項からなり、以下の文法に要約される。変数 x は項であり、項 t_1 から変数 x を抽象化した $\lambda x.t_1$ も項であり、項 t_1 を他の項 t_2 に適用した $t_1 t_2$ も項である。

$t ::=$

$$x$$
$$\lambda x.t$$
$$t\,t$$

ラムダ計算において関数適用は左結合とする。つまり、 $s\,t\,u$ は $(s\,t)\,u$ となる。

また､抽象の本体はできる限り右側へと拡大する。例えば $\lambda x.\ \lambda y.\ x\,y\,x$ は $\lambda x.(\lambda.y((x\,y)\,x))$ となる。

ラムダ計算には変数のスコープが存在する。抽象 $\lambda x.t$ の本体 t の中に変数 x がある時、 x の出現は束縛されていると言う。同様に、 λx は t をスコープとする束縛子であると言う。なお、 x を囲む抽象によって束縛されていない場所の x の出現は自由であると言う。例えば $x\ y$ や $\lambda y.\ x\ y$ における x の出現は自由だが、 $\lambda x.x$ や $\lambda z.\lambda x.\lambda y.x(y\ z)$ における x の出現は束縛されている。$(\lambda x.x)\ x$ においては、最初の x の出現は束縛されているが、2 つ目の出現は自由である。

ラムダ計算において、計算とは引数に対する関数の適用である。抽象に対して項を適用した場合、抽象の本体に存在する束縛変数に適用する項を代入したもので書き換える。図式的には

$$(\lambda x.t_{12})t_2 \to [x \mapsto t_2]t_{12}$$

と記述する。ここで $[x \mapsto t_2]t_{12}$ とは、t_12 中の自由な x を全て t_2 で置換した項を意味する。例えば、 $(\lambda x.x)\ y$ は y となり、項 $(\lambda x.x(\lambda x.x))(y\ z)$ は $y\ z\ (\lambda x.x)$ となる。

なお、 $(\lambda x.t_{12})t_2$ という形の項を簡約基 (redex, reducible expression) と呼び、上記の規則で簡約基を置換する操作をベータ簡約と呼ぶ。ラムダ計算のための評価戦略には数種類の戦略がある。

- 完全ベータ簡約

  任意の簡約基がいつでも簡約されうる。つまり項の中からどの順番で簡約しても良い。

- 正規順序簡約

  最も左で最も外側の簡約基が最初に簡約される。

- 名前呼び

正規順序の中でも抽象の内部での簡約を許さない。名前呼びの変種は Algol-60 や Haskell で利用されている。なお、Haskell においては必要呼びという最適化された変種を利用している。

- 値呼び

  ほとんどの言語はこの戦略を用いている。基本的には最も左の簡約基をを簡約するが、右側が既に値 (計算が終了してもう簡約できない閉じた項) になっている簡約基のみを簡約する。

値呼び戦略は関数の引数が本体で使われるかに関わらず評価され、これは正格と呼ばれる。名前呼びなどの非正格な戦略は引数が使われる時のみ評価され、これは遅延評価とも呼ばれる。

ラムダ計算において、複数の引数は、関数を返り値として返す高階関数として定義できる。項 s が二つの自由変数 x と y を含むとすれば、 $\lambda x.\lambda y.s$ と書くことで二つの引数を持つ関数を表現できる。これは x に v が与えられた時、y を受けとり、 s の抽象内の自由な x を v に置き換えた部分を置換する関数、を返す。例えば $(\lambda x.\lambda y.s)\ v\ w$ は $(\lambda y.[x \mapsto v]s)w$ に簡約され、 $[y \mapsto w][x \mapsto v]s$ に簡約される。なお、複数の引数を取る関数を高階関数に変換することはカリー化と呼ばれる。

ラムダ計算の帰納的な項は以下のように定義される。

定義 **3.14** V を変数名の加算集合とする。項の集合は以下を満たす最小の集合 T である。

$$\begin{aligned} \text{任意の } x \in V \text{ について } x \in T \\ t_1 \in T \text{ かつ } xinV \text{ ならばλ}x.t \in T \\ t_1 \in T \text{ かつ } t_2 \in T \text{ ならば } t_1\ t_2 \in T \end{aligned}$$

また、形式的な自由変数の定義を与える。

定義 **3.15** 項 t の自由変数の集合は $FV(t)$ と書き、以下のように定義される。

$$\begin{aligned} FV(x) &= \{x\} \\ FV(\lambda.t_1 x) &= FV(t_1) \setminus \{x\} \\ FV(t_1\ t_2) &= FV(t_1) \cup FV(t_2) \end{aligned}$$

記号 $\setminus$ は集合に対する二項演算子であり、$S \setminus T := x \in S : x \notin T$ である。つまり、t_1 の内部の自由変数の集合から x を抜いた集合である。

最後に代入について定義する。代入の操作は直感的には置換であるが、変数の束縛に注意しなくてはならない。例えば抽象への代入を以下のように定義する。

$$[x \mapsto s](\lambda y.t_1) = \lambda y.[x \mapsto s]t_1$$

この場合、束縛変数の名前によっては定義が破綻してしまう。例えば以下のようになる。

$$[x \mapsto y](\lambda x.x) = \lambda x.y$$

λ よって束縛されているはずの x が書き変わっている。これはスコープとして振る舞っていないので誤っている。この問題は項 t 内の変数 x の自由な出現と束縛された出現を区別しなかったために出現した誤りである。

