高水準動的型付け言語間FFIにおける Type Description Helperの構築

池崎 翔哉[1,a)]　山崎 徹郎[1,b)]　千葉 滋[1,†1,c)]

概要：他言語インターフェース（FFI）はある言語から他の言語のライブラリを使用するための仕組みである。プログラミング言語が実用的であるために、既に他言語で書かれたライブラリ資産を使用可能であることが重要である。近年では豊富な型を有する動的型付け言語にライブラリ資産が溜まってきているため、このような言語との FFI が望まれている。しかし型が豊富であるがゆえに、ホスト言語とライブラリ言語間の型変換規則が複雑なものとなり型変換規則の記述量が多くなってしまうという問題がある。本発表は Type Description Helper という型変換規則の半自動的な導出器を提案する。この Type Description Helper を用いることでユーザーが記述する必要のある型変換規則の量をおさえることができる。Type Description Helper による導出は不完全であるため、一部の導出できない型変換規則を手動で記述する必要がある。Type Description Helper による型変換規則の導出には Log-based 法と Type-inference-based 法の二種類の方法がある。Log-based 法は他言語関数呼び出し時のログから動的に型変換規則を導出する。Type-inference-based 法はソースコードを静的に解析することで型変換規則を導出する。また、この静的解析によって明らかに適用不可能な型の引数を検出することも可能である。本発表では具体例として Python 及び Euslisp の 2 言語を選定し、この間の FFI を作成すると共に Type Description Helper の実装を行い、その有用性を確認した。

Building Type Description Helper in FFI between High-level Dynamically Typed Languages

Ikezaki Shoya[1,a)]　Yamazaki Tetsurou[1,b)]　Chiba Shigeru[1,†1,c)]

Abstract:
A foreign function interface (FFI) is a mechanism that enables a programming language to use libraries written in another foreign language. The FFI is important since the language that does not have access to the rich libraries that already exist is not considered practical. Recently, the FFI between dynamically typed languages that have various types is required because a great number of useful libraries are written in those languages. However, it takes a high cost to describe the rule of type conversion between host and library languages since the rule of it is complex on account of type-richness. This presentation proposes Type Description Helper that derives the rule of type conversion semi-automatically. The Type Description Helper reduces the amount of the rule of type conversion that users have to write. As the rule of type conversion that is derived by Type Description Helper is incomplete, users have to write the rule of type conversion that has not yet derived. Type Description Helper derives the rule of type conversion in two ways: log-based approach and type-inference-based approach. The log-based approach derives the rule of type conversion dynamically from the log of foreign function calls. The type-inference-based approach derives the rule of type conversion by analyzing the source code statically. At the same time, it can detect some invalid arguments that cannot be applied by foreign function. In this presentation, we choose Python and Euslisp as an example and implement FFI and Type Description Helper, then checked that it is useful.

1. はじめに

ある言語が他言語で書かれたコードを関数単位で使用できるようにする機能のことを FFI（Foreign Function Interface）という。プログラミング言語が実用的であるためには、その言語が豊富なライブラリ資産を使用可能であることが重要である。洗練されたライブラリ資産の再利用はその分のコードを書く手間を削減し開発コストを下げるとともに、不要なバグを埋め込む危険性も減少させる。FFI により、あるプログラミング言語が同言語で書かれたライブラリ資産だけでなく他言語のライブラリを使用できるようになる。ホスト言語とライブラリ言語の間で値の参照渡しをできるようにしたり、その言語の有する関数やクラスについての情報をやりとりしたりできるように実装するのは多少複雑ではあるが、プロキシパターン [1] やリフレクション [2] といった既存の手法を用いて実装可能であると知られている。

従来は C 言語との FFI が重要であると考えられてきた。システムコールをはじめとした様々なライブラリ資産が溜まっている言語は C 言語であったので、C 言語の機能を他のホスト言語から使用できるようにすることは、ホスト言語がアクセス可能なライブラリ資産の質と量をあげる上で重要であった。そのため、Java, Python, Ruby, Rust といった現在幅広く用いられている言語では、C 言語との FFI が開発されている。

一方で、近年では C 言語以外の豊富な型を持つ動的型付け言語にも多様なライブラリ資産が溜まっており、それらとの FFI も重要になっている。例を挙げると、Python は NumPy[3], SciPy[4], PyTorch[5], TensorFlow[6] など数多くの強力なライブラリを有する。ROS (Robot Operating System) [7] の開発言語のひとつでもあるようにロボティクスの場でも広く用いられている。また、Lisp 言語の一つであり効率的なロボット開発のためのプログラミング言語である EusLisp[8] は、直感的に逆運動学を解くことのできるライブラリや、HRP2 といったロボットの制御ライブラリ群 [9][10] も持ち合わせている。こうした言語の有するライブラリの再利用が望まれている。

一般にある言語が他言語の関数を使用する際には型変換が必要となるが、この型変換規則の記述量が多くなってしまい、問題である。他言語関数使用の際 FFI はある言語のデータを他言語へと送るが、言語間で有する型は異なるため、同じデータでも両言語でそれを表現する型は異なる。そのため型同士の対応付けである型変換が必要となる。このとき、対象言語が共に豊富な型を有する場合、型同士の対応関係が多対多となる。豊富な型の一例として、今回の実装のターゲットでもある Python, Euslisp 両言語についてシーケンス型に対応する型を図 1 に示す。このような言語の FFI においては。型に対するデフォルトの変換規則を用意しておくだけでは対応しきれない。デフォルトの型変換の規則から外れて型変換を行いたい事例が多発してしまう。ユーザーは型変換規則の記述を静的ないし動的に行う必要があるが、複雑な型変換をユーザー任せで毎回記述させるのは親切ではない。

本研究では、そのような言語間の型変換規則の半自動的な導出器である Type Description Helper を提案する。これは半自動的に型変換規則を導出するため、導出できた分だけ必要な記述量は減る。これが導出可能な型変換規則は値に対するものと関数に対するものの 2 種類である。この Type Description Helper によりユーザーが記述する必要のある型変換規則の量を減らすことができると共に、明らかに適用不可能な型の引数の検出も可能になる。他言語関数の呼び出し時にホスト言語のオブジェクトをそのまま渡すかのような引数記述が可能になり、一部の型エラーは他言語関数の実行前にホスト言語側のみで発見することが可能になる。

