2024 年度 北見工業大学 工学部
情報デザイン・コミュニケーション工学コース
学士論文

Python のパフォーマンス向上のための多言語トランスコンパイラの実装

数理波動システム研究室
著者　北原玲
指導教員　杉坂純一郎

目次

第 1 章

序論

1.1 研究背景

Python は可読性・保守性が高く、手動での編集がしやすいプログラミング言語である。しかし、その実行速度は一行ずつコードを実行するインタプリタ言語あるため、極めて低速となっている。Python でソフトウェア開発を行う場合、高速な処理が必要な場面では、実行速度の速い C 言語や Fortran といったコンパイル言語で実装されたライブラリを呼び出すことで、処理を行っている。このように、異なる言語を組み合わせて開発が行われる場合、呼び出し元のソースコードとライブラリのソースコードをそれぞれ別の言語で編集する必要がある。この時、可読性と保守性の高い単一の言語から複数の言語のコードを生成することができれば、ソフトウェア開発をより効率的に進めることができる。また、大規模な開発では、Python のようなインタプリタ言語の実行速度を向上させるために、コード生成時に最適化処理を行う必要がある。これらを実現するには、人工知能を用いたコードの自動変換 [1][2] かトランスコンパイラ [3][4][5] を使用するのが現実的である。しかし、人工知能を用いたコードの自動変換はプログラマーの意図しないコードを生成する場合があるため、本研究では、トランスコンパイラを用いることとした。

トランスコンパイラとは、あるプログラミング言語を同水準の言語に変換するソフトウェアの総称である。Python で使用されるトランスコンパイラとしては、Cython が挙げられる。Cython は、Python の拡張モジュールの作成の労力を軽減するために作られたプログラミング言語であり、所謂 Python のスーパーセットである。その処理系では、ソースコードを C や C++ に変換し、コンパイルすることで Python の拡張モジュールを生成する。このモジュールを使用することで Python コードの処理速度を上昇させることが可能になる。しかし、Python のソースコードを最適化しているわけではなく、あくまで拡張モジュールを生成しているに過ぎない。また、高速フーリエ変換に使われるライブラリ FFTW も、Cython によって生成される拡張モジュールと同じように Ocaml というプログラミング言語のソースコードを C 言語のコードに変換することで実装されている。

実行速度ではなく、可読性や保守性を上昇させるために、トランスコンパイラが用いられる場合もある。TypeScript はその典型的な例である。TypeScript は Cython と同様に JavaScript のスーパーセットである。TyepScript の文法を端的に表すと、静的型付けを行う JavaScript であり、JavaScript では不可能なクラスの作成可能であるなど、JavaScript よりも高い保守性と可読性を持っている。ただし、Cython とは違い、実行時には JavaScript に変換されてから実行されるため、プログラムの高速化には寄与しない。同じような言語としては、Erg[8] という言語が存在する。こちらは、Python のバイトコードにトランスコンパイルを行う言語であり、静的型付け言語である Erg でコーディングすることによって、動的型付け言語である

Python よりも早くバグを発見できる。しかし、最適化処理には重点を置いておらず、不要な変数の削除しか行わない。

また、多言語に対して変換可能なトランスコンパイラも存在し、その例として、Haxe[9] が挙げられる。Haxe はそのソースコードから C++、Python、JavaScript などの複数の言語への変換が可能である。その特徴を生かして、多言語によるソフトウェア開発用フレームワークである nMorph[6] にも使用されている。トランスコンパイル時に最適化を行うこともでき、極めて利便性の高いトランコンパイル言語となっている。しかし、その構文には高階関数の冗長性を無くすことや関数の部分適応を行うためのカリー化など、より可読性を上げる余地が残されている。

そこで、本研究では Aqualis[7] という多言語トランスコンパイラを使用する。Aqualis とは、保守性・可読性の高い Aqualis のソースコードから他の言語のソースコードファイルを生成するトランスコンパイラである。Aqualis は F♯[10][11] のライブラリであるが、F♯ の関数型プログラミングの特徴を活かし、疑似的なトランスコンパイル言語として機能している。現在は、C 言語、Fortran、HTML、LaTeX への変換が可能である他、図面やグラフの生成も可能となっている。また、実行速度の速いコンパイル言語である C 言語及び Fortran に変換し、プログラムを高速に実行することが可能である。更に、Aqualis ではコード生成時に最適化処理を行える。しかし、前述の通り現在の Aqualis には Python への変換機能は存在しない。

1.2　研究目的

そこで本研究の目的は、Aqualis に Python のソースコード生成機能を追加することと、実装後に Aqualis を用いて最適化処理を行った Python のコードを生成することである。Aqualis によって最適化処理を行ったコードは、同様の実行結果となる手動で生成した Python コードと実行時間の比較を行う。この時の計測時間を短縮することで目的が達成されたこととする。

1.3　本論文の構成

本論文では、2 章で F♯ の記法について簡単に述べる。3 章では、Aqualis の基本文法、特徴について述べる。4 章では、Aqualis への Python への変換機能の実装の詳細を述べる。5 章では、Aqualis で最適化処理を個なった Python コードと手書きの Python コードの実行時間を比較し、考察を行う。6 章では、本論文のまとめ、結論を述べる。

第 2 章

F♯

この章では F♯ の文法について示す。F♯ はマイクロソフトリサーチによって 2005 年に開発された言語である。関数型プログラミングを軸足としながらオブジェクト指向プログラミングや手続き型プログラミングなどの利点を持つマルチパラダイムプログラミング言語である。

2.1 関数定義

ソースコード 2.1 はどちらも引数 x に 1 を加えた値を返す関数を定義している。

ソースコード 2.1 F♯ の関数定義 1

```
1 let f x = x + 1
2 let f(x) = x + 1
```

右辺は関数 f に引数 x を適応した場合の値を示している。ソースコード 2.2

ソースコード 2.2 F♯ の関数定義 2

```
1 let f = fun x -> x + 1
```

は、関数 f そのもの定義を記述した例である。ソースコード 2.1 はソースコード 2.2 を略記したものとなっており、前述のコードと同じように、引数 x に 1 を加えた値を返す関数を定義している。また、ソースコード 2.2 の右辺はラムダ式となっており、匿名の関数オブジェクトとして定義されている。

引数に関数を受け取る高階関数を簡潔に記述することもできる。ソースコード 2.3 は高階関数を定義し、使用した例である。

ソースコード 2.3 F♯ の高階関数

```
1 let iterator (x:int) (y:int->unit) =
2     for i in 1..x do
3         y i
4
5 iterator 5 (fun i -> printfn "%d" i)
6
7 iterator 5 <| fun i ->
8     printfn "%d" i
```

このコードでは、反復処理を行う関数 “iterator” が定義されている。“x:int” は引数 x の型が整数型であることを示し、“y:int->unit” は y が整数型を受け取り、何も値を返さない関数であることを表す。5 行目

と 7・8 行目では、1 行目で定義した関数を使用している。どちらも実行結果として、“1 2 3 4 5” を画面出力する。5 行目のラムダ式 “fun i -> printfn "%d" i” は、第二引数である関数を表し、整数 i を受け取り、その値を画面出力する関数である。7・8 行目では逆順パイプライン関数 “<|” を利用して、関数を第二引数として記述している。

型推論を用いてより簡潔に高階関数を記述できる他、自動的にカリー化が行われるため、関数の部分適用を容易に行うことができる。ソースコード 2.4 は、ソースコード 2.3 の関数 iterator の型推論を用いた例とそれを部分適用した関数の例である。

ソースコード 2.4 F♯ の部分適用

```
1 let iterator x y =
2     for i in 1..x do
3         y i
4
5 let iteratorFive = iterator 5
6
7 iteratorFive <| fun i ->
8     printfn "%d" i
```

このコードにおける関数 iterator は、引数の型注釈を行わずに定義されている。また、5 行目で定義されている関数 “iteratorFive” は、iterator を部分適用し、5 回引数として渡された関数を繰り返す。

2.2 演算子の定義

F♯ では、演算子のユーザー定義や既存の演算子のオーバーロードが可能である。Aqualis で用いられている演算子は、これらの特徴が用いられている。例えば、y の値を x に代入する演算子を定義する場合、ソースコード 2.5 のようなコードになる。

ソースコード 2.5 F♯ の演算子定義

```
1 let (<==) x y =
2     //代入するコードを生成する処理
```

この演算子の定義後、

ソースコード 2.6 定義された演算子

```
1 a <== 1
```

と記述することで、変数 a に 1 を代入するコードを出力することができる。

2.3 判別共用体

判別共用体とは、F♯ のデータ型の一種である。代数的データ型とも呼ばれ、ほかのデータ型を組み合わせて形成されている。これにより、定義したケースの内いずれかのデータ型をとることができる。例えば、

ソースコード 2.7 F♯ の判別共用体

```
1 type Shape =
2     | Rectangle of width : float * length : float
```

```
3       | Circle of radius : float
4       | Prism of width : float * float * height : float
```

ソースコード 2.7 のように判別共用体 Shape を宣言することができる。この判別共用体は、“Rectangle”、“Circle”、“Prism” の三つのケースのいずれかの情報を持つ。それらには、それぞれ長方形の横と縦の長さ（float）、円の半径（float）、角柱の横と縦及び高さ（float）のフィールド（項目）を持っている。判別共用体は、数値や変数、数式コードなど様々な型を指定することが可能である。Aqualis を構成する F♯ ライブラリでは、主に型定義に用いられており、ソースコード 2.8 のような型が定義されている。

ソースコード 2.8　Aqualis を構成する判別共用体例

```
1  type bool0 =
2      |False
3      |True
4      |Eq of num0*num0
5      |NEq of num0*num0
6      |Greater of num0*num0
7      |GreaterEq of num0*num0
8      |Less of num0*num0
9      |LessEq of num0*num0
10     |AND of bool0 list
11     |OR of bool0 list
12     |Null
13
14     //num0型とbool0型を用いたメソッドの定義
15
16 and num0 =
17     |Str_c of string
18     |Int_c of int
19     |Dbl_c of double
20     |Var of Etype*string
21     |Par of Etype*string*string*num0
22     |Inv of Etype*num0
23     |Add of Etype*num0*num0
24     |Sub of Etype*num0*num0
25     |Mul of Etype*num0*num0
26     |Div of Etype*num0*num0
27     |Pow of Etype*num0*num0
28     |Exp of Etype*num0
29     |Sin of Etype*num0
30     |Cos of Etype*num0
31     |Tan of Etype*num0
32     |Asin of Etype*num0
33     |Acos of Etype*num0
34     |Atan of Etype*num0
35     |Atan2 of num0*num0
36     |Abs of Etype*num0
37     |Log of Etype*num0
38     |Log10 of Etype*num0
```

```
39      |Sqrt of Etype*num0
40      |Idx1 of Etype*string*num0
41      |Idx2 of Etype*string*num0*num0
42      |Idx3 of Etype*string*num0*num0*num0
43      |Formula of Etype*string
44      |Sum of Etype*num0*num0*(num0->num0)
45      |Let of Etype*num0*num0
46      |NaN
47
48      //num0型を用いたメソッドの定義
```

上記のコードでは、論理演算用のデータ型“bool0”と変数及び算術演算用のデータ型“num0”が定義されている。num0 型の定義で使用されている“Etype”は、値の型を格納している判別共用体である。F♯では、基本的に定義されたメソッドなどは自身よりも上のコード上では使用できない。そこで、このコードでは“and”を使うことで bool 型と num0 型を同時に定義している。

bool0 型は、False(偽)、True(真)、Eq(等しい)、NEq(等しくない)、Greater(より大きい)、GreaterEq(以上)、Less(より小さい)、LessEq(以下)、AND(論理積)、OR(論理和)、Null(空値) の 7 つのケースを持つ。False、True、Null はフィールドを持たず、Eq、NEq、Greater、GreaterEq、Less、LessEq は左辺と右辺の式 (num0) をフィールドとして持つ。また、AND、OR は論理型 (bool0) のリストをフィールドとして持っている。