そこで、x を束縛する項に対しては置換行なわないように定義を変える。

- $y = x$ の場合
$$[x \mapsto s](\lambda y.t_1) = \lambda y.t_1$$
- $y \neq x$ の場合
$$[x \mapsto s](\lambda y.t_1) = \lambda y.[x \mapsto s]t_1$$

この場合は束縛された変数を上書きしないが、逆に自由変数を束縛するケースが発生する。具体的には以下である。

$$[x \mapsto z](\lambda z.x) = \lambda z.z$$

項 s 中の自由変数が項 t に代入されて束縛される現象は変数捕獲と呼ばれる。これを避けるためには t の束縛変数の名前が s の自由変数の名前と異なることを保証する必要がある。変数捕獲を回避した代入操作は捕獲回避代入と呼ばれる。代入における名前の衝突を回避するために項の束縛変数の名前を一貫して変更することで変数捕獲を回避する方法も存在する。束縛変数の名前を一貫して変更することをアルファ変換と呼ばれる。これは関数抽象に対する束縛変数は問わないという直感からくるもので、 $\lambda x.x$ も $\lambda y.y$ も振舞いとしては同じ関数であるとみなすものである。捕獲回避の条件を追加した代入の定義は以下のような定義となる。

- 変数への代入
$$[x \mapsto s]x = s$$

- 存在しない変数への代入 ($y \neq x$ の時)

$$[x \mapsto s]y = y$$

- 抽象内の項への代入 ($y \neq x$ かつ y が s の自由変数でない)

$$[x \mapsto s](\lambda y.t_1) = \lambda y.[x \mapsto s]t_1$$

- 適用への代入

$$[x \mapsto s](t_1\ t_2) = (t_1[x \mapsto s])([x \mapsto s]t_2)$$

この定義は少なくとも代入が行なわれる際には正しく代入が行なえる。さらに、抽象が束縛している変数を名前では無く数字として扱う名無し表現も存在する。これは De Brujin 表現と呼ばれ、コンパイラ内部などでの項表現として用いられる。

最終的な型無しラムダ計算 λ の項の定義と評価の要約を示す。

定義 **3.16** $\rightarrow$ (型無し)

項

$t ::=$	項
$\lambda x.t$	ラムダ抽象
$t\ t$	関数適用

値

$v ::=$	値
$\lambda x.t$	ラムダ抽象値

評価 ($t \rightarrow t'$)

$$\frac{t_1 \rightarrow t_1'}{t_1\ t_2 \rightarrow t_1' t_2} \quad \text{E-APP1}$$

$$\frac{t_2 \rightarrow t_2'}{v_1\ t_2 \rightarrow v_1 t_2'} \quad \text{E-APP2}$$

$$(\lambda x.t_{12})\ v_2 \rightarrow [x \mapsto v_2]t_{12} \quad \text{E-APPABS}$$

項は変数かラムダ抽象か関数適用の 3 つにより構成される。また、ラムダ抽象値は全て値である。加えて評価は関数適用を行なう E-APPABS 計算規則と、適用の項を書き換える E-APP1 と E-APP2 合同規則により定義される。

この定義からも評価戦略と評価順序が分かる。関数を適用する E-APPABS は左側が抽象であり、右側が値である v_2 の時にしか適用されない。逆に、規則 E-APP1 の t_1 は任意の項にマッチするため関数部分が値でない関数適用に用いる。一方、E-APP2 は左辺が値であるようになるまで評価されない。よって、関数適用 $t_1\ t_2$ の評価順は、まず E-APP1 を用いて t_1 が値となった後に E-APP2 を用いて t_2 を値とし、最後に E-APPABS で関数を適用を行なう。

3.5　単純型付きラムダ計算

型無しラムダ計算に対して単純型を適用する場合、ラムダ抽象の型について考える必要がある。ラムダ抽象は値を取って値を返すため、関数として考えることもできる。差し当たり Bool 型における関数の型を $\lambda x.t :\rightarrow$ と定義する。この定義においては $\lambda x.true$ についても $\lambda x.\lambda y.y$ のような型も同一の型を持つ。この二つの項は値を適用すると値を返すという点では同じであるが、前者は $true$ を返し、後者は $\lambda y.y$ を返す。これでは関数を適用した際に返す値の型は関数の型から予測できず、加えてどの値に対して適用可能かも分からない。そのために引数にどのような型を期待しているのか、正しい型の値を適用するとどの型の値を返すのかを型情報に追加する。具体的には以下のように $\rightarrow$ を $T_1 \rightarrow T_2$ の形をした無限の型の族に置き換える。

定義 **3.17**　型 Bool 上の単純型の集合は次の文法により生成される。

$t ::=$	型 :
$T \rightarrow T$	関数の型
$Bool$	ブール値型

なお、型構築子 $\rightarrow$ は右結合である。つまり $T_1 \rightarrow T_2 \rightarrow T_3$ は $T_1 \rightarrow (T_2 \rightarrow T_3)$ となる。

