特別研究報告書

OCaml用FFIライブラリCtypesの参照渡し関数への拡張

指導教員：五十嵐 淳 教授

京都大学工学部情報学科

佐藤 新太

2022年2月3日

OCaml 用 FFI ライブラリ Ctypes の参照渡し関数への拡張

佐藤 新太

内容梗概

プログラミングにおいて，他言語で記述された関数をバインドし，呼び出すことはよく行われる．この仕組みを *FFI (Foreign Function Interface)* と呼ぶ．FFI を利用して他言語の関数を使用するためには，その関数をバインドするためのプログラムが必要である．そのプログラムを**バインディング**と呼ぶ．バインディングは，引数や返り値のデータ表現や型の変換を行い，バインド元のプログラムからバインド先の関数の呼び出しを可能にする．

バインド先の言語とバインド元の言語において，型システムやデータ表現，実行環境などが異なる場合，バインディングの手動での記述は煩雑で間違えやすい．この問題の解決への１つのアプローチに，バインド先の関数型の記述を元にしたバインディングの自動生成がある．C に対する FFI を扱う OCaml のライブラリである Ctypes は，そのような機能を持つ．Ctypes は，OCaml のプログラムとして表現されるバインド先の関数の型の記述を元に，その関数をバインドして呼び出す仕組みを提供する．その仕組みを**結合メカニズム**と呼ぶ．Ctypes が提供している結合メカニズムの１つである**段階的バインディング**は，バインド先の関数型の記述を元に C と OCaml のバインディングを生成し，その生成されたバインディングを元に静的に他言語関数をリンクする．Ctypes を用いることで，OCaml と C の型システムやデータ表現の差異を考慮し，手動で記述するバインディングを減らすことが可能である．

本研究では，Ctypes を拡張することで，バインド先の関数型の記述に基づき機械的に生成できるバインディングを増やすことによる，ソフトウェアの堅牢性の向上を目指す．そのために，Ctypes を使用するオープンソースプロジェクトにおいて，手動で記述されているバインディングの例を調査した．この調査から得られた主な所見は次の通りである．(i) 参照渡しを用いた C の関数をバインドする際，その引数を OCaml では返り値として扱えるようにラッパー関数を実装する例がある．(ii) そのような引数のメモリの確保などを管理するためのモジュールを実装する例がある．(iii) 確保したメモリのサイズを検証する関数を用意し，バインディングの利用側でその関数を用いて検証する例がある．(iv)

C の関数の int 型の返り値を OCaml の例外に紐づけている例がある．この調査から，プログラミング言語の慣例の差異や受け渡すデータ間に満たしたい制約がある場合，依然として Ctypes を用いた上でバインディングを手動で記述する例が多いことが分かった．この調査結果を踏まえ，参照渡しを用いる関数をバインドする場合，その関数の型や確保するメモリのサイズの情報に基づいてメモリを確保し，OCaml ではその変数を返り値として扱えるような仕組みを提供するツールが有用だと判断した．

本研究では，Ctypes を拡張することで，バインド先の関数の型や確保するメモリのサイズの情報を元に C と OCaml のバインディングを機械的に生成するツール Cbuf を実装する．Cbuf では，バインディングの生成に必要な情報を記述するため，Ctypes でバインド先の関数型を記述する際に用いられる FOREIGN インターフェースを拡張する．Ctypes と同様に，関数型の記述において C のオブジェクト型や関数型は OCaml のプログラムとして表現される．Cbuf の設計は Ctypes の段階的バインディングに基づいており，関数型の記述に基づいて C と OCaml のバインディングの生成を行う．Cbuf によって生成されたバインディングは，メモリの確保やバインド先の C の関数の呼び出し，関数の引数や返り値の型変換などを行う．Cbuf の結合メカニズムは，Ctypes が標準で提供する結合メカニズムと同様に，FOREIGN を満たしたモジュールとして実装する．

同じ C の参照渡し関数に対する，Ctypes の段階的バインディングと Cbuf を使用したバインディングを比較し，提案手法の評価を行った．比較の結果，Cbuf を使用することで手動で記述する必要があるコード量は平均で半分程度に減った．また，2 つのバインディングを使用した C の関数の呼び出しにかかる時間に大きな差は見られなかった．このツールを利用することにより，参照渡しを用いる C の関数のバインディングにおいて手動で記述するプログラムが減り，それによりエラーの機会が減ることでソフトウェアの堅牢性が向上することが期待される．

An Extension of OCaml FFI library Ctypes for Call-By-Reference Functions

Arata Sato

Abstract

It is common to bind and call functions written in a language L_1 from a program written in another language L_2 in programming. This mechanism is called *FFI (Foreign Function Interface)*. To use functions of L_1 using FFI, we need a program to bind the functions. Such programs are called *bindings*. A binding converts the data representation, types of the arguments, and the type of the return values between those of L_1 and L_2, so that a program written in L_2 can call a function written in L_1.

Writing a binding manually is cumbersome and error-prone if there are mismatches between L_1 and L_2 in type systems, data representation, and runtime environments. One approach to solve this problem is to generate bindings from type definitions of foreign functions. Ctypes is one of the tools with such functionality, and it handles FFI to call a C function from OCaml. Ctypes supports a mechanism to generate a binding from its type definitions, which is expressed as an OCaml program. This mechanism is called a *binding mechanism*. *Staged binding*, one of the binding mechanisms provided by Ctypes, generates C and OCaml programs from type definitions of foreign functions and statically links foreign functions using the generated programs. Ctypes reduces the amount of manually written bindings, taking the differences in the type system and data representation between OCaml and C into account.

In this study, we aim to improve the robustness of software by extending Ctypes to increase the variety of bindings that can be generated systematically from the type definitions of foreign functions. To identify a useful extension in practice, we surveyed manually written bindings used in open source projects that use Ctypes. The findings from this survey are as follows: (i) To bind call-by-reference C functions, one prefers to write a binding manually so that the arguments can be handled as return values in OCaml; (ii) One implements modules for managing memory allocation for such values; (iii)

One implements functions for verifying the size of allocated memory; and (iv) One often associates `int` with an OCaml exception. From this survey, we observed that there are many cases where, even when Ctypes is used, bindings are written manually, especially when there are differences in programming language conventions or restrictions that need to be met between the data to be passed. Based on this survey, we identified that, to handle FFI for call-by-reference functions, we would need a tool with the following functionalities: automated memory allocation based on the type and size of the call-by-reference function; and a mechanism that allows OCaml to handle the call-by-reference variables as return values.

Inspired by these findings, we propose a method that automatically generates bindings between C and OCaml based on information about the type of a foreign function and the size of memory to be allocated. We also implement a tool to demonstrate this method by extending Ctypes. We named the tool Cbuf. Cbuf extends the `FOREIGN` interface used to describe foreign function types in Ctypes to describe additional information necessary for generating bindings. As with Ctypes, C object types and function types are represented as OCaml programs. The design of Cbuf is based on staged binding of Ctypes, and it generates bindings between C and OCaml from type definitions of foreign functions. The binding generated by Cbuf allocates memory, calls foreign functions, and converts types of function arguments and return values. The binding mechanism of Cbuf is implemented as modules that satisfy FOREIGN as well as the binding mechanisms provided by Ctypes.

