Region-Based Memory Management in Cyclone

Miyazato Shuntaro

~~Region-Based Memory Management in Cyclone~~

Rustはいい（らしい）ぞ

Miyazato Shuntaro

目次

- はじめに（この発表の目的・論文の概要） 4-7p
- Rustとは, この授業の観点で見たRust　8-27p
- 論文の中身　28-45p
- まとめ　46p

はじめに

目的

- メモリ管理の2大潮流、「GCを使う」と「使わない」の後者について学ぶ取っ掛かり

- Rustがなぜ押されているのか？なぜ学習コストが高いと言われているのか？の紹介

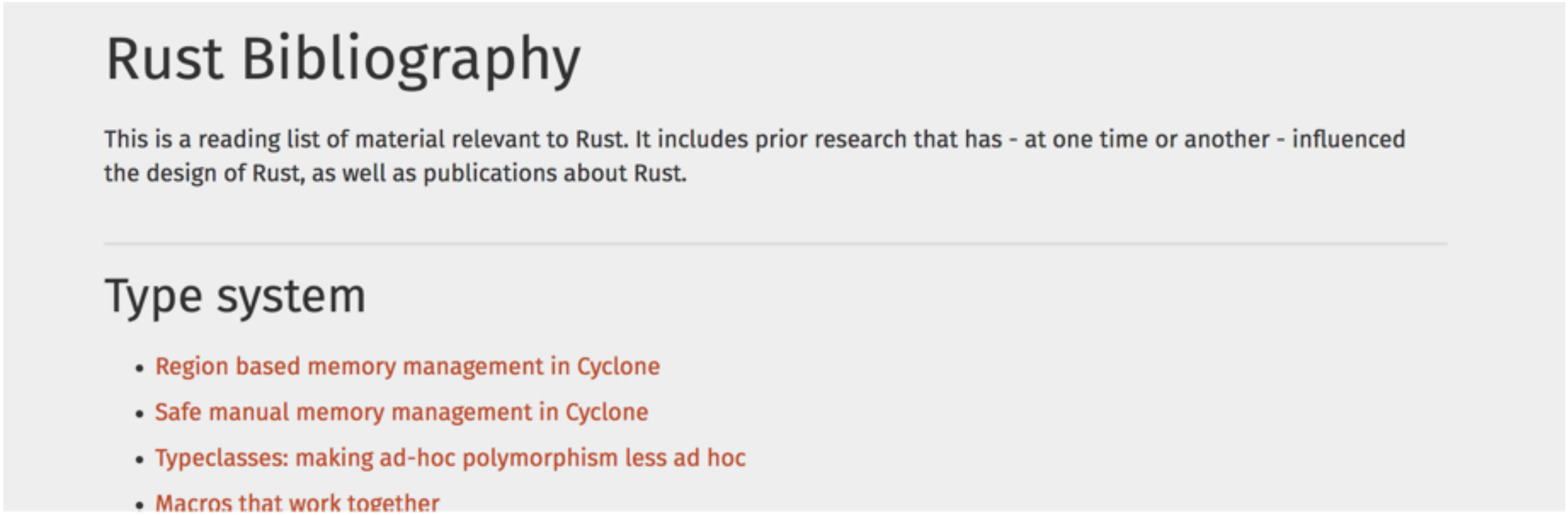
- 論文がおまけみたいになっちゃったのはすみません

論文概要

- Region-based Memory Management in Cyclone（2004）
- CycloneというC言語の方言で型安全な言語を開発者が紹介した論文
- Rustはこの言語の考え方をだいぶ参考にしている

論文概要

- ちなみに, Rustは公式サイトで, 設計思想の元になった論文一覧を紹介している（この論文が一番上に）



Rust Bibliography

This is a reading list of material relevant to Rust. It includes prior research that has - at one time or another - influenced the design of Rust, as well as publications about Rust.