提案手法を試すにあたりホスト言語には Python を、そこから呼び出す他言語には Euslisp を選定し、従来型の FFI および提案手法の型マッピングの記述インターフェースを実装した。これらは両言語共にオブジェクト指向でありプログラミングパラダイムは大きく外れることがなく、共に動的型付け言語であり互いによく似た言語であるが、型が豊富でありその FFI は上述した問題を抱えている。また実際に、Euslisp を Python から使いたいという要望は実際に我々の周囲のロボット研究グループの中にあり、実用上の意義もある。

以降、第 2 章では我々の動機を、第 3 章では提案手法を示す。第 5 章では関連研究を示し、最後に第 6 章ではまとめと今後の展望について記す。

2. 従来の型変換の規則の記述法と問題点

ホスト言語とライブラリ言語で有する型が異なる場合、FFI は型変換を行う。この型変換の規則は明らかではないため、ユーザーが型変換の規則を記述する必要がある。既存の型変換の規則を記述する方法としては値に対して型変換の規則を記述する手法、関数に対して型変換の規則を記述する手法の 2 つがあるが、いずれも記述量が多くなってしまうという問題がある。本章ではこれらの手法およびその具体例をそれぞれ説明し、問題点を示す。最後に実際にこれら手法を全て用いて実装したときの様子を具体的に見ていく。

[1] 情報処理学会
IPSJ, Chiyoda, Tokyo 101–0062, Japan
[†1] 現在，東京大学大学院情報理工学系研究科
Presently with Graduate School of Information Science and Technology, University of Tokyo
[a)] ikezaki@csg.ci.i.u-tokyo.ac.jp
[b)] yamazaki@csg.ci.i.u-tokyo.ac.jp
[c)] chiba@acm.org

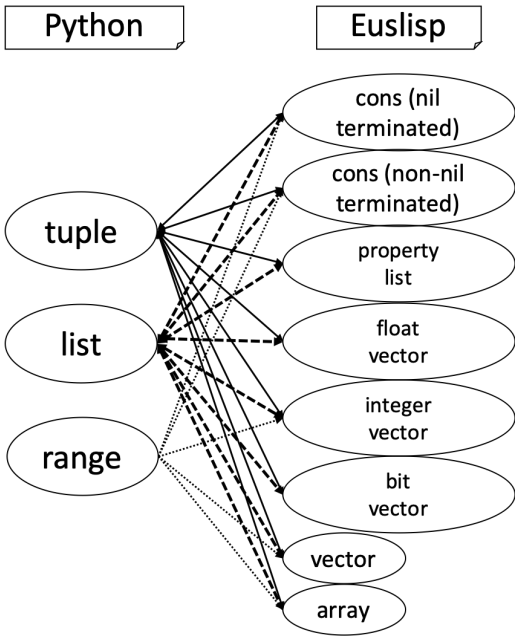

図 1　Python と Euslisp のシーケンス型とそのマッピング

ソースコード 1　Python の 3×3 list を用いて、C++言語の double 型 3 × 3 行列を要素にもつ vector に変換する例

```
1  translate_1 = [[1.0, 2.0, 3.0],
2                 [4.0, 5.0, 6.0],
3                 [7.0, 8.0, 9.0]]
4  translate_2 = [[1.0, 0.0, 0.0],
5                 [0.0, 1.0, 0.0],
6                 [0.0, 0.0, 1.0]]
7  translate_3 = [[0.866, -0.5, 1.73],
8                 [0.5, 0.866, 1.0],
9                 [0.0, 0.0, 1.0]]
10
11 cp_vec = Vector(
12 translate_1.data_as(cp_array((cp_double *
       3) * 3)),
13 translate_2.data_as(cp_array((cp_double *
       3) * 3)),
14 translate_3.data_as(cp_array((cp_double *
       3) * 3)),
15 )
```

2.1　値に対する型変換の規則の記述

値に対する型変換規則の記述とはユーザーがホスト言語の各データに対して変換されてほしいライブラリ言語の型情報を適宜付与するというものである。ホスト言語とライブラリ言語の型が往々にして多対多対応していることが問題であるため、ライブラリ言語の型と一対一対応する型をライブラリ言語側に用意しておけばよい。こういった対応する型が明らかなオブジェクトしか他言語関数に渡せないことにすれば他言語側で適切に関数実行が可能である。

値に対する型変換はデフォルトの型変換の規則を補完する形で、ユーザーが必要に応じて動的に値に対し行う。ライブラリ言語側にホスト言語の型と一対一対応する型が存在する場合はその型へ自動的に変換を行う。ライブラリ言語側にホスト言語の型と対応する型が複数存在する場合はそのうちの一つへ自動的に変換を行う。しかしながら、このデフォルトの型変換の規則から外れて他の候補の型へと変換を行いたい場合は、データに対してユーザーが毎回明示的に型変換を行う必要がある。

この値に対する型変換の規則の記述の具体的な説明のため、Python と C++の仮想の FFI を考える。Python と C の FFI である ctypes のように、単純な整数や文字列に対するデフォルトの型変換規則を設定することは可能であるが、それ以外のデータ型を使用する場合型変換規則をユーザーが書く必要があるだろう。ソースコード 1 に Python の 3 × 3 list を用いて、C++言語の double 型 3 × 3 行列を要素にもつ vector に変換する場合の仮想的な API を示す。1 行目から 9 行目にて Python の 3×3 list を作成している。12, 13, 14 行目はそれぞれ Python の list を C++の array に変換している。11 行目から 15 行目でそれらを初期値としてもつような vector を作成している。cp_vec は C++上の vector オブジェクトを指す Python の変数である。

この値に対する型変換の規則の記述手法には、関数呼び出し時にユーザーが柔軟に型の指定を行うことができるという利点がある一方で、豊富な型を有する言語間の FFI の場合、型変換の規則の記述量が多くなるという問題がある。ソースコード 1 の例をとっても一つの他言語のデータを作るだけでもこれだけ記述が必要になっている。一つの他言語関数に渡す値は複数あることが多い上、何度も他言語関数を呼び出すコードを書くことを考えると、一つ一つのホスト言語のデータに対して変換規則を記述するのは大変であると言える。

