SecHack365

Althea

ちなみに、アルテアって読みます

～安全で安定したコードを簡単に書ける言語～

開発駆動コース 川合ゼミ 八木橋拓之

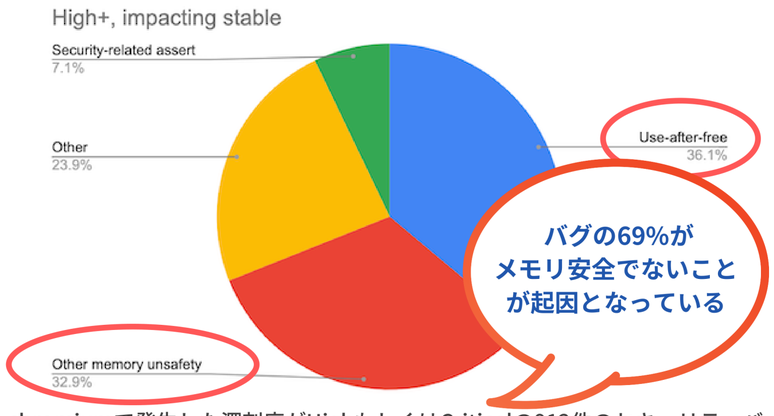
プログラミング時に発生する3つの課題を

① 性能が安定しない

Discord内のGoで書かれたサーバーのレスポンス時間とシステムCPU時間 [1]

現在主流の言語で採用されているGCの多くはStop The Worldという処理が完全に停止する期間があるため、安定した性能を実現することは難しい。

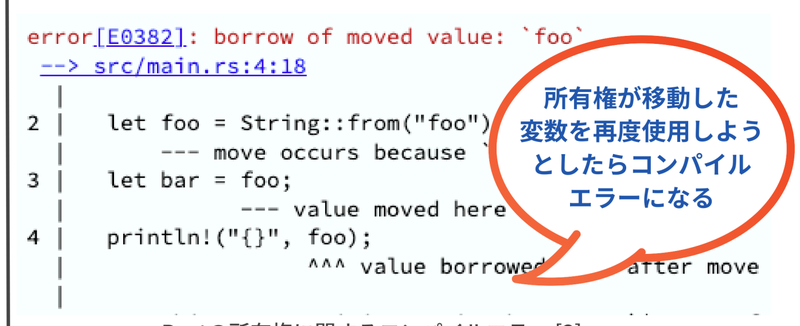
② 手動メモリ管理による脆弱性



chromiumで発生した深刻度がHighもしくはCriticalの912件のセキュリティバグの分類 [2]

手動でメモリ管理を行う言語は、Use-after-freeやメモリリーク等の問題が発生する可能性があり、場合によっては重大な脆弱性につながる。

③ 学習コストの増大



Rustの所有権に関するコンパイルエラー [3]

所有権やライフタイムは、管理や学習にコストがかかる。2016年に行われたRustの調査では、Rustの課題について1900人以上から回答があり、その中の4人に1人が学習曲線についてコメントしている。[4]

所有権付き参照カウントで解決

参照カウントをベースとしているため、メモリの解放処理が処理全体に分散し、Stop The Worldが発生しない。

HTTPサーバーを起動するコード例。handle内でメモリを解放するため、基本的に1リクエストごとにメモリが解放される。

Altheaコード

```
struct Foo {
  bar: i32,
}

func main() i32 {
  listen_and_serve(
    80,
    func handle() string {
      let foo: Foo = Foo {
        bar: 1,
      }
      "Welcome to Althea Server!"
    }
  )
}
```

コンパイル
(handle内を抜粋)