判別共用体 num0 は、30 個のケース識別子を持っている。17 行目から 19 行目の Str_c、Int_c、Dbl_c は、上から文字列型、整数型、実数型の値を格納する。20 行目の Var は変数のことであり、値の型と変数名がフィールドとして格納されている。21 行目の Par は、丸括弧に囲まれた式を示し、式の型と丸括弧を格納する二つの文字列、num0 型の式を持っている。22 行目の Inv は、符号の反転を行うケースを示し、対象の値の型と式 (num0) をフィールドとして格納している。23 行目から 26 行目の Add、Sub、Mull、Div は四則演算のケースであり、それぞれ解の型と被演算子 (num0) を持っている。27 行目から 39 行目及び 44 行目のケースは数学関数のデータ型であり、上から Pow(累乗関数)、Exp(指数関数)、Sin(正弦関数)、Cos(余弦関数)、Tan(正接関数)、Asin(逆正弦関数)、Acos(逆余弦関数)、Atan(逆正接関数)、Atan2(二つの引数をとる逆正接関数)、Abs(絶対値)、Log(自然対数)、Log10(常用対数)、Sqrt(平方根)、Sum(総和) が定義されている。これらは、基本的にフィールドとして解の型と引数 (num0) を持っている。40 行目から 42 行目は配列のデータ型が定義されており、Idx1(一次元配列)、Idx2(二次元配列)、Idx3(三次元配列) となっている。それぞれの配列の型、変数名、配列のインデックスがフィールドとして定義されている。43 行目の Formula は、文字列化した式を持つケースであり、文字列化した式の値の型と文字列化した式をフィールドとして定義している。44 行目の Let は、数式の結果を保存するためのデータ型であり、結果の型と保存先の変数 (num0)、結果を出力する演算式 (num0) を持つ。45 行目の NaN は非数を表しており、フィールドは持たない。

2.4 パターンマッチ

パターンマッチとは、ある入力に対してパターンが一致するかどうかを調べ、一致していたパターンに該当する結果式を実行する分岐処理のことである。本項では、F♯におけるパターンマッチ方法の内、Aqualis を実装するうえで頻繁に使用されている match 式による識別子パターンについて解説を行う。match 式は、パターンマッチのための式であり、if 式の代用として非常によく使われている。識別子パターンは、前項で

紹介した判別共用体のケースや例外識別子、アクティブパターンなどのパターンマッチが有効な識別子を持つ値に使用可能である。また、パターンマッチに有効な識別子は、二文字以上かつ大文字で始まるものに限定される。ソースコード 2.7 の判別共用体 Shape を用いた match 式は、ソースコード 2.8 のようになる。

ソースコード 2.9 F♯ のパターンマッチ式

```
1 let matchShape shape =
2     match shape with
3     | Rectangle(height = h; width = w) ->
4         printfn $"Rectangle with height %f{h} and width %f{w}"
5     | Circle(r) -> printfn $"Circle with radius %f{r}"
6     |_ -> ()
```

上記のコードでは、渡された引数に対してパターンマッチを行う関数“matchShape”を宣言している。名前付きフィールドがある判別共用体の場合、その値を抽出するために等号によって値を入れる変数を指定する必要がある。そのため、3 行目のパターンマッチでは、Rectangle の持つフィールド“height”と“width”の値がそれぞれ変数 h、w に指定されている。また、複数のフィールドを指定する際は、セミコロンを用いてフィールドを区切る。5 行目の Circle のパターンマッチでは、フィールドが一つだけであるから、名前付きであってもフィールド名を省略することができる。6 行目では、Rectangle 及び Circle 以外のパターンの処理が記述されている。この際に使用されているパターン記号“_”は、ワイルドカードパターンを意味している。

第 3 章

Aqualis

この章では、Aqualis の基本的な文法、よく使うメソッドを紹介を行う。

3.1 変数宣言

変数宣言ではラムダ式を用いてソースコード 3.1 のように記述する。

ソースコード 3.1 Aqualis の変数宣言 1

```
1 ch.i <| fun x ->
2     x <== 1
3     print.c x
```

これは整数型の変数 x を宣言したときの例である。1 行目の “fun x ->” 以降の式がラムダ式となっている。“ch.i” とあるが、これは変数の生成と使用に関するクラス “ch” に属する整数型変数の宣言メソッド “i” である。他には、実数（倍精度浮動小数点）を宣言する “d”、複素数型を宣言する “z” メソッドなどが存在している。この例では “x <== 1” で x に 1 を代入し、“print.c x” で変数 x の画面出力をしている。x の使用範囲はインデントで指定できる。上記のコードでは 2 行目から 3 行目が x の使用可能な範囲である。この性質は、プログラムのメモリ消費を抑えることに寄与している。

ソースコード 3.2 Aqualis の変数宣言 2

```
1 ch.i <| fun x ->
2     x <== 1
3     print.c x
4 ch.i <| fun y ->
5     y <== 2
6     print.c y
```

ソースコード 3.2 は、使用可能範囲の異なる二つの変数を宣言する例である。Aqualis はこのコードを読み込み、ソースコード 3.3 のような一つの変数のみを宣言し、使用するコードを生成する。

ソースコード 3.3 Aqualis で生成された C 言語コード 1

```
1 int i0001;
2 int main
3 {
4     i0001 = 1;
```

```
5       printf("%8d\n",i0001);
6       i0001 = 2;
7       printf("%8d\n",i0001);
8   }
```

1 行目の命令文“int i0001;”は、整数型の変数 i0001 を宣言している。3 行目・5 行目の“printf("%8d\n",i0001);”は、i0001 を画面出力する命令文である。上記のコードにおいて、変数宣言は 1 行目のみであり、変数 i0001 が使いまわされている。

ソースコード 3.3 で用いられている変数名“i0001”は、Aqualis によって自動的に割り振られているものである。同じ型で使用範囲が重なる変数がある場合は、末尾の数字が i0002、i0003、i0004…のように増えていく。また、使用範囲を気にせず使用できる変数を生成したい場合は、var クラスのメソッドを利用する。つまり、これによって宣言される変数は Aqualis のメインコード及び関数コード内でグローバル変数のように機能する。ただし、インライン化しない関数の定義内で参照することはできないので注意が必要である。

ソースコード 3.4 Aqualis のグローバル変数

```
1   ch.i <| fun x ->
2       x <== 1
3       print.c x
4   let a = var.i0("a")
5   a <== 1
6   print.c a
```

ソースコード 3.4 は、変数 x の使用可能範囲外で、var.i0(…) によって整数型変数 a が宣言されている。引数として受け取っている文字列"a" は、生成されたソースコード上での変数名となる。ソースコード 3.5 に実際に生成される C 言語のコードを示す。

ソースコード 3.5 Aqualis で生成された C 言語コード 2

```
1   int a;
2   int i0001;
3   int main{
4       i0001 = 1;
5       printf("%8d\n",i0001);
6       a = 1;
7       printf("%8d\n",i0001);
8   }
```

上記のコードでは、ソースコード 3.3 とは違い、変数 i0001 以外にも変数 a が宣言されている。

また、配列を持つ変数の場合はソースコード 3.6 のように記述する。

ソースコード 3.6 Aqualis の一次元配列生成

```
1   ch.i1 2 <| fun n1 ->
2       n1 <== 0
```

これは要素数 2 の整数型の一次元配列 n1 を宣言したときの例である。“ch.i1”とあるが、整数型の二次元配列の場合は“ch.i2”、三次元配列の場合は“ch.i3”に変わる。この例では、“n1 <== 1”で配列の要素全てに 1 を代入している。このコードによって生成される C 言語のソースコードは 3.7 のようになる。

ソースコード 3.7 Aqualis で生成された C 言語コード 3

```
1  int ic0001;
2  int *i10001;
3  int i10001_size[1]={ -1 };
4  int main()
5  {
6      i10001_size[0] = 2;
7      i10001=(int *)malloc(sizeof(int)*i10001_size[0]);
8      for(ic0001=1; ic0001<=i10001_size[0]; ic0001++)
9      {
10         i10001[ic0001-1] = 0;
11     }
12     i10001_size[0] = -1;
13     free(i10001);
14     return 0;
15 }
```

1 行目で宣言されている変数“ic0001”は、8 行目から 11 行目にかけて行われている配列の全要素に 0 を代入するための反復処理で用いられるカウンタ変数となっている。2 行目の“int *i10001”は、動的メモリ確保のできる整数型のポインタ変数 i10001 を宣言する命令文である。この i10001 が配列を格納する変数となる。2 行目の“int i10001_size[1]= -1 ;”は、-1 で初期化された一つの要素を持つ整数型の一次元配列 i10001_size を宣言する命令文である。この i10001_size は、i10001 の要素数を格納するための配列となっている。Aqualis で配列を宣言するときは必ず動的メモリ確保のできる変数と配列の要素を格納する配列がトランスコンパイル後のソースコードに生成される。6 行目で用いられている“(int *)malloc(…)”は、(…) で指定されたサイズのメモリ配列を整数型として返す関数である。7 行目以降は、配列使用後として値の初期化が行われている。8 行目の命令文 free(i10001); はメモリの解放を行う命令文である。この命令文も自動生成されるため、メモリの解放忘れを防止できる。

3.2 四則演算

Aqualis の四則演算は、他のプログラミング言語と同じような文法で書くことができる。

ソースコード 3.8 Aqualis の四則演算

```
1 z <== x + y
2 z <== x - y
3 z <== x * y
4 z <== x / y
```

代入の演算子として“<==”が定義され、使用されている。

Aqualis では、第 2 章で説明した通り判別共同体 num0 が演算に用いられている。そのため、Aqualis の演算は、演算子をオーバーロードして num0 型に変換して行われる。これによって生成された num0 型変数をパータンマッチによって、各言語の対応するソースコードに変換している。

3.3 条件分岐

条件分岐にはソースコード 3.9 のようなコードを用いる。

ソースコード 3.9 Aqualis の条件分岐 1

```
1 br.if1 (z .= x+y) <| fun ()->
2     print.t "z = x+y"
```

“z .= x+y” は条件式であり、“.=” は左辺と右辺が等しいかを確認する論理演算子である。これが真であるとき、インデントで区切られた内部処理を実行する。上記のコードでは、文字列を画面出力するメソッド “print.t” によって、文字列 “z = x+y” が出力される。条件が偽となった時の処理を記述したい場合は、ソースコード 3.10 のように” br.if2” を使用する。

ソースコード 3.10 Aqualis の条件分岐 2

```
1 br.if2 (z .>= x+y)
2     <| fun ()->
3       print.t "z >= x+y"
4     <| fun ()->
5       print.t "z <= x+y"
```

“.>=” は左辺が右辺以上であるかを確認する論理演算子である。条件式 “z .>= x+y” を満たしている場合、文字列 “z >= x+y” が出力され、満たしていなければ “z <= x+y” が出力される。条件式を複数追加するときはソースコード 3.11 のように記述する。

ソースコード 3.11 Aqualis の条件分岐 3

```
1 br.branch <| fun b->
2     b.IF (z .> x+y) <| fun ()->
3         print.t "z > x+y"
4     b.IF (z .= x+y) <| fun ()->
5         print.t "z = x+y"
6     b.EL <| fun ()->
7         print.t "z < x+y"
```

z>(x+y) を満たす場合に文字列 “z > x+y” を出力、z>(x+y) を満たさず z=x+y を満たす場合に “z = x+y” を出力、どちらの条件も満たさない場合に “z < x+y” を出力する。