$\lambda x.t$ に対して型を割り当てる時、明示的に型付けする方法と暗黙的に型付けする方法がある。明示的に型付けを行なう場合はプログラマが項に型の注釈を記述する。暗黙的に型付けを行なう場合は型検査器に情報を推論させ、型を再構築させる。型推論は λ 計算の文献内では型割り当て体系と呼ぶこともある。今回は明示的に型を指定する方法を取る。

λ 抽象の引数の型が分かれば、結果の型は本体 t_2 となる。ここで、t_2 内における x の出現は型 T_1 の項を表すと仮定する必要がある。これは引数に対して正しい型の値が渡さ

れたにも関わらず抽象内で異なる型として振る舞うのを禁止するためである。この λ 抽象の型付けは以下の T-ABS によって定義される。

$$\frac{x : T_1 \vdash t_2 : T_2}{\vdash \lambda x : T_1 . t_2 : T_1 \to T_2} \qquad \text{T-ABS}$$

項は抽象を入れ子で持つ可能性があるため、引数の仮定は複数持ちうる。このため型付け関係は二項関係 $t : T$ から、三項関係 $\Gamma \vdash t : T$ となる。ここで Γ とは t に表われる自由変数の型の仮定の集合である。Γ は型付け文脈や型環境と呼ばれ、$\Gamma \vdash t : T$ は「型付け文脈 Γ において項 t は型 T を持つ」と読む。空の文脈は $\emptyset$ と書かえることもあるが、通常は省略して $\vdash t : T$ と書く。また、型環境に対する , 演算子は Γ の右に新しい束縛を加えて拡張する。

新しい束縛と既に Γ に表われている束縛は混同しないように名前 x は Γ に存在しない名前から選ばれるものとする。これはアルファ変換により λ 抽象の束縛名は一貫して変更ができるため、常に満たせる。

ラムダ抽象に型を持たせる規則の一般的な形は

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2} \qquad \text{T-ABS}$$

であり、変数の型付け規則は

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{T-VAR}$$

である。$x : T \in T$ は 、Γ において x に仮定された型は T である、と読む。

最後に関数適用の型付け規則を定義する。

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \qquad \text{T-APP}$$

もし t_1 が T_{11} の引数を T_{12} の計算結果に移す関数へ評価され、t_2 が型 T_{11} の計算結果にに評価されるのであれば、t_1 を t_2 に適用した計算結果は T_{12} の型を持つ。ブール定数と条件式の型付け規則は型付き算術式と時と同様である。

最終的な純粋単純型付きラムダ計算の規則を示す。純粋とは基本型を持たないという意味であり、純粋単純型付きラムダ計算にはブールのような型を持たない。この純粋単純型付きラムダ計算でブール値を扱う場合は型環境 Γ を考慮してブール値の規則を追加すれば良い。

定義 **3.18** $\rightarrow$ (型付き) の構文

$t ::=$	項
x	変数
$\lambda x : T.t$	ラムダ抽象
$t\ t$	関数適用

$v ::=$	項
$\lambda x : T.t$	ラムダ抽象値

$T ::=$	型
$T \rightarrow T$	関数型

$\Gamma ::=$	文脈
$\emptyset$	空の文脈
$\Gamma, x : T$	項変数の束縛

定義 **3.19** $\rightarrow$ (型付き) の評価 $(t \rightarrow t')$

$$\frac{t_1 \rightarrow t_1'}{t_1\ t_2 \rightarrow t_1'\ t_2} \quad \text{E-APP1}$$

$$\frac{t_2 \rightarrow t_2'}{v_1\ t_2 \rightarrow v_1\ t_2'} \quad \text{E-APP2}$$

$$(\lambda x : T_{11}.t_{12})v_2 \rightarrow [x \mapsto v_2]t_{12} \quad \text{E-APPABS}$$

定義 **3.20** $\rightarrow$ (型付き) の型付け $(\Gamma \vdash t : T)$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{T-VAR}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.t_2 : T_1 \rightarrow T_2} \quad \text{E-ABS}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : t_{12}} \quad \text{T-APP}$$

単純型付きラムダ計算の型付け規則のインスタンスも型付き算術式のように導出木をすることで示せる。例えば $(\lambda x : Bool\ x)\ true$ が空の文脈において $Bool$ を持つことは以下の木で表せる。

$$\dfrac{\dfrac{\dfrac{\dfrac{x : Bool \in x : Bool}{x : Bool \vdash x : Bool}\ \text{T-VAR}}{\vdash \lambda x : Bool.x : Bool \rightarrow Bool}\ \text{T-ABS} \qquad \dfrac{}{\vdash true : Bool}\ \text{T-TRUE}}{\vdash (\lambda x : Bool.x)\ true : Bool}\ \text{T-APP}$$

純粋型付きラムダ計算の型システムにおいて、閉じた項に対して進行定理と保存定理は成り立つ [12] [13]。 閉じた項、という制限が付いているのは $f\ true$ といった自由変数が存在する項は正規形ではあるが値でないからである。しかし、開いた項に関しては評価が行なえないために型システムの検査対象に含まれない。

3.6　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 **3.21** レコードの拡張に用いる新しい構文形式

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

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

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

定義 **3.22** レコードの拡張に用いる新しい評価規則

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

$$\frac{t_1 \to t_1'}{t_1.l \to t_1'.l} \quad \text{E-PROJ}$$

$$\frac{t_j \to t_j'}{\{l_i = v_i^{i \in 1..j-1}, l_j = t_j, l_k = t_k^{k \in j+1..n}\} \to \{l_i = v_i^{i \in 1..j-1}, l_j = t_j', l_k = t_k^{k \in j+1..n}\}} \quad \text{E-RCD}$$