LLVM IR

```
define i8* @handle() {
entry:
  %malloccall = tail call i8* bitcast (i8* (i64)* @malloc to i8*
(i32)*)(i32 ptrtoint ({ i32, i32 }* getelementptr ({ i32, i32 }, { i32,
i32 }* null, i32 1) to i32))
  %alc_1 = bitcast i8* %malloccall to { i32, i32 }*
  %field_0_ptr = getelementptr inbounds { i32, i32 }, { i32, i32 }*
%alc_1, i32 0, i32 0
  store i32 1, i32* %field_0_ptr, align 4
  %raw = bitcast { i32, i32 }* %alc_1 to i8*
  call void @free(i8* %raw)
  %malloccall1 = tail call i8* bitcast (i8* (i64)* @malloc to i8*
(i32)*)(i32 ptrtoint ([26 x i8]* getelementptr ([26 x i8], [26 x i8]*
null, i32 1) to i32))
  %return_tmp = bitcast i8* %malloccall1 to [26 x i8]*
  store [26 x i8] c"Welcome to Althea Server!\00", [26 x i8]*
%return_tmp, align 1
  %return_tmp2 = getelementptr inbounds [26 x i8], [26 x i8]*
%return_tmp, i32 0, i32 0
  ret i8* %return_tmp2
}
```

コンパイラが自動的にメモリ管理を行うため、手動メモリ管理による問題や脆弱性は発生しない。

構造体用に確保したヒープ領域を、その構造体を所有している変数がスコープから抜けるタイミングで自動的に解放。

Altheaコード

```
struct Foo {
  bar: i32,
}

func main() i32 {
  let foo: Foo = Foo {
    bar: 1,
  }
  0
}
```

コンパイル

LLVM IR

```
define i32 @main() {
entry:
  %malloccall = tail call i8* bitcast (i8*
(i64)* @malloc to i8* (i32)*)(i32 ptrtoint
({ i32, i32 }* getelementptr ({ i32, i32 },
{ i32, i32 }* null, i32 1) to i32))
  %alc_1 = bitcast i8* %malloccall to { i32,
i32 }*
  %field_0_ptr = getelementptr inbounds {
i32, i32 }, { i32, i32 }* %alc_1, i32 0, i32
0
  store i32 1, i32* %field_0_ptr, align 4
  %raw = bitcast { i32, i32 }* %alc_1 to i8*
  call void @free(i8* %raw)
  ret i32 0
}
```

所有権をコンパイラが自動で管理するため、コードを書く時に管理や学習は不要。

RustではコンパイルエラーになるコードでもAltheaでは実行可能。

Rustコード

```
fn main() {
  let foo = String::from("foo");
  let bar = foo;
  println!("{}", foo);
}
```

コンパイル ✗

```
error[E0382]: borrow of moved value: `foo`
 --> src/main.rs:4:18
  |
2 |   let foo = String::from("foo");
  |       --- move occurs because `foo` has type `String`,
which does not implement the `Copy` trait
3 |   let bar = foo;
  |             --- value moved here
4 |   println!("{}", foo);
  |                  ^^^ value borrowed here after move
```

Altheaコード

```
func main() i32 {
  let foo = "foo"
  let bar = foo
  println(foo)
  0
}
```

コンパイル ✓

```
foo
```

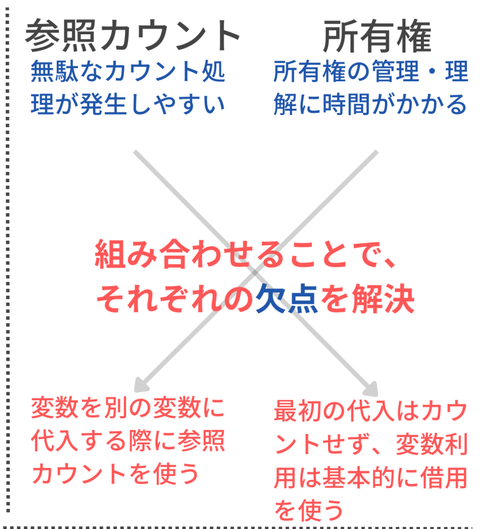
所有権付き参照カウント

所有権と参照カウントを組み合わせたメモリ管理手法。

ヒープ領域割り当てごとに所有権が割り当てられ、その後の利用は基本的に借用となる。**最初の所有権と全ての借用は参照カウントを使用しない**。**同じ値を所有したい場合のみ、参照カウントが増加**する。