We compared the staged binding of Ctypes and the binding using Cbuf for the same C functions to evaluate the proposed method. As a result of the comparison, the use of Cbuf reduced the amount of code written manually by about half on average. In addition, there was no significant difference in the time required to call a C function using the two bindings. Our tool is expected to improve software robustness by reducing the amount of code that needs to be written manually to bind the call-by-reference C function, thereby reducing the chance of errors.

OCaml 用 FFI ライブラリ Ctypes の参照渡し関数への拡張

目次

1 序論

1.1 背景

プログラミングにおいて，他言語で記述された関数をバインドし，呼び出すことはよく行われる．この仕組みを *FFI(Foreign Function Interface)* と呼ぶ．例えば，参照渡しによって渡された引数 `out` にデータを書き込むような C の関数 sample を，OCaml から呼び出す場合を考える．:

```
int sample(unsigned char *out);
```

FFI を利用して他言語の関数を使用するためには，その関数をバインドするためのプログラム (**バインディング**) が必要である．関数 sample に対応するバインディングは，OCaml で引数の値をバイト列として扱いたい場合，C の `unsigned char*` 型の値を OCaml の `bytes` 型の値に型変換を行う．このようにバインディングは，引数や返り値のデータ表現や型の変換を行い，バインド元のプログラムからバインド先の関数の呼び出しを可能にする．

バインド先の言語とバインド元の言語において，型システムやデータ表現，実行環境などが異なる場合，バインディングの手動での記述は煩雑で間違えやすい．この問題の解決への 1 つのアプローチに，バインド先の関数型の記述を元にしたバインディングの自動生成がある．C に対する FFI を扱うライブラリである Ctypes [1] は，そのような機能を持つ．Ctypes では，バインド先の関数の型を OCaml のプログラムとして記述することができる．Ctypes は，この関数型の記述を元に C と OCaml のバインディングを生成し，その生成されたバインディングを元に静的に他言語関数をリンクする．例えば，Ctypes を用いた関数 sample の関数型の記述を以下に示す．:

```
let sample = foreign "sample" (ocaml_bytes @-> returning int)
```

以上のように，C の関数名，引数と返り値の型を OCaml のプログラムとして記述することができる．OCaml の値である関数 sample の型は，`bytes -> int` となる．Ctypes の関数型の記述については後の章で詳しく説明する．この記述を元にバインディングを生成し，そのバインディングを利用して，関数 sample を呼び出すことができる．このように，Ctypes を用いることで，OCaml と C の型システムやデータ表現の差異を考慮し，手動で記述するバインディングを減らすことが可能である．

1.2 本研究について

本研究では，Ctypes を拡張することで，バインド先の関数型の記述に基づき機械的に生成できるバインディングを増やすことによる，ソフトウェアの堅牢性の向上を目指す．そのために Ctypes を利用するオープンソースプロジェクトにおいて，手動で記述されているバインディングの例を調査した．この調査の結果，プログラミング言語の慣例の差異や受け渡すデータ間に満たしたい制約がある場合，Ctypes を用いた上で依然としてバインディングを手動で記述する例が多いことが分かった．特に，参照渡しを用いた C の関数をバインドする際，適当なサイズのメモリを確保し，その参照をバインド先の関数に渡す OCaml のラッパー関数を実装する例が多かった．そのラッパー関数では，参照渡しに用いた変数をラッパー関数の返り値として扱うような例が散見された．例えば，関数 sample に対応するラッパー関数は以下のようになる．

```
let sample_wrapper () =
  let b = Bytes.create 4 in (* 4バイトのバイト列を作成する *)
  let _ = sample b in       (* バイト列を関数 sample に適用する *)
  b                         (* バイト列を全体の返り値として返す *)
```

この調査結果から，このようなラッパー関数内の処理を行うバインディングを，関数型の記述を元に生成するツールが有用だと判断した．

我々は，Ctypes を拡張することで，バインド先の関数の型や確保するメモリのサイズの情報を元に C と OCaml のバインディングを機械的に生成するツール Cbuf を実装する．Ctypes と同様に，関数型の記述において C のオブジェクト型や関数型は OCaml のプログラムとして表現される．例えば，関数 sample の関数型の記述を以下に示す．

```
let sample_cbuf = foreign "sample"
  (void @-> retbuf (buffer 4 ocaml_bytes) (returning int))
```

このように，`retbuf` や `buffer` を用いて，参照渡し変数を指定する．OCaml の値である関数 `sample_cbuf` の型は，`unit -> (bytes * int)` となる．また，確保するメモリのサイズを関数型の記述の一部として含める．Cbuf の関数型の記述については後の章で詳しく説明する．Cbuf はこの関数型の記述に基づいて C と OCaml のバインディングの生成を行う．Cbuf によって生成されたバインディングは，メモリの確保やバインド先の C の関数の呼び出し，関数の引数や

返り値の型変換などを行う．sample_cbuf を直接呼び出すことで，先に示したラッパー関数と同等の処理を行うことができる．

また，同じ C の参照渡し関数に対する，Ctypes の段階的バインディングと Cbuf を使用したバインディングを比較し，提案手法の評価を行う．具体的には，2 つのバインディング間で手動で記述する必要があるコード量と C の関数の呼び出しにかかる時間を比較する．

1.3 本報告書の構成

本報告書は次のような構成になっている．第 2 章で Ctypes について概説し，第 3 章で Ctypes を利用するライブラリの事前調査について述べる．第 4 章で本研究で提案するツールである Cbuf の設計と実装について述べる．第 5 章で Cbuf を用いたバインディングを，手動でのコードの記述量と関数の呼び出し速度の観点から定量的に評価する．第 6 章で関連研究について議論し，第 7 章で結論と今後の課題を述べる．

2 Ctypes

Ctypes は OCaml において C に対する FFI を扱うためのライブラリである．この章では Ctypes について，本研究に関連する部分に焦点を当てて概説する．

2.1 C のオブジェクト型の表現

C において，オブジェクト型はメモリ上の値のレイアウトを規定する型である．Ctypes では typ 型の値が C のオブジェクト型を表す．ソースコード 1 に抽象型 typ といくつかのオブジェクト型を表す値のシグネチャを示す．それぞれ

```
1 type 'a typ
2
3 val void : unit typ
4 val string : string typ
5 val int : int typ
```

ソースコード 1: 抽象型 typ とオブジェクト型のシグネチャ

```
type ('a, 'b) pointer
type 'a ptr = ('a, [ `C ]) pointer
type 'a ocaml = ('a, [ `OCaml ]) pointer

val ocaml_bytes : bytes ocaml typ
val ptr : 'a typ -> 'a ptr typ
```

ソースコード 2: pointer 型のシグネチャと，pointer 型を用いた型定義と値

の typ のパラメータは C のオブジェクト型に対応する OCaml の型を表す．例えば，OCaml の値 `val string : string typ` は C の `char *`型と OCaml の string 型に対応する．また，オブジェクト型のポインタは pointer 型を用いて表現する．ソースコード 2 に pointer 型のシグネチャと，pointer 型を用いた型の定義，typ 型の値のシグネチャなどの例を示す．

