Fail-Safe C のためのインターフェイス定義言語

末永 幸平 * †　　大岩 寛 ‡　　住井 英二郎 ‡　　米澤 明憲 ‡

† 東京大学理学部情報科学科

‡ 東京大学大学院情報理工学系研究科コンピュータ科学専攻

概 要

安全な C 言語処理系である Fail-Safe C から外部関数を呼び出すための wrapper を, 半自動的に生成する手法を提案する. 我々は外部関数のインターフェイスを記述するための言語を定義し, その言語で記述された情報をもとに wrapper を自動生成するというアプローチをとった. 我々の定義した言語では, 型に**属性**と呼ばれる追加情報を与えて, wrapper の行うべき作業を記述する. 我々はいくつかの外部関数について, wrapper を介することで生ずるオーバーヘッドを測定した.

1　序論

Fail-Safe C [7] は東京大学の米澤研究室において, 大岩らによって開発されている C 言語の安全な実装である. この実装では, プログラム実行の安全性を, 動的検査によって保証している. この安全性を効率的に保証するために, Fail-Safe C では通常の C と異なるデータ表現を採用している. 具体的には, すべてのメモリブロックにはサイズや型情報を持ったヘッダが付加されている. また, ポインタや整数は 2 ワードで表現されている.

このようにデータ表現が異なっているので, Fail-Safe C から通常の処理系でコンパイルされた関数を呼ぶには, 引数のデータ表現を変換しなくてはならない. さらに安全性を保証するには, 引数の NULL チェックなど, 何らかの事前条件を検査しなければならないような関数もある. これらの作業を行う wrapper をすべての関数について手で書くのは, 面倒なだけでなく誤りを起こしやすい.

本研究では, そのような wrapper を半自動的に生成するためのインターフェイス定義言語を設計, 実装した. 設計に当たっては, Objective Caml から C 言語の関数を呼び出す wrapper を生成するための IDL である, CamlIDL [4] を参考にした. 我々は単純なテストプログラムによって, wrapper を介することで生ずる種々のオーバーヘッドを計測した. さらに, より現実的な例として, ファイルのコピーを行うプログラムの性能を測定した.

2　Fail-Safe C

2.1　データの内部表現

この節では, Fail-Safe C の動作と独自の内部表現について, その概要を説明する. 詳細は [7] を参照されたい.

メモリブロック　Fail-Safe C で確保されるメモリブロックはすべてヘッダを持っており, ヘッダはサイズ情報と *TypeInfo* と呼ばれる型情報を持っている. メモリアクセスのたびに, サイズ情報を用いて boundary check が行われる. TypeInfo は, 型名と *handler method* と呼ばれる関数へのポインタのテーブルをメンバとして持つ構造体である.

*kohei@yl.is.s.u-tokyo.ac.jp

Handler method　C では, ポインタを別の型のポインタにキャストすることができる．このため, ポインタを介したメモリアクセスの際に, ポインタの静的な型とメモリブロックに格納されているデータの型が一致しないことがある．この型の不一致は, 危険な振る舞いを引き起こすことがあるので, メモリアクセスの際には型チェックを行って, メモリブロック内のデータの型に従ってアクセスする必要がある．しかし, メモリブロック内でのデータの並び方は型によって異なるため, 型ごとに固有のメモリアクセス用の関数が必要となる．

Fail-Safe C では, このようなメモリアクセス用関数として, TypeInfo 内に handler method と呼ばれる関数を用意している．このうち, データを読むための handler method は, 引数で指定された場所から, 指定された大きさのデータを読み, `void *`にキャストして返す．データを書くための handler method は, 引数で指定された場所に, 引数として渡された `void *`型のデータを書く．このとき, アラインメントのチェックなどの必要な検査は, 各々のメモリブロック内のデータの並びに従って適切に行われる．メモリブロック自身が持っている handler method を使うことで, ポインタの静的な型が実際の型と異なっている場合でも, 安全にメモリアクセスを行うことができる．

Fat pointer　ポインタは, 指しているメモリブロックの base と, その base からの offset の 2 ワードで表現される． これにより, メモリアクセスの際の boundary check が高速に行える．また, ヘッダは必ず base の手前に置かれるので, ヘッダへのアクセスを安全かつ高速に行うことができる．さらに, ポインタ演算の際に二つのポインタの base を比較することで, 異なるメモリブロックを指すポインタの間の無効な演算 (比較や減算) を検出することができる．

base の最下位ビットは, ポインタがキャストされたことがあるかどうかを示す**キャストフラグ**になっている． このフラグが立っていなければ, handler method を使うことなく, 通常の方法で安全にメモリアクセスを行うことができる．フラグが立っている場合は, ポインタの静的な型がメモリ中のデータの型と一致するとは限らないので, handler method を使ってメモリアクセスをする必要がある．