Type system

- Region based memory management in Cyclone
- Safe manual memory management in Cyclone
- Typeclasses: making ad-hoc polymorphism less ad hoc
- Macros that work together

Rustとは

Rustの特徴

- Mozillaが開発してるよ静的型付け言語（2006年始動）
- 手続き型プログラミング、オブジェクト指向プログラミング、関数型プログラミングなどの実装手法をサポートしているよ
- 自称、「速度、安全性、並行性の3つのゴールにフォーカスしたシステムプログラミング言語」（？）
  →LinuxをC言語で書いてるみたいにOSとかデータベースとか書ける！

この授業の観点で見た Rust

型システム（あまり話さない）

- クラスなんてものはない
- トレイトっていうやつ（Haskellの型クラスみたいなの）で, 型にどんなメソッドが必要かという制約を与える
- （型に関しては）部分型多相ではなくパラメトリック多相（＋アドホック多相）
- 継承っぽいことも出来る！型推論が強い！パターンマッチがある！

メモリ管理

- ガベージコレクションを使わない（CやC++の潮流）
- なぜ使わない？→ 速くしたいから

- C系統の強さ（メモリ管理をプログラマがある程度制御出来る）+ 関数型の強さ（強い型システムと型安全）+ コンパイル時に全てエラーが出る = つよい！！

ここがヤバイよC言語

- こういうのがコンパイルエラー出ない

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char p[2] = "aa";
    free(p);
    free(p);
    free(p+1);
    *p = 'b';
    return 0;
}

```

ここがヤバいよC言語

- こういうのがコンパイルエラー出ない（warningは流石に出ました）

```c
#include <stdio.h>
#include <stdlib.h>
int *foo()
{
  int a[1];
  a[0] = 0;
  return a;
}
int main()
{
  int *p = foo();
  p[0] = 5;
  printf("%d", p);
  return 0;
}
```

ここがヤバイよC言語

- 寿命を超えた変数のアクセス
- メモリの解放し忘れ（メモリリーク）
- 「使わなくなったよ」っていう情報に気づいてくれる仕組みが必要

防ぐ方法1

- ガベージコレクションで実行時（ランタイム）になんやかや

- 例：すごくたくさん（OCaml, Python, Haskell…）

- メリット：プログラマが面倒な事を考えなくていい！

- デメリット：メモリ解放の為にプログラムが一時停止する+それはプログラマの予期せぬタイミングである場合が多い（もちろん授業で紹介されたような色々頭の良い方法が研究されてますが…）

防ぐ方法2

- C++11のスマートポインタ

- スコープ外になる（使用期間が終わる）と確保したメモリを自動的に解放してくれるポインタ型だと思ってれば大丈夫らしい

- （C++にガベージコレクションがないわけではない, これを使うという選択肢もあるという話）

- 大体のプログラムがスマートポインタで”事足りる”為, 明示的にfreeしたりGCをしたりしなくて良いという発想

スマートポインタ

- shared_ptrとunique_ptrだけ知ってれば今はOK

- こんな事が出来る（雰囲気だけ分かればOK）

```cpp
#include <iostream>
#include <memory>
using namespace std;
int main()
{
  shared_ptr<double> p_first(new double);
  *p_first = 0.0;
  {
    shared_ptr<double> p_copy = p_first;
    *p_copy = 21.2;
  } // 'p_copy'は破壊されるが、割り当てられたdoubleは破壊されない
  cout << *p_first; //21.2ってちゃんと表示される、しかし
  //cout << *p_copy;  //これはエラー
  return 0; // 'p_first'が破壊され、同時に割り当てられたdoubleも破壊される
}

```

スマートポインタ

- shared_ptrとunique_ptrの違いは所有権（書き換える権利）を持てるオブジェクトが複数か単一か

- 例えばさっきのコードのsharedをuniqueにするとコピーした時点で怒られる