2.2　関数に対する型変換の規則の記述

関数に対する型変換規則の記述とは、使用する全関数について型変換の規則をユーザーがあらかじめ記述しておくという手法である。関数ごとに引数として渡されたホスト言語のデータをライブラリ言語のどの型へと変換するかについて、ユーザーが静的ないし動的に指定する必要がある。ほとんどの関数では、同じ引数なら異なる値が渡されたとしても同じ変換規則で変換を行ってよい。この観察に基づくと、関数の引数ごとに変換規則を指定し、呼び出しごとに自動的に変換を行うようにすることで型変換規則の記述を減らすことが可能になる。

具体例を挙げると、OMG（Object Management Group）により仕様が策定された CORBA（Common Object Request Broker Architecture）[11] という技術は、関数に対する型変換の規則の記述を行う FFI の一種である。CORBA は実際には様々な言語の組み合わせの FFI の構築を容易にするため、ホスト言語の型とライブラリ言語の型を直接

変換するのではなく、ある共通の中間言語 IDL（Interface definition language）を介して型変換を行う。その結果ホスト言語の型と ID の型 L、IDL の型とライブラリ言語の型という形でデータの型変換が行われ、ホスト言語から他言語関数を使用することができる。CORBA を使用する際、ユーザーは使用する他言語関数ごとに C 言語のプロトタイプ宣言のようなものを静的に記述する必要がある。ここで記述される型は IDL の型である。他言語関数の型は記述されているため、このプロトタイプ宣言をみれば CORBA のシステムが引数、戻り値について型変換を行えるというものである。これは関数に対して型変換規則の記述を行っているとみなせる。

図 2 に CORBA のアーキテクチャーの模式図を示す。クライアントコード、サーヴァントコードは任意の CORBA 対応言語（C, C++, Java など）で書かれたアプリケーションコードである。各アプリケーションコードでは IDL とのインターフェースが記述されている必要がある。各関数に対する型変換規則が記述された IDL ファイルをもとに IDL コンパイラが、ホスト言語側、ライブラリ言語側で型変換および通信を担うコード（Proxy, Skelton）を生成する。重要な点は関数に対して型変換規則を記述しさえすれば、ホスト言語から他言語関数にデータを送れるようになるということである。

以下、具体的に C++で書かれたクライアントコードから、挨拶を返す Java で書かれたサーヴァントコードを使用する例を見ていく。ソースコード 2 に OMG IDL を用いたリモートインターフェースの記述の例を、ソースコード 3 に Java による IDL インターフェースの実装の例を示す。ソースコード 2 ではソースコード 3 中の Java の各関数に対して、プロトタイプ宣言のような形で変換されるべき IDL の型を記述している。Java の package, signature interface, java.lang.String といった型がそれぞれ IDL の module, interface, string といった型に対応していることがわかる。このように記述することで、同様にしてインターフェースが記述された CORBA オブジェクトが sayHello メソッドを呼び出して結果を受け取ったり、shutdown メソッドを呼び出して手続きを終了させたりすることができるようになる。

この手法には一度宣言時に記述しさえすれば他言語関数の使用の際に型変換規則の記述の必要がなくなり、値に対する型変換の記述よりは記述量が減るという利点がある。一般に使用する他言語関数の数は他言語関数に送り込む値の数よりは少ないと言える。それでも、豊富なライブラリ資源を再利用する目的のためには、やはり多くの型変換の規則の記述が必要になるという問題がある。

2.3 既存手法を用いた Python/Euslisp FFI の実装例

これまでの節で述べた 2 つの既存手法を用いて型変換の規則の記述を行う場合の実装を、Python/Euslisp FFI を例として具体的に見ていく。そして両手法とも記述量が多い問題があるということを改めて確認する。

Python から Euslisp へのデフォルトの型変換の規則を表 1 に、Euslisp から Python へのデフォルトの型変換の規則を表 2 に示す。一般に FFI における引数や戻り値の扱い方として、コピー渡しと参照渡しの 2 つが存在するが、今回は引数に関してはコピー渡しを、戻り値に関しては参

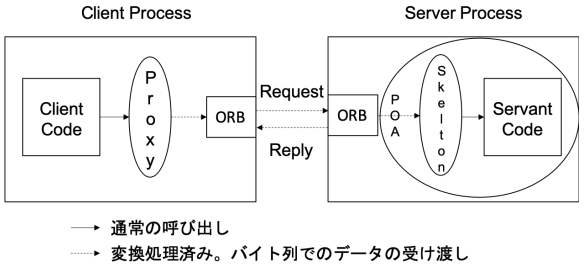

図 2 CORBA のアーキテクチャー

ソースコード 2 Java オブジェクトに対する IDL を用いたインターフェースの記述

```
1  module HelloApp {
2      interface Hello {
3          string sayHello(string);
4          oneway void shutdown();
5      };
6  };
```

ソースコード 3 Servant プログラムの抜粋 (Java)

```
1  // https://docs.oracle.com/javase/jp/1.5.0/
       guide/idl/tutorial/GSserver.html
2  import HelloApp.*;
3  // 本来org.omg.CosNamingやorg.omg.
       CORBAなどのモジュールもimportする
4
5  class HelloImpl extends HelloPOA {
6    private ORB orb;
7    public void setORB(ORB orb_val) {
8      orb = orb_val;
9    }
10   public String sayHello(String name) {
11     return "\nHello, " + name + "!\n";
12   }
13   public void shutdown() {
14     orb.shutdown(false);
15   }
16 }
17
18 public class HelloServer {
19   public static void main(String args[]) {
20   // 初期化、参照の取得などを本来行う
21   // クライアントを待つ処理を行う
22   }
23 }
```