Fat pointer とメモリブロック, および型情報の構造を, 図 1 に示す．

Fat integer　通常の C では, ポインタを十分なサイズの整数にキャストできることが保証されている．このため Fail-Safe C では, 整数も base と offset の 2 ワードで表現される．通常の整数は, base を 0 とする．(したがって, 任意の整数をポインタとして用いるようなプログラムは, Fail-Safe C では動作しない.) ポインタを整数にキャストする際は, base と offset をそのまま保持する．この整数を再びポインタにキャストするときは, base からヘッダの型情報をたどり, キャストしようとしているポインタの型と一致していなければ, 前述のフラグを立てる．これにより, ポインタを整数へキャストし再びポインタに戻した場合でも, メモリアクセスのチェックを正しく行うことができる．なお, 2 ワードで表現されるのはポインタを保持するのに十分なサイズを持つ整数だけで, たとえば `char` 型の整数のようにサイズの小さな整数は, 通常の C と同じように表現される．

2.2　Fail-Safe C の保証する安全性

この節では, Fail-Safe C がどのような安全性を保証しようとしているかを述べる．C 言語の安全でない動作のほとんどは, 正しくないメモリアクセスに起因する．このため Fail-Safe C は, メモリアクセスの安全性を保証することに重点を置いて設計されている．しかし, C 言語の安全でない振る舞いは, メモリアクセス以外にも存在する．

[8, pp. 7–8] でも述べられているように, あるプログラミング言語が安全であるためには, その言語で書かれたすべてのプログラムの振る舞いが, 言語仕様から決定できることが必要である．しかし,

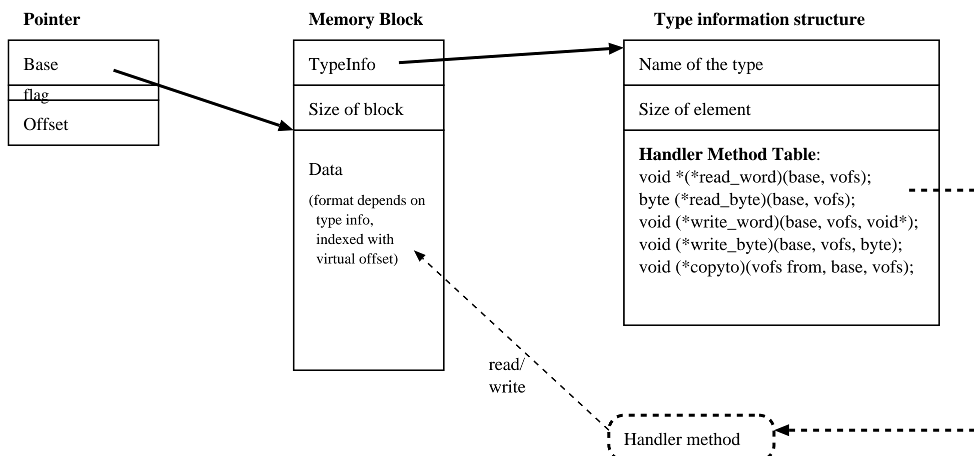

図 1: Fat pointer とメモリブロックおよび型情報の構造

ANSI-C の仕様では, 0 による除算が行われた場合など, 多くの場合の振る舞いが “不定である” と表現されている. このことを考えると, Fail-Safe C の安全性は以下のように説明できる.

> 多くの人が C 言語の意味論として認めるような意味論が一つ固定されたと考える[1]. プログラムがこの意味論において不定な振る舞いをしようとしたときには, Fail-Safe C は確実にそのプログラムを終了させる.

この説明をより厳密にするためには, C の意味論を形式的に定めなくてはならないが, 本論文では C の意味論について議論することはせず, この非形式的な定義を使うことにする.

3 wrapper の内容

3.1 wrapper の保証する安全性

この節では, 我々の IDL が生成する wrapper が保証する安全性について説明する. 我々は, IDL を設計するに当たって 3 つの仮定をおいた. まず 1 つ目は, 保証しようとする安全性を, Fail-Safe C の保証する安全性に限るということである. これにより, たとえば fork bomb のような Fail-Safe C が関与しない危険な振る舞いを無視することになる.

2 つ目は, Fail-Safe C 自体はバグを含んでいないということである. これによって, 外部関数の呼び出し前には, Fail-Safe C の保証する安全性は必ず成り立っていることになる.

3 つ目は, IDL プロセッサに与えられたインターフェイス記述が, 実際に外部関数の実装に合っているということ, つまりインターフェイス記述は間違っていないということである. この仮定によって, インターフェイス記述そのものが間違っていた場合は, 安全性は必ずしも保証されないことになる.

以上の仮定をまとめて, Fail-Safe C IDL の生成する wrapper が保証する安全性は次のようになる.

> 以下の条件が成り立っている場合, Fail-Safe C IDL の生成する wrapper は外部関数の呼び出しを安全に実行する:
>
> - 外部関数の呼び出し前に, Fail-Safe C の保証する安全性が成り立っている.
> - 外部関数のインターフェイス記述が,

[1] この意味論として ANSI-C をとることも考えられるが, ほとんどのプログラムは ANSI-C の範囲を超えており, また Fail-Safe C も ANSI-C の範囲にないプログラムをサポートしているので, “多くの人が認める意味論” とした.