```cpp
#include <iostream>
#include <memory>
using namespace std;
int main()
{
  unique_ptr<double> p_first(new double);
  *p_first = 0.0;
  {
    unique_ptr<double> p_copy = p_first;//p_copyに書き換える権利を渡す操作なのでエラー
    *p_copy = 21.2;
  }
  cout << *p_first;
  //cout << *p_copy;
  return 0; //
}

```

大まかな仕組み

- オブジェクトを指しているポインタの数をカウントしてお
  き, カウントが0になると自動で解放

- 少しだけ型を使いにくくして（制限を付けて）機械側に分
  かりやすくしてあげようという発想か？

Q：あれ？これ参照カウンタでは？？

- A：参照数はオブジェクトではなくポインタに紐付けられている事, 実行時ではなくコンパイル時に色々やってくれる事, などからガベージコレクションとは別物と捉えられている（風潮）なんだと思う. けど結局参照カウンタ方式とも呼ばれてる. 紛らわしい.（のでもしかしたら自分に誤解があるかも）

だがしかし

- 完璧じゃなかったよC++

- 所有権を「動かして」自分自身は用済みになる操作が,利便性の為に存在するが, 用済みになっても使えてしまう（コンパイルエラーにはならない）

```cpp
#include <iostream>
#include <memory>
using namespace std;
int main()
{
  unique_ptr<double> p_first(new double);
  *p_first = 0.0;
  unique_ptr<double> p_second(new double);
  *p_second = 21.2;
  cout << *p_first;
  cout << *p_second;
  p_second = move(p_first);
  cout << *p_first;
  cout << *p_second;
  return 0; //
}

```

解決策

- コンパイルエラーを吐く（Rustがやってくれました）

```
fn main() {
    let mut x = "1".to_string();//xが文字を指すポインタ
    let y = "2".to_string();//yが文字を指すポインタ
    println!("{}", x);//1
    println!("{}", y);//2
    x = y;//所有権移動
    println!("{}", x);//2
    println!("{}", y); //ここでエラーが出る
}

```

完璧じゃなかったよC++

- 一度スコープ外のポインタに渡すと, スコープ外になったのにアクセス出来ちゃう