ソースコード 4 Client プログラムの抜粋 (C++)

```
#include "Hello.h"
// 本来 OB/CORBA.h や OB/Cosnaming.h などのヘッダーも include する
int main(int argc, char** argv)
{
  CORBA::ORB_var orb;
  try {
    // 初期化、参照の取得などを本来行う
    // "manager" へのメソッドコールは Java プロセス中の対応するオブジェクトへのメソッドコールとなる
    CORBA::String_var response_string;
    CORBA::Char* name;
    name = CORBA::string_alloc(10);
    strcpy(name, "PRO2019-5")
    response_string = manager->sayHello(name);
    cout << response_string.in() << endl;
    // "Hello, PRO2019-5 !" と表示される

    manager->shutdown();
  } catch(const CORBA::Exception& e) {
    cerr << e << endl;
  }
  // 終了処理を本来行う
  return 0;
}
```

照渡しを基本としている。ライブラリ言語である Euslisp が関数型言語でもありプリミティブを引数にとる関数が多いことから、引数として Euslisp のオブジェクトに対するプロキシーオブジェクトが来た場合を除いて、対応する Euslisp のオブジェクトをコピー生成することにしている。ここで型変換規則が必要となる。一方、他言語関数呼び出しの戻り値については、同一性の問題から数値および真偽値以外はその Euslisp オブジェクトに対するプロキシーオブジェクト [1] を作成して返すようになっている。このプロキシーオブジェクトに対しても Python の型を指定し、その型のもつインターフェースを継承させることは可能であるが、今回は行っていない。この場合、戻り値に対する型変換規則は与えないでよいが、一般にはプロキシーオブジェクトを使用する場合も型変換規則を与える必要がある。なお、今回この Euslisp のオブジェクトへのプロキシーオブジェクトを対応する Python のオブジェクトへと変換する場合、to_python メソッドを呼ぶことで変換が行われる。

初めにリストをうけとり逆順にして返す Euslisp の reverse 関数を Python から使用することを考える。Python 側のコードはソースコード 5 のようになる。第 2 行で他言語関数の Python バインドである reverse が Python の list を引数として受け取っている。この他言語関数の Python バインドである list_concatenate は、引数として Python オブジェクトを受け取った時、それに対応する Euslisp オブジェクトをコピー生成し、関数実行のための適切な S 式を Euslisp 側に送り込むことを内部で行なっている。デフォルトの型変換の規則によれば Python の list はコピーされ Euslisp の nil-terminated list へと自動的に変換されるため、このような場合は引数として Python の list を渡すだけで Euslisp 側で正しく引数を受け取り、型エラーを起こすことなく関数の処理が完了する。reverse 関数コールの戻り値は Euslisp の cons オブジェクトのプロキシーである。ここでは Python の list に変換するために to_python なるメソッドを呼んでいる。なお、リフレクション [2] を用いて Euslisp 側のシンボルテーブルを Python 側のパッケージオブジェクトの属性として自動的に生成しているため、Python ライブラリに対するメソッドコールのような記述での他言語関数の呼び出しが可能になっている。

次にソースコード 6 のような 2 つの行列を引数にとり、それらの 0, 0 成分の和を求める Euslisp の関数を Python から使用することを考える。デフォルトの型変換の規則からでは Euslisp の array は生成できないため、ユーザーが明示的な型変換の規則の記述を行う必要がある。第 2.1 節や第 2.2 節で示したように、型変換の規則の記述には値に対する記述と関数に対する記述がある。前者は REPL を用いた開発時など、実行時に型の記述を指定したい際に便利である。後者は他言語関数を複数回呼ぶ際や、宣言的に型の記述をしたい際に便利である。

前者のユーザーによる値に対する型変換の規則の記述の API を表 3 に示す。第一列のコンストラクタを第二列の型の Python のオブジェクトに適用すると、そのデータがコピーされ第四列に示した型の Euslisp のオブジェクトが生成され、それを参照する第三列に示したプロキシーオブジェクトが返される。前者の方法で Python から Euslisp の sum-of-0-0-element-of-arr 関数を呼び出すコードはソースコード 7 のようになる。引数部の Python の list に対し EusArray なる型コンストラクタの記述を与えることで型変換を行なっている。この型変換コンストラクタは引数として第二列のような Python オブジェクトを受け取った時、それに対応する Euslisp オブジェクトを生成し、その Euslisp オブジェクトのプロキシーオブジェクトを Python 側で生成して返している。その結果 Euslisp 側では正しく array として引数を受け取り、型エラーを起こすことなく関数の処理が終了する。値に対する型変換の規則の記述を行う手法では型コンストラクタを何度も呼ぶ必要が出てしまう。他言語関数の引数は複数あることもあるし、他言語関数を複数回呼ぶことを考えると型変換規則のユーザー記述量は多い。

後者の方法で Python から Euslisp の sum-of-0-0-element-of-arr 関数を呼び出すコードはソースコード 8 のようになる。冒頭の set_params という関数は他言語関数に対して型変換規則の登録を行う関数である。このようにユーザーが

表 1 Python から Euslisp へのデフォルトの型変換規則

Object (Python)	Object (Euslisp)	Description
int	integer	
float	float	
None, False	nil	
True	t	
string	string	
list, tuple, xrange	list	
dict	hash-table	
any other object	N/A	TypeError

ソースコード 5 Euslisp の reverse 関数を Python から呼び出す例

```
1  # LISP は Euslisp の標準関数が入ったパッケージ
2  LISP.reverse([1,2,3]).to_python()
3  # => [3,2,1]
4
5  type(LISP.reverse([1,2,3]))
6  # => <class 'pyeus.cons'>
7  # proxy object that points to cons in Euslisp
```

関数呼び出し前に、関数に対し型変換の規則を 1 度動的に記述すれば、以降はその通りに引数が型変換されて、結果の受け取りができるようになる。第 3 行以降で他言語関数の引数として Python の list が渡されているが、内部的に array へと型変換が行われてから Euslisp 側へ渡されるため、この場合も型エラーを起こすことなく関数の処理が終了する。この例において値に対する型変換規則の記述のときよりも記述量が減っていることが見てとれる。しかしながら、ソースコード 9 に示したような、内部で他言語関数を複数使用するような関数を使うことを考える。使用する他言語関数が増えれば増えるほど型変換規則の記述が必要となるため、ユーザーが記述する必要のある型変換規則の量は依然として多いと言える。