実際の実装に適合している.

3.2 Case study: Linux システムコール

一般的な wrapper について考える前に, Linux の read, accept システムコールを例にとって, それぞれどのような wrapper を生成すれば良いかを考えることにする.

3.2.1 read システムコール

read システムコールの宣言は

```
int read(int fd, char *buf, int n);
```

となっている. fd はファイルディスクリプタ, n は buf が指しているメモリブロックのバイト数である. read システムコールを呼び出した結果は, 返り値が-1 でない場合は, ファイルから buf の指すメモリブロックに, 返り値の示すバイト数だけデータが読み込まれる. 返り値が-1 であった場合は, 何らかのエラーのためにデータが読み込めなかったことを示す. この場合, グローバル変数 errno にエラーの原因を示す整数が書き込まれる.

事前条件 まずは, read を安全に呼び出すための事前条件について考える. n はバイト数であるから, n ≥ 0 であることを確かめなければならない. また, buf の指すメモリブロックには, ファイルから読んだ内容が書き込まれるので, buf が NULL ポインタでないことを確かめなければならない. これは buf の base 部分が 0 でないことで確かめることができる. さらに, 実際にバッファの大きさが n バイトあること, すなわち buf から buf + n − 1 までの範囲がアクセス可能であることも確かめなければならない. これは buf の指すメモリブロックのサイズ情報から確かめられる.

データ表現の変換 (呼び出し前) 事前条件が成り立っていることを確認した上で, 引数に対して Fail-Safe C の表現から通常の C の表現への変換を行う. 整数である fd と n は, base と offset の 2 ワード表現を, 1 ワードの表現に変換する. また, buf の指しているメモリブロックに対応して n バイトのバッファを確保し, ポインタである buf をその先頭へのポインタに変換する.[2]

データ表現の変換 (呼び出し後) 関数が返った後は, 通常の C の表現から Fail-Safe C の表現への変換を行う. まず返り値を Fail-Safe C の表現に変換する. 返り値は通常の整数なので, base 部分が 0 の整数に変換する.

また, 返り値_ret が-1 でなければ, buf から buf + _ret − 1 の内容は書き換わっているので, この変更を反映しなければならない. これには, 呼び出し前に確保したメモリから, buf に変換を行いながら書き込みを行えばよい. 書き込みが終わった後に, 確保したメモリを解放する.

また, 返り値が-1 であれば, グローバル変数 errno にエラーの内容が書き込まれる. このように, 外部関数から変更する可能性のあるグローバル変数には, 通常の C の表現を保持する領域と, Fail-Safe C の表現を保持する領域の二つを用意しておく. この場合は, errno の値を通常の C の表現で保持している領域から, Fail-Safe C の表現で保持している領域に, 変換を行ったうえでコピーする.

3.2.2 accept システムコール

続いて, accept システムコールについて考える. accept システムコールの宣言は

```
int accept(int socket,
           struct sockaddr *address,
           unsigned int *address_len);
```

となっている. sockaddr 構造体は

```
struct sockaddr {
    unsigned short sa_family;
```

[2] 実際には Fail-Safe C の char の表現は通常の C と同じなので, read についてはこの変換は必要ない.

```
    char        sa_data[14];
};
```

と宣言されている．socket はソケットのディスクリプタ，address は sockaddr 構造体へのポインタ，address_len は address の指す構造体のサイズが書かれた整数へのポインタである．address は NULL でも良い．その場合，address_len は無視される．

address には，使用するソケットの種類に合った構造体へのポインタを，sockaddr 構造体へのポインタにキャストして渡す．たとえば，IP 用のソケットでは，sockaddr_in 構造体へのポインタを sockaddr 構造体へのポインタにキャストして渡す．このようなソケットの種類に応じた構造体は，先頭に unsigned short 型のメンバを含んでいるということのみが保証されている．このため，sa_data メンバの静的な型は，実際に指しているメモリブロックの型とは異なっている場合がある．

accept の返り値が-1 でない場合，返り値は新しく生成されたソケットのディスクリプタである．この場合，address が NULL で無ければ，address の指す先に，生成されたソケットの通信先のアドレスが書かれ，そのサイズが address_len の指す先に書かれる．返り値が-1 である場合は，何らかのエラーが起こっており，グローバル変数 errno にエラーの原因を示す整数が書き込まれる．

事前条件　まずは，このシステムコールを呼び出すための事前条件について考える．address が NULL でない場合，次の 4 つの条件が成り立っていなければならない:

- address->sa_data が NULL でないこと．
- address_len が NULL でないこと．
- *address_len ≥ 0.
- address の指すメモリブロックのサイズが，address_len バイト以上であること．

このうち，1 つ目と 2 つ目の条件は，それぞれ address->sa_data と address_len の base 部分が 0 でないことによって確かめられる．4 つ目の条件は，address の指すメモリブロックのヘッダ情報から確かめられる．