```
1  #include <iostream>
2  #include <memory>
3  using namespace std;
4  int main()
5  {
6    double *x;
7    {
8      double y = 0;
9      x = &y;
10   }
11   //cout << y; //そんなものはないとコンパイルで言われる
12   cout << *x; //でもこれは0と表示される
13   return 0; //
14 }
15
```

解決策

- コンパイルエラーを吐く

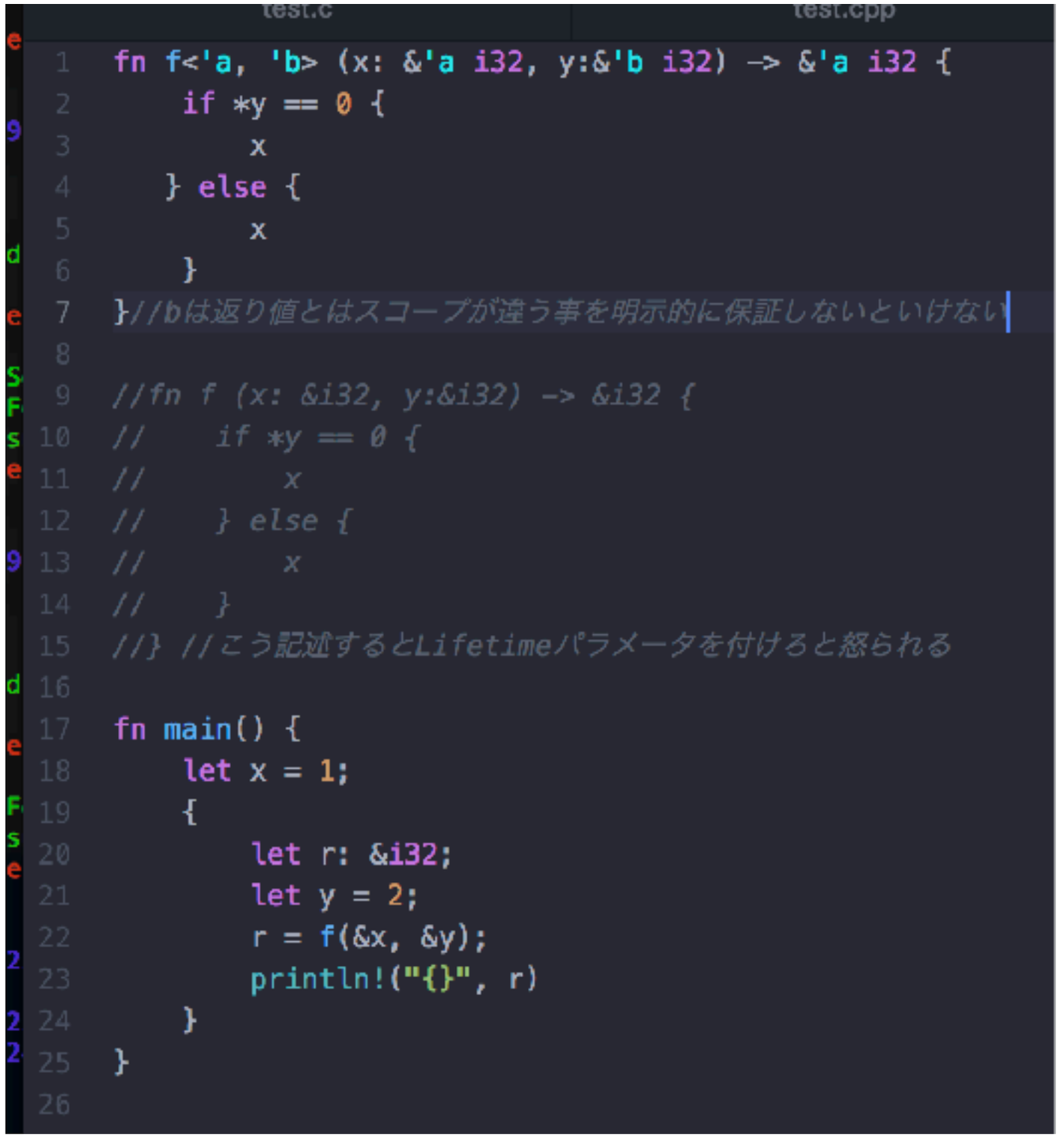
- これはRustのライフタイムという概念. 一言で言うと
  「寿命の短いものを長いものに渡すな」