ソースコード 6 2 つの配列の (0, 0) 成分の和を求める Euslisp 関数の例

```
1  (in-package  "EUS_LIB")
2
3  (defun sum-of-0-0-elment (arr1 arr2)
4    (+ (aref arr1 0 0) (aref arr2 0 0)))
```

3. Type Description Helper

本章では型変換規則の半自動的な導出器である Type Description Helper を提案する。この Type Description Helper は一部の型変換規則を半自動的に導出するため、記述が必要な型変換規則の量を減らすことができる。Type Description Helper は値に対する型変換規則を導出する

ソースコード 7 Euslisp の sum-of-0-0-element 関数を、Python から値に対する型変換規則の記述とともに呼び出す例

```
1  result1 = EUS_LIB.sum_of_0_0_element(
       EusArray([[1,2],[3,4]]), EusArray
       ([[5,6],[7,8]]))
2
3  result2 = EUS_LIB.sum_of_0_0_element(
       EusArray([[9,10],[11,12]]), EusArray
       ([[13,14],[15,16]]))
4
5  result3 = EUS_LIB.sum_of_0_0_element(
       EusArray([[17,18],[19,20]]), EusArray
       ([[21,22],[23,24]]))
```

ソースコード 8 Euslisp の sum-of-0-0-element 関数を、Python から関数に対する型変換規則の記述とともに呼び出す例

```
1  set_params(EUS_LIB.sum_of_0_0_element,
       EusArray, EusArray)
2
3  result1 = EUS_LIB.sum_of_0_0_element
       ([[1,2],[3,4]], [[5,6],[7,8]])
4
5  result2 = EUS_LIB.sum_of_0_0_element
       ([[9,10],[11,12]], [[13,14],[15,16]])
6
7  result3 = EUS_LIB.sum_of_0_0_element
       ([[17,18],[19,20]], [[21,22],[23,24]])
```

Log-based 法と、関数に対する型変換規則を導出する Type-inference-based 法の 2 種類を組み合わせて導出を行う。これらは既存手法に対し表 4 のような関係にある。第 22.1 節で説明した値に対する型変換規則の記述を自動化したものが Log-based 法であり、2.2 節で説明した関数に対する型変換規則の記述を自動化したものが Type-inference-based 法である。どちらから得られる型変換規則も完全なものではないため、一部の導出できない型変換規則はユーザーが手動で記述する必要がある。

3.1 Log-based な型変換規則の導出

Log-based 法は他言語関数呼び出し時のログから動的に型変換規則を導出する。これはユーザーが関数の引数部で型変換を行っていたら関数がそれを記憶し、以降は特に記述がない限りその記憶の通りに型が変換されて他言語関数を呼び出し、結果の受け取りができるというものである。

このシステムの模式図を図 3 に示す。図中左側がホスト言語の環境で右側がライブラリ言語の環境である。使用したい他言語関数のプロキシに対し関数呼び出しを行なった時の様子を表している。図中の mapping table なる関数オブジェクトの属性は型マッピングの時に適用すべき型コン

表 2 Euslisp から Python へのデフォルトの型変換規則

Object (Euslisp)	Class of the object (Python)	after to_python conversion
integer	int	N/A
float	float	N/A
nil	None	N/A
t	True	N/A
symbol	class 'pyeus.symbol'	string
function symbol	class 'pyeus.compiled_code'	string
string	class 'pyeus.string'	string
(non-nil-terminated) list	class 'pyeus.cons'	list
property list	class 'pyeus.cons'	list
list	class 'pyeus.cons'	list
integer-vector	class 'pyeus.integer_vector'	list
float-vector	class 'pyeus.float_vector'	list
bit-vector	class 'pyeus.bit_vector'	list
vector	class 'pyeus.vector'	list
array	class 'pyeus.array'	list
hash-table	class 'pyeus.hash_table'	dict
pathname object	class 'pyeus.pathname'	str
any other instance of 'MyClass'	class 'pyeus.MyClass'	N/A
class object (not an instance)	class 'pyeus.metaclass'	N/A

表 3 Python から Euslisp への明示的な型変換に用いるコンストラクタ

Constructor (Python)	Args (Python)	Class of the object (Python)	Object (Euslisp)
EusSym	string	class 'pyeus.symbol'	symbol
EusFuncSym	string	class 'pyeus.compiled_code'	function symbol
EusStr	string	class 'pyeus.string'	string
EusCons	list, tuple, xrange	class 'pyeus.cons'	(non-nil-terminated) list
EusPlist	list, tuple	class 'pyeus.cons'	property list
EusList	list, tuple, xrange	class 'pyeus.cons'	(nil-terminated) list
EusIntVec	list, tuple, xrange	class 'pyeus.integer_vector'	integer-vector
EusFloatVec	list, tuple	class 'pyeus.float_vector'	float-vector
EusBitVec	list, tuple	class 'pyeus.bit_vector'	bit-vector
EusVec	list, tuple, xrange	class 'pyeus.vector'	vector
EusArray	list, tuple, xrange	class 'pyeus.array'	array
EusHash	dict	class 'pyeus.hash_table'	hash-table
EusPath	str	class 'pyeus.pathname'	pathname object

ストラクタを登録するためのフィールドである。まず明示的な値に対する型変換などにより proxy function が proxy object を引数に呼ばれた場合を考える。その場合、図中の logger がそのクラス情報を元に適切な型コンストラクタたちを mappping table に登録する。そして send sexpression ルーチンに処理が渡され、proxy object を引数に foreign function の関数呼び出しがなされる。次に proxy function が proxy object 以外のオブジェクトを引数に呼ばれた場合を考える。その場合、図中の mapper が mapping table を見に行く。型コンストラクタが登録されていなかったならば既存手法のようにデフォルトの型変換規則を適用する。登録されていたならばその型コンストラクタを各引数に対して適用し、戻り値の proxy object で引数を置き換え、send sexpression ルーチンに処理が渡される。前述の場合と同様に以降は proxy object を引数に foreign function の関数呼び出しがなされる。