定義 **3.23** レコードの拡張に用いる新しい型付け規則

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

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

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。

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

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。

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

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

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。

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

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 $S_1 \to S_2$ の関数を別の型 $T_1 \to T_2$ 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。

3.7　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 2.9 に示していた DataSegment の一つに注目する (リスト 3.2)。

リスト 3.2: akashaContext の DataSegment である AkashaInfo

```
1 struct AkashaInfo {
2     unsigned int minHeight;
3     unsigned int maxHeight;
4     struct AkashaNode* akashaNode;
5 };
```

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 3.3) 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 3.3: CbC の Meta DataSegment である Context

```
1  struct Data { /* data segments as types */
2      struct Tree { /* ... */ } tree;
3      struct Node { /* ... */ } node;
4
5      struct IterElem { /* ..  */ } iterElem;
6      struct Iterator { /* ... */ } iterator;
7      struct AkashaInfo { /* ... */} akashaInfo;
8      struct AkashaNode { /* ... */} akashaNode;
9  };
10
11
12 struct Context { /* meta data segment as subtype */
13     /* ... */
14     struct Data **data;
15 };
```

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 **3.24** 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 **3.25** CodeSegment の定義

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

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

定義 **3.26** Meta CodeSegment の定義

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

ここで具体的なコード (リスト 3.4) と比較してみる。

リスト 3.4: 具体的な CbC における CodeSegment

```
1 __code getMinHeight_stub(struct Context* context) {
2     goto getMinHeight(context, &context->data[Allocate]->allocate, &
    context->data[AkashaInfo]->akashaInfo);
3 }
4
5 __code getMinHeight(struct Context* context, struct Allocate* allocate,
    struct AkashaInfo* akashaInfo) {
6     /* ... */
7     goto getMinHeight_stub(context);
8 }
```

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 $\{allocate : Allocate, akashaInfo : AkahsaInfo\} \to Context$ 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 $X \to X$ とする際の導出木は以下である。

$$\dfrac{\dfrac{}{X <: X}\ \text{S-REFL} \quad \dfrac{}{Conttext <: X}\ \text{S-MDS}}{X \to Context <: X \to X}\ \text{S-ARROW}$$

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。

第4章　証明支援系言語 Agda による証明手法

第 3 章では形無し算術式と型付き算術式による型システムの定義、ラムダ計算によるプログラミング言語の抽象化、部分型の導入と CbC の型が部分型で示せることを確認した。部分型を用いて具体的な CbC の型システムを定義する前に、型システムの一方のメリットである証明について触れる。依存型という型を持つ証明支援系言語 Agda を用いて証明が行なえることを示す。

4.1 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 [14]。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 ⇒ を導入する。

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

$\Rightarrow \mathcal{I}$ 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。

$$\dfrac{\begin{array}{cc}\vdots & \vdots \\ A & B\end{array}}{A \wedge B} \wedge\mathcal{I}$$

$$\dfrac{\begin{array}{c}\vdots \\ A\end{array}}{A \vee B} \vee 1\mathcal{I}$$

$$\dfrac{\begin{array}{c}\vdots \\ B\end{array}}{A \vee B} \vee 2\mathcal{I}$$

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

- 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。

$$\frac{\begin{array}{c}\vdots\\ A \wedge B\end{array}}{A} \wedge 1\mathcal{E}$$

$$\frac{\begin{array}{c}\vdots\\ A \wedge B\end{array}}{B} \wedge 2\mathcal{E}$$

$$\frac{\begin{array}{ccc} & [A] & [B] \\ \vdots & \vdots & \vdots \\ A \vee B & C & C \end{array}}{C} \vee \mathcal{E}$$

$$\frac{\begin{array}{cc} \vdots & \vdots \\ A & A \Rightarrow B \end{array}}{B} \Rightarrow \mathcal{E}$$

記号 $\vee, \wedge, \Rightarrow$ 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

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

$$\dfrac{\dfrac{[A]_{(1)} \quad \dfrac{[(A \Rightarrow B) \wedge (B \Rightarrow C)]_{(2)}}{(A \Rightarrow B)} \wedge 1\mathcal{E}}{B} \quad \dfrac{[(A \Rightarrow B) \wedge (B \Rightarrow C)]_{(2)}}{(B \Rightarrow C)} \wedge 2\mathcal{E}}{\dfrac{\dfrac{C}{A \Rightarrow C} \Rightarrow \mathcal{I}_{(1)}}{((A \Rightarrow B) \wedge (B \Rightarrow C)) \Rightarrow (A \Rightarrow C)} \Rightarrow \mathcal{I}_{(2)}}$$

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 $(A \Rightarrow B) \wedge (B \Rightarrow C)$ 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 $((A \Rightarrow B) \wedge (B \Rightarrow C)) \Rightarrow (A \Rightarrow C)$ となる。

証明の手順はこうである。まず条件 $(A \Rightarrow B) \wedge (B \Rightarrow C)$ と A の 2 つを仮定する。条件を $\wedge 1\mathcal{E}$ $\wedge 2\mathcal{E}$ 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 $(A \Rightarrow B) \wedge (B \Rightarrow C)$ 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 $((A \Rightarrow B) \wedge (B \Rightarrow C)) \Rightarrow (A \Rightarrow C)$ が導けたために証明終了となる。