```
1  fn main() {
2      let mut x = "1";
3      {
4          let y = "2".to_string();
5          x = &y;//borrowed value does not live long enoughというエラー
6      }
7      println!("{}", x);
8  }
9
```

ライフタイム

- 実は本来書くべきものをコンパイラが推論してくれている

- アノテーションが必要な時もある
  （Python3の型アノテーションのように）



結局Rustは

- 所有権, ライフタイムという考え方でコンパイル時にはすでに変な参照を見つけている
- あと一つ, データ競合を避ける為に「書き換え権限のある参照」を制限する借用という仕組みがあるが省略
  （C++だと未定義動作が起こる）

- Rustの概観のお話終わり

（やっと）

論文の中身

（実は）

大事な部分は既に話している

Region-Based Memory Management in Cyclone

- リージョンベースのメモリ管理 in Cyclone

- リージョンってなんですか

- ライフタイムの観点で見て「生きている」メモリの領域をリージョン（領域）と読んでいる（そのまま）

- 発想の初出は1994年
  （この論文は2004年でRust誕生は2006年）

リージョンベース管理

- Rust for Semanticists（http://asaj.org/talks/lola16/#rust:goals）曰く

- 「Rustの安全性は2つの1990年代のアイデアで出来ている. 線形型とリージョンベース管理だ」

- 線形型は線形論理（古典論理を弱めた論理）に基づいた型で, 「仮定に1度使った命題は消えてもう使えない」という特徴があるもの（あらゆるデータが必ず1回使われる事に関係があるらしいよ, わからんけど）

開発者の主張

- C言語は危険（解放後の参照とか解放し忘れとか）. 型安全であるべき
- でもメモリ管理を人間が支配したい
- コンパイル時に全部決められたら爆速
- でもアノテーションはなるべく書きたくない, 推論して欲しい
- だからリージョンとその推論を使います！w

ここはRustと違うよ

- ガベージコレクションは完全に駆逐していません. ヒープ領域はお掃除してもらってます
- あくまでC言語の方言なのでC言語と文法をほぼ変えず, 実験で大量のC言語コードをCycloneに書き換えたが, 変更はなんと約8%（行数で）

リージョンとは

- 型と同じでデータ（ポインタ）につくラベルでメモリの確保を示しているもの
  （int型ポインタでリージョン名がρLのp）
- 型と直交的な概念なので, 型が違えば変数が別物であるように, リージョンが違えば型が同じでも全く別物

- LIFOに確保, 解放される（スタックに置かれる）

$$\texttt{int*}\rho_L\ \texttt{p;}$$

エラーのイメージ

- コンパイラがリージョンを推論→リージョンがσとρで違うなら型が違うのと同じ！

```
1   fn main() {
2       int*σ p;
3       ρ {
4           int x = 0;
5           p = &x; //int*σにint*ρ型の代入！
6       }
7       *p = 123;
8   }
9
```

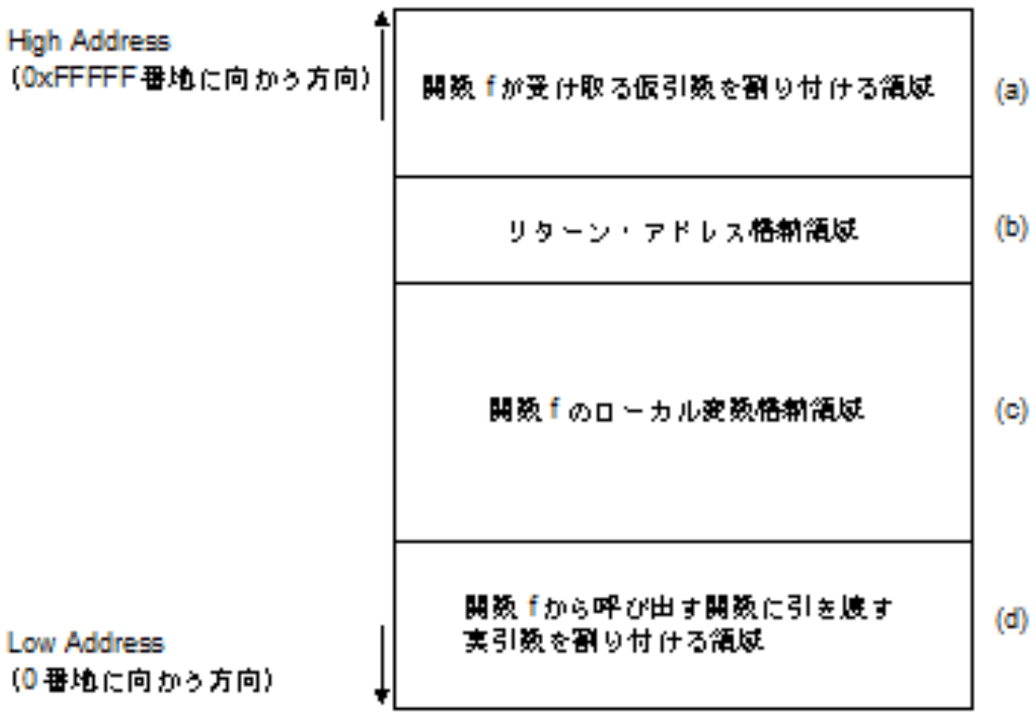
リージョンの種類

- スタックリージョン
- ヒープリージョン
- 動的リージョン
- （関数リージョン）

関数リージョンと
スタックリージョン

- 大体普通のメモリ管理のスタック領域のイメージで考えれば大丈夫です

- 引数の数だけのリージョンが名付けられて作られ,リージョンスタックに積まれる→関数が終わったら解放される



動的リージョン

- region r{文}のブロックで確保されるリージョン（rがリージョン名）
- サイズがコンパイル時に決まっていない為に実行時はヒープ領域に割り当てられるが, ブロックを抜けたら解放される. 「ページ」が連なった連結リストとして表現される. サイズが増えるごとに伸びていく.

ヒープリージョン

- グローバル関数, mallocを確保する（Cycloneはfreeがないⅰ仕様を選んだのでどちらも静的に確保されます）

- ずっと存在するので解放されない

- それとCycloneではヒープ領域はガベージコレクション出来るのであまり工夫していない？

リージョン多相性

- 構造体や関数は宣言時にアクセスするregionが具体的に分からなくてもよい

- どう実現しているのかは書いていなかったが, 型のパラメータ多相と似ている？