3.4 反復処理

反復処理を行うソースコードの例を 3.12 に示す。

ソースコード 3.12 Aqualis の範囲指定された反復処理 1

```
1 iter.range-5 5 <| fun i->
2     print.c i
```

この例では、反復処理に関するクラス “iter” の指定された範囲でループを行うメソッド “range” が用いられている。1 行目で定義されたカウンタ変数 i の値は引数-5 から 5 までの範囲を 1 ずつ変化していく。こ

れを実行すると、-5 から 5 までの 11 個の整数が出力される。カウンタ変数が 1 から始まるループの場合、ソースコード 3.13 のような記述もできる。

ソースコード 3.13 Aqualis の範囲指定された反復処理 2

```
1 iter.range 10 <| fun i->
2     print.c i
```

この例では、実行すると 1 から 10 までの 10 個の整数が出力されることになる。また、無限ループはソースコード 3.14 のようなコードで生成することができる。

ソースコード 3.14 Aqualis の無限ループ

```
1 iter.loop <| fun (ex,i) ->
2     print.c i
3     br.if1 (i.>=100) <| fun () ->
4         ex() //ここでループ脱出
```

無限ループのメソッド loop は、カウンタ変数と脱出関数を 1 行目で定義する。脱出関数は条件式と組み合わせて使用する。この例では、カウンタ変数が 100 以上になるまでカウンタ変数を画面出力し、無限ループから脱出する。

3.5 関数

Aqualis では、関数を F♯ と同じように定義できる。そのため、カリー化は既に実装されており、ソースコード 3.15 のように関数の部分適用を行うことができる。

ソースコード 3.15 Aqualis の関数定義

```
1 ch.i <| fun x ->
2     let multiply a b = a * b
3     let twice = multiply 2
4     x <== twice 10
5     print.c x
```

上記のコードでは、乗算を行う関数 “multiply” とそれを部分適用して引数の二倍の値を返す関数 “twice” が定義されている。また、Aqualis では最適化処理として関数のインライン展開が行われている。そのため、生成されるコードでは関数は生成されずに、呼び出し元に関数の内部コードが展開されることになる。よって、ソースコード 3.15 を C 言語に変換するとソースコード 3.16 のようになる。

ソースコード 3.16 Aqualis で生成された C 言語コード 4

```
1 int i0001;
2 int main()
3 {
4     i0001 = 20;
5     printf("%8d\n",i0001);
6     return 0;
7 }
```

上記のコードにおいて、関数 multiply と double は定義されていない。そして、変数 x にあたる i0001 に “double 10” がインライン展開された結果、その解である “20” が代入されている。

インライン展開を行わない関数を定義したい場合は“func”メソッドを用いる。func は、Aqualis でソースコード 3.17 のように用いられる。

ソースコード 3.17 Aqualis の非インライン展開関数

```
let f(z:num1,x:num0,y:num0) =
    func "func1" <| fun () ->
        z.farg <| fun z ->
        x.farg <| fun x ->
        y.farg <| fun y ->
            x <== 2
            z[2] <== x + y
ch.id <| fun (x,y) ->
ch.d1 2 <| fun z1 ->
    x <== 1
    y <== 2
    z1 <== 0
    f(z1,x,y)
    print.cccc x y z1[1] z1[2]
```

上記のコードでは、一つの配列と二つの変数を引数として受け取る関数 f が宣言され、関数内で更新した変数と配列の要素を画面出力している。func は 1 行目で宣言された関数 f の中で用いられ、3 行目から 5 行目で“.farg”を用いて関数 f で指定された引数の関数内での名前、仮引数名を定義している。Aqualis では、func を使用する場合、原則として“.farg”による仮引数名の定義を行わなければならない。また、Aqualis の非インライン展開の関数は引数を参照渡しとして受け取る。実際にソースコード 3.17 から生成した C 言語のコードをソースコード 3.18 に示す。

ソースコード 3.18 Aqualis で生成された C 言語コード 5

```
int ic0001;
int i0001;
double d0001;
double *d10001;
int d10001_size[1]={-1};
/*============================================
    ============================================*/
/* Subroutine name: func1 */
/* arg01 */
/* arg01_size */
/* arg03 */
/* arg04 */
/*============================================
    ============================================*/
void func1(double *arg01, int *arg01_size, int *arg03, double *arg04)
{
  (*arg03) = 2;
  arg01[1] = (*arg03)+(*arg04);
}

int main()
```

```
22 {
23   d10001_size[0] = 2;
24   d10001=(double *)malloc(sizeof(double)*d10001_size[0]);
25   i0001 = 1;
26   d0001 = 2;
27   for(ic0001=1; ic0001<=d10001_size[0]; ic0001++)
28   {
29     d10001[ic0001-1] = 0;
30   }
31   func1(d10001, d10001_size, &i0001, &d0001);
32   printf("%8d%27.17e%27.17e%27.17e\n",i0001,d0001,d10001[0],d10001[1]);
33   d10001_size[0] = -1;
34   free(d10001);
35   return 0;
36 }
```

コードの 15 行目から 19 行目で、引数として二つの実数型の変数ポインタと二つの整数型の変数ポインタを受け取り、戻り値を持たない関数 func1 が生成されている。6 行目から 14 行目は、宣言される関数の関数名と仮引数名をコメント文で表している。31 行目で実引数として配列である d10001 及び d10001_size と変数のポインタ&i0001 及び&d0001 が渡されている。配列の実引数がポインタ型ではないのは、関数に渡す際に C 言語の配列は自動的に配列のポインタが渡されるためである。

3.6 クラス (構造体)

クラスの定義はソースコード 3.19 のように記述する。

ソースコード 3.19 Aqualis のクラス定義

```
1  type Class1(sname_,name) =
2      static member sname = "Class1"
3      new(name) =
4          str.reg(Class1.sname,name)
5          testClass1(Class1.sname,name)
6      member public __.x1   = str.i0(sname_,name,"x1")
7      member public __.y1   = str.d0(sname_,name,"y1")
8
9  type Class1_1(sname_,name,size1) =
10     inherit base1(Structure(Class1.sname),Var1(size1,name))
11     new(name,size1) =
12         str.reg(Class1.sname,name,size1)
13         testClass1_1(Class1.sname,name,A1(size1))
14     new(name) = testClass1_1(name,0)
15     member this.Item with get(i:num0) = Class1(sname_,this.Idx1(i).code)
16     member this.Item with get(i:int ) = Class1(sname_,this.Idx1(i).code)
17     member public this.allocate(n:num0) = this.allocate(n)
18     member public this.allocate(n:int) = this.allocate(n.I)
```

これは整数型と実数型のフィールドを持つ“Class1”と Class1 を要素の型として持つ一次元配列を生成する“Class1_1”を定義した例である。Class1_1 は、10 行目の“inherit base1(…)”で Class1 の軽傷

を行っている。“static member sname = "Class1"” はトランスコンパイル後のクラス名を Class1 に指定している。3 行目、11 行目、14 行目に登場する “new(…)” は、そのクラスがどのような値を受け取り、その結果どのような処理を行うかを定義するのに用いられている。4 行目、12 行目で使用されている “str.reg(…)” は、トランスコンパイル後に生成するクラスとクラスのフィールドのリスト登録を行っている。この際にクラスを構成する要素がメソッドのみであった場合、リストに登録されない。つまり、変換後のソースコードではフィールドを持ったクラス以外は定義されないことになる。6 行目、7 行目では “str.i0(…)” と “str.d0(…)” によって、それぞれ整数型のフィールド “x1” と実数型のフィールド “y1” が定義されている。なお、6 行目、7 行目、17 行目、18 行目に登場している “member public …” が対象のクラスのメソッド及びフィールドにアクセスする方法の定義に用いられている。6 行目、7 行目で定義されたメソッドにより、“変数名. フィールド名” で各フィールドにアクセスできる様になっている。15 行目、16 行目の “member this.Item with get(…)” は、配列のインデクサーを定義してる。これによって、クラス型の配列の要素の取得を行えるようになっている。17 行目、18 行目で用いられている “this.allocate(…)” は、配列のメモリを確保するメソッドである。Class1_1 型の配列を生成する際に要素数が指定されなかった場合、後から要素数を指定する際に使用される。

前述した通り、Aqualis ではフィールドのみが変換後のクラスの内部に記述される。記述されなかったメソッドはメソッドが呼び出された際に、呼び出し元にメソッドの内部コードのみが展開されて実装される。その為、ソースコード 3.19 から C 言語のコードを生成した場合、フィールドのみがクラスとして定義され、ソースコード 3.20 のようになる。

ソースコード 3.20 Aqualis で生成された C 言語コード 6

```
1  typedef struct _Class1
2  {
3      int x1;
4      double y1;
5  } Class1;
```

クラス Class1 が定義され、整数型のフィールド x1 と実数型のフィールド y1 だけがクラスの内部に記述されている。Aqualis においてクラスはソースコード 3.21 のように使用される。

ソースコード 3.21 Aqualis のクラス使用例

```
1  let cc = Class1("c")
2  cc.x1 <== 1
3  cc.y1 <== 2.0
4  print.c cc.x1
5  print.c cc.y1
6  let dd = Class1_1("d")
7  dd.allocate(4)
8  dd.foreach <| fun i ->
9      dd[i].x1 <== 1
10     dd[i].y1 <== 2.0
11     print.cc dd[i].x1 dd[i].y1
```

上記のコードは、クラス型の変数と配列を生成し、各変数と要素のフィールドにそれぞれ 1 と 2.0 を代入後、それを画面出力するプログラムとなっている。1 行目の “let cc = Class1("c")” でクラス型の変数 cc を、6 行目の “let dd = Class1_1("d")” でクラスを要素に持つ配列 dd を生成している。各クラスに

引数として渡された“"c"”及び“"d"”はトランスコンパイル後の変数名である。ただし、dd の生成時に配列の要素数が渡されなかったため、7 行目の“dd.allocate(…)”で動的メモリ確保を行っている。また、8 行目で用いられているメソッド“dd.foreach”は、配列 dd の全要素に対して操作を行うメソッドであり、dd の要素数分だけ反復処理を行う。

ソースコード 3.21 から生成される C 言語のコードはソースコード 3.22 のようになる。

ソースコード 3.22 Aqualis で生成された C 言語コード 7

```
1  Class1 *d;
2  int d_size[1]={ -1 };
3  Class1 c;
4  int ic0001;
5  int main()
6  {
7      c.x1 = 1;
8      c.y1 = 2.0E0;
9      printf("%8d\n",c.x1);
10     printf("%27.17e\n",c.y1);
11     d_size[0] = 4;
12     d=(Class1 *)malloc(sizeof(Class1)*d_size[0]);
13     for(ic0001=1; ic0001<=d_size[0]; ic0001++)
14     {
15         d[ic0001-1].x1 = 1;
16         d[ic0001-1].y1 = 2.0E0;
17         printf("%8d%27.17e\n",d[ic0001-1].x1,d[ic0001-1].y1);
18     }
19     return 0;
20 }
```

1 行目から 4 行目では、使用するクラス型のポインタ変数*d、d の要素数を格納する配列 d_size、クラス型の変数 c、カウンタ変数 ic0001 がそれぞれ宣言されている。1 行目、2 行目、11 行目、12 行目の命令文及び for 文中のフィールドへのアクセスを行うコードは、Class1_1 の内部コードによって生成されたコードとなっている。

第 4 章

Python への変換機能の実装

この章では、Aqualis に Python への変換機能を実装する際に、既に実装されている C 言語や Fortran の生成コードから関数名や記号を置き換えただけでは実装できなかった機能について述べる。また、この章と関連性の薄い LaTeX 及び HTML についてのソースコードは省略する。