4.2 Curry-Howard Isomorphism

4.1 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 4.2 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。

$$x : A$$
$$\lambda x.x : A \to A$$

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。

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

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)

4.3　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 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であり、型を引数に取る関数や値を取って型を返す関数、型の型などが記述できる。この節では Agda の文法的な紹介を行なう [**?**]。

まず Agda のプログラムは全てモジュールの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` のモジュール名定義はリスト 4.1 のようになる。

リスト 4.1: Agda のモジュールの定義する

```
1 module AgdaBasics where
```

また、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。

まず Agda におけるデータ型について触れていく。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 に似て代数的なデータ構造のパターンマッチを扱うことができ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。

リスト 4.2: Agda におけるデータ型 Bool の定義

```
1 data Bool : Set where
2   true  : Bool
3   false : Bool
```

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。

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。関数の型は単純型付き λ 計算と同様に $\rightarrow$ を用いる。なお、$\rightarrow$ に対しては糖衣構文 `->` も用意されている。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 4.3 のようになる。

リスト 4.3: Agda における関数 not の定義

```
1 not : Bool -> Bool
2 not  true = false
3 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。パターンマッチは上からマッチされていくため、優先順位が存在する。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コンストラクタ以外となる。例えばリスト 4.4 の not は x には true しか入ることは無い。

リスト 4.4: Agda におけるパターンマッチ

```
1 not : Bool -> Bool
2 not false = true
3 not x     = false
```

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。

単純型で利用した自然数もデータ型として定義できる (リスト 4.5)。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 4.5: Agda における自然数の定義

```
1 data Nat : Set where
2   zero : Nat
3   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 4.6 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演算子のように振る舞う。また、Agda は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。

リスト 4.6: Agda における自然数の加算の定義

```
1 _+_ : Nat -> Nat -> Nat
2 zero  + m = m
3 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 4.7 のように書ける。

リスト 4.7: 依存型を持つ関数の定義

```
1 identity : (A : Set) -> A -> A
2 identity A x = x
3
4 identity-zero : Nat
5 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 4.8 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 4.8: Agda における暗黙的な引数を持つ関数

```
1  id : {A : Set} -> A -> A
2  id x = x
3
4  id-zero : Nat
5  id-zero = id zero
6
7  id' : {A : Set} -> A -> A
8  id' {A} x = x
9
10 id-true : Bool
11 id-true = id {Bool} true
```

Agda には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 4.9 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 4.9: Agda におけるレコード型の定義

```
1  record Point : Set where
2    field
3      x : Nat
4      y : Nat
5
6  makePoint : Nat -> Nat -> Point
7  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 4.10 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 4.10 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 4.10 の 7,8 行目のように書ける。

リスト 4.10: 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 4.11 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 4.11: Agda における部分型制約

```
1 record Eq (A : Set) : Set where
2   field
3     _==_ : 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 4.12 のようになる。

リスト 4.12: Agda における部分型関係の構築

```
1 _==Nat_ : Nat -> Nat -> Bool
2 zero    ==Nat zero    = true
3 (suc n) ==Nat zero    = false
4 zero    ==Nat (suc m) = false
5 (suc n) ==Nat (suc m) = n ==Nat m
6
7 instance
8   natHas== : Eq Nat
9   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 4.13: Agda における部分型を使う関数の定義

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

この `elem` 関数はリスト 4.14 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。なお、 `--` の後はコメントである。

リスト 4.14: 部分型を持つ関数の適用

```
1 listHas4 : Bool
2 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 4.15 に示す。

リスト 4.15: Agda におけるモジュールのインポート

```
1 import Data.Nat                    -- import module
2 import Data.Bool as B              -- renamed module
3 import Data.List using (head)      -- import Data.head function
4 import Level renaming (suc to S) -- import module with rename suc to S
5 import Data.String hiding (_++_) -- import module without _++_
6 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 4.16 である。

リスト 4.16: Agda における Parameterized Module

```
1 module Sort (A : Set) (_<_ : A -> A -> Bool) where
2   sort : List A -> List A
3   sort = ...
4
5 open import Sort Nat  Nat._<_  as N
6 open import Sort Bool Bool._<_ as B
```

4.4 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 ($n \neq m \rightarrow Sn \neq Sm$)
- 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 4.17 のようになる。

リスト 4.17: Agda における自然数型 Nat の定義

```
1 module nat where
2
3 data Nat : Set where
4   O : Nat
5   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 4.18)。

リスト 4.18: Agda における自然数型に対する加算の定義

```
1 open import nat
2 module nat_add where
3
4 _+_ : Nat -> Nat -> Nat
5 O + m     = m
6 (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 4.19: Relation.Binary.Core による等式を示す型 ≡

```
1 data _≡_ {a} {A : Set a} (x : A) : A → Set a where
2   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 4.20)。

リスト 4.20: Agda における 3 + 1 の結果が 4 と等しい証明

```
1 open import Relation.Binary.PropositionalEquality
2 open import nat
3 open import nat_add
4
5 module three_plus_one where
6
7 3+1 : (S (S (S O)))  +  (S O) ≡  (S (S (S (S O))))
8 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 : $x \equiv y \to y \equiv x$

  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 : $f \to x \equiv y \to fx \equiv fy$

  証明した等式に同じ関数を与えても等式は保たれる。

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

  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 4.21)。

リスト 4.21: Agda における加法の交換法則の証明