これにより一度ユーザーが関数の引数部で型変換を行ったら、以降はそのとおりに型変換が自動的に行われる。ユーザーが記述する必要のある型変換規則の量は削減される。REPL 駆動で開発やデバッグを行い動的な型変換規則の記述が行いたい場合に特に便利である。

3.2 Type-inference-based な型情報の記述

Type-inference-based 法はソースコードを静的に解析することで型変換規則を導出する。この方法では FFI ライブラリが型推論をもとに、考えられる型変換規則の候補を自動的に絞る。絞られた候補を元に型変換をするほか、ユーザーから明示的な型変換規則の指定が与えられた際にその引数の型の正当性を検証する。このシステムの模式図を図 4 に示す。mapping table の登録を行うものとして

表 4　型変換規則の記述の自動性とその型変換規則の記述先

型変換規則の記述の自動性 ＼ 記述先	値	関数
手動	2.1 節で示した手法	2.2 節で示した手法
自動	Log-based 法	Type-inference-based 法

ソースコード 9　内部で複数の Euslisp 関数を使用するような関数を、Python から関数に対する型変換の指定とともに呼び出す例

```
1  # LISP は Euslisp の標準関数が入ったパッケージ
2  # index が複数入った2次元 vector と array の2引数をうけとり、その index の指す array の要素の和を求める
3  def sum_indirect(arr, ind_vec):
4    sum = 0
5    dims = LISP.array_dimensions(arr)
6    dim_x = LISP.car(dims)
7    dim_y = LISP.car(LISP.cdr(dims))
8    num = LISP.length(ind_vec)
9    for i in range(num):
10     indices = LISP.svref(ind_vec, i)
11     ind_x = LISP.car(indices)
12     ind_y = LISP.car(LISP.cdr(indices))
13     if ind_x < dim_x and ind_y < dim_y:
14       sum += LISP.aref(arr, ind_x, ind_y)
15     else:
16       raise IndexError
17   return sum
18
19 # sum_indirectを使用
20 set_params(LISP.array_dimensions, EusArray)
21 set_params(LISP.svref, EusVec)
22 set_params(LISP.aref, EusArray)
23
24 sum_indirect([[1,2,3], [4,5,6], [7,8,9]],
       [[0,0], [1,1], [2,2]])
25 # => (1+5+9=) 15
26 sum_indirect([[1,2,3], [1,2,3], [1,2,3]],
       [[0,0], [1,0], [2,0]])
27 # => (1+1+1=) 3
```

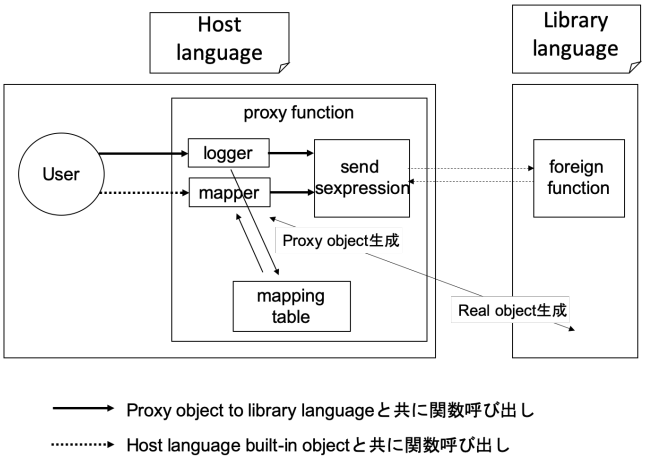

図 3　Log-based な型変換規則の導出の模式図

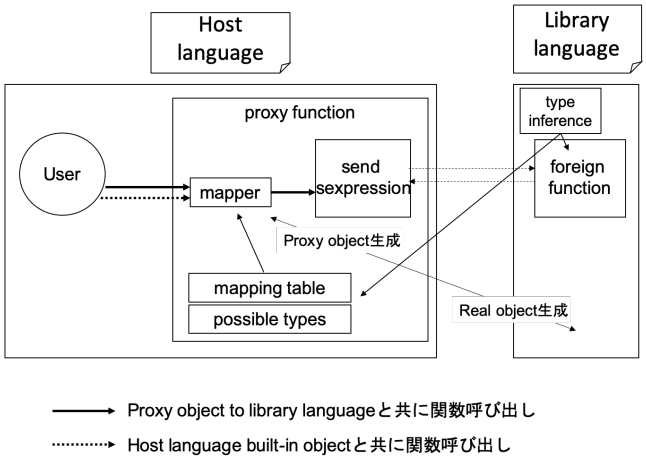

図 4　Type-inference-based な型変換規則の導出の模式図

logger の代わりに type inference が書かれている。FFI ライブラリから他言語のライブラリをロードする際、この型推論器は各他言語関数のボディの情報から、関数引数の型として可能性の残されている選択肢を取得する。型を絞り込むことができたら、その他言語関数に対応する proxy function の mapping table に登録を行う。絞りきれずともこの選択肢を関数の属性 possible types として保持しておく。mapper の基本的な挙動は変わらないが、proxy object が渡された場合はその型について、proxy object 以外の オブジェクトが渡された場合はデフォルトの型変換規則適用後の型について possible types とマッチするか検証を行う。これにより、ライブラリ言語で実行をしたら型エラーを起こしてしまうようなユーザーによる無効な型変換規則の記述に対し、ホスト言語側で事前に型エラーとして弾くことができる。以降は前節同様 foreign function の呼び出しがなされる。

これにより引数の型が明確な他言語関数に関しては型変換が自動的に行われる。ユーザーが記述する必要のある型変換規則の量は削減される。前節の手法は型変換規則の記述量を削減することはできるが、やはりその型変換規則自体の信頼性に関してはユーザーに任せていた。対しこの手法は記述量の削減に加えて型変換規則の信頼性の問題に対しても部分的に取り組むことができる。

4. Python/Euslisp FFI における提案手法の実装

本章では前章の提案手法を Python/Euslisp FFI に実装した場合の挙動を具体的なコードと共に見ていく。そしてこれら Type Description Helper を補助的に用いることで