4.1 整数同士の除算において剰余を無視する除算演算子

4.1.1 剰余無視の除算演算子

整数同士の除算において、小数点を切り捨てた値を得たいとき、Aqualis では、ユーザー演算子である“./”が用いられる。“./”がオーバーロードされると、除算のデータ型である Div に変換される。また、通常の除算演算子“/”もオーバーロードによって、Div 型に変換される。これは、C 言語と Fortran が静的型付け言語であることに起因する。静的型付け言語では、整数同士の除算の商は整数型になるため、小数点の切り捨てが自動的に行われるからである。その為、num0 から変換されて生成されるソースコードでは、整数同士の除算を行うだけで“./”と同じ結果が得られる。対して、除算演算子“/”を用いて整数同士の除算を行う場合、データ損失を防ぐために演算子のオーバーロード時に被演算子が実数型に変換され、計算が行われる。つまり、変換後における演算子“/”と“./”の違いは、被演算子のデータ型であり、使用する演算子は等しいことになる。例として演算子“/”と“./”を用いる Aqualis のコードをソースコード 4.1 に示す。

ソースコード 4.1　Aqualis の整数同士の除算

```
1 ch.diii <| fun (z1,z2,x,y)->
2     x <== 4
3     y <== 2
4     z1 <== x / y
5     z2 <== x ./ y
```

上記のコードを C 言語に変換すると、

ソースコード 4.2　Aqualis で生成した C 言語 8

```
1 double d0001
2 int i0002
3 int i0003
4 int i0004
5 int main()
```

```
6  {
7      i0003 = 4
8      i0004 = 2
9      d0001 = (double)(i0003)/(double)(i0004)
10     i0002 = i0003/i0004
11 }
```

のようなソースコード 4.2 が生成される。コードの 9 行目“d0001 = (double)(i0003)/(double)(i0004)”は、ソースコード 4.1 の 4 行目に対応し、被演算子が実数型に変換されて計算されているのがわかる。同じように、10 行目の“i0002 = i0003/i0004”は、ソースコード 4.1 の 5 行目に対応している。この命令文が演算子“./”を用いて生成されたものであり、使用している演算子が 9 行目と同じ除算演算子“/”となっている。

この演算子“./”の機能を Python で再現するにあたり、問題になったのは、Python が動的型付け言語である点である。動的型付け言語で整数同士の除算を行った場合、商の型は自動的に実数型が割り当てられてしまう。その為、Python で小数点を切り捨てる除算を行う場合、専用の演算子“//”を用いなくてはならない。したがって、演算子“./”用に判別共用体 num0 にデータ型を追加するか、num0 を対応する別言語のソースコードに変換するメソッドの Div に関する分岐を追加する必要が生じた。そこで本研究では、num0 型変数を各言語のコードに変換するためのパターンマッチ関数“this.code”の C 言語及び Fortoran の構成を参考にしつつ、Div に関する分岐に変更を加えることにした。

4.1.2 Fortran 及び C 言語への変換コード

任意の num0 型の変数“this”を別言語のソースコードに変換するメソッド“this.code”の定義コードはソースコード 4.3 の通りである。

ソースコード 4.3 this.code の定義コード

```
1  member this.code with get() =
2      match p.lang with
3      |LaTeX ->
4          ...
5      |HTML ->
6          ...
7      |Fortran ->
8          match this with
9          |Var(_,x) -> x
10         |Int_c x -> p.ItoS(x)
11         |Dbl_c x -> p.DtoS(x)
12         |Str_c x -> "\""+x+"\""
13         |Par(_,_,_,x) -> "("+x.code+")"
14         |Inv(_,x) -> "-"+x.code
15         |Add (_,x,y) -> x.code+"+"+y.code
16         |Sub (_,x,y) -> x.code+"-"+y.code
17         |Mul (_,x,y) -> x.code+"*"+y.code
18         |Div (_,x,y) -> x.code+"/"+y.code
19         |Pow (_,x,y) -> x.code+"**("+y.code+")"
20         |Exp(_,x) -> "exp("+x.code+")"
```

```
21          |Sin(_,x) -> "sin("+x.code+")"
22          |Cos(_,x) -> "cos("+x.code+")"
23          |Tan(_,x) -> "tan("+x.code+")"
24          |Asin(_,x) -> "asin("+x.code+")"
25          |Acos(_,x) -> "acos("+x.code+")"
26          |Atan(_,x) -> "atan("+x.code+")"
27          |Atan2 (x,y) -> "atan2("+x.code+","+y.code+")"
28          |Abs(_,x) -> "abs("+x.code+")"
29          |Log(_,x) -> "log("+x.code+")"
30          |Log10(_,x) -> "log10("+x.code+")"
31          |Sqrt(_,x) -> "sqrt("+x.code+")"
32          |Idx1(_,u,n1) -> u+"("+n1.code+")"
33          |Idx2(_,u,n1,n2) -> u+"("+n1.code+","+n2.code+")"
34          |Idx3(_,u,n1,n2,n3) -> u+"("+n1.code+","+n2.code+","+n3.code+")"
35          |Formula(_,s) -> s
36          |Sum(t,n1,n2,f) ->
37              let g = num0.ch t
38              g.clear()
39              num0.looprange n1 n2 <| fun n ->
40                  g <== g + (f n)
41              g.code
42          |Let(_,v,_) -> v.code
43          |NaN -> "NaN"
44      |C99 ->
45          match this with
46          |Var(_,x) -> x
47          |Int_c x -> p.ItoS(x)
48          |Dbl_c x -> p.DtoS(x)
49          |Str_c x -> "\""+x+"\""
50          |Par(_,_,_,x) -> "("+x.code+")"
51          |Inv(_,x) -> "-"+x.code
52          |Add (_,x,y) -> x.code+"+"+y.code
53          |Sub (_,x,y) -> x.code+"-"+y.code
54          |Mul (_,x,y) -> x.code+"*"+y.code
55          |Div (_,x,y) -> x.code+"/"+y.code
56          |Pow (Zt,x,y) -> "cpow("+x.code+","+y.code+")"
57          |Exp(Zt,x) -> "cexp("+x.code+")"
58          |Sin(Zt,x) -> "csin("+x.code+")"
59          |Cos(Zt,x) -> "ccos("+x.code+")"
60          |Tan(Zt,x) -> "ctan("+x.code+")"
61          |Asin(Zt,x) -> "casin("+x.code+")"
62          |Acos(Zt,x) -> "cacos("+x.code+")"
63          |Atan(Zt,x) -> "catan("+x.code+")"
64          |Pow (_,x,y) -> "pow("+x.code+","+y.code+")"
65          |Exp(_,x) -> "exp("+x.code+")"
66          |Sin(_,x) -> "sin("+x.code+")"
67          |Cos(_,x) -> "cos("+x.code+")"
68          |Tan(_,x) -> "tan("+x.code+")"
69          |Asin(_,x) -> "asin("+x.code+")"
```

```
70          |Acos(_,x) -> "acos("+x.code+")"
71          |Atan(_,x) -> "atan("+x.code+")"
72          |Atan2 (x,y) -> "atan2("+x.code+","+y.code+")"
73          |Abs(_,x) ->
74              match x.etype with
75              |Zt ->  "cabs("+x.code+")"
76              |Dt ->  "fabs("+x.code+")"
77              |_  ->  "abs("+x.code+")"
78          |Log(Zt,x) -> "clog("+x.code+")"
79          |Log(_,x) -> "log("+x.code+")"
80          |Log10(Zt,x) -> "clog10("+x.code+")"
81          |Log10(_,x) -> "log10("+x.code+")"
82          |Sqrt(Zt,x) -> "csqrt("+x.code+")"
83          |Sqrt(_,x) -> "sqrt("+x.code+")"
84          |Idx1(_,u,n1) -> u+"["+(n1-Int_c 1).code+"]"
85          |Idx2(_,u,n1,n2) ->
86              let size1 = Var(It 4,u+"_size[0]")
87              u+"["+(((n2-Int_c 1)*size1+(n1-Int_c 1))).code+"]"
88          |Idx3(_,u,n1,n2,n3) ->
89              let size1 = Var(It 4,u+"_size[0]")
90              let size2 = Var(It 4,u+"_size[1]")
91              u+"["+((n3-Int_c 1)*size1*size2+
92                     (n2-Int_c 1)*size1+(n1-Int_c 1)).code+"]"
93          |Formula(_,s) -> s
94          |Sum(t,n1,n2,f) ->
95              let g = num0.ch t
96              g.clear()
97              num0.looprange n1 n2 <| fun n ->
98                  g <== g + (f n)
99              g.code
100         |Let(_,v,_) -> v.code
101         |NaN -> "NaN"
```

1 行目の “with get()” はこのメソッドが読み取り専用であることを示している。2 行目の “match p.lang with …” は、出力するソースファイルの言語の場合分けを表している。7 行目の “|Fortran ->” 以降の命令はは Fortran のソースファイルを出力する場合、44 行目の “|C99->” 以降の命令は C 言語のソースファイルを出力する場合を表している。8 行目、45 行目で登場する “match this with …” は、num0 型変数のデータ型の場合分けを表している。C 言語のソースファイルを出力する場合の定義コードにおいて、解が複素数 (Zt) 型である時のパターン分岐が 56 行目から 63 行目、78 行目、80 行目などに記述されおり、Python のソースコードに変換する定義コードを作成する際に特に参考になった。

4.1.3 Python への変換コード

追加した Python のソースコードに変換する定義コードをソースコード 4.4 に示す。

ソースコード 4.4 this.code に追加した定義コード

```
1 |Python ->
2     match this with
```

```
3       |Var(_,x) -> x
4       |Int_c x -> p.ItoS(x)
5       |Dbl_c x -> p.DtoS(x)
6       |Str_c x -> "\""+x+"\""
7       |Par(_,_,_,x) -> "("+x.code+")"
8       |Inv(_,x) -> "-"+x.code
9       |Add (_,x,y) -> x.code+"+"+y.code
10      |Sub (_,x,y) -> x.code+"-"+y.code
11      |Mul (_,x,y) -> x.code+"*"+y.code
12      |Div (It _,x,y) -> x.code+"//"+y.code
13      |Div (_,x,y) -> x.code+"/"+y.code
14      |Pow (_,x,y) -> "math.pow("+x.code+","+y.code+")"
15      |Exp(Zt,x) -> "cmath.exp("+x.code+")"
16      |Sin(Zt,x) -> "cmath.sin("+x.code+")"
17      |Cos(Zt,x) -> "cmath.cos("+x.code+")"
18      |Tan(Zt,x) -> "cmath.tan("+x.code+")"
19      |Asin(Zt,x) -> "cmath.asin("+x.code+")"
20      |Acos(Zt,x) -> "cmath.acos("+x.code+")"
21      |Atan(Zt,x) -> "cmath.atan("+x.code+")"
22      |Exp(_,x) -> "math.exp("+x.code+")"
23      |Sin(_,x) -> "math.sin("+x.code+")"
24      |Cos(_,x) -> "math.cos("+x.code+")"
25      |Tan(_,x) -> "math.tan("+x.code+")"
26      |Asin(_,x) -> "math.asin("+x.code+")"
27      |Acos(_,x) -> "math.acos("+x.code+")"
28      |Atan(_,x) -> "math.atan("+x.code+")"
29      |Atan2 (x,y) -> "math.atan2("+x.code+","+y.code+")"
30      |Abs(_,x) -> "abs("+x.code+")"
31      |Log(Zt,x) -> "cmath.log("+x.code+")"
32      |Log(_,x) -> "math.log("+x.code+")"
33      |Log10(Zt,x) -> "cmath.log10("+x.code+")"
34      |Log10(_,x) -> "math.log10("+x.code+")"
35      |Sqrt(Zt,x) -> "cmath.sqrt("+x.code+")"
36      |Sqrt(_,x) -> "math.sqrt("+x.code+")"
37      |Idx1(_,u,n1) -> u+"["+(n1-Int_c 1).code+"]"
38      |Idx2(_,u,n1,n2) ->
39          u+"["+(n1-Int_c 1).code+","+(n2-Int_c 1).code+"]"
40      |Idx3(_,u,n1,n2,n3) ->
41          u+"["+(n1-Int_c 1).code+","+(n2-Int_c 1).code+","+(n3-Int_c 1).code+"]"
42      |Formula(_,s) -> s
43      |Sum(t,n1,n2,f) ->
44          let g = num0.ch t
45          g.clear()
46          num0.looprange n1 n2 <| fun n ->
47              g <== g + (f n)
48          g.code
49      |Let(_,v,_) -> v.code
50      |NaN -> "NaN"
```