具体的に、変数代入時は「左辺は右辺の値を所有したい」、関数呼出時は「仮引数は実引数の値を借用したい」として扱う。右辺値と実引数値はそれぞれリテラルと変数を使用でき、リテラルは「所有されたい」、変数は「借用させたい(所有したままにしたい)」として扱う。そして、**それぞれのパターンでマッチングを行う(2x2=4パターン)**。パターンごとの実装は下記「実装」の項目のようになる。メモリの解放はスコープから抜けるタイミングで行われる。

参照カウントと所有権はそれぞれ単体でもメモリ管理手法として使用されるが、組み合わせることでそれぞれの欠点を打ち消し合うことができる。



実装

変数にリテラルを代入

```
struct Foo {
  bar: i32,
}

func main() i32 {
  let foo: Foo = Foo {
    bar: 1,
  }
  0
}
```

コンパイル

```
define i32 @main() {
entry:
  %malloccall = tail call i8* bitcast (i8*
(i64)* @malloc to i8* (i32)*)(i32 ptrtoint
({ i32, i32 }* getelementptr ({ i32, i32 },
{ i32, i32 }* null, i32 1) to i32))
  %alc_1 = bitcast i8* %malloccall to { i32,
i32 }*
  %field_0_ptr = getelementptr inbounds {
i32, i32 }, { i32, i32 }* %alc_1, i32 0, i32
0
  store i32 1, i32* %field_0_ptr, align 4
  %raw = bitcast { i32, i32 }* %alc_1 to i8*
  call void @free(i8* %raw)
  ret i32 0
}
```

変数定義したスコープにヒープ領域を確保し、スコープを抜けるタイミングで解放

変数に変数を代入

```
struct Foo {
  bar: i32,
}

func main() i32 {
  let foo: Foo = Foo {
    bar: 1,
  }
  let foo2: Foo = foo
  0
}
```

コンパイル
(抜粋)

```
define i32 @main() {
entry:
  ; ~~~ 構造体定義は省略 ~~~
  %rc_ptr = getelementptr inbounds { i32, i32 }, { i32, i32 }* %alc_1, i32
0, i32 1
  %rc = load i32, i32* %rc_ptr, align 4
  %increment_rc = add i32 %rc, 1
  store i32 %increment_rc, i32* %rc_ptr, align 4
  %rc_ptr1 = getelementptr inbounds { i32, i32 }, { i32, i32 }* %alc_1,
i32 0, i32 1
  %rc2 = load i32, i32* %rc_ptr1, align 4
  %decrement_rc = sub i32 %rc2, 1
  store i32 %decrement_rc, i32* %rc_ptr1, align 4
  %is_zero = icmp sle i32 %decrement_rc, 0
  br i1 %is_zero, label %free, label %else

free:                                             ; preds = %entry
  %raw = bitcast { i32, i32 }* %alc_1 to i8*
  call void @free(i8* %raw)
  br label %else

else:                                             ; preds = %free, %entry
  ret i32 0
}
```

右辺値を所有したいが変数も借用させたい(所有したままにしたい)ため、参照カウントを更新

関数の実引数がリテラル

```
struct Foo {
  bar: i32,
}

func main() i32 {
  let baz_val: i32 = baz(Foo {
    bar: 1,
  })
  baz_val
}

func baz(foo: Foo) i32 {
  0
}
```

コンパイル

```
define i32 @main() {
entry:
  %malloccall = tail call i8* bitcast (i8* (i64)*
@malloc to i8* (i32)*)(i32 ptrtoint ({ i32, i32 }*
getelementptr ({ i32, i32 }, { i32, i32 }* null, i32 1)
to i32))
  %alc_1 = bitcast i8* %malloccall to { i32, i32 }*
  %field_0_ptr = getelementptr inbounds { i32, i32 }, {
i32, i32 }* %alc_1, i32 0, i32 0
  store i32 1, i32* %field_0_ptr, align 4
  %alc_2 = call i32 @baz({ i32, i32 }* %alc_1)
  %raw = bitcast { i32, i32 }* %alc_1 to i8*
  call void @free(i8* %raw)
  ret i32 %alc_2
}

define i32 @baz({ i32, i32 }* %0) {
entry:
  ret i32 0
}
```