C で管理されるメモリに配置される値のポインタは ptr 型で表現し，OCaml の場合は ocaml で表現する．ocaml_bytes は OCaml の bytes 型と C の `unsigned char*`型に対応する typ 型の値である．bytes 型の値は OCaml が管理するメモリに配置されるので，typ の型パラメータは `bytes ocaml` 型になる．また，ptr は typ 型の値を引数にとり，そのポインタを表す typ 型の値に変換する関数である．

2.2 C の関数型の表現

C の関数型は関数の引数と返り値を規定する型である．C の関数をバインドするために，C の関数型を OCaml のプログラムで表現する必要がある．C の関数型は Ctypes が提供する foreign, @->, returning などの関数を使用して記述する．それらの関数は Ctypes が提供するインターフェースである FOREIGN に含まれる．ソースコード 3 に FOREIGN の一部を示す．

fn 型は C の関数型を表し，2 つの関数@->と returning によって fn 型の値を構成する．関数 foreign は C の関数名と fn 型の値を引数にとり，OCaml の値を返す．例えば，標準出力に文字列を出力する C の関数 puts のシグネチャと，OCaml のプログラムとして記述した関数型は以下のようになる．
C の関数 puts のシグネチャ:

```
module type FOREIGN = sig
  type 'a fn
  type 'a return

  val ( @-> ) : 'a typ -> 'b fn -> ('a -> 'b) fn
  val returning : 'a typ -> 'a return fn

  val foreign : string -> ('a -> 'b) fn -> ('a -> 'b)
  ...
end
```

ソースコード 3: FOREIGN シグネチャの一部

```
int puts(const char *s);
```

OCaml のプログラムとして記述される関数型:

```
let puts = foreign "puts" (string @-> returning int)
```

C の関数 puts は char 型のポインタを引数にとり，int 型の値を返す関数である．returning の左の string は C の関数 puts の引数の型に対応し，returning の引数の int は返り値の型に対応する．foreign が返す OCaml の値 puts の型は string -> int となる．

2.3 モジュール性

C の関数を紐付けて呼び出す仕組みを結合メカニズムと呼ぶ．Ctypes では 1 つのバインドされる関数型の記述に対して複数の結合メカニズムを適用することができる．結合メカニズムは FOREIGN インターフェースを満たしたモジュールの実装として提供される．バインドされる関数型の記述を FOREIGN を満たすモジュールをパラメータとするファンクタの中に置くことで複数の結合メカニズムを入れ替えることが可能である．ソースコード 4 に C の関数 puts の関数型の記述を含むファンクタの例を示す．この例では，モジュール Puts のパラメータに渡すモジュールを入れ替えることで，puts の結合メカニズムを変更することができる．Ctypes は動的バインディング，段階的バインディングなどの結合メカニズムを提供している．動的バインディングは動的ライブラリを libffi

```
module Puts(F: FOREIGN) = struct
  open F
  let puts = foreign "puts" (string @-> returning int)
end
```

ソースコード 4: puts の関数型の記述を含むファンクタ Puts

[2] を用いてバインドする．一方，段階的バインディングは C と OCaml のコード生成を用いて，OCaml が標準で用意する FFI の仕組み [3] を用いてバインドする．

2.4 段階的バインディング

段階的バインディングは，関数型の記述を元に C と OCaml のコードを生成し，それらの生成されたコードを用いて C の関数を紐付け，OCaml から呼び出す結合メカニズムである．この節では，Cbuf を実装する際に参考にした結合メカニズムである段階的バインディングについて概説する．

2.4.1 アーキテクチャ

段階的バインディングでは，関数型の記述を含むファンクタを，3 つの異なる `FOREIGN` を実装したモジュールに適用する．以下に 3 つのモジュールについて順に説明する．

Foreign_GenC 関数型を元に C のスタブコードを生成する

Foreign_GenML 関数型を元に OCaml のプログラムを生成する

Foreign_GeneratedML 関数型の記述を元に C の関数を呼び出す

`Foreign_GenC` と `Foreign_GenML` は Ctypes が提供するモジュールであり，`Foreign_GeneratedML` は `Foreign_GenML` によって生成されるモジュールである．

図 1 に段階的バインディングを使用して C の関数を使用する際の全体のアーキテクチャを示す．黒枠で示した C の関数を OCaml から利用する流れを順に説明する．(1) プログラマは関数型の記述を含む Puts のようなファンクタを作成する．(2) OCaml のプログラムと C のスタブコードを生成するため，`Foreign_GenC` と `Foreign_GenML` を Puts に適用するモジュールを作成する．このモジュールを bindgen モジュールと呼ぶことにする．(3) bindgen モジュールを実行し，赤

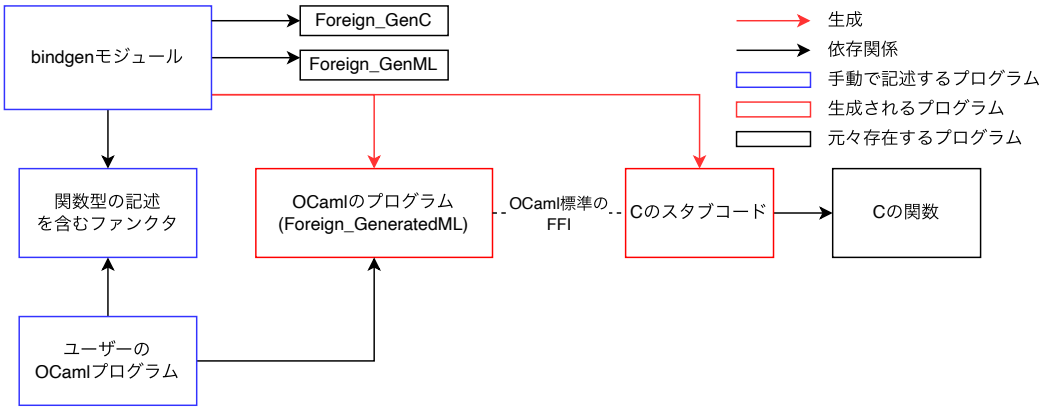

図 1: 段階的バインディングのアーキテクチャ

```
1 value caml_puts(value x1)
2 {
3     char* x2 = CTYPES_ADDR_OF_FATPTR(x1);
4     int x3 = puts(x2);
5     return Val_long(x3);
6 }
```

ソースコード 5: 生成される C のコード

枠で示したプログラムを生成する．(4) ユーザーの OCaml プログラムにおいて，Foreign_GeneratedML を関数型の記述を含むファンクタに適用し，関数型の記述から C の関数を呼び出す．

以上の流れで，最終的に使用するプログラムが静的にコンパイル，リンクされ，ユーザーの OCaml プログラムからバインド先の C の関数が利用可能になる．

2.4.2 C のコード生成

Foreign_GenC をモジュール Puts に適用することによって C のコードを生成する．：Puts(Foreign_GenC)