大部分は C 言語と Fortoran の定義コードと変更はない。演算子 “./” と “/” の場合分けは、“./” の解が整数であることを利用して、Div 型に格納されている商の型が整数型 (It _) であるパターン “|Div (It _,x,y) -> x.code+"//"+y.code”(12 行目) を “|Div (_,x,y) ->”(13 行目) の前に追加した。

4.1.4　Python コードの生成

ソースコード 4.1 によって生成される Python のコードをソースコード 4.5 に示す。

ソースコード 4.5　Aqualis で生成した Python コード 1

```
1 d0001 = 0
2 i0002 = 0
3 i0003 = 0
4 i0004 = 0
5 i0003 = 4
6 i0004 = 2
7 d0001 = float(i0003)/float(i0004)
8 i0002 = i0003//i0004
```

上記のコードの 1 行目から 4 行目では、使用する変数の生成が行われている。Python では、変数の宣言ができないため、数値や文字列の場合は初期値として “0” が、配列の場合は空の配列が生成されるようになっている。8 行目で行われている整数同士の余剰を無視する除算では Aqualis の “./” に当たる演算子 “//” がで用いられており、演算子 “./” を問題なく実装できている。

4.2　多重ループからの脱出

4.2.1　多重ループからの脱出

Aqualis における多重ループからの脱出は、iter クラスの脱出機能を持つメソッドを使用する際に自動で生成される脱出関数を用いて行われる。この脱出関数はどのような位置からでも指定されたループを脱出することが求められる。例えば、

ソースコード 4.6　Aqualis の二重ループ

```
1 iter.loop <| fun (ex,i) ->
2     iter.loop <| fun (ey,j) ->
3         br.if1 (j.>=5) <| fun () ->
4             ex()
5         print.cc i j
```

ソースコード 4.6 のような loop を二つ用いた二重ループでは外ループの脱出関数 ex() を用いたときは外ループと内ループを、内ループの脱出関数 ey() を用いたときは内ループのみを脱出しなければならない。

この機能を C 言語や Fortran では goto 文によって実装している。しかし、Python には goto 文がないため、if 文と break 文を用いて脱出関数を実装する必要があった。

4.2.2　Fortran 及び C 言語への変換コード

脱出関数を生成するメソッド this.getloopvar_exit の定義コードはソースコード 4.7 の通りである。

ソースコード 4.7 this.getloopvar_exit の定義コード

```
1  member this.getloopvar_exit code =
2      match lan with
3      |Fortran ->
4          let goto = this.goto_label.ToString()
5          this.goto_label <- this.goto_label + 1
6          let exit() = this.codewrite("goto "+goto+"\n")
7          let counter = this.loopvar.getAutoVar()
8          code(goto,counter,exit)
9          this.loopvar.setVar(It 4,A0,counter,"")
10     |C99 ->
11         let goto = "_"+this.goto_label.ToString()
12         this.goto_label <- this.goto_label + 1
13         let exit() = this.codewrite("goto "+goto+";\n")
14         let counter = this.loopvar.getAutoVar()
15         code(goto,counter,exit)
16         this.loopvar.setVar(It 4,A0,counter,"")
17     |LaTeX ->
18         ...
19     |HTML ->
20         ...
```

“match lan with …” は、出力するソースファイルの場合分けを表している。3 行目の “|Fortran ->” の後の命令から 9 行目の命令までは Fortran のソースファイルを出力するとき、10 行目の “|C99->” の命令文から 16 行目の命令文までは C 言語のソースファイルを出力するときを表している。ここでは、Fortran を出力するときで説明する。getloopvar_exit メソッドには “code” という名前の引数があるが、これはメソッドの使用時にラムダ式で定義される関数である。この関数の処理は、ソースコードの 8 行目で実行されている。その際に、引数として 4 行目で定義されている goto 文用のラベル番号と 6 行目で定義されている goto 文を生成する脱出関数 exit、7 行目で定義されているカウンタ変数 counter が渡されている。また、6 行目の “let exit() = this.codewrite("goto "+goto+";\n")” で使用されている “p.codewrite(…)” は、括弧の中の文字列をソースファイルに書き込む命令文である。4 行目で用いられている “this.goto_label.ToString()” の “.goto_label” は goto ラベル番号用の数値を取得するためのメソッドであり、その初期値は 10 となっている。また、“.ToString()” は、数値を文字列に変換するメソッドである。ソースコードの 5 行目で goto ラベルの名前が被らないようにラベル番号用の数値へ 1 を加算を行っている。9 行目では、使用し終わったカウンタ変数を未使用のカウンタ変数のリストに登録している。これにより、Fortran のソースコードで同じカウンタ変数を使い回すことができ、無駄なメモリの消費を抑えている。

脱出関数を用いるメソッドの例としては第 2 章で紹介した loop メソッドが挙げられる。loop メソッドのソースコードは 4.8 となっている。

ソースコード 4.8 loop メソッドの定義コード

```
1  static member loop code =
2      match p.lang with
3      |Fortran ->
4          p.getloopvar_exit <| fun (goto,v,exit) ->
```

```
5             let cnt = Var(It 4,v)
6             cnt <== 1
7             p.codewrite("do\n")
8             p.indentInc()
9             code(exit,cnt)
10            cnt <== cnt + 1
11            p.indentDec()
12            p.codewrite("end do"+"\n")
13            p.codewrite(goto+" continue"+"\n")
14    |C99 ->
15        p.getloopvar_exit <| fun (goto,v,exit) ->
16            let cnt = Var(It 4,v)
17            cnt <== 1
18            p.codewrite("for(;;)\n")
19            p.codewrite("{"+"\n")
20            p.indentInc()
21            code(exit,cnt)
22            cnt <== cnt + 1
23            p.indentDec()
24            p.codewrite("}"+"\n")
25            p.codewrite(goto+":;\n")
26    |LaTeX ->
27        ...
28    |HTML ->
29        ...
```

loop メソッドの引数“code”も getloopvar_exit メソッドの“code”と同様であり、反復処理の内部処理を表している。4 行目及び 15 行目の“p.getloopvar_exit”は、ソースコード 4.7 で定義されているメソッド“getloopvar_exit”を呼び出すメソッドである。5 行目と 6 行目及び 16 行目と 17 行目でカウンタ変数の定義と初期化を行っている。7 行目から 11 行目及び 18 行目から 24 行目は、それぞれ Fortran と C 言語における無限ループの生成している。9 行目、10 行目、21 行目、22 行目では、ループの内部処理が書かれており、関数 code に引数として脱出関数 exit とカウンタ変数 cnt が渡されている。19 行目、25 行目で goto ラベルを生成するコードが記述されている。

4.2.3　Python への変換コード

まず、getloopvar_exit に追加したソースコードを 4.9 に示す。

ソースコード 4.9　getloopvar_exit に追加した定義コード

```
1     |Python ->
2         let goto = this.goto_label.ToString()
3         this.goto_label <- this.goto_label + 1
4         let exit() = this.codewrite("flag="+goto+"\nbreak\n")
5         let counter = this.loopvar.getAutoVar()
6         code(goto,counter,exit)
7         this.loopvar.setVar(It 4,A0,counter,"")
```

上記のコードで exit は、ラベル番号が格納された変数 flag とループを脱出する break を生成する関数をなっている。次に loop メソッドはソースコード 4.10 のようになった。

ソースコード 4.10 loop に追加した定義コード

```
1  |Python ->
2      p.getloopvar_exit <| fun (goto,counter,exit) ->
3          let cnt = Var(It 4,counter)
4          cnt <== 1
5          p.codewrite("while True:\n")
6          p.indentInc()
7          code(exit,cnt)
8          p.codewrite("flag = "+goto+"\n")
9          cnt <== cnt + 1
10         p.indentDec()
11         if goto="10" then
12             p.exit_reset
13         else
14             p.codewrite("if flag < "+goto+":\n")
15             p.indentInc()
16             p.codewrite("break\n")
17             p.indentDec()
```

8 行目で内部処理を記述後、9 行目で変数 flag を現在実行されているループのラベル番号に更新を行う。12 行目から 18 行目の条件分岐 “goto="10" then …else …” は、最初に脱出関数を生成したループかそうでないかの判定を行っている。最初のループだった場合は、ラベル番号を再利用するために関数 “p.exit_reset” を用いてラベル番号の初期化を行う。そうでない場合は、連続脱出のための条件文のソースコードを生成するコードが実行される。連続脱出のための条件は、先程まで実行されていたループのラベル番号よりも変数 flag の値が小さいことである。前述のループ内で脱出が行われなかった場合、変数 flag の値は条件文で用いられているラベル番号と等しいため、脱出は行われない。また、iter クラスの脱出機能を持たないメソッドにも同じような連続脱出の条件文を生成する処理が記述されており、どのような状態からも脱出関数が指定したループを脱出可能になっている。

4.2.4 Python コードの生成

ソースコード 4.6 によって生成される Python のソースコードを 4.11 に示す。

ソースコード 4.11 Aqualis で生成した Python コード 2

```
1  ic0001 = 1
2  while True:
3      ic0002 = 1
4      while True:
5          if ic0002 > 5:
6              flag = 10
7              break
8          print("%8d%8d" %(ic0001,ic0002))
9          flag = 11
10         ic0002 = ic0002+1
```

```
11      if flag < 11:
12          break
13      flag = 10
14      ic0001 = ic0001+1
```

上記のコードでは、無限ループ“while True:”が二重に生成されている。ループの末尾ではそれぞれのカウンタ変数と脱出用のフラグ変数“flag”の値の更新が行われている。また、内ループのすぐ外にはフラグ変数を利用した連続脱出用の条件文が生成されている。脱出関数 ex() は、7 行目の“flag = 10”と 8 行目の“break”を出力した。連続脱出用の条件文は“flag < 11”であるため、5 行目の条件文を満たせば内ループから外ループの外まで脱出できる。

また、ex() を ey() に入れ替えた場合、ソースコード 4.12 のようなコードになる。

ソースコード 4.12　Aqualis で生成した Python コード 3

```
1  ic0001 = 1
2  while True:
3      ic0002 = 1
4      while True:
5          if ic0002 > 5:
6              flag = 11
7              break
8          print("%8d%8d" %(ic0001,ic0002))
9          flag = 11
10         ic0002 = ic0002+1
11     if flag < 11:
12         break
13     flag = 10
14     ic0001 = ic0001+1
```

脱出関数 ey() は、7 行目の“flag = 11”と 8 行目の“break”を生成している。連続脱出用の条件文は“flag < 11”であるため、5 行目の条件文を満たした場合、内ループのみ脱出する。

4.3　クラスを要素に持つ配列の生成

4.3.1　クラスを要素に持つ配列の生成

クラスを要素に持つ配列を宣言する場合、Aqualis では静的な配列確保は行われず、動的なメモリ確保が必要である。この動的確保は、C 言語では malloc、Fortran では allocate を使用して実装されている。一方、Python ではメモリ管理が自動化されており、手動でメモリを確保する操作は基本的に不要であり、推奨されていない。