データ表現の変換 (呼び出し前)　続いて，引数を Fail-Safe C の表現から通常の C の表現へと変換する．まず，socket は整数なので，2 ワード表現を 1 ワード表現に変換する．

次に，address != NULL ならば，*address_len バイトのメモリ領域を確保して，address の指す構造体の内容をコピーする．このとき，構造体の各メンバのデータ表現も変換する必要がある．sa_family メンバの型は short で，Fail-Safe C と通常の C の間で表現の違いがないので，そのままコピーしてよい．sa_data メンバについては，実際に指しているメモリブロックと同じ大きさだけコピーする．このメンバは，先に述べたように，静的な型と異なる型のメモリブロックを指している可能性があるが，handler method を使うことで安全にコピーを行うことができる．

最後に，address_len が NULL でないときには，整数を用意して*address_len の値を 2 ワード表現から 1 ワード表現に変更して書き込み，その整数へのポインタを address_len の変換結果とする．

データ表現の変換 (呼び出し後)　関数が返った後は，通常の C の表現から Fail-Safe C の表現への変換を行う．返り値の整数と，グローバル変数 errno については，3.2.1 で述べた read システムコールの場合と同様の変換を行う．

返り値が-1 でなければ，address と address_len の指す先が書き換えられている可能性があるので，その変更を反映する．address_len については，ラッパー内で用意した整数の表現を Fail-Safe C の表現に変換し，address_len の指す先に書き込む．address についても，各メンバの表現を変換しながら，変更を書き戻す．このとき，sa_data メンバの変換では，呼び出し前の変換と同様に handler method

を使って書き戻す.

3.3 IDL が必要とする情報

ここまでの内容から, wrapper の生成には以下のような情報が必要であることが分かる.

- ポインタが NULL になってもよいかどうか
- ポインタについて, それを介してどの範囲のメモリがアクセス可能でなければならないか
- 関数について, 関数から返ったあとにどのような副作用 (ポインタを渡しているメモリへの変更, グローバル変数の変更など) が生じるか
- その他, read の例における $n \geq 0$ など, 関数呼び出しの前に成り立たせておかなければならない条件

3.4 制御の流れ

この節では, wrapper を用いたアプリケーションの制御の流れについて説明する. 図 1 に, 関数呼び出しの流れを示す. この図では, main, wrapper_f, f の 3 つの関数と, グローバル g を説明のために用いる. 先に説明したように, 外部関数がグローバル変数にアクセスする場合, 変数 1 つにつき 2 つの領域が用意される. この図では, Fail-Safe C 表現の値を保持する領域を fsc_g, 通常の C の表現の値を保持する領域を g で示している.

以下, 関数呼び出しの流れを順に説明する. main は Fail-Safe C でコンパイルされた関数である. この関数は通常の C 処理系でコンパイルされた関数 f を呼び出している. この呼び出しは通常の C 処理系でコンパイルされた wrapper である wrapper_f への呼び出しで置き換えられる [図中の (1)]. wrapper_f は事前条件のチェックと引数を変換した上で, 外部関数の f を呼び出す [図中の (2)]. f がグローバル変数 g の内容を変更する場合, wrapper_f は変更を fsc_g に反映させる [図中の (5)]. 最後に f が返ったら [図中の (3)], wrapper_f は返り値の変換などを行って, 制御を main に戻す [図中の (4)].

3.5 一般的な wrapper の構成

ここまでの例をもとにして, 一般的な wrapper の構成を考える.

wrapper は以下の 5 つのフェーズからなる.

- Precondition Checking
- Decoding and Allocation
- Call
- Deallocation and Encoding
- Return

以下, 各フェーズについて, その内容を説明する.

Precondition Checking

関数の引数について, 指定された事前条件が成り立っているかどうかを確認するフェーズである. 事前条件としては, ポインタが NULL であるかどうか, 指定した範囲が安全にアクセス可能であるかどうか, n $\geq$ 0 のようにユーザが指定する条件などがある.

Decoding and Allocation

このフェーズではデータ表現の変換を行う. 整数については, base と offset の 2 ワード表現を base + offset で 1 ワード表現に直す. ポインタについては, 指しているメモリブロックと同じ大きさのメモリを確保し, データ表現の変換を行いながら内容をコピーする.