生成される C のコードの例をソースコード 5 に示す．関数 puts の関数型の記述に対して，ラッパー関数 caml_puts が生成される．このラッパー関数をスタブコードと呼ぶ．caml_puts の引数，返り値の型 value は，OCaml が提供している OCaml の値を表す型である．

```
module CI = Cstubs_internals

external caml_puts : _ CI.fatptr -> int = "caml_puts"

type 'a return = 'a
type 'a fn =
 | Returns  : 'a CI.typ   -> 'a return fn
 | Function : 'a CI.typ * 'b fn  -> ('a -> 'b) fn
let returning t = Returns t
let (@->) f p = Function (f, p)
let foreign : type a b. string -> (a -> b) fn -> (a -> b) =
  fun name t -> match t, name with
| Function
    (CI.View {CI.ty = CI.Pointer _; write = x2; _},
     Returns (CI.Primitive CI.Int)),
  "puts" ->
  (fun x1 -> let CI.CPointer x4 = x2 x1 in let x3 = x4 in
   ↪  caml_puts x3)
| _, s ->  Printf.ksprintf failwith "No match for %s" s
...
```

ソースコード 6: 生成される OCaml のコード

CTYPES_ADDR_OF_FATPTR, Val_long はそれぞれ引数と返り値の型を puts に合わせてキャストする.

2.4.3 OCaml のコード生成

Foreign_GenML をモジュール Puts に適用することによって OCaml のコードを生成する. : `Puts(Foreign_GenML)`

生成される OCaml のコードの例をソースコード 6 に示す. external キーワードは OCaml が提供している, スタブコードを OCaml のプログラムと紐づける記法である. スタブコード caml_puts を, OCaml の関数 `caml_puts : _ CI.fatptr -> int` として OCaml のプログラムから呼び出すことが可能である. また, 生成された

モジュールは FOREIGN を満たしており，foreign，@->，returning などの実装を含む．foreign はバインドされる C の関数型を表す fn 型の値と，関数名をパターンとするパターンマッチにより，引数の前処理をした後，適切なスタブコードを呼び出す．

3 Ctypes を使用しているライブラリの調査

オープンソースプロジェクトにおいて Ctypes がどのように使用されているかについて，事前調査を行った．事前調査を行った目的は主に次の 2 つである．:

1. Ctypes を使ったバインディングにおいて，ツールを実装することで解決できる問題を探す．
2. そのようなツールがサポートするべき機能を理解する．

この章では，まず事前調査に用いたオープンソースプロジェクトの概要を述べ，次に所見が得られたコードやパターンの例をいくつか挙げる．最後に，得られた所見について調査結果を述べる．

3.1 調査に使用したプロジェクト

この事前調査では，ocaml-sodium [4] と ocaml-argon2 [5] という 2 つのオープンソースプロジェクトを使用した．それぞれのライブラリについて，使用されている結合メカニズムやソースコードに含まれる主なモジュールを概観する．

3.1.1 ocaml-sodium

ocaml-sodium は，C で記述された暗号を扱うライブラリである libsodium を OCaml から使えるように FFI によってバインドしたライブラリである．

ocaml-sodium は Ctypes の段階的バインディングを使用してバインディングを生成している．ライブラリの関数のバインディングに関連するモジュールは，Sodium_bindings，Sodium_bindgen，Sodium_storage，Sodium の 4 つである．Sodium_bindings モジュールは Ctypes を使用して記述されたバインドされる関数型の記述を含む．関数型の記述は FOREIGN をパラメータとするファンクタとして提供される．Sodium_bindgen モジュールは段階的バインディングにより C と OCaml のコード生成を行う際のエントリーポイントを含むモジュールである．Sodium_storage モジュールは参照渡し変数を管理するモジュールを提供する．このモジュールについては次の節で詳しく述べる．Sodium モジュー

```
let hash_raw fun_name =
  foreign fun_name
    (uint32_t (* t_cost *) @-> uint32_t (* m_cost *)
    @-> uint32_t (* parallelism *) @-> string
    (* pwd *) @-> size_t (* pwdlen *)
    @-> string (* salt *) @-> size_t
    (* saltlen *) @-> ptr void (* hash *)
    @-> size_t (* hashlen *) @-> returning ErrorCodes.t)
```

ソースコード 7: ocaml-argon2/src/argon2.ml

ルはバインドした関数を OCaml のライブラリとしてエクスポートするモジュールである．

3.1.2 ocaml-argon2

ocaml-argon2 は，Argon2 という鍵導出関数の C の参照実装を OCaml から使用できるように FFI によってバインドしたライブラリである．ocaml-argon2 は Ctypes の動的バインディングを使用してライブラリをバインドしている．ライブラリの関数のバインディングに関連する主なモジュールは Argon2 の 1 つである．Argon2 モジュールはバインドされる関数型の記述などが含まれる．このモジュールでは，Ctypes が提供する動的バインディングを使用するためのモジュールである Foreign を open している．Foreign モジュールが FOREIGN の実装として提供する 1 つの結合メカニズムのみを用いるので，バインドされる関数型の記述はファンクタではなく，Argon2 モジュールに直接記述されている．

3.2 コードやパターンの例

この節では，調査したプログラムの断片の例を挙げ，問題点などについて議論する．

ソースコード 7 は，FOREIGN を用いて記述された Argon2 に含まれる関数型である．Argon2 では関数の引数と返り値の型が同じ関数が複数ある場合，関数名をパラメータとし，1 つの関数型の記述で複数の関数をバインドしている．ソースコード 7 の hash_raw を内部で呼び出すラッパー関数をソースコード 8 に示す．allocate_n は Ctypes が提供する関数であり，引数に typ 型の値と int

```
let hash_raw hash_fun ~t_cost ~m_cost ~parallelism ~pwd ~salt
  ↪  ~hash_len =
  ...
  let hash = allocate_n char ~count:hash_len |> to_voidp in
  let s_hash_len = Unsigned.Size_t.of_int hash_len in

  match
    hash_fun u_t_cost u_m_cost u_parallelism pwd s_pwd_len salt
     ↪  s_salt_len hash
      s_hash_len
  with
  | ErrorCodes.OK ->
      let hash = string_from_ptr (from_voidp char hash)
       ↪  ~length:hash_len in
      Result.Ok hash
  | e -> Result.Error e
```

ソースコード 8: ocaml-argon2/src/argon2.ml

```
module type S = sig
  type t
  type ctype

  val ctype      : ctype typ

  val create     : int -> t
  val length     : t -> int
  val to_ptr     : t -> ctype
  val to_bytes   : t -> Bytes.t
  val of_bytes   : Bytes.t -> t
  ...
end
```

ソースコード 9: ocaml-sodium/lib/sodium_storage.ml

型の count をとり，長さが count の配列を作成し，その配列のポインタを ptr 型の値として返す．allocate_n によって作成されたポインタ hash は hash_fun の引数として適用される．ソースコード 8 の hash_raw の引数 hash_fun はソースコード 7 の hash_raw を fun_name に適用したものである．hash_fun によってバインド先の C の関数が呼び出され，hash が指すポインタのデータが書き換えられる．エラーが発生していない場合は hash が指すポインタのデータを string に変換して返り値とし，エラーが発生した場合はエラーを返す．このように，Argon2 は引数にポインタ変数を渡してその変数に副作用によってデータを書き込み，その変数を返り値に変換するようなバインディングが含まれる．また，バインドされた C の関数の返り値が int 型の値で，エラーコードを表す場合，それを OCaml の値と紐づけるようなコードが含まれることが分かる．