ソースコード 10 Euslisp の sum-of-0-0-element 関数を、Python から Log-based な型変換規則の導出と共に呼び出す例

```
1  result1 = TEST.sum_of_0_0_element(EusArray
       ([[1,2],[3,4]]), EusArray
       ([[5,6],[7,8]]))
2  # type mapping is registered automatically
3
4  result2 = TEST.sum_of_0_0_element
       ([[9,10],[11,12]], [[13,14],[15,16]])
5
6  result3 = TEST.sum_of_0_0_element
       ([[17,18],[19,20]], [[21,22],[23,24]])
```

型変換規則が半自動的に導出されることを確認していく。

ソースコード 6 に挙げた Euslisp の sum-of-0-0-element 関数を、Log-based な型変換規則の導出と共に使用する場合のコードをソースコード 10 に示す。すでに見たように 1,2 行目の EusArray といった型コンストラクタはライブラリ言語である Euslisp に対応するオブジェクトを生成し、そのプロキシーオブジェクトを返す（この例でいうと#a((1 2) (3 4)), #a((5 6) (7 8)) といったオブジェクトを生成し、それと結びついたプロキシークラス class 'pyeus.array' のインスタンスを作成して返す)。1, 2 行目でプロキシー関数はプロキシーオブジェクトを引数に呼ばれるため図 3 の logger が TEST.sum_of_0_0_element 関数の mapping table 属性に型コンストラクタ EusArray, EusArray が登録される。以降の 5, 6 行目、8, 9 行目ではホスト言語である Python のオブジェクトを引数にプロキシ関数呼び出しが行われているので 3 の mapper が mapping table を元に EusArray なる型コンストラクタを自動的に適用する。本来なら 7 のようにして毎回 EusArray なる型変換を書くところが、このように一度値に対する型情報付きで関数を呼べば、以降は変換可能な Python オブジェクトを渡せば自動的に変換されるようになる。なお、ジェネリックな関数を使用するケースなどでは同一関数を様々な引数の型で呼び出すことが考えられるが、そのような場合では再び手動で明示的な型指定が必要になる。

ソースコード 6 に挙げた Euslisp の sum-of-0-0-element 関数を、Type-inference-based な型変換規則の導出と共に使用する場合のコードをソースコード 11 に示す。FFI ライブラリがライブラリ言語である Euslisp のファイルをロードした時点で図 3 の型推論器が一連の他言語関数を対象に型推論を行う。sum-of-0-0-element 関数の引数 arr1, arr2 は共に aref 関数の引数となっており、aref 関数は array のみを引数にとる関数であるため、今回引数の型は共に array であると特定される。TEST.sum_of_0_0_element 関数の mapping table 属性に型コンストラクタ EusArray, EusArray が登録される。また、possible types 属性についても同様に登

ソースコード 11 Euslisp の sum-of-0-0-element 関数を、Python から Type-inference-based な型変換規則の導出と共に呼び出す例

```
1  # type mapping has already been registered
       automatically!
2  result1 = TEST.sum_of_0_0_element
       ([[1,2],[3,4]]), [[5,6],[7,8]]))
3
4  result2 = TEST.sum_of_0_0_element
       ([[9,10],[11,12]], [[13,14],[15,16]])
5
6  # result3 = TEST.sum_of_0_0_element(EusList
       ([[17,18],[19,20]]), EusList
       ([[21,22],[23,24]]))
7  # raises TypeError
```

録される。以降の 5, 6 行目ではホスト言語である Python の組み込みオブジェクトを引数にプロキシ関数呼び出しが行われているので 4 の mapper が mapping table を元に EusArray なる型コンストラクタを自動的に適用する。8, 9 行目ではリストのプロキシオブジェクトを引数にプロキシ関数呼び出しが行われているが、この引数は possible types 属性を見るに正当でないため、この時点でホスト言語側のみで TypeError が上げられる。本来なら 7 のようにして毎回 EusArray なる型変換を書くところが、型推論がうまくいくと変換可能な Python オブジェクトを渡せば最初から自動的に変換されるようになる。また、明らかに適用不可能な型の引数とともに他言語関数が呼ばれた場合、型エラーをホスト言語側のみで検知できるようになっている。なお型推論で型を絞りきれず、デフォルトの型変換規則から外れた型指定を行いたい場合は、手動で明示的な型指定が必要になる。

5. 関連研究

本研究では動的型付け言語間の FFI における型変換規則の半自動的な導出器である Type Description Helper を提案した。Type Description Helper によりユーザーが記述する必要のある型変換規則の量をおさえることができること、明らかに適用不可能な型の引数の検出が可能であることを確認した。このような動的型付け言語間の FFI に関する論文はごく僅かであるため、本章では静的片付け言語との FFI も含めて関連研究について述べることにする。

FFI を利用する際のユーザーの記述量削減に関する研究としては、静的型付け言語との FFI におけるグルーコードの自動生成が挙げられる。SWIG（Simplified Wrapper and Interface Generator）[12] は、C/C++コードと Tcl, Python, Perl といったスクリプト言語のバインディングの自動生成を行う。具体的には、ヘッダーファイルの情報とユーザーが与えるインターフェースファイルのみを元に、各 C/C++関数に対してアクセスするためのラッパー関数

を自動生成する。FIG[13] は C コードと Moby[14] 言語のバインディングの自動生成を行う。FIG は SWIG と似ているが、こちらは C のラッパー関数を生成するのではなくユーザー定義の型変換規則に基づき Moby コードを生成するため、データレベルでの互換性があるという特徴がある。このように型変換規則の記述は必要ではあるものの、使用する個々の関数に対する型変換規則の記述に関しては自動化がなされている。

FFI を利用する際のユーザーコードの検証に関する研究としては、静的型付け言語との FFI における静的解析・動的解析が存在する。前者の一例として OCaml/C FFI において型の検証を行うシステムである multi-lingual type inference system[15] が挙げられる。これは C の型で表現された OCaml の型と、OCaml の型で表現された C の型を有する型言語を定義することで、両言語間で完全な型情報の追跡を可能にしている。後者の一例として Java/C FFI において JVM の状態に関する制約・型に関する制約・リソースに関する制約が守られているか検証を行うプログラムである Jinn[16] が挙げられる。他言語関数呼び出しを行う Java の関数を、上記制約を適宜検証するラッパー関数に自動で置き換えることで網羅的な動的解析を可能にしている。