Call

このフェーズでは関数呼び出しを行う. ここまでの二つのフェーズが正常に終了していて, 関数の実

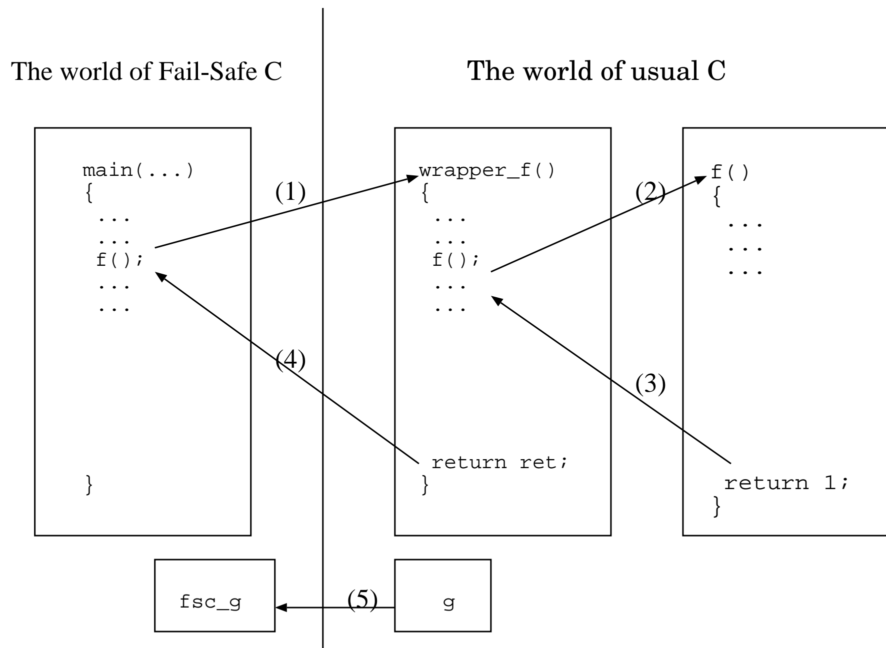

図 2: wrapper を用いたアプリケーションの関数呼び出しの流れ

装が記述されたインターフェイスに合致していれば,この関数呼び出しは安全に実行することができる.(関数呼び出し後に成り立っていなければならないなんらかの事後条件があるとしても, wrapper はその条件のチェックを行わない. 3.1 節で述べたように, インターフェイス記述は外部関数の実装に適合していると仮定しているので, 関数呼び出し前に事前条件が成り立っていれば, 事後条件は成り立っていると仮定できる.)

Deallocation and Encoding

返り値の表現の変換や, 副作用を Fail-Safe C の表現に反映する作業を行うフェーズである. 返り値と, ポインタ渡ししたメモリブロックについては Decoding and Allocation フェーズと逆の作業を行えばよい. グローバル変数に副作用が生じうる場合は, 通常の変数の他に Fail-Safe C 表現の変数を用意して前者の変更を後者に反映する. (逆に, 関数の呼び出し前に Fail-Safe C 表現のグローバル変数の値を変換して, 通常の変数に書き込むことが必要になる関数も考えられるが, 現時点でそのような関数を扱っていないので実装していない.)

Return

返り値がある関数の場合は, Fail-Safe C の表現に変換した値を返す.

4 言語の詳細

4.1 文法

表 1 に, Fail-Safe C IDL の文法の一部を示す.

type は C の型である. ident は C の識別子として使える文字列, pure-expr は副作用を伴わない C の expression である. なお, pure-expr の中で返り

attributes	::= ϵ | [{ attribute, }]
attribute	::= ident
	| ident ({ pure-expr, })
	| * attribute
global-decl	::= attributes type declarator
struct-decl	::= struct {
	{ attributes type declarator }
	}
declarator	::= { *[const] } direct-declarator
direct-declarator	::= ident
	| (declarator)
	| direct-declarator [[pure-expr]]
	| direct-declarator (param-list)
param-list	::= { attributes type declarator, }
function-decl	::= attributes_1 type { * [const] } ident (param-list) attributes_2

表 1: Fail-Safe C IDL の文法: { T } で T の 0 個以上の並びを, { T, } で T の一つ以上のコンマ区切りリストを表す.

値を参照する必要があるときのために, 返り値を示す予約語として _ret が用意されている.

attributes は属性と呼ばれる, wrapper を生成するために必要となる型の追加情報である. *の付いた属性はポインタ型の属性として与えられる. T*型に属性*A が与えられた場合は, その指す先すなわち T に属性 A が与えられたことになる.

global-decl は外部関数からアクセスする可能性のあるグローバル変数の宣言である. 属性を伴った C のグローバル変数の宣言になっている.

struct-decl は, インターフェイス記述に使う構造体の宣言である. C の構造体の宣言に, 各メンバの属性を加えたものになっている.

function-decl は, 関数のインターフェイスの宣言である. attributes_1 には, 返り値の型に与えられる属性を記述する. 一方, attributes_2 には事前条件など, この関数自体の属性を記述する.

4.2 属性とその意味

続いて, 我々の IDL が用意する属性を示す.

always_null, maybe_null, never_null これらの属性はポインタ型に与えられる. それぞれ, その型の変数が常に NULL でなければならないこと, NULL であったり NULL でなかったりすること, NULL であってはならないことを示す. read システムコールについては, buf に never_null が与えられることになる.