```
1 open import Relation.Binary.PropositionalEquality
2 open import nat
3 open import nat_add
4 open ≡-Reasoning
5
6 module nat_add_sym where
7
```

```
8  addSym : (n m : Nat) -> n + m ≡ m + n
9  addSym O       O   = refl
10 addSym O    (S m)  = cong S (addSym O m)
11 addSym (S n)   O   = cong S (addSym n O)
12 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 ≡ ⟨*expression*⟩ 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 4.22 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 4.22: ≡ - Reasoning を用いた証明の例

```
1  open import Relation.Binary.PropositionalEquality
2  open import nat
3  open import nat_add
4  open ≡-Reasoning
5
6  module nat_add_sym_reasoning where
7
8  addToRight : (n m : Nat) → S (n + m) ≡ n + (S m)
9  addToRight O m     = refl
10 addToRight (S n) m = cong S (addToRight n m)
11
12 addSym : (n m : Nat) → n + m ≡ m + n
13 addSym O       O   = refl
14 addSym O    (S m)  = cong S (addSym O m)
15 addSym (S n)   O   = cong S (addSym n O)
16 addSym (S n) (S m) = begin
17   (S n) + (S m)  ≡⟨ refl ⟩
18   S (n + S m)    ≡⟨ cong S (addSym n (S m)) ⟩
19   S ((S m) + n)  ≡⟨ addToRight (S m) n ⟩
20   S (m + S n)    ≡⟨ refl ⟩
21   (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

CbC の項を部分型を用いて Agda 上に記述していく。DataSegment と CodeSegment の定義、CodeSegment の接続と実行、メタ計算を定義し、 Agda 上で実行できることを確認する。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 上で証明していく。

5.1　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 5.1 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 5.1: Agda における DataSegment の定義

```
1 record ds0 : Set where
2   field
3     a : Int
4     b : Int
5
6 record ds1 : Set where
7   field
8     c : Int
```

5.2　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 $I \to O$ 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 5.2 のようになる。

リスト 5.2: Agda における CodeSegment 型の定義

```
1 data CodeSegment {l1 l2 : Level} (I : Set l1) (O : Set l2) : Set (l ⊔ l1
    ⊔ l2) where
2   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 5.3 のようになる。

リスト 5.3: Agda における CodeSegment の定義

```
1  cs2 : CodeSegment ds1 ds1
2  cs2 = cs id
3
4  cs1 : CodeSegment ds1 ds1
5  cs1 = cs (\d -> goto cs2 d)
6
7  cs0 : CodeSegment ds0 ds1
8  cs0 = cs (\d -> goto cs1 (record {c = (ds0.a d) + (ds0.b d)}))
9
10 main : ds1
11 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。

5.3　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 5.4 のようになる。

リスト 5.4: Agda における goto の定義

```
1 goto : {l1 l2 : Level} {I : Set l1} {O : Set l2}
2    -> CodeSegment I O -> I -> O
3 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。

5.4　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 5.5 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 5.5: Agda における Meta DataSegment の定義

```
1 module subtype {l : Level} (Context : Set l) where
2
3 record DataSegment {ll : Level} (A : Set ll) : Set (l ⊔ ll) where
4   field
5     get : Context → A
6     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。

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

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 $S_1 \to S_2$ の関数を、部分型 $T_1 \to T_2$ 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。

5.5 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 5.6)

リスト 5.6: Agda における Meta CodeSegment の定義

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

5.6　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 5.7 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 5.7: Agda におけるメタレベル実行の定義

```
1 exec : {l1 l2 : Level} {I : Set l1} {O : Set l2}
2        {{_ : DataSegment I}} {{_ : DataSegment O}}
3        → CodeSegment I O → Context → Context
4 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 5.8 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 5.8: Agda における Meta Meta DataSegment の定義例

```
1 ...
2 open import subtype Context as N
```

```
3
4  record Meta : Set where
5    field
6      context : Context
7      c'      : Int
8      next    : N.CodeSegment Context Context
9
10 open import subtype Meta as M
11 ...
```