関数呼出したスコープに無名変数でヒープ領域を確保し、スコープを抜けるタイミングで解放

関数の実引数が変数

```
struct Foo {
  bar: i32,
}

func main() i32 {
  let foo: Foo = Foo {
    bar: 1,
  }
  let baz_val: i32 = baz(foo)
  baz_val
}

func baz(foo: Foo) i32 {
  0
}
```

コンパイル

```
define i32 @main() {
entry:
  %malloccall = tail call i8* bitcast (i8* (i64)* @malloc
to i8* (i32)*)(i32 ptrtoint ({ i32, i32 }* getelementptr
({ i32, i32 }, { i32, i32 }* null, i32 1) to i32))
  %alc_1 = bitcast i8* %malloccall to { i32, i32 }*
  %field_0_ptr = getelementptr inbounds { i32, i32 }, {
i32, i32 }* %alc_1, i32 0, i32 0
  store i32 1, i32* %field_0_ptr, align 4
  %alc_2 = call i32 @baz({ i32, i32 }* %alc_1)
  %raw = bitcast { i32, i32 }* %alc_1 to i8*
  call void @free(i8* %raw)
  ret i32 %alc_2
}

define i32 @baz({ i32, i32 }* %0) {
entry:
  ret i32 0
}
```

実引数値を借用したく変数も借用させたいため、借用

言語機能

列挙型宣言

```
enum Result {
  Ok(i32),
  Err(string),
}
```

構造体宣言

```
struct User {
  id: Result,
}
```

変数定義

```
func main() i32 {
  let foo: Foo = Foo {
    bar: Result::Ok(1),
  }
  let baz: string = "This is baz.\n"
  let arr: [i8; 8] = [1; 8]
```

パターンマッチ

```
  match foo {
    Foo { bar: bar } ⇒ match bar {
      Result::Ok(val) ⇒ {
        println("This is Foo.\n")
      }
    },
  }
}
```

HTTPサーバー起動

```
func main() i32 {
  listen_and_serve(
    80,
    func handle() string {
      "Welcome to Althea Server!"
    }
  )
}
```