ソースコード 9 に，ocaml-sodium に含まれる参照渡し変数を管理するモジュールのシグネチャの一部を示す．参照渡し変数を管理するモジュールをここではストレージと呼ぶ．このシグネチャに含まれる関数のうちいくつかについて説明する．

create　指定したサイズ分のメモリを確保し，参照渡し変数を生成する．

```
module Bytes = struct
  type t = Bytes.t
  type ctype = Bytes.t ocaml

  let ctype = ocaml_bytes

  let create     len = Bytes.create len
  let length     byt = Bytes.length byt
  let to_ptr     byt = ocaml_bytes_start byt
  let to_bytes   byt = Bytes.copy byt
  let of_bytes   byt = Bytes.copy byt
  ...
end
```

ソースコード 10: ocaml-sodium/lib/sodium_storage.ml

```
let box_keypair = F.(foreign (prefix^"_keypair") (ocaml_bytes
  ↪  @-> ocaml_bytes @-> returning int))
```

ソースコード 11: ocaml-sodium/lib_gen/sodium_bindings.ml

length　参照渡し変数のサイズを返す

to_ptr　ptr 型の値に変換する

to_bytes　bytes 型の値に変換する

of_bytes　bytes 型の値を t 型の値に変換する

ocaml-sodium では，シグネチャS の実装としてストレージを提供する．ソースコード 10 に，bytes 型の参照渡し変数を管理するストレージの実装である Bytes モジュールを示す．Bytes では 1 バイト単位でメモリの確保などを行い，Ctypes が提供する ocaml_bytes_start 関数を使って ptr 型への変換を行う．このように，参照渡し変数を扱うためのモジュールを，Ctypes の利用側が独自に実装している例があることが分かる．

ソースコード 11 は ocaml-sodium の，FOREIGN を用いて記述した関数型の例

```
let random_keypair () =
  let pk, sk = Storage.Bytes.create public_key_size,
               Storage.Bytes.create secret_key_size in
  let ret =
    C.box_keypair (Storage.Bytes.to_ptr pk)
    ↪  (Storage.Bytes.to_ptr sk) in
  assert (ret = 0); (* always returns 0 *)
  sk, pk
```

ソースコード 12: ocaml-sodium/lib/sodium.ml

```
let verify_length str len fn_name =
  if T.length str <> len then raise (Size_mismatch fn_name)

let to_public_key str =
  verify_length str public_key_size "Box.to_public_key";
  T.to_bytes str
```

ソースコード 13: ocaml-sodium/lib/sodium.ml

である．box_keypair は，libsodium の crypto_kx_keypair などの関数をバインドする．crypto_kx_keypair のシグネチャを以下に示す．

```
int crypto_kx_keypair(unsigned char pk[crypto_kx_PUBLICKEYBYTES],
                      unsigned char sk[crypto_kx_SECRETKEYBYTES]);
```

crypto_kx_keypair は，unsigned char 型の配列を 2 つ引数に取り，副作用によってその配列にデータを書き込むような関数である．

ソースコード 12 に box_keypair を呼び出す関数の例を示す．public_key_size, secret_key_size は int 型の値である．まず，ソースコード 10 で見た Bytes を用いて，サイズがそれぞれ public_key_size，secret_key_size の参照渡し変数 pk，sk を確保している．それらの参照渡し変数を box_keypair に引数として渡し，データを書き込んでいる．

ソースコード 13 にストレージのサイズを検証する関数の例を示す．verify_length

はストレージのサイズを検証し，サイズが適切でなかった場合に Size_mismatch 例外を発生させる．to_public_key はストレージを bytes 型のエイリアスである public key 型の値に変換する関数である．変換する前に verify_length を用いてストレージのサイズの検証を行っている．このように，Ctypes の利用側が参照渡し変数のサイズを検証するようなコードの例がある．

3.3 調査結果

この調査により，次の所見が得られた．(i) ソースコード 8，12 のように，参照渡しによって C に渡した変数を，OCaml で返り値として扱えるようにラッパー関数を実装する例がある．(ii) ソースコード 9，10 のように，そのような変数を扱うため，C に参照渡し変数として渡すポインタ変数のメモリの確保を管理するためのモジュールを実装する例がある．(iii) ソースコード 13 のように，参照渡し変数のサイズを検証する関数を用意し，バインディングの利用側でその関数を用いて検証する例がある．(iv) ソースコード 8 のように，C の関数の int 型の返り値を OCaml の例外に紐づけている例がある．

ocaml-argon2 に含まれる関数型の記述は 8 例あり，その内参照渡し関数のラッパー関数を作っている例は 3 あった．それらのラッパー関数は全て，C の関数の int 型の返り値を OCaml の例外に紐づけていた．また，ocaml-sodium に含まれる関数型の記述は 76 例あり，その内参照渡し関数のラッパー関数を作っている例は 36 あった．そのうち例外への紐付けを行っているものは 4 つであった．

これらの所見を受け，変数の参照渡しを用いる関数をバインドする場合，その関数の型や確保するメモリのサイズの情報に基づいてメモリを確保し，OCaml ではその変数を返り値として扱えるような仕組みを提供するツールが有用だと判断した．

4 Cbuf

この章では，本研究で提案するツールである Cbuf [1)] の設計と実装について述べる．まず，Ctypes の関数型の記述をどのように拡張したかについて述べる．次に，Cbuf によって生成されるコードと，コード生成の実装について述べる．

[1)] ソースコードは https://github.com/atrn0/ocaml-ctypes-cbuf で公開している．

```
type _ cbuffers =
  | LastBuf :
      int * ('a, 'b) Ctypes.pointer Ctypes.typ
      -> ('a, 'b) Ctypes.pointer cbuffers
  | ConBuf :
      ('a, 'b) Ctypes.pointer cbuffers * ('c, 'd) Ctypes.pointer
      ↪  cbuffers
      -> ('a * 'c, [ `OCaml ]) Ctypes.pointer cbuffers
```

ソースコード 14: Cbuf.cbuffers

4.1 関数型の記述の拡張

Cbuf では C の関数型に加え，参照渡しによって C に渡される変数とその型や，確保するメモリのサイズを記述できるよう，Ctypes の関数型の表現を拡張した．参照渡しによって C に渡す変数は，C の関数型において引数の最初または最後に連続している例が多いため，Cbuf では参照渡しを用いる変数の列を記述し，その列を引数の最初または最後のどちらとして扱うかを指定できるようにした．また現在の Cbuf では，参照渡しに用いる変数の型として ocaml_bytes のみをサポートする．記述するメモリのサイズの単位はバイトとする．必要な情報を記述するために新しく追加した関数や型を順に説明する．

型 cposition は，参照渡しを用いる変数の列を，C において引数の最初または最後のどちらとして扱うかを表す型である．`First は最初，`Last は最後を表す．: `type cposition = [ `First | `Last ]`