本研究では Aqualis で生成される Python の配列に NumPy モジュールの ndarray(多次元配列) を採用した。このため、配列全体のデータ型を指定することは可能だが、指定できるのは整数型や実数型などの標準的なデータ型に限られる。よって、クラスなどユーザー定義型を直接扱うことはできない。そこで、ndarray のデータ型に Python オブジェクトを指す object 型を指定し、クラス型のインスタンスを要素とするリストを ndarray に渡す必要がある。

4.3.2 Fortran 及び C 言語への変換コード

Aqualis の配列のメモリ割り当ては、各次元の配列ごとに定義されている allocate メソッドによって行われる。ソースコード 4.13 は一次配列用の allocate の定義コードである。

ソースコード 4.13 allocate の定義コード

```
1  member this.allocate(n1:num0) =
2      match x with
3      |Var1(size1,name) ->
4          if p.debugMode then
5              p.errorIDinc()
6              p.comment("***debug array1 allocate check: "
7                        +p.errorID.ToString()+"****************************")
8              br.branch <| fun b ->
9                  b.IF (this.size1 .=/ -1) <| fun () ->
10                     print.t ("ERROR"+p.errorID.ToString()+
11                              " array "+name+" is already allocated")
12             p.comment("**************************************************")
13         match p.lang with
14         |Fortran ->
15             match size1 with
16             |A1(0) ->
17                 this.size1 <== n1
18                 p.codewrite("allocate("+name+"(1:"+this.size1.code+")"+")"+"\n")
19             |_ ->
20                 p.codewrite("(Error:055-001 「"+name+"」は
21                             可変長1次元配列ではありません")
22         |C99 ->
23             match size1 with
24             |A1(0) ->
25                 this.size1 <== n1
26                 p.codewrite(name+"="+"("+typ.tostring(p.lang)+" *)"
27                             +"malloc("+"sizeof("+typ.tostring(p.lang)+")*"
28                             +this.size1.code+");\n")
29             |_ ->
30                 p.codewrite("(Error:055-001 「"+name+"」は
31                             可変長1次元配列ではありません")
32         |LaTeX ->
33             ...
34         |HTML ->
35             ...
36     |_ -> ()
```

2 行目の“match x with …”は変数と部分配列の場合分けを表している。3 行目の“|Var1(size1,name) ->”から 35 行目までの命令は一次配列“this”が変数だった時の処理を、36 行目の“|_ -> ()”はそれ以外の時は処理を行わない事を表している。4 行目から 12 行目までは、Aqualis をデバックモードで実行した際の処理が記述されている。13 行目以降以降のコードは、格言言語毎に場合分けされている。このメソッド

の構成は C 言語と Fortran で違いは無いため、ここでは Fortran のコードを出力するときについて説明する。13 行目にある“match size1 with …”は、変数及び各次元の配列の場合分けを表している。16 行目の“|A1(0) ->”の後の命令は、一次元配列だった場合の処理が記述されており、Fortran における動的メモリ確保のソースコードの生成が行われている。17 行目の“this.size1 <== n1”は、配列精製時に用意される配列のサイズ格納用配列に指定されたサイズの数値を渡している。このサイズ格納用配列は一次元配列の場合は一次元、二次元配列の場合は二次元の配列が用意される。19 行目にある“|_ ->”以降の命令では一次元配列以外の場合、エラー文を出力する処理が記述されている。

4.3.3　Python への変換コード

追加した Python のソースファイルを出力する場合の定義コードはソースコード 4.14 のようになった。

ソースコード 4.14　allocate に追加した定義コード

```
1     |Python ->
2         match size1 with
3         |A1(0) ->
4             this.size1 <== n1
5             match typ with
6             |Structure(sname) ->
7                 p.codewrite(name+" = numpy.array(["+sname+"() for _ in range(
8                                 int("+this.size1.code+"))], dtype=object)\n")
9             |It _ |It 1       ->
10                p.codewrite(name+" = numpy.zeros("+this.size1.code+",
11                                dtype=int)\n")
12            |Zt               ->
13                p.codewrite(name+" = numpy.zeros("+this.size1.code+",
14                                dtype=numpy.complex128)\n")
15            |_                ->
16                p.codewrite(name+"="+"numpy.zeros("+this.size1.code+")\n")
17        |_ ->
18            p.codewrite("(Error:055-001 「"+name+"」は
19                          可変長1次元配列ではありません")
```

5 行目の“match typ with …”は、一次元配列“this”のデータ型による場合分けを表している。6 行目の“|Structure(sname) ->”の後の命令は構造体型、9 行目の“|It _ |It 1 ->”の後の命令は整数型、12 行目の“|Zt ->”の後の命令は複素数型、15 行目の“|_ ->”の後の命令には実数型の処理が記述されている。7 行目で記述されている Python の関数“numpy.array(…)”は、リストを“dtype”で指定された型の多次元配列に変換できる。これを用いてリスト内包表記で生成したクラス型の要素を持ったリストを多次元配列に変換している。10 行目、13 行目、15 行目で登場している“numpy..zeros(…)”は、指定された要素数と“dtype”で指定された型の 0 配列を生成する Python の関数となっている。

4.3.4　Python コードの生成

定義コード追加後に Aqualis によって生成される Python コードはソースコード 4.15 のようになる。

ソースコード 4.15　Aqualis で生成した Python コード 4

```
d = numpy.array([], dtype=object)
d_size = numpy.array([-1])
c = Class1()
ic0001 = 0
c.x1 = 1
c.y1 = 2.0E0
print("%8d" %(c.x1))
print("%27.17e" %(c.y1))
d_size[0] = 4
d = numpy.array([Class1() for _ in range(int(d_size[0]))], dtype=object)
for ic0001 in range(1, d_size[0]+1, 1):
    d[ic0001-1].x1 = 1
    d[ic0001-1].y1 = 2.0E0
    print("%8d%27.17e" %(d[ic0001-1].x1,d[ic0001-1].y1))
```

allocate(…) メソッドが生成したのは、10 行目と 11 行目のコードである。10 行目は allocate(…) で指定された配列のサイズを配列変数 d_size に格納している。そして、11 行目ではリスト内包表記でクラス型の配列を生成し、それを object 型の多次元配列に変換して配列変数 d に格納している。これによって、クラスの要素に持つ配列が生成され、利用できるようになった。よって、問題なく実装できていると言える。

4.4 参照渡しの関数

4.4.1 参照渡しの関数

Aqualis によって生成される非インライン展開の関数は、原則として引数の参照渡しを行う。参照渡しとは、関数に引数を渡す際にその変数への参照 (アドレス) を渡す手法のことである。C 言語、Fortran では変数にアドレスが振られている。そのため、C 言語では引数として参照渡しを行うことができ、Fortran の場合はデフォルトとして参照渡しが行われている。よって、既存の二つの言語では問題なく実装されている。

Python の場合は、参照渡しに近い動きをするものと引数に対して値のコピーを渡す値渡しという方法に近い動きをするものに分かれる。これは Python がオブジェクト指向の言語であるが故である。オブジェクト指向の言語において、アドレスは変数ではなく、値に与えられる。そのアドレスを変数が格納し、それを用いて値を参照する形となっている。その為、関数の引数は変数でも値ではなく値のアドレスが渡される“共有渡し”と言われる形となる。よって、数値や文字列などは内容が変更を加えると値のアドレスそのものが変わるので、関数外の変数には変更が共有されない。対して、配列などの場合は配列内の要素を変更するだけならば、配列そのもののアドレスは変わらないため、関数外の変数にも変更が共有される。

よって、本研究では関数の戻り値を利用して引数へ再代入を行うことで、擬似的な参照渡しを実現することにした。

4.4.2 C 言語への変換コード

関数の引数で参照渡しを行うメソッド func の追加作業では C 言語を参考に行われたため、ソースコード 4.16 に示す func の定義コードでは Fortran に関するソースコードは省略する。

ソースコード 4.16 func の定義コード

```
let func (projectname:string) (code:unit->unit) =
```

```
2     let dir_ = p.dir
3     let fdeclare (typ:Etype,vtp:VarType,name:string) =
4         ...
5     match p.lang with
6     |Fortran ->
7         ...
8     |C99 ->
9         p.param_main.funlist <- projectname::p.param_main.funlist
10        p.param_add(C99, dir_, projectname)
11        //ここから関数定義。p.paramは関数用のものに変わる
12        p.paramClear()
13        //メインコード生成
14        p.indentInc()
15        code()
16        p.indentDec()
17        p.cclose()
18        p.indentInc()
19        p.declareall()
20        p.indentDec()
21        p.vclose()
22        //ソースファイル(関数部分)出力
23        p.hwrite("/*================================================
24                ==========================================*/"+"\n")
25        p.hwrite("/* Subroutine name: "+projectname+" */\n")
26        for _,(_,_,nm) in p.arglist do
27            p.hwrite("/* "+nm+" */\n")
28        p.hwrite("/*================================================
29                ==========================================*/"+"\n")
30        let argvar =
31            let cat acc i =
32                let _,(typ,vtp,n) = p.arglist.[i]
33                let cm = if (i=p.arglist.Length-1) then "" else ", "
34                match vtp with
35                |A1(_)|A2(_)|A3(_) ->
36                    acc + typ.tostring(p.lang) + " *" + n + cm
37                |_ ->
38                    acc + typ.tostring(p.lang) + " *" + n + cm
39            List.fold cat "" [0..p.arglist.Length-1]
40        p.hwrite("void "+projectname+"("+argvar+")"+"\n")
41        p.hwrite("{\n")
42        p.indentInc()
43        //グローバル変数の定義
44        p.hwrite(File.ReadAllText(dir_+"\\"+projectname+"_var"+".bee"))
45        File.Delete(dir_+"\\"+projectname+"_var"+".bee")
46        //メインコード
47        p.hwrite(File.ReadAllText(dir_+"\\"+projectname+"_code.bee"))
48        File.Delete(dir_+"\\"+projectname+"_code.bee")
49        p.pclose()
50        File.Delete(dir_+"\\"+projectname+"_par.bee")
```

```
51          p.indentDec()
52          p.hwrite("}\n")
53          p.hclose()
54          //呼び出しコードを記述
55          let args =
56              let cat acc i =
57                  let n,(typ,vtp,_) = p.arglist.[i]
58                  let cm = if (i=p.arglist.Length-1) then "" else ", "
59                  match typ,vtp,(n.StartsWith "(*") with
60                  |(It _|Dt|Zt|Structure _),A0,false ->
61                      acc + "&" + n + cm
62                  |(It _|Dt|Zt|Structure _),A0,true  ->
63                      let n_ = n.Substring(2,n.Length-3)
64                      acc + n_ + cm
65                  |_ -> acc + n + cm
66              List.fold cat "" [0..p.arglist.Length-1]
67          //もとの関数に戻る
68          p.param_back()
69          p.codewrite(projectname + "(" + args + ");\n")
70      |LaTeX ->
71          ...
72      |HTML ->
73          ...
```