この属性が与えられた変数については, Precondition フェーズでそれぞれの条件が満たされているかどうかを確かめる.

can_access_in_elem(e_1, e_2) ポインタ型に与えられる. e_1, e_2 は pure-expr である. この属性が与えられたときには, その型の変数を p としたときに $p+e_1$ から $p+e_2$ までが読み書き可能でなければな

らないことを示す．`read` システムコールでは `buf` に `can_access_in_elem(0, n - 1)` が与えられる．

この属性が与えられた変数については, Precondition Checking フェーズで実際に指定された範囲が読み書き可能であることをメモリブロックのヘッダ情報から確認し, Decoding and Allocation フェーズで, 内容を変換しながらコピーする．

`can_access_in_byte(`e`)`　ポインタ型に与えられる． e は pure-expr である． この属性が与えられた変数 p について, 指しているメモリブロックの型に関係なく, p から e バイトが p を介して読み書き可能でなければならないことを示す． `accept` システムコールの例では, `address` に `can_access_in_byte(*address_len)` が与えられる[3]．

この属性が与えられた変数については, Precondition Checking フェーズで, 指定されたバイト数が読み書き可能であることメモリブロックのヘッダ情報から確認し, Decoding and Allocation フェーズで, 内容を変換しながらコピーする．

`string`　char 型へのポインタ, char 型の配列型に与えられる．この型の変数が `NULL` 文字で終端が示された C の文字列であることを示す．この属性が与えられた変数については Precondition Checking フェーズで実際に `NULL` で終端が示されていることを確認し, Decoding and Allocation フェーズでメモリを確保してその内容をコピーする．

`write_global(`e_1`,` **ident**`)`　関数宣言に与えられる．ident はグローバル変数の名前である．この関数の呼び出しを行った後で, $e_1 \neq 0$ が成り立っているときには, グローバル変数 ident の値が書き換えられていることを示す．`read` システムコールの場合, `write_global(_ret == -1, errno)` が与えられる．

[3] ここで `address_len` の指す先を参照するので, `address_len` の事前条件は `address` の事前条件よりも先にチェックしなければならない．

この属性が指定されていたときには, Deallocation and Encoding フェーズで, 従来のグローバル変数の内容を Fail-Safe C 表現のグローバル変数の方に書き戻す．

`write(`e`,` **ident**`,` e_1`,` e_2`)`　関数宣言に与えられる．ident は引数のうち `const` でないポインタか `const` でない配列である． この関数の呼び出しを行った後で, $e \neq 0$ が成り立っていれば, ident $+ e_1$ から ident $+ e_2$ が書き換えられている可能性があることを示す．`read` システムコールでは `write(_ret != -1, buf, 0, _ret - 1)` が与えられる．

この属性が与えられた時には, Deallocation and Encoding フェーズで指定された範囲のメモリについて, 書き戻しを行う．e_1, e_2 は省略可能である．省略された場合は `can_access_in_elem` や `string` で示されたアクセス可能な領域全体を書き戻す．

`precond(`e`)`　関数宣言に与えられる．それぞれ関数の呼び出しの前, 呼び出しの後に $e \neq 0$ が成り立ってなければならないことを示す．`read` システムコールの場合, `precond(n >= 0)` が与えられる．

この属性で指定された条件は, それぞれ Precondition Checking フェーズでチェックされる．

4.3　インターフェイス記述の例

インターフェイス記述の例を表 3 に示す．Linux システムコールの `open`, `read`, `write`, `accept` についてインターフェイスを記述した．

関数の引数の名前の有効範囲は, その関数宣言全体に及ぶ．たとえば `read` では, `buf` に与えられる属性に `nbytes` が現れている．

なお, 3.5 で述べた wrapper の各フェーズは, 実際には互いに順序が前後することがある．たとえば, `read` に与えられている `precond(nbytes >= 0)` は, `nbytes` の decode が終わってからでないと検証できない．

```
int errno;

struct sockaddr {
    unsigned short sa_family;
    [never_null] char *sa_data;
};

int open([never_null, string] const char *path, int flags)
  [write_global(_ret == -1, errno)];
int read(int fildes,
         [never_null,
          can_access_in_elem(0, nbytes - 1),
          write(_ret != -1, 0, _ret - 1)] char *buf,
         int nbytes)
  [precond(nbytes >= 0), write_global(_ret == -1, errno)];
int write(int fildes,
          [never_null, can_access_in_elem(0, nbytes - 1)] const char *buf,
          int nbytes)
  [precond(nbytes >= 0), write_global(_ret == -1, errno)];
int accept(int socket,
           [can_access_in_byte(*address_len),
            write(_ret != -1)] struct sockaddr *address,
           [write(_ret != -1)] int *address_len)
  [precond(address == NULL || address_len != NULL),
   write_global(_ret == -1, errno)];
```

図 3: インターフェイス記述の例

5 実験結果

これまでに述べた IDL プロセッサを Objective Caml を用いて実装した．この実装が生成した wrapper を使って, wrapper を介して外部関数を呼び出すプログラムを Fail-Safe C のランタイムライブラリとリンクして実行した場合と, 外部関数を直接呼び出すプログラムを gcc でコンパイルして実行した場合の, 呼び出しにかかる時間を測定した．なお, 後者でのコンパイルオプションは`-O3`とした．プログラムの実行は, Sun UltraSPARC-II 400 MHz CPU, メインメモリ 13.0 GB のマシン上で行った．