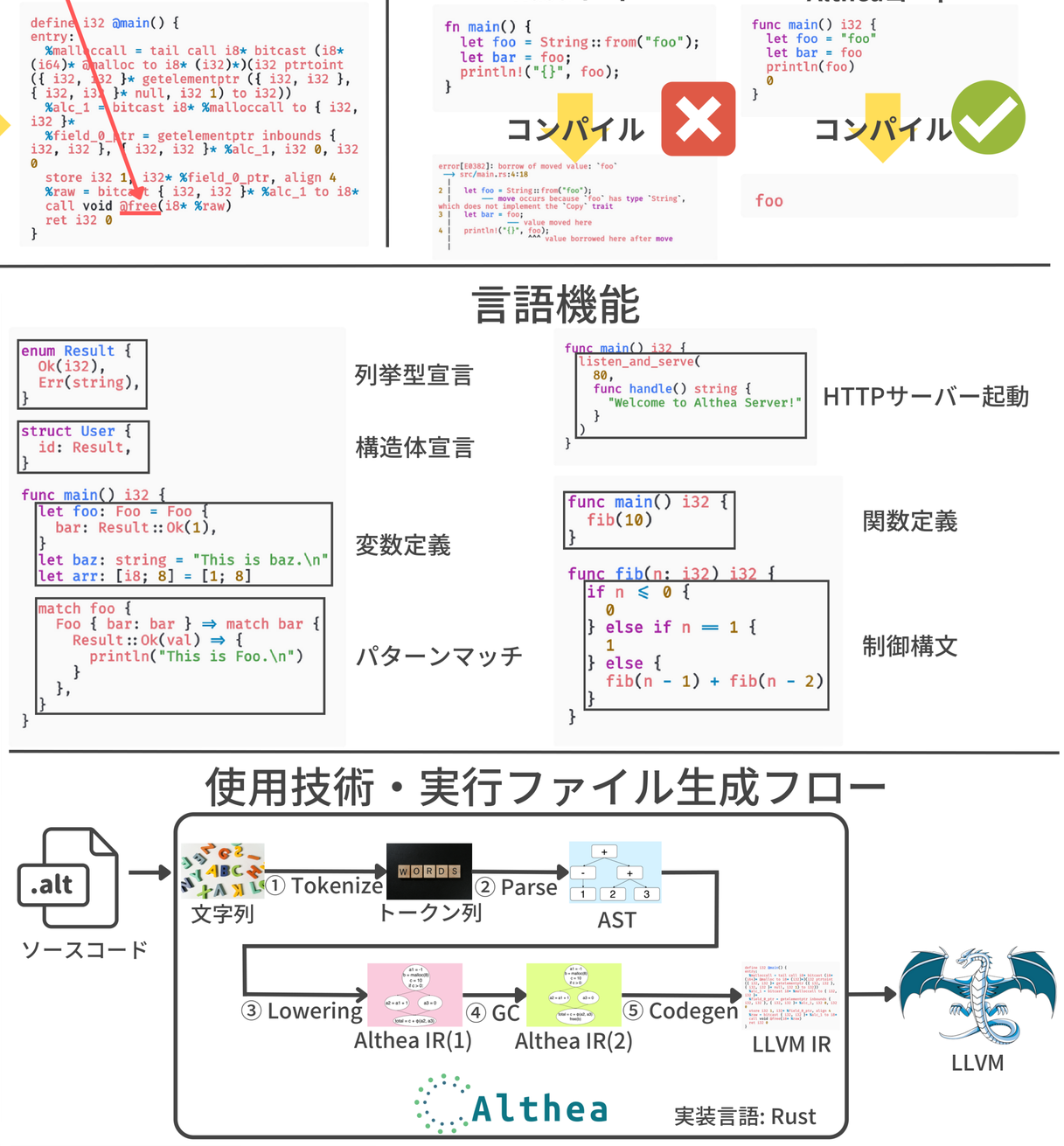
関数定義

```
func main() i32 {
  fib(10)
}

func fib(n: i32) i32 {
  if n ⩽ 0 {
    0
  } else if n == 1 {
    1
  } else {
    fib(n - 1) + fib(n - 2)
  }
}
```

制御構文

使用技術・実行ファイル生成フロー



使用用途

個人的に興味がある分野をベースに大きく2つの方針を考えている。今後の開発予定も含まれる。

HTTPサーバー

Stop The Worldによって性能要件を満たすことが難しいサーバーや、大規模にメモリを使用するサーバーにより効果的。サポートが進んでおり、現在2ファイルでHTTPサーバーを立てることが可能(コードはFAQを参照)。今後は様々なHTTPメソッドのハンドリングやクエリパラメータの取得をサポート予定。

インメモリDB

本言語は大規模なメモリを使用する際に効果的なため、インメモリDBを開発する際にも効果的だと考えている。インメモリDB開発のために、様々なデータ構造を標準ライブラリから提供することで簡単に高性能なインメモリDBを開発できることを目指していく。

[1]: https://discord.com/blog/why-discord-is-switching-from-go-to-rust
[2]: https://www.chromium.org/Home/chromium-security/memory-safety
[3]: https://play.rust-lang.org/?gist=dfcf3a4e1ca60dcc34ac459c00fa84de
[4]: https://blog.rust-lang.org/2016/06/30/State-of-Rust-Survey-2016.html
[5]: https://github.com/yagipy/althea
[6]: https://github.com/yagipy/althea/blob/main/doc/FAQ.md

GitHub [5]　FAQ [6]

謝辞
本作品の開発にあたりお世話になりました、開発駆動コース・トレーナーの川合秀実氏、アシスタントの古川菜摘氏、トレーニーの中神悠太氏をはじめSecHack365に関わる全ての方々に多大なる感謝を申し上げます。ありがとうございました。