ソースコード 14 に示した型 cbuffers は，参照渡しを用いる変数の列を表す型である．コンストラクタ LastBuf は列の右端の要素を表し，確保するメモリのサイズと，typ 型の値を保持する．コンストラクタ ConBuf は 2 つの cbuffer 型の値をとり，Ctypes.pointer のパラメータに含まれる型変数'a と'c から'a * 'c 型のタプルを生成する．ConBuf の最初の引数は LastBuf のみをサポートする．例えば，`ConBuf (LastBuf (1, t1), LastBuf (2, t2))` は許容するが，`ConBuf (ConBuf (LastBuf (1, t1), LastBuf (2, t2)), LastBuf (4, t3))` は許容しない．

ソースコード 15 に示した型 Cbuf.fn は，Ctypes において関数型を表す型

```
type _ fn =
  ...
  | Buffers :
    cposition * ('a, 'b) Ctypes.pointer cbuffers * 'c fn
    -> ('a * 'c) fn
```

ソースコード 15: Cbuf.fn

```
let ( @* ) l r = ConBuf (l, r)
let retbuf ?(cposition = `Last) buf return = Buffers (cposition,
  ↪ buf, return)
let buffer i ty = LastBuf (i, ty)
```

ソースコード 16: Cbuf において新しく追加した演算子と関数

Ctypes.fn を拡張し，コンストラクタ Buffer を追加したものである．Buffer は cposition，参照渡しを用いる変数の列，バインド先の C の関数の返り値の型を保持する．

ソースコード 16 に Cbuf において新しく追加した演算子と関数を示す．@* は参照渡しを用いる変数の列を表す cbuffers 型の値を構成する．retbuf は cposition，@*によって構成した cbuffers 型の値，returning によって構成したバインド先の C の関数の返り値の型を表す値を引数にとり，Cbuf.fn 型の値を構成する．buffer は確保するメモリのサイズと typ 型の値を引数にとり，LastBuf を使って cbuffers 型の値を構成する．buffer によって構成された値は@*のオペランドとして用いる．

Ctypes.FOREIGN を拡張した Cbuf.FOREIGN を示す．：

```
module type FOREIGN = Ctypes.FOREIGN with type 'a fn = 'a fn
```

Buffers コンストラクタを利用するため，fn に Cbuf.fn を指定している．

以上が新しく追加した型と関数，演算子である．これらを用いて関数型を記述する例を以下に示す．

ソースコード 17 に示した関数 last_cbuf は，int 型の値と，2 つの unsigned char 型のポインタを引数にとる．また，返り値は int 型である．関数 last_cbuf

```c
int last_cbuf(int in, unsigned char *out1, unsigned char *out2);
```

ソースコード 17: C の関数 last_cbuf

```
open Cbuf
module C (F : FOREIGN) = struct
  open F
  (* val last_cbuf : int -> ((bytes * bytes) * int) *)
  let last_cbuf =
    foreign "last_cbuf" (int @->
            retbuf ~cposition:`Last
            (buffer 4 ocaml_bytes @* buffer 8 ocaml_bytes)
            (returning int))
end
```

ソースコード 18: last_cbuf の関数型の記述

は参照渡しにより 2 つの unsigned char 型のポインタ変数を引数として受け取り，それらの変数が指すメモリにそれぞれ 4 バイト，8 バイトのデータを書き込む関数である．関数 last_cbuf をバインドするための Cbuf における関数型の記述をソースコード 18 に示す．参照渡しと C の返り値の型は retbuf を用いて記述する．まず，参照渡しを用いた変数は C において引数の最後に連続しているので，cposition に `Last を指定する．次に buffer と@*を用いて参照渡しを用いる変数の列を構成する．ソースコード 18 の 5 行目の値は ConBuf (LastBuf (4, ocaml_bytes), LastBuf (8, ocaml_bytes)) となり，型は(bytes * bytes, [ `OCaml]) pointer cbuffers となる．また，6 行目で C における返り値の型 int を指定している．最終的に構成される OCaml の値 last_cbuf の型は，int -> ((bytes * bytes) * int) となる．以上のように，C の引数で参照渡しを用いる変数を foreign によって記述すると，それらの変数が OCaml で返り値に含まれるように型変換を行う．Cbuf では 1 つの C の関数型の記述を複数の結合メカニズムに適用するため，Cbuf.FOREIGN をパラメータに持つファンクタの中に関数型を記述する．

```
1 value caml__3_last_cbuf(value x19, value x18, value x17)
2 {
3    int x20 = Long_val(x19);
4    unsigned char* x23 = CTYPES_PTR_OF_OCAML_BYTES(x18);
5    unsigned char* x24 = CTYPES_PTR_OF_OCAML_BYTES(x17);
6    int x25 = last_cbuf(x20, x23, x24);
7    return Val_long(x25);
8 }
```

ソースコード 19: Cbuf によって生成される C のコード

4.2 コード生成

Cbuf の結合メカニズムは Ctypes の段階的バインディングと同様のアーキテクチャを持つ．つまり Cbuf は，関数型の記述を含むファンクタを 3 つの異なる FOREIGN の実装に適用することで，C と OCaml のプログラムの生成，生成されたプログラムを用いた C の関数の OCaml のプログラムへの紐付けを行う．この節では，Cbuf の C と OCaml コードの生成について述べる．

4.2.1 C コードの生成

Cbuf における C のコード生成について，ソースコード 18 に示したファンクタを用いて説明する．Cbuf では Ctypes の段階的バインディングと同様に，`Foreign_Cbuf_GenC` をモジュール C に適用することでスタブコードを生成する．生成される `last_cbuf` に対応するスタブコードをソースコード 19 に示す．引数に含まれる `x18` と `x17` は参照渡し変数として扱われ，`unsigned char` のポインタにキャストされる (4，5 行目)．それらの変数は `last_cbuf` に引数として渡され，データが書き込まれる (6 行目)．

4.2.2 OCaml コードの生成

Cbuf における OCaml のコード生成について，ソースコード 18 に示したファンクタを用いて説明する．Cbuf では Ctypes の段階的バインディングと同様に，`Foreign_Cbuf_GenML` をモジュール C に適用することで OCaml のコードを生成する．生成される `last_cbuf` に対応する OCaml のコードの，段階的バインディングにおいて生成されるコードとの差分であるパターンマッチ部分を切り出してソースコード 20 に示す．段階的バインディングによって生成される OCaml

```
1 | Function (CI.Primitive CI.Int,
2     Buffers (_, ConBuf (LastBuf (4, CI.OCaml CI.Bytes),
3                         LastBuf (8, CI.OCaml CI.Bytes)),
4     Returns (CI.Primitive CI.Int))),
5   "last_cbuf" -> (fun x8 ->
6     let x10 = Bytes.create 8 in let x9 = Bytes.create 4 in
7     let x11 = caml__3_last_cbuf x8 (Ctypes.ocaml_bytes_start x9)
8       (Ctypes.ocaml_bytes_start x10)
9     in ((x9, x10), x11))
```

ソースコード 20: Cbuf によって生成される OCaml のコード

```
1 module C = C (Foreign_Cbuf_GeneratedML)
2 let (out1, out2), ret = C.last_cbuf 3
```