Chiba[17] は高水準言語間インターフェースに適したデザインとして、FFI に替わりコードマイグレーションを提案している。標準の処理系を用いた Ruby/Python FFI である PyCall[18] と埋め込みドメイン特化言語を組み合わせることで、Ruby コード中に Python ライクなコードブロックを記述して使用できるようになるとともにコードブロックのシンタックスチェックも行うことが可能となる。両言語のシンタックスを混同することに由来する他言語インターフェース部分でのエラーはつきものだが、そのようなユーザーコストを削減する試みである。本研究と着眼点は異なるが関連研究として位置付けることができるだろう。

動的型付け言語間の FFI の実装自体に関する研究は、Ramos ら [19][20] の Racket/Python FFI、Barrett ら [21] の PHP/Python FFI などがある。前者は Python のセマンティクス、組み込みデータタイプを Racket 上で全て実装することで、Racket プラットフォーム上での両言語の実行が可能となっている。後者は Python インタプリタの RPython 実装である PyPy[22] と PHP インタプリタの RPython 実装である HippyVM[23] を組み合わせることで、ソースコードエディター Eco 上での両言語の実行が可能となっている。どちらの研究も高速化を図るため独自の処理系、環境を作り FFI の構築を行なっているが、本研究では標準の処理系および環境での FFI の構築を対象としたため、このような構築手法はとらなかった。また、いずれもユーザーの記述コストや検証という観点からの取り組みはなされていない。

6. まとめと今後の展望

本研究では型が豊富な動的型付け言語間 FFI における型変換において、型変換規則の半自動的な導出器である Type Description Helper の提案を行った。対象言語の一例として Python, Euslisp を選定し、FFI および Type Description Helper の実装を行い、ユーザーによる型変換規則の記述の必要量の削減が可能であることを確認した。

今後の展望としては、本稿執筆時点では型推論機構を用いた半自動的な型変換規則の導出はまだ改善の余地があるため、型推論機構をより充実させる。ソースコードでのメソッドコールなどにも対応した型推論機構ができればより広範な型の推定が可能となるだろう。洗練された型推論機構を用いて、実在する Euslisp ライブラリ群に対して実験を行うことで、定量的な評価も行えるようになるはずである。

参考文献

[1] E. Gamma, R. Helm, R. Johnson, & J. Vlissides. (1994). Design Patterns. Addison-Wesley

[2] Smith, B. C. (1984, January). Reflection and semantics in Lisp. In Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (pp. 23-35). ACM.

[3] Travis Oliphant. (2006). NumPy. https://numpy.org/.

[4] Travis Oliphant, Pearu Peterson & Eric Jones. (2001). SciPy. https://www.scipy.org/.

[5] Adam Paszke, Sam Gross & Soumith Chintala and Gregory Chanan. (2016). PyTorch. https://pytorch.org/.

[6] Google Brain. (2015). TensorFlow. https://www.tensorflow.org/.

[7] Willow Garage and Stanford Artificial Intelligence Laboratory. (2007). https://www.ros.org/.

[8] Matsui, T., & Inaba, M. (1990). Euslisp: An object-based implementation of lisp. Journal of Information Processing, 13(3), 327-338.

[9] Okada, K., Ogura, T., Haneda, A., Kousaka, D., Nakai, H., Inaba, M., & Inoue, H. (2004, April). Integrated system software for HRP2 humanoid. In IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 (Vol. 4, pp. 3207-3212). IEEE.

[10] Okada, K., Ogura, T., Haneda, A., Fujimoto, J., Gravot, F., & Inaba, M. (2005, July). Humanoid motion generation system on hrp2-jsk for daily life environment. In IEEE International Conference Mechatronics and Automation, 2005 (Vol. 4, pp. 1772-1777). IEEE.

[11] Vinoski, S. (1997). CORBA: integrating diverse applications within distributed heterogeneous environments. IEEE Communications magazine, 35(2), 46-55.

[12] Beazley, D. M. (1996, July). SWIG: An easy to use tool for integrating scripting languages with C and C++. In TCL' 96,

[13] Reppy, J., & Song, C. (2006, October). Application-specific foreign-interface generation. In Proceedings of the 5th international conference on Generative programming and component engineering (pp. 49-58). ACM.

[14] Fisher, K., & Reppy, J. (1999, May). The design of a class mechanism for Moby. In ACM SIGPLAN Notices (Vol. 34,

No. 5, pp. 37-49). ACM.

[15] Furr, M., & Foster, J. S. (2005, June). Checking type safety of foreign function calls. In ACM SIGPLAN Notices (Vol. 40, No. 6, pp. 62-72). ACM.

[16] Lee, B., Wiedermann, B., Hirzel, M., Grimm, R., & McKinley, K. S. (2010). Jinn: synthesizing dynamic bug detectors for foreign language interfaces. ACM Sigplan Notices, 45(6), 36-49.

[17] Chiba, S. (2019, October). Foreign language interfaces by code migration. In Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (pp. 1-13). ACM.

[18] Kenta Murata. (2016). PyCall: Calling Python functions from the Ruby language. https://github.com/mrkn/pycall.rb.

[19] Ramos, P. P., & Leitão, A. M. (2014). Implementing Python for DrRacket. In 3rd Symposium on Languages, Applications and Technologies. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[20] Ramos, P. P., & Leitão, A. M. (2014, August). Reaching Python from Racket. In Proceedings of ILC 2014 on 8th International Lisp Conference (p. 32). ACM.

[21] Edd Barrett, Carl Friedrich Bolz, Lukas Diekmann, & Laurence Tratt. (2016). Fine-grained Language Composition: A Case Study. In 30th European Conf. on Object-Oriented Programming (ECOOP 2016), Vol. 56. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 3:1–3:27.

[22] (2007). PyPy. http://pypy.org/

[23] (2014). HippyVM. http://hippyvm.baroquesoftware.com/