整数の変換に伴うオーバーヘッド　与えられた整数引数に 1 を加えた結果を返す外部関数 succ を作成して, 通常の C でコンパイルした．この関数を 0 から 9999999 までの整数に順に適用するプログラムを作り, succ の wrapper を介して動作させた場合と, そうでない場合について動作時間の比較を行った．この実験で得られた結果は, 整数の decoding にかかるオーバーヘッドを表していると考えられる．結果を表 2 に示す．wrapper を介した場合, 6%のオーバーヘッドを伴っている．

ポインタの変換に伴うオーバーヘッド　次に, 要素数 10^7 個の `char` 型の配列を受け取り, それぞれの要素に 1 を加える外部関数 arraysucc を作成した．実際に配列を用意した上でこの関数を呼び出す関数を作り, 動作時間の比較を行った．この実験で得られた結果は, ポインタ型の引数について, メモリを確保してその内容をコピーすることで発生するオーバーヘッドを表していると考えられる．結果を表 2 に示す．wrapper を介した場合, 199%のオーバーヘッドを伴っており, メモリの確保と内容のコピーにかかるオーバーヘッドが, 整数の変換に伴うオーバーヘッドに比べて非常に大きいことを調べている．

この実験では, wrapper のそれぞれのフェーズが wrapper の全実行時間のどれくらいの割合を占めているかも測定した．結果を表 3 に示す．wrapper の全実行時間の中で, 約 55%ほどが Decoding and Allocation フェーズに費やされている．

より現実的な例でのオーバーヘッド　最後に, Linux システムコールの `open`, `read`, `write` を用いて, 10^6 バイトのファイルをコピーするプログラムを作成し, システムコールの wrapper を介した場合とそうでない場合で動作時間の比較を行った．結果を 2 に示す．wrapper を介した場合, 58%のオーバーヘッドがかかっている．

この実験についても, wrapper の各フェーズでの実行時間を測定した．`read` と `write` の wrapper での測定結果を表 4 に示す．cp では, 外部関数内でファイルアクセスを行うため, Call フェーズに多くの時間が費やされている．しかし, このフェーズを除いて wrapper 自体の実行時間に注目すると, やはり Decoding and Allocation フェーズが実行時間の多くを占めていることが分かる．

5.1 考察

以上の実験から, wrapper のオーバーヘッドの多くはは Decoding and Allocation フェーズに起因していることが分かった．したがって, wrapper 全体のオーバーヘッドを削減するには, このフェーズの作業を減らすことが重要である．

考えられる方法の一つに, Decoding and Allocation フェーズで行われるメモリブロックの内容のコピーを省略する方法がある．たとえば `read` システムコールの場合, 引数 `buf` の指すメモリブロックの内容は読み出されることがなく, かつ必ず上書きされる．したがって, 関数呼び出し前の内容のコピーは不要である．

また, Fail-Safe C と通常の C でメモリブロックの構造が同じである場合は, 呼び出し前にバッファを確保せずに, もとのメモリブロックのアドレスを外部関数に渡すことができる．たとえば, `read` システムコールの引数 `buf` に, `char` 型の値が格納されるメモリブロックを指しているポインタが渡された

	succ	arraysucc	cp
with wrapper	234	597	144
without wrapper	220	200	91
overhead	6	199	58

表 2: 各々のプログラムでのオーバーヘッド (単位:msec)

	P	D	C	E	合計
実行時間 (msec)	0	326	110	160	596
割合 (%)	0	54.7	18.5	26.8	-

表 3: ポインタの変換に伴うオーバーヘッド. P: Precondition Checking, D: Decoding and Allocation, C: Call, E: Deallocation and Encoding

場合がこれに当たる．このような条件を実行時に検査するコードを出力することで, バッファの確保とコピーのオーバーヘッドを削減することができる．

6 関連研究

言語間のインターフェイスに関する研究 関数型言語から外部関数を呼び出す wrapper を生成する IDL として, H/Direct [3, 2] や CamlIDL [4] がある．前者は Haskell の, 後者は Objective Caml の IDL である．Scheme の処理系にも外部関数を呼び出すための機構を備えているものがある．たとえば Chez Scheme は, 独立した IDL は持たないものの, `foreign-procedure` 関数に外部関数のインターフェイスを与えて, 外部関数を呼び出す Scheme の関数を作ることができる．これらの IDL はデータ表現の変換に重きをおいており, 我々の IDL で行っているような事前条件のチェックを行わない．たとえば, 我々の言語で用意されている `can_access_in_elem` 属性に対応する属性として, CamlIDL では `size_is` 属性が用意されているが, wrapper はこの属性が指定されてもバッファサイズのチェックを行わない．このため, 引数として渡されたバッファのサイズが十分に大きくない場合, 外部関数呼び出しが安全に行えないことがある．