ソースコード 21: last_cbuf を呼び出す例

のコード (ソースコード 6) と同様に，fn 型の値と関数名をパターンとするパターンマッチによりバインド先の関数を呼び出す．パターンマッチの本体では，関数型の記述で指定したサイズ分のバイト列を作り，変数 x10 と x9 に代入している (6 行目)．その後，それらの変数を Ctypes.ocaml_bytes_start を用いて bytes 型から Ctypes の bytes ocaml 型に変換し，caml__3_last_cbuf に適用する (7，8 行目)．最後に，参照渡し変数 x9，x10 と，caml__3_last_cbuf の返り値をタプルとして返す (9 行目)．生成された OCaml のコードは FOREIGN を満たすモジュールとなっており，ファンクタ C に適用することで，関数型の記述からバインド先の関数を直接呼び出すことが可能である．この生成されたモジュールをここでは Foreign_Cbuf_GeneratedML とする．

last_cbuf を呼び出す例をソースコード 21 に示す．
まず，Foreign_Cbuf_GeneratedML をファンクタ C に適用する (1 行目)．次に，適用した結果のモジュールを用いて，last_cbuf を呼び出す (2 行目)．out1，out2 は参照渡し変数に対応し，ret は int 型のバインド先の関数の返り値に対応する．以上のように，参照渡し関数をバインドする際，Cbuf のユーザーはメ

```
let box_keypair = F.(foreign (prefix^"_keypair")
  (ocaml_bytes @-> ocaml_bytes @-> returning int))
```

ソースコード 22: box_keypair の関数型の記述 (Ctypes)

```
let box_keypair = F.(foreign (prefix^"_keypair")
  (void @-> retbuf (buffer 32 ocaml_bytes @* buffer 32
  ↪  ocaml_bytes) (returning int)))
```

ソースコード 23: box_keypair の関数型の記述 (Cbuf)

モリの確保やラッパー関数を手動で書くことをせずに，返り値として参照渡し変数を扱うことが可能である．

5　評価

この章では，まず ocaml-sodium のバインディングを Cbuf によって置き換えた例をいくつか示す．次に Cbuf を用いた C の関数呼び出しにかかる時間に関しての実験の手法とその結果について述べ，最後に考察を述べる．

5.1　置き換えの例

我々は，ocaml-sodium の Ctypes を使った 6 つのバインディングを Cbuf によって置き換えた．バインディングを置き換える際，ocaml-sodium の API の互換性は維持した．また，ocaml-sodium において用意されていたテストは全て通過した．

バインディングを Cbuf を用いて置き換えた例を以下に示す．それぞれの例について，ラッパー関数の記述量の変化を議論する．記述量は，広く使用されているフォーマッターである ocamlformat の標準のプロファイル (default) を使用してプログラムをフォーマットした後に比較した．

C.Box.box_keypair の関数型の記述をソースコード 22，23 に示す．最初の 2 つの引数が参照渡しを用いる関数型であり，box_keypair では参照渡し変数に必要なメモリのサイズを指定している (5 行目)．

```
let random_keypair () =
  let pk, sk = Storage.Bytes.create public_key_size,
               Storage.Bytes.create secret_key_size in
  let ret =
    C.box_keypair (Storage.Bytes.to_ptr pk)
     ↪  (Storage.Bytes.to_ptr sk) in
    assert (ret = 0); (* always returns 0 *)
    sk, pk
```

ソースコード 24: box_keypair のラッパー関数 (Ctypes)

```
let random_keypair (): keypair =
  let (pk, sk), ret = C.box_keypair () in
  assert (ret = 0); (* always returns 0 *)
  sk, pk
```

ソースコード 25: box_keypair のラッパー関数 (Cbuf)

```
let sign_seed_keypair = F.(foreign (prefix^"_seed_keypair")
  (ocaml_bytes @-> ocaml_bytes @-> ocaml_bytes
   @-> returning int))
```

ソースコード 26: sign_seed_keypair の関数型の記述 (Ctypes)

```
let sign_seed_keypair = F.(foreign (prefix^"_seed_keypair")
  (ocaml_bytes @-> retbuf ~cposition:`First
  (buffer 32 ocaml_bytes @* buffer 64 ocaml_bytes)
  (returning int)))
```

ソースコード 27: sign_seed_keypair の関数型の記述 (Cbuf)

```
let seed_keypair seed =
  let pk, sk = Storage.Bytes.create public_key_size,
               Storage.Bytes.create secret_key_size in
  let ret =
    C.sign_seed_keypair (Storage.Bytes.to_ptr pk)
    (Storage.Bytes.to_ptr sk) (Storage.Bytes.to_ptr seed) in
  assert (ret = 0);
  sk, pk
```

ソースコード 28: sign_seed_keypair のラッパー関数 (Ctypes)

C.Box.box_keypair のラッパー関数 Sodium.Box.random_keypair の置き換えをソースコード 24，25 に示す．ソースコード 24 ではバイト列を作るコード (2，3 行目) を手動で書いているが，ソースコード 25 では書く必要がないことがわかる．また，ラッパー関数のコードの記述量は 11 行から 5 行に減った．

C.Sign.sign_seed_keypair の関数型の記述をソースコード 26，27 に示す．最初の 2 つの引数が参照渡しを用いる関数型であり，sign_seed_keypair では参照渡し変数に必要なメモリのサイズを指定している (7 行目)．

C.Sign.sign_seed_keypair のラッパー関数 Sodium.Sign.seed_keypair

```
1 let seed_keypair seed =
2   let (pk, sk), ret = C.sign_seed_keypair (Storage.Bytes.to_ptr
    ↪  seed) in
3   assert (ret = 0);
4   sk, pk
```

ソースコード 29: sign_seed_keypair のラッパー関数 (Cbuf)

表 1: ラッパー関数の記述量 (行数) の変化

関数名	置き換え前	置き換え後
Box.random_keypair	11	5
Box.precompute	11	7
Sign.random_keypair	11	5
Sign.seed_keypair	11	4
Sign.secret_key_to_seed	7	4
Sign.secret_key_to_public_key	7	4

の置き換えをソースコード 28，29 に示す．ソースコード 28 はバイト列を作るコード (2，3 行目) を手動で書いているが，ソースコード 29 では書く必要がないことがわかる．また，ラッパー関数のコードの記述量は 11 行から 4 行に減った．

`Sodium.Box.random_keypair`，`Sodium.Sign.seed_keypair` を含めて 6 つの関数型の記述そのラッパー関数を Cbuf によって置き換えた．表 1 に置き換えによるコードの記述量の変化の定量的な結果を示す．全体で 58 行のコードが 29 行に減少した．ラッパー関数のコードの記述量は平均で 50%減少した．

以上の結果から，参照渡し関数のバインドにおいて，関数の Cbuf を用いることによって手動でのプログラムの記述量が大幅に削減されることが確認できた．これによりエラーの機会が減ることが期待される．

表 2: 関数の実行時間の差 (マイクロ秒)

関数名	実行時間の差
arity1	-2894 ± 976
arity2	1279 ± 1578
arity3	-658 ± 2390

5.2 実験

我々は次のような計算機環境で実験を行った:

CPU Intel Core i5-8279U (2.4 GHz)

RAM 16GiB

OS macOS 12.1

Cbuf の Ctypes に対するオーバーヘッドを評価するため，3 つの C の関数 (arity1, arity2，arity3) を用意した．これらの関数はそれぞれ 2，3，4 つの引数をとり，最初の引数が `int` 型で，残りの引数は `unsigned char` のポインタであり，返り値は void 型である．これらの関数は第一引数として受け取った `int` 型の値を最後の引数として受け取った `unsigned char` のポインタに書き込む．これらの関数を Ctypes の段階的バインディングと Cbuf を用いて呼び出すため，それぞれの結合メカニズムに対応する関数型の記述を用意した．`unsigned char` 型のポインタに対応する引数は全て参照渡し変数として扱った．Ctypes の段階的バインディングについては，参照渡し変数として C の関数に渡すバイト列を作成するため，ラッパー関数を用意した．段階的バインディングによってバインドしたラッパー関数と Cbuf によってバインドした関数をそれぞれ 100 万回呼び出し，実行時間の差 (Cbuf の実行時間 − 段階的バインディングの実行時間) を計測した．実行時間の差の計測はそれぞれ 100 回行い，その結果を平均値 ± 標準偏差と表記した．

実験結果を表 2 に示す．arity2 は段階的バインディングを用いた方が実行時間が短かったが，arity1 と arity3 は Cbuf の方が短かった．これより Cbuf は段階的バインディングに対して，呼び出しにかかる時間のオーバーヘッドは十分小さいと言える．以上の結果は，Cbuf によって既存の Ctypes によるバインディングが十分置き換え可能であることを示唆する．

6 関連研究

Furr ら [6] は，OCaml の C に対する FFI における型安全性の検証を行う型システムを提案している．彼らは，C の型を OCaml でエンコードすることに加え，OCaml の型を C でエンコードすることによって，FFI を通じてでも型情報を正確に追跡できるシステムを構築している．このシステムではガベージコレクションの情報も追跡するため，OCaml が管理するヒープへのポインタを安全に扱うことが可能である．Cbuf では OCaml のヒープを C から追跡しないため，C から OCaml ヒープを意図せず書き換えることなどが可能であり，課題として残っている．また，本論文でも議論したように，型システムやデータ表現，実行環境などが異なる場合に，正確に FFI を利用する難しさを議論している．

SWIG [7] は C や C++で記述されたプログラムを他の言語から呼び出すためのツールである．このツールは，Ctypes の段階的バインディングや Cbuf のように，関数型の情報から，バインディングを生成することによって他言語関数の呼び出しを可能にする．SWIG は Ctypes や Cbuf と異なり，C/C++のヘッダファイルを元にバインディングを生成する．SWIG は Perl，Python，Ruby，Tcl などの言語のバインディングを生成することが可能である．Ctypes や Cbuf と異なり，OCaml などバインディング元の言語で関数型の記述をする必要が無い．SWIG は外部ツールとして利用する必要があるが，一方 Ctypes は関数型の記述や関数のバインド，呼び出しが OCaml のみで完結する．

上野ら [8] は ML 系高階関数型言語である SML # における他言語関数インターフェースを提案している．この他言語関数インターフェースは，ML のプログラムとして関数型の宣言が可能である．この方式は，データ表現レベルの相互運用性によって，Ctypes や Cbuf で生成するような多くのスタブコードを不要としている．一方，関数などの変換が必要なデータに関しては，C の関数型からデータ変換を行うスタブコードを生成する機構を備える．

Blume [9] が提案する SML/NJ の他言語関数インターフェースは，C の型システムを ML でエンコードすることによって，データの検査や操作を可能にする．これによって，型安全に C の値を ML から操作することが可能である．Ctypes や Cbuf も同様に，C のオブジェクト型や関数型を OCaml でエンコードすることによって，型安全に他言語関数の呼び出しを可能にしている．

7　結論

本研究では，OCaml の FFI ライブラリである Ctypes を拡張することで，バインド先の参照渡し関数の型や確保するメモリのサイズの情報を元に C と OCaml のバインディングを機械的に生成するツールである Cbuf を実装した．Cbuf を用いて Ctypes を用いた既存のバインディングを置き換え，手動で記述するプログラムの記述量の変化などを定量的に評価した．また，実験によって，Ctypes と Cbuf の C の関数の呼び出しにかかる時間を比較した．

バインディングの置き換えから，参照渡し関数のバインドにおいて，関数の Cbuf を用いることによって手動でのプログラムの記述量が大幅に削減されることが確認できた．これによりエラーの機会が減ることが期待される．また，実験によって関数の呼び出し時間は Ctypes と比べて大きな差がないことを確認した．この実験結果から，Cbuf によって既存の Ctypes によるバインディングが十分置き換え可能であることが言える．

参照渡し変数の型については，Cbuf は現在 OCaml の `bytes` 型のみをサポートしている．また，事前調査の結果に見られた，C の関数の返り値の OCaml の例外への紐付けは，Cbuf は現在サポートしていない．柔軟な型の指定や例外の紐付けのサポートは，今後の課題である．

謝辞

本研究の機会を頂き，多くのご指導を賜りました五十嵐淳教授，末永幸平准教授，和賀正樹助教に深く感謝いたします．また，毎週のミーティングでアドバイスをいただいた国立情報学研究所の関山太朗助教，多くの助言を頂いた研究室の皆様に深く感謝いたします．

参考文献

[1] Yallop, J., Sheets, D. and Madhavapeddy, A.: A modular foreign function interface, *Science of Computer Programming*, Vol. 164, pp. 82–97 (2018). Special issue of selected papers from FLOPS 2016.

[2] Green, A.: The libffi home page, `http://sourceware.org/libffi`. (Accessed on 02/01/2022).

[3] Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D. and Vouillon, J.: The OCaml Manual - Chapter 20 Interfacing C with OCaml, `https://ocaml.org/manual/intfc.html`. (Accessed on 02/01/2022).

[4] Sheets, D., Zotov, P. and Canou, B.: ahrefs/ocaml-sodium: Binding to libsodium 1.0.9+, a shared library wrapper for djb's NaCl, `https://github.com/ahrefs/ocaml-sodium`. (Accessed on 02/01/2022).

[5] Roché, L.: Khady/ocaml-argon2: Ocaml bindings to Argon2, `https://github.com/Khady/ocaml-argon2`. (Accessed on 02/01/2022).

[6] Furr, M. and Foster, J. S.: Checking Type Safety of Foreign Function Calls, *ACM Trans. Program. Lang. Syst.*, Vol. 30, No. 4 (2008).

[7] Beazley, D. M.: SWIG : An Easy to Use Tool for Integrating Scripting Languages with C and C++, USENIX Tcl/Tk Workshop (1996).

[8] 上野雄大, 大堀淳: SML#の外部関数インターフェース, コンピュータ ソフトウェア, Vol. 27, No. 2, pp. 2_142–2_168 (2010).

[9] Blume, M.: No-Longer-Foreign: Teaching an ML compiler to speak C "natively", *Electronic Notes in Theoretical Computer Science*, Vol. 59, pp. 36–52 (2001).