2 行目で定義されている関数“dir_”はソースファイルの出力先ディレクトリを取得する処理を実行する。9 行目のコードは、関数の名前を格納しているリストに func メソッドを用いて定義された関数名 (projectname) の追加を行っている。10 行目で使用されている関数“param_add”で、新しい関数の追加を行っている。11 行目で新しい関数の内部処理を定義するために、“paramClear”を使用して関数のパラメータの初期化を実行している。以降、関数のパラメータは新しく定義される関数用のものになる。15 行目で関数のメインコードの変換の実行が行われ、C 言語の文法に沿った関数の内部処理が書かれた一時ファイルが生成される。また、19 行目の“p.declareall()”によって関数内で用いる var クラスのメソッドによって生成された変数を宣言するソースコードの書かれた一時ファイルが生成されている。他のメソッドと違い、メインコードが先に変換されたのは、関数のメインコードが記述されるよりも前で変数宣言を行うためである。実際に関数のメインコードが生成される 47 行目より前の 44 行目で変数用の一時ファイルから読み込まれた変数の定義が行われている。23 行目から 29 行目では、関数名と仮引数名を纏めたコメント文の生成を行う。30 行目から 39 行目までで定義されている“argvar”は仮引数のリストから生成された文字列である。40 行目から 53 行目で“argvar”を仮引数として持つ関数を生成し、そのメインコードは定義される関数用の一時ファイルから読み込まれている。5 行目から 66 行目までで定義されている“args”は、呼び出しコードに使用される引数のリストから作られた文字列となっている。68 行目の“p.paramClear_back()”で関数が呼び出された関数に戻り、69 行目で新しく定義された関数を呼び出す場合の処理が記述されている。

4.4.3 Python への変換コード

ソースコード 4.17 に追加した定義コードを示す。

ソースコード 4.17 func に追加した定義コード

```
1         |Python ->
2             p.param_main.funlist <- projectname::p.param_main.funlist
3             p.param_add(Python, dir_, projectname)
4             //ここから関数定義。p.paramは関数用のものに変わる
5             p.paramClear()
6             //メインコード生成
7             p.indentInc()
8             code()
9             p.indentDec()
10            p.cclose()
11            p.indentInc()
12            p.declareall()
13            p.indentDec()
14            p.vclose()
15            //ソースファイル(関数部分)出力
16            p.hwrite("#=============================================
17                    ==============================================="+"\n")
18            p.hwrite("# Subroutine name: "+projectname+"\n")
19            for _,(_,_,nm) in p.arglist do
20                p.hwrite("# "+nm+"\n")
21            p.hwrite("#=============================================
22                    ==============================================="+"\n")
23            let argvar =
24                let cat acc i =
25                    let _,(_,vtp,n) = p.arglist.[i]
26                    let cm = if (i=p.arglist.Length-1) then "" else ", "
27                    match vtp with
28                    |A1(_)|A2(_)|A3(_) ->
29                        acc + n + cm
30                    |_ ->
31                        acc + n +  cm
32                List.fold cat "" [0..p.arglist.Length-1]
33            let re_argvar =
34                let cat acc i =
35                    let _,(_,vtp,n) = p.arglist.[i]
36                    let cm = if (i=p.arglist.Length-1) then "" else ", "
37                    match vtp with
38                    |A1(_)|A2(_)|A3(_) ->
39                        acc
40                    |_ ->
41                        acc + n +  cm
42                List.fold cat "" [0..p.arglist.Length-1]
43            //呼び出しコードを記述
44            let args =
45                let cat acc i =
46                    let n,(typ,vtp,_) = p.arglist.[i]
47                    let cm = if (i=p.arglist.Length-1) then "" else ", "
48                    match typ,vtp,(n.StartsWith "(*") with
```

```
49                      |(It _|Dt|Zt|Structure _),A0,false ->
50                          acc + n + cm
51                      |(It _|Dt|Zt|Structure _),A0,true  ->
52                          let n_ = n.Substring(2,n.Length-3)
53                          acc + n_ + cm
54                      |_ -> acc + n + cm
55                  List.fold cat "" [0..p.arglist.Length-1]
56              let re_args =
57                  let cat acc i =
58                      let n,(_,vtp,_) = p.arglist.[i]
59                      let cm = if (i=p.arglist.Length-1) then "" else ", "
60                      match vtp with
61                      |A1(_)|A2(_)|A3(_) ->
62                          acc
63                      |_ ->
64                          acc + n +  cm
65                  List.fold cat "" [0..p.arglist.Length-1]
66              p.hwrite("def "+projectname+"("+argvar+"):"+"\n")
67              p.indentInc()
68              //グローバル変数の定義
69              p.hwrite(File.ReadAllText(dir_+"\\"+projectname+"_var"+".bee"))
70              File.Delete(dir_+"\\"+projectname+"_var"+".bee")
71              //メインコード
72              p.hwrite(File.ReadAllText(dir_+"\\"+projectname+"_code.bee"))
73              File.Delete(dir_+"\\"+projectname+"_code.bee")
74              p.pclose()
75              File.Delete(dir_+"\\"+projectname+"_par.bee")
76              p.hwrite(p.indentSpace+"return "+re_argvar+""+"\n")
77              p.indentDec()
78              p.hclose()
79              //もとの関数に戻る
80              p.param_back()
81              p.codewrite(re_args+" = "+projectname+"("+args+")\n")
```

変更箇所は、3 カ所存在する。一カ所目は、33 行目から 42 行目のコードで戻り値として渡す仮引数群“re_argvar”を定義している箇所である。この際、再代入時の処理時間短縮のため、関数内でも編集可能である配列の仮引数は除外されている。二カ所目は、戻り値を受け取る変数群“re_args”が定義されている 56 行目から 64 行目のコードである。この際にも同じ理由から配列の変数は除外される。3 カ所目は、関数の呼び出しについて記述されている 81 行目のコードとなっている。81 行目の処理によって生成される Python コードは、“re_args”が呼び出した関数の戻り値を受け取るものになっている。

4.4.4 Python コードの生成

追加後に 3.17 から生成した Python のソースコードは 4.18 のようになった。

ソースコード 4.18 Aqualis で生成した Python コード 5

```
1 i0001 = 0
2 d0001 = 0
```

```
3  d10001 = numpy.array([])
4  d10001_size = numpy.array([-1])
5  #=============================================
6      ===============================================
7  # Subroutine name: func1
8  # arg01
9  # arg01_size
10 # arg03
11 # arg04
12 #=============================================
13     ===============================================
14 def func1(arg01, arg01_size, arg03, arg04):
15     arg03 = 2
16     arg01[1] = arg03+arg04
17     return arg03, arg04
18
19 d10001_size[0] = 2
20 d10001=numpy.zeros(d10001_size[0])
21 i0001 = 1
22 d0001 = 2
23 d10001[:]=0
24 i0001, d0001 = func1(d10001, d10001_size, i0001, d0001)
25 print("%8d%27.17e%27.17e%27.17e" %(i0001,d0001,d10001[0],d10001[1]))
26 del d10001
```

宣言されている関数“func1”は、四つの仮引数を受け取り、配列以外の変数を戻り値として返している。24 行目に記述された呼び出し元でも配列以外の変数が戻り値を受け取っている。よって、Aqualis のメソッド“func”は、問題なく実装できたと判断できる。

第 5 章

計測時間の比較

5.1　最適化処理：インライン展開

3 章で述べた通り、Aqualis は最適化処理の一つとして、関数は基本的にインライン展開されるようになっている。これによって、関数呼び出しに伴うオーバーヘッドを削減することができる。

この削減が処理の効率化にどの程度貢献するのか、実行時間を比較するために Aqualis で生成されたコードと手書きのコードをそれぞれ二種類用意した。二種類のソースコードの動きとしては、関数 `h` を 1000 回呼び出すものと関数 `h` を引数に受け取る高階関数 `f` を 1000 回呼び出すものになっている。ソースコード 5.1、5.2 に作成した Aqualis のコードを示す。

ソースコード 5.1　Aqualis: 関数 `h`

```
1 ch.i <| fun a ->
2     a <== 0
3     let h(x:num0) = x+1
4     iter.range 1 1000 <| fun i ->
5         a <== h(i)
```

ソースコード 5.2　Aqualis: 高階関数 `f`

```
1 ch.i <| fun a ->
2     a <== 0
3     let h(x:num0) = x+1
4     let f(x:num0,g:num0->num0) = g x + 1
5     iter.range 1 1000 <| fun i ->
6         a <== f(i, h)
```

上記の二つのコードから生成された Python のソースコードは 5.3、5.4 のようになる。

ソースコード 5.3　Aqualis で生成した Python コード: 関数 `h`

```
1 i0001 = 0
2 ic0001 = 0
3 for ic0001 in range(1, 1000, 1):
4     i0001 = ic0001+1
```

ソースコード 5.4　Aqualis で生成した Python コード: 高階関数 `f`

```
1 i0001 = 0
```

```
2  ic0001 = 0
3  for ic0001 in range(1, 1000, 1):
4      i0001 = ic0001+2
```

次に手書きで作成したした Python のコードをソースコード 5.5、5.6 に示す。

ソースコード 5.5　Python: 関数 h

```
1  def h(x):
2      return x+1
3  a = 0
4  for i in range(1, 1000, 1):
5      a = h(i)
```

ソースコード 5.6　Python: 高階関数 f

```
1  def h(x):
2      return x+1
3  def f(x, g):
4      return g(x)+1
5  a = 0
6  for i in range(1, 1000, 1):
7      a = f(i, h)
```

手書きで作成されたソースコード 5.5、5.6 には関数が定義されているのに対し、Aqualis によって生成されたコードであるソースコード 5.3、5.4 では関数の定義がされていない。しかし、ソースコード上の呼び出し元に関数の内部コードが展開されており、Aqualis が問題なく最適化処理を実行できているのが分かる。

5.2　測定結果

表は、5 回実行したときの平均の計算時間である。プログラムは OS:Windows11、PC:WindowsPC(CPU:Intel(R) Core(TM) i5-9400)、環境:Windows Subsystem for Linux で実行した。

表 5.1　関数 h の実行時間

生成方法	実行時間 [µs]
手書き	136.2
Aqualis	82.6

表 5.2　高階関数 f の実行時間

生成方法	実行時間 [µs]
手書き	206.6
Aqualis	73.3

5.3 考察

手書きのコードと Aqualis によって生成されたコードを比較した結果、関数 h の場合、高階関数 f の場合共に実行時間が短縮された。また、関数 h では手書きのコードの実行時間の約 60%、高階関数 f においては約 35% 程度の実行時間まで短縮することができた。高階関数 f の方が、関数 h の実行時間よりも相対的に短縮できた理由としては、高階関数 f が関数 h を呼び出した時間だけ手書きのコードの実行時間が増加した事が考えられる。このことから Aqualis の最適化処理によって Python の実行時間の短縮が可能であることが分かった。これにより、本研究の目的は達成された。

第 6 章

結論

本研究では、Aqualis に Python のソースコード生成機能を実装し、Aqualis を用いて最適化処理による Python の実行時間の短縮を図った。

6.1 Python への変換機能の実装

4 章で述べた通り、Aqualis に Python への変換機能を実装した。これにより、Aqualis のソースファイルから Python のソースファイルを生成することが可能になり、他言語との連携が従来よりも簡単になった。

6.2 最適化処理

5 章での Aqualis の生成コードと手書きのコードの比較では、実行時間の短縮に成功した。他のプログラムでも同じ結果が得られることが予想される。ゆえに、本研究の目的が達成された。

付録 A

編集したメソッド一覧

表 A.1　array1 のメソッド

メソッド名	引数	役割
__.size1	-	変数の要素数
this.allocate	n1:num0	配列メモリの割り当て
this.deallocate	-	メモリの解放
<==	v1:num1,v2:num1	配列のコピー
<==	v1:num1,v2:num0	配列の全要素への代入

表 A.2　array2 のメソッド

メソッド名	引数	役割
__.size1	-	変数の要素数
__.size2	-	変数の要素数
this.allocate	n1:num0,n2:num0	配列メモリの割り当て
this.deallocate	-	メモリの解放
<==	v1:num2,v2:num2	配列のコピー
<==	v1:num2,v2:num0	配列の全要素への代入