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 5.9 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 5.9: Agda における Meta Meta CodeSegment の定義と実行例

```
1  ...
2  -- meta level
3  liftContext : {X Y : Set} {{_ : N.DataSegment X}} {{_ : N.DataSegment Y}}
       -> N.CodeSegment X Y -> N.CodeSegment Context Context
4  liftContext {{x}} {{y}} (N.cs f) = N.cs (\c -> N.DataSegment.set y c (f (
       N.DataSegment.get x c)))
5
6  liftMeta : {X Y : Set} {{_ : M.DataSegment X}} {{_ : M.DataSegment Y}} ->
       N.CodeSegment X Y -> M.CodeSegment X Y
7  liftMeta (N.cs f) = M.cs f
8
9  gotoMeta : {I O : Set} {{_ : N.DataSegment I}} {{_ : N.DataSegment O}} ->
       M.CodeSegment Meta Meta -> N.CodeSegment I O -> Meta -> Meta
10 gotoMeta mCode code m = M.exec mCode (record m {next = (liftContext code)
       })
11
12 push : M.CodeSegment Meta Meta
13 push = M.cs (\m -> M.exec (liftMeta (Meta.next m)) (record m {c' =
       Context.c (Meta.context m)}))
14
15 -- normal level
16
17 cs2 : N.CodeSegment ds1 ds1
18 cs2 = N.cs id
19
20 cs1 : N.CodeSegment ds1 ds1
21 cs1 = N.cs (\d -> N.goto cs2 d)
22
23 cs0 : N.CodeSegment ds0 ds1
24 cs0 = N.cs (\d -> N.goto cs1 (record {c = (ds0.a d) + (ds0.b d)}))
25
```

```
26 -- meta level (with extended normal)
27 main : Meta
28 main = gotoMeta push cs0 (record {context = (record {a = 100 ; b = 50 ; c
       = 70}) ; c' = 0 ; next = (N.cs id)})
29 -- 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。

5.7　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 5.10 のようになっている。

リスト 5.10: CbC における構造体 stack の定義

```
1 struct SingleLinkedStack {
2     struct Element* top;
3 } SingleLinkedStack;
4 struct Element {
5     union Data* data;
6     struct Element* next;
7 } 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 5.11: Agda における Maybe の定義

```
1 data Maybe {a} (A : Set a) : Set a where
2   just    : (x : A) → Maybe A
3   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 5.12 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 5.12: Agda における片方向リストを用いたスタックの定義

```
1  data Element (a : Set) : Set where
2    cons : a -> Maybe (Element a) -> Element a
3
4  datum : {a : Set} -> Element a -> a
5  datum (cons a _) = a
6
7  next : {a : Set} -> Element a -> Maybe (Element a)
8  next (cons _ n) = n
9
10 record SingleLinkedStack (a : Set) : Set where
11   field
12     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 **??**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 5.13: スタックを利用するための DataSegment の定義

```
1  record Context : Set where
2    field
3      -- fields for stack
4      element : Maybe A
5
6
7  open import subtype Context as N
8
9  record Meta  : Set^^e2^^82^^81 where
10   field
11     -- context as set of data segments
12     context : Context
13     stack   : SingleLinkedStack A
14     nextCS  : N.CodeSegment Context Context
15
16 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 5.14 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 5.14: CbC における SingleLinkedStack を操作する Meta CodeSegment

```
1  __code pushSingleLinkedStack(struct SingleLinkedStack* stack,union Data*
       data, __code next(...)) {
2      Element* element = new Element();
3      element->next = stack->top;
4      element->data = data;
5      stack->top = element;
6      goto next(...);
7  }
8
9  __code popSingleLinkedStack(struct SingleLinkedStack* stack, __code next(
       union Data* data, ...)) {
10     if (stack->top) {
11         data = stack->top->data;
12         stack->top = stack->top->next;
13     } else {
14         data = NULL;
15     }
16     goto next(data, ...);
17 }
```

次に Agda における定義をリスト 5.15 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 5.15: 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。

5.8　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 SingleLinkedStack 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 **5.1** 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 5.16 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 5.16: Agda におけるスタックの性質の定義 (1)

```
1 pushOnce : Meta → Meta
2 pushOnce m = M.exec pushSingleLinkedStackCS m
3
4 popOnce : Meta → Meta
5 popOnce m = M.exec popSingleLinkedStackCS m
6
7 push-pop-type : Meta → Set1
8 push-pop-type meta =
9     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 5.17)。

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 Code-Segment 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 5.17: Agda におけるスタックの性質の証明 (1)

```
1  id-meta : ℕ → ℕ → SingleLinkedStack ℕ → Meta
2  id-meta n e s = record { context = record {n = n ; element = just e}
3                         ; nextCS = (N.cs id) ; stack = s}
4
5  push-pop-type : ℕ → ℕ  → ℕ → Element ℕ → Set1
6  push-pop-type n e x s = M.exec (M.csComp {meta} (M.cs popOnce) (M.cs
       pushOnce)) meta ≡ meta
7    where
8      meta = id-meta n e record {top = just (cons x (just s))}
9
10 push-pop : (n e x : ℕ) → (s : Element ℕ) →  push-pop-type n e x s
11 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 5.18 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 5.18: Agda におけるスタックの性質の定義 (2)

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

```
4
5  n-pop : {m : Meta} {{_ : M.DataSegment Meta}} (n : ℕ) → M.CodeSegment
       Meta Meta
6  n-pop {{mm}} (zero)      = M.cs {{mm}} {{mm}} id
7  n-pop {m} {{mm}} (suc n) = M.cs {{mm}} {{mm}} (\m → M.exec {{mm}} {{mm}}
       (n-pop {m} {{mm}} n) (popOnce m))
8
9  pop-n-push-type : ℕ → ℕ → ℕ → SingleLinkedStack ℕ → Set1
10 pop-n-push-type n cn ce s = M.exec (M.csComp {meta} (M.cs popOnce) (n-
       push {meta} (suc n))) meta
11                           ≡ M.exec (n-push {meta} n) meta
12   where
13     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 5.19 のように証明できる。