安全な C の処理系に関する研究 Fail-Safe C 以外にも, 安全な C の処理系に関する研究は行われている．SafeC [1] は, 通常の C のポインタの代わりに *safe pointer* を用いることで, メモリアクセスの安全性を保証する．safe pointer は, メモリブロックの base アドレスやメモリブロックがスタック上にあるかどうかなどの追加情報を持つポインタである．Fail-Safe C もポインタに追加情報を持たせるというアプローチをとっているが, Fail-Safe C の fat pointer の方がサイズが小さく, よりシンプルになっている．また, Fail-Safe C は handler method を使ってメモリアクセスを行うので, SafeC よりも柔軟なメモリアクセスが行える．

CCured [6, 5] はポインタの使われ方をコンパイル時に解析することで, 実行時のチェックをできるだけ省くというアプローチをとっている．このため, 必要となる動的なチェックの量は, Fail-Safe C よりも CCured の方が少ない．一方, Fail-Safe C は動的チェック時にキャストフラグをチェックして handler method の呼び出しを省くことで, 動的チェック自体の効率を上げている．また, SafeC のところで述べたのと同様に, Fail-Safe C は handler method を用いるので, より柔軟なメモリアクセスを行える．

また, CCured で外部関数を呼び出すには, wrapper を書く必要がある．この wrapper には, 通常の

	P	D	C	E	Total
`read` の wrapper の実行時間 (msec)	1	16	46	14	77
`write` の wrapper の実行時間 (msec)	1	16	73	4	94

表 4: `read` と `write` の wrapper の各フェーズでの実行時間. P: Precondition Checking, D: Decoding and Allocation C: Call, E: Deallocation and Encoding

C の表現で引数が渡され, その引数の読み書きは安全であると仮定される. 従って, 外部関数を呼び出す際の安全性は必ずしも保証されない. 一方, Fail-Safe C では, wrapper に Fail-Safe C の表現で引数を渡し, wrapper 内で読み書きの安全性を確認した上で, 引数の表現を変換して外部関数を呼び出す. そのため, インターフェイス記述が実装に適合していれば, 外部関数の呼び出しを安全に行えることが保証される.

7 結論と今後の課題

本研究では, Fail-Safe C から従来の C コンパイラでコンパイルされた関数を呼び出すための wrapper を半自動的に生成するための手法を提案した.

今後の課題として, 現存する実用的なプログラムに我々のアプローチを適用することが挙げられる. 現時点では, Fail-Safe C の処理系が完成していないため, そのようなプログラムについて, 提案した手法の有効性を検証することができない. 処理系が完成し次第, より大きなプログラムについても実験を行う予定である. またその過程で, システムコールや C のライブラリ関数を調査してそれらに共通な性質を抽出することにより, 新たな属性を付け加えることも考えている.

また, 外部関数の中には, システムコール, C の標準ライブラリ関数, それらを使用する既存のライブラリといった, いくつかの異なるレベルが存在するが, このうち, どこまでを外部関数として扱うべきか検証することも重要である. OS のカーネルを Fail-Safe C でコンパイルするなどして, より低レベルな関数まで Fail-Safe C で扱うことにすれば, プログラムの安全性はより高まる. しかし, その場合 Fail-Safe C に起因するオーバーヘッドが, OS の性能に大きく影響することも考えられる. また, メモリマップト I/O など, ハードウェアに大きく依存しているために Fail-Safe C で扱うのが難しいと思われる操作も存在する. これらのトレードオフについての実験を行い, どこまでを外部関数とするか検証する必要がある.

謝辞

本研究を進めるにあたり, 東京大学の米澤研究室のメンバーから多くのアドバイスを頂きました. また, 査読者の方々からも有益なコメントを頂きました. ここに感謝の意を表します.

参考文献

[1] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *SIGPLAN Conference on Programming Language Design and Implementation*, pp. 290–301, 1994.

[2] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon L. Peyton Jones. H/Direct: A binary foreign language interface for Haskell. In *International Conference on Functional Programming*, pp. 153–162, 1998.

[3] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon L. Peyton Jones. Calling hell from

heaven and heaven from hell. In *International Conference on Functional Programming*, pp. 114–125, 1999.

[4] Xavier Leroy. *CamlIDL Users Manual*. INRIA Recquencourt, July 2001. http://camlidl.inria.fr/camlidl.

[5] George C. Necula. *CCured: User Manual*, February 2003. http://manju.cs.berkeley.edu/ccured/.

[6] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages*, pp. 128–139, 2002.

[7] Yutaka Oiwa, Tatsurou Sekiguchi, Eijiro Sumii, and Akinori Yonezawa. Fail-Safe ANSI-C Compiler: An approach to making C programs secure (progress report). In *Proceedings of International Symposium on Software Security, Tokyo, Japan, November 8–10, 2002*, Vol. 2609 of *Lecture Notes in Computer Science*. Springer-Verlag, February 2003.

[8] Benjamin C. Pierce. *Types and Programming Languages*. MIT press, 2002.