表 A.3　array3 のメソッド

メソッド名	引数	役割
__.size1	-	変数の要素数
__.size2	-	変数の要素数
__.size3	-	変数の要素数
this.allocate	n1:num0,n2:num0,n3:num0	配列メモリの割り当て
this.deallocate	-	メモリの解放
<==	v1:num3,v2:num3	配列のコピー
<==	v1:num3,v2:num0	配列の全要素への代入

表 A.4 bessel のメソッド 1

メソッド名	引数	役割
besselj0	x:num0,code:num0->unit	0 次の第 1 種ベッセル関数
bessely0	x:num0,code:num0->unit	0 次の第 2 種ベッセル関数
besselh0	x:num0,code:num0->unit	0 次の第 3 種ベッセル関数
besselj1	x:num0,code:num0->unit	1 次の第 1 種ベッセル関数
bessely1	x:num0,code:num0->unit	1 次の第 2 種ベッセル関数
besselh1	x:num0,code:num0->unit	1 次の第 3 種ベッセル関数

表 A.5 br のメソッド

メソッド名	引数	役割
branch_	el:Branch,cond:bool0,code:unit->unit	条件分岐式を生成

表 A.6 codestr のメソッド

メソッド名	引数	役割
section	s:string,code:unit->unit	コードの階層構造を作成
subsection	s:string,code:unit->unit	コードの階層構造を作成
subsection	s:string,code:unit->unit	コードの階層構造を作成
header	c:char,s:string	コードの階層構造を作成
footer	c:char,s:string	コードの階層構造を作成
h1	s:string,code:unit->unit	コードの階層構造を作成
h2	s:string,code:unit->unit	コードの階層構造を作成
h3	s:string,code:unit->unit	コードの階層構造を作成
h4	s:string,code:unit->unit	コードの階層構造を作成
h5	s:string,code:unit->unit	コードの階層構造を作成

表 A.7 compile のメソッド

メソッド名	引数	役割
func	projectname:string,code:unit->unit	参照渡しの関数を作成
Compile	lglist:seq<Language>,dir:string,projectname:string, (aqver:string,codever:string),code:unit->unit	トランスコンパイル

表 A.8　expr のメソッド

メソッド名	引数	役割
this.code	-	論理演算のコード生成
looprange	i1:num0,i2:num0,code:unit->unit	反復処理のコード生成
this.code	-	変数及び算術演算のコード生成
todouble	x:num0	実数型への変換
%	x:num0,y:num0	余剰の計算
powr	x:num0,y:num0	累乗の計算
<==	x:num0,y:num0	代入
===	x:num0,y:num0	等式 (TeX、HTML のみ)
=|=	x:num0,y:num0	等式 (TeX、HTML のみ)

表 A.9　fft1 のメソッド

メソッド名	引数	役割
fft1	planname:string,data1:num1,data2:num1,fftdir:int	一次元高速フーリエ変換

表 A.10　fft2 のメソッド

メソッド名	引数	役割
fft2	planname:string,data1:num2,data2:num2,fftdir:int	二次元高速フーリエ変換

表 A.11　io のメソッド

メソッド名	引数	役割	
io	fileAccess	filename:list<num0>,readmode:bool0, isbinary:bool0,code:string->unit	ファイルへのアクセス
io	Write	fp:string,lst:list<num0>	テキストファイルへの書き込み
io	Write_bin	fp:string,v:num0	バイナリファイルへの書き込み
io	Read	fp:string,iostat:num0,lst:list<num0>	テキストファイルの読み込み
io	Read_bin	fp:string,iostat:num0,v:num0	バイナリファイルの読み込み
io	fileAccess	filename:list<num0>,readmode:bool0, isbinary:bool0,code:string->unit	区切り無のファイルへのアクセス
io	Write	fp:string,lst:list<num0>	区切り無のファイルへの書き込み

表 A.12 iter1 のメソッド

メソッド名	引数	役割
loop	code:(unit->unit)*num0->unit	無限ループ
whiledo	cond:bool0,code:unit->unit	条件を満たす間ループ
range	i1:num0,i2:num0,code:num0->unit	指定された範囲のループ (増加)
range_exit	i1:num0,i2:num0, code:(unit->unit)*num0->unit	指定された範囲のループ (脱出可)
range_reverse	i1:num0,i2:num0,code:num0->unit	指定された範囲のループ (減少)
range_reverse _exit	i1:num0,i2:num0, code:(unit->unit)*num0->unit	指定された範囲のループ (減少) (脱出可)
range_interval	i1:num0,i2:num0, ii:num0,code:num0->unit	指定された範囲のループ (指定された間隔で減少)
range_interval _exit	i1:num0,i2:num0,ii:num0, code:(unit->unit)*num0->unit	指定された範囲のループ (脱出可) (指定された間隔で減少)

表 A.13 lapack のメソッド

メソッド名	引数	役割
solve_simuleq	matrix:num2,y:num1	連立方程式の求解
solve_simuleqs	matrix:num2,y:num2	連立方程式の求解
inverse_matrix	mat2:num2,mat1:num2	逆行列の計算
rank	rank:num0,mat:num2,cond:num0	行列の階数
eigen_matrix	eigenvalues:num1*eigenvectors:num2,mat1:num2	非対称複素行列の固有値
eigen_matrix2	eigenvalues1:num1*eigenvalues2:num1* eigenvectors:num2,mat1:num2,mat2:num2	非対称複素行列の一般化固有値
determinant	matrix:num2,code:num0->unit	行列式の常用対数を計算
svd	mat1:num2,u:num2*s:num1*vt:num2	特異値分解

表 A.14 main のメソッド

メソッド名	引数	役割
Language	-	言語を指定
this.tostring	lang: Language	言語設定に従って型名を生成

表 A.15　num0 のメソッド

メソッド名	引数	役割	
num0	uj	-	虚数単位
num0	pi	-	円周率
num0	todouble	x:num0	実数型への変換
num0	toint	x:num0	整数型への変換
num0	floor	v:num0	小数点以下切り捨て
num0	ceil	v:num0	小数点以下切り上げ
num0	conj	v:num0	共役複素数
num0	x.re	-	実部
num0	x.im	-	虚部

表 A.16 param のメソッド

メソッド名	引数	役割
__.getAutoVarName	n:int	ナンバリング自動変数
loopvar	-	ループのカウンタに現在使用できる変数インデックス
i1_cache_var	-	複数の代入文で続けて使用できる一時変数 (整数 1 次元配列)
d1_cache_var	-	複数の代入文で続けて使用できる一時変数 (実数 1 次元配列)
z1_cache_var	-	複数の代入文で続けて使用できる一時変数 (複素数 1 次元配列)
i2_cache_var	-	複数の代入文で続けて使用できる一時変数 (整数 2 次元配列)
d2_cache_var	-	複数の代入文で続けて使用できる一時変数 (実数 2 次元配列)
z2_cache_var	-	複数の代入文で続けて使用できる一時変数 (複素数 2 次元配列)
i3_cache_var	-	複数の代入文で続けて使用できる一時変数 (整数 3 次元配列)
d3_cache_var	-	複数の代入文で続けて使用できる一時変数 (実数 2 次元配列)
z3_cache_var	-	複数の代入文で続けて使用できる一時変数 (複素数 3 次元配列)
__.Stype	typ:Etype	変数の型名を文字列に変換
this.declare	typ:Etype*vtp:VarType*name:string*param:string	変数宣言のコード
this.declareall	unit	宣言されたすべての変数を一時ファイルに書き込み
__.DtoS	d:double	実数型の数値を文字列に変換
this.addarg	typ:Etype*vtp:VarType*n:string,code:VarType*string->'a	関数定義の引数を追加
this.codewrite	ss:string	コードを一時ファイルに書き込み
this.comment	s:string	コメント文
this.getloopvar	code:string*string*	ループカウンタ変数と
_exit	(unit->unit)->unit	ループ脱出先 goto ラベルを作成し、code 内の処理を実行

表 A.17 print のメソッド

メソッド名	引数	役割	
print	s	lst:list<num0>	変数リストを画面表示

表 A.18 random のメソッド

メソッド名	引数	役割
random	code:((num1->unit)->unit)*(num0->unit)->unit	0〜1 の一様乱数を取得
random_s	seed_:int,code:((num1->unit)->unit)*(num0->unit)->unit	0〜1 の一様乱数を取得

表 A.19 setting のメソッド

メソッド名	引数	役割
abort	unit	プログラムの実行を強制終了
stop	unit	何かのキーを押すまで
stop	unit	実行を一時停止
stop	string	一時停止前に表示する文字列

表 A.20 shellscript のメソッド

メソッド名	引数	役割
this.AddProcess	unit	ソースファイルのコンパイル・実行するスクリプトファイルを生成
__.AddProcess	address:string	ソースファイルのコンパイル・実行するスクリプトファイルを生成

表 A.21 structure のメソッド

メソッド名	引数	役割
this.Def_Structure	unit	構造体定義のコードを作成
mem	vname:string*name:string	構造体メンバへのアクセス
this.reg	sname:string,name:string	構造体定義追加・変数宣言
this.reg	sname:string,name:string,size1:int	構造体定義追加・変数宣言
this.reg	sname:string,name:string,size1:int,size2:int	構造体定義追加・変数宣言
this.reg	sname:string,name:string,size1:int,size2:int,size3:int	構造体定義追加・変数宣言

表 A.22 var のメソッド

メソッド名	引数	役割
init_z	re:double,im:double	複素数の初期値
init_i1	v:list<int>	整数型配列の初期値
init_d1	v:list<double>	実数型配列の初期値
init_z1	v:list<double*double>	複素数型配列の初期値

謝辞

本研究の遂行にあたり、指導教員の杉坂准教授には大変お世話になりました。私の卒業論文や発表資料の作成にも、夜遅くまでお付き合いいただきました。深く感謝いたします。最後に、数理波動システム研究室の皆様には、本研究の遂行にあたり多大な激励頂きました。ここに感謝の意を表します。

2025 年 2 月 北原玲

参考文献

[1] M. Szafraniec, B. Roziere, H. Leather, F. Charton, P. Labatut, and G. Synnaeve, “Code translation with compiler representations,” arXiv preprint arXiv:2207.03578, 2022.

[2] B. Roziere, M.A. Lachaux, L. Chanussot, and G. Lample, “Unsupervised translation of programming languages,” Advances in neural information processing systems, vol.33, pp.20601–20611, 2020.

[3] E. Ilyushin and D. Namiot, “On source-to-source compilers,” International Journal of Open Information Technologies, vol.4, no.5, pp.48–51, 2016.

[4] A. Bastidas Fuertes, M. Pérez, and J. Meza Hormaza, “Transpilers: A systematic mapping review of their usage in research and industry,” Applied Sciences, vol.13, no.6, p.3667, 2023.

[5] I. Joseph, I.P. David, and E.J. Garba, “Software framework of an all-in-one transpiler for development of wora applicatons,” ScienceOpen Preprints, 2022.

[6] A.B. Fuertes, M. Pérez, and J. Meza, “nmorph framework: An innovative approach to transpiler-based multi-language software development,” IEEE Access, vol.11, pp.124386–124429, 2023.

[7] 杉坂純一郎, “Sugisaka/Aqualis”, https://github.com/Sugisaka/Aqualis.

[8] 芝山駿介, “erg-lang/erg”, https://github.com/erg-lang/erg.

[9] Nicolas Cannasse, “Haxe Foundation”,https://github.com/HaxeFoundation.

[10] 荒井省三, いげ太, 実践 F♯ 関数型プログラミング入門, 株式会社技術評論社, 2011.

[11] Microsoft, “Microsoft Learn/.NET/F♯ ドキュメント”, https://learn.microsoft.com/ja-jp/dotnet/fsharp/.