リスト 5.19: Agda におけるスタックの性質の証明 (2)

```
1  pop-n-push-type : ℕ → ℕ → ℕ → SingleLinkedStack ℕ → Set1
2  pop-n-push-type n cn ce s = M.exec (M.csComp (M.cs popOnce) (n-push (suc
       n))) meta
3                            ≡ M.exec (n-push n) meta
4    where
5      meta = id-meta cn ce s
6
7  pop-n-push : (n cn ce : ℕ) → (s : SingleLinkedStack ℕ) → pop-n-push-
       type n cn ce s
8  pop-n-push zero cn ce s    = refl
9  pop-n-push (suc n) cn ce s = begin
10   M.exec (M.csComp  (M.cs popOnce) (n-push  (suc (suc n)))) (id-meta cn
       ce s)
11   ≡⟨ refl ⟩
12   M.exec (M.csComp (M.cs popOnce) (M.csComp  (n-push  (suc n)) (M.cs
       pushOnce))) (id-meta cn ce s)
13   ≡⟨ exec-comp (M.cs popOnce) (M.csComp  (n-push  (suc n)) (M.cs pushOnce
       )) (id-meta cn ce s) ⟩
14   M.exec (M.cs popOnce) (M.exec (M.csComp  (n-push  (suc n)) (M.cs
       pushOnce)) (id-meta cn ce s))
15   ≡⟨ cong (\x → M.exec (M.cs popOnce) x) (exec-comp (n-push (suc n)) (M.
       cs pushOnce) (id-meta cn ce s)) ⟩
16   M.exec (M.cs popOnce) (M.exec (n-push  (suc n))(M.exec (M.cs pushOnce)
       (id-meta cn ce s)))
17   ≡⟨ refl ⟩
18   M.exec (M.cs popOnce) (M.exec (n-push  (suc n)) (id-meta cn ce (record
       {top = just (cons ce (SingleLinkedStack.top s))})))
19   ≡⟨ sym (exec-comp (M.cs popOnce) (n-push (suc n)) (id-meta cn ce (
       record {top = just (cons ce (SingleLinkedStack.top s))}))) ⟩
20   M.exec (M.csComp  (M.cs popOnce) (n-push  (suc n))) (id-meta cn ce (
       record {top = just (cons ce (SingleLinkedStack.top s))}))
21   ≡⟨ pop-n-push n cn ce (record {top = just (cons ce (SingleLinkedStack.
       top s))}) ⟩
```

```
22    M.exec (n-push n) (id-meta cn ce (record {top = just (cons ce (
       SingleLinkedStack.top s))}))
23    ≡⟨ refl ⟩
24    M.exec (n-push n) (pushOnce (id-meta cn ce s))
25    ≡⟨ refl ⟩
26    M.exec (n-push n) (M.exec (M.cs pushOnce) (id-meta cn ce s))
27    ≡⟨ refl ⟩
28    M.exec (n-push (suc n)) (id-meta cn ce s)
29    ■
30
31
32
33  n-push-pop-type : ℕ →  ℕ  → ℕ → SingleLinkedStack ℕ → Set1
34  n-push-pop-type n cn ce st = M.exec (M.csComp (n-pop n) (n-push n)) meta
       ≡ meta
35    where
36      meta = id-meta cn ce st
37
38  n-push-pop : (n cn ce : ℕ) → (s : SingleLinkedStack ℕ) → n-push-pop-
       type n cn ce s
39  n-push-pop zero    cn ce s = refl
40  n-push-pop (suc n) cn ce s = begin
41    M.exec (M.csComp (n-pop (suc n)) (n-push (suc n))) (id-meta cn ce s)
42    ≡⟨ refl ⟩
43    M.exec (M.csComp (M.cs (\m → M.exec (n-pop n) (popOnce m))) (n-push (
       suc n))) (id-meta cn ce s)
44    ≡⟨ exec-comp (M.cs (\m → M.exec (n-pop n) (popOnce m))) (n-push (suc n
       ))  (id-meta cn ce s) ⟩
45    M.exec (M.cs (\m → M.exec (n-pop  n) (popOnce m))) (M.exec (n-push (
       suc n)) (id-meta cn ce s))
46    ≡⟨ refl ⟩
47    M.exec (n-pop n) (popOnce (M.exec (n-push (suc n)) (id-meta cn ce s)))
48    ≡⟨ refl ⟩
49    M.exec (n-pop n) (M.exec (M.cs popOnce) (M.exec (n-push (suc n)) (id-
       meta cn ce s)))
50    ≡⟨ cong (\x → M.exec (n-pop n) x) (sym (exec-comp (M.cs popOnce) (n-
       push (suc n)) (id-meta cn ce s))) ⟩
51    M.exec (n-pop n) (M.exec (M.csComp (M.cs popOnce) (n-push (suc n))) (id
       -meta cn ce s))
52    ≡⟨ cong (\x → M.exec (n-pop n) x) (pop-n-push n cn ce s) ⟩
53    M.exec (n-pop n) (M.exec (n-push n) (id-meta cn ce s))
54    ≡⟨ sym (exec-comp (n-pop n) (n-push n) (id-meta cn ce s)) ⟩
55    M.exec (M.csComp (n-pop n) (n-push n)) (id-meta cn ce s)
56    ≡⟨ n-push-pop n cn ce s ⟩
57    id-meta cn ce s
58    ■
```

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

6.1　今後の課題

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

[13] 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.

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

[15] 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.

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

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

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

発表履歴

- 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.