```
1  struct List <ρ1, ρ2> {
2      int *ρ1 head;
3      struct List <ρ1, ρ2> *ρ2 tail;
4  };
5
```

リージョンsubtyping

- ρ1はρ2より先にスタックに積まれて後に取り出される
  →長生きするのでρ2にρ1を代入出来る（ρ1がρ２として振る舞う）
- （リージョンじゃない方の）型は完全に同じでないとキャスト出来ないことに注意！

```
1  ρ1 {
2      ρ2 {
3      }
4  }
5
```

アノテーションの省略

- デフォルトルールによりリージョンは自然に推論されている（ので一々面倒な指示をしなくていい）

- 1. ローカル変数に関してはunification basedで推論する（わからない…）

- 2. 関数の引数には汎用的に解釈出来る新しいリージョンを充てる

- 3. それ以外の場合に関してはヒープリージョンを充てる

その他

- 存在型っていう複数の型を統一的に扱えるヤバい型があってダングリングポインタが出て来がちでマズいけど凄い工夫で安全にしたらしい
  （特定の関数がアクセスする可能性のあるリージョンのリストと生きているリージョンのリスト2つを比較して云々）

- 安全だという数学的証明

- わからなすぎる

実験

- 複数のC言語のプログラム（そういうベンチマークらしい. 内容は本質的でなさそう）をCycloneに書き換えたらなんと約8%ほどの行数しか書き換えなくて良かった
- アノテーションもほとんど明示しなくて良かった（行数にして約0.5%）
- メモリの安全性を保ちつつ, 実行速度は0.9~1.36倍でほぼ変わらない（ガベージコレクションを使わない場合）

論文まとめ

- リージョンという, 型のようなタグをデータに付けることによってメモリ管理を安全に行うことが出来る
  （寿命を超えた参照や解放し忘れを防げる）

- リージョンは本来の型とは別物だが, 多相や部分型などのような構造が可能で利便性を持たせられる

- コンパイラに推論させることによって人間側の負担を増やさないことが出来る

全体まとめ

- RustはC言語やC++のようにメモリ管理を制御出来て速く, 関数型やオブジェクト指向のように型システムが安全で豊富で, コンパイル時にエラーが分かる良い言語らしい

- ただしその分人間側が考慮する制限が増えているので難しい（コンパイル時に機械が教えてくれるが）

- 所有型・ライフタイム ・リージョン管理といった発想は全然新しくなく, むしろRustは古い研究成果を丁寧に応用して出来た言語と言える