実践情報セキュリティと
アルゴリズム ４
線形型システムの実装

大阪大学　大学院工学研究科　電気電子情報通信工学
王　イントウ
wang@comm.eng.osaka-u.ac.jp

続・Rust言語

impl

特定の型に対する関数を定義
オブジェクト指向型言語のクラスメソッドに相当

```
struct Vec2 {
    x: f64,
    y: f64
}

impl Vec2 {
    fn new(x: f64, y: f64) -> Vec2 {
        Vec2{x: x, y: y}
    }

    fn norm(&self) -> f64 {
        (self.x * self.x + self.y * self.y).sqrt()
    }

    fn set(&mut self, x: f64, y: f64) {
        self.x = x;
        self.y = y;
    }
}
```

```
fn my_func9() {
    let mut v = Vec2::new(10.0, 5.0);
    println!("v.norm = {}", v.norm());
    v.set(3.8, 9.1);
    println!("v.norm = {}", v.norm());
}
```

trait

Javaのinterface or Haskellの型クラスといった機能
ある型が実装すべき関数を定義

Addトレイトの例
Addトレイトを実装した型は+が利用できる

```
trait Add<RHS=Self> {
    type Output;

    fn add(self, rhs: RHS) -> Self::Output;
}
```

```
use std::ops::Add;

struct Vec2 {
    x: f64,
    y: f64
}

impl Add for Vec2 {
    type Output = Vec2;

    fn add(self, rhs: Vec2) -> Vec2 {
        Vec2 {
            x: self.x + rhs.x,
            y: self.y + rhs.y,
        }
    }
}

fn my_func10() {
    let v1 = Vec2{x: 10.0, y: 5.0};
    let v2 = Vec2{x: 3.1, y: 8.7};
    let v = v1 + v2; // +演算子が利用可能。v1とv2の所有権は移動

    println!("v.x = {}, v.y = {}", v.x, v.y);
}
```

?演算子

?演算子を使うと、OptionやResult型の値を取り出し、エラーの場合リターンする
というコードを簡略に記述可能

getの返り値がOption型の場合

```
let a = get(expr)?;
```

は、以下に等しい

```
let a = match get(expr) {
    Some(e) => e,
    None => return None,
};
```

getの返り値がResult型の場合

```
let a = get(expr)?;
```

は、以下に等しい

```
let a = match get(expr) {
    Ok(e) => e,
    Err(e) => return Err(e),
};
```

LinkedList

リンクリスト
要素の型はすべて同じ必要あり

```
use std::collections::LinkedList;

fn main() {
    let mut l = LinkedList::new();
    l.push_back(10);  // 最後尾に追加
    l.push_back(20);  // 同上
    l.push_front(40); // 先頭に追加
    l.pop_back();     // 最後尾から削除
    l.pop_front();    // 先頭から削除
}
```

HashMap

辞書
それぞれのkeyと、それぞれのvalueの型はすべて同じ必要あり

```
use std::collections::HashMap;

fn main() {
    let mut m = HashMap::new();
    // インサート
    m.insert("a".to_string(), 20);
    m.insert("b".to_string(), 40);

    // 取得
    println!("a = {}", m.get(&”a".to_string()).unwrap()); //.unwrapは?でも変えられる

    // 削除
    m.remove(&"b".to_string());

    // 変更
    *m.get_mut(&"a".to_string()).unwrap() = 5;
}
```

Box

ヒープメモリ上に値を生成
C言語のmallocに相当

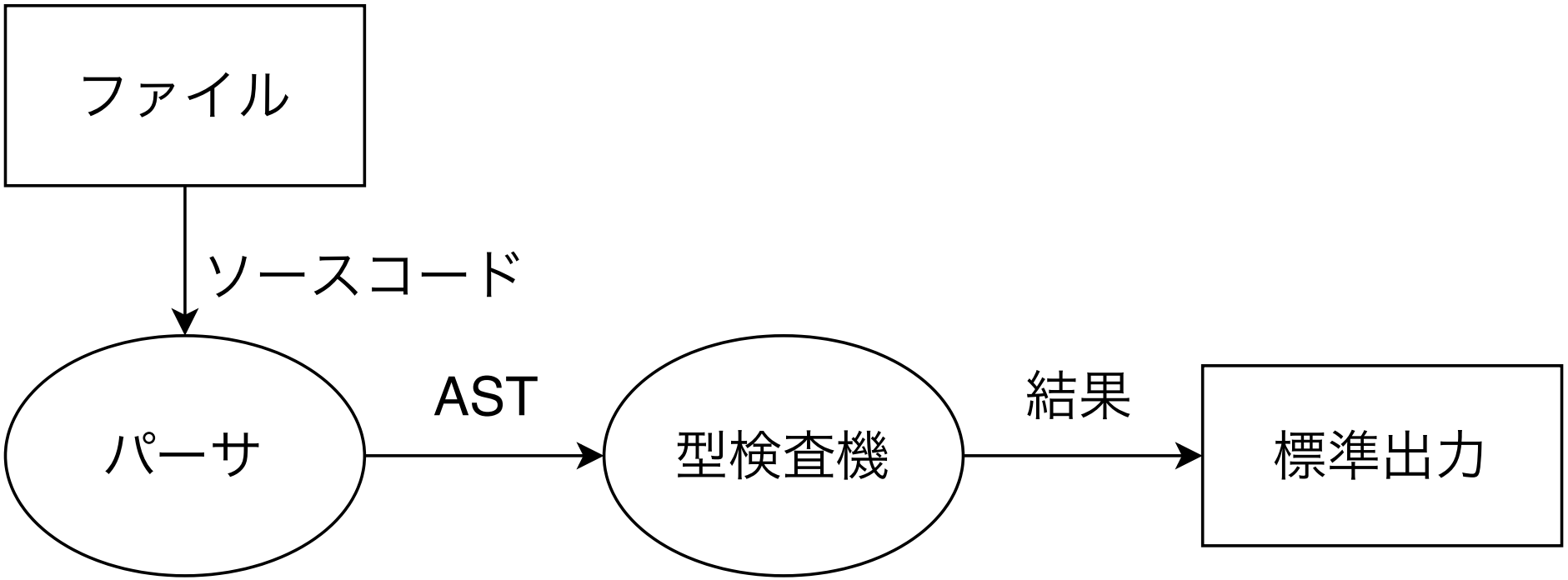
```
fn main() {
    let val = Box::new(30);
}
```

線形型システムの実装

ファイル構成

ファイル	説明
Cargo.toml	cargo 用のファイル
./src/main.rs	main 関数用ファイル
./src/parser.rs	パーサ
./src/typing.rs	型検査器
./src/helper.rs	補助的なコード用のファイル（8 章と同じ）
./codes	LinZ のサンプルコードを含むディレクトリ

流れ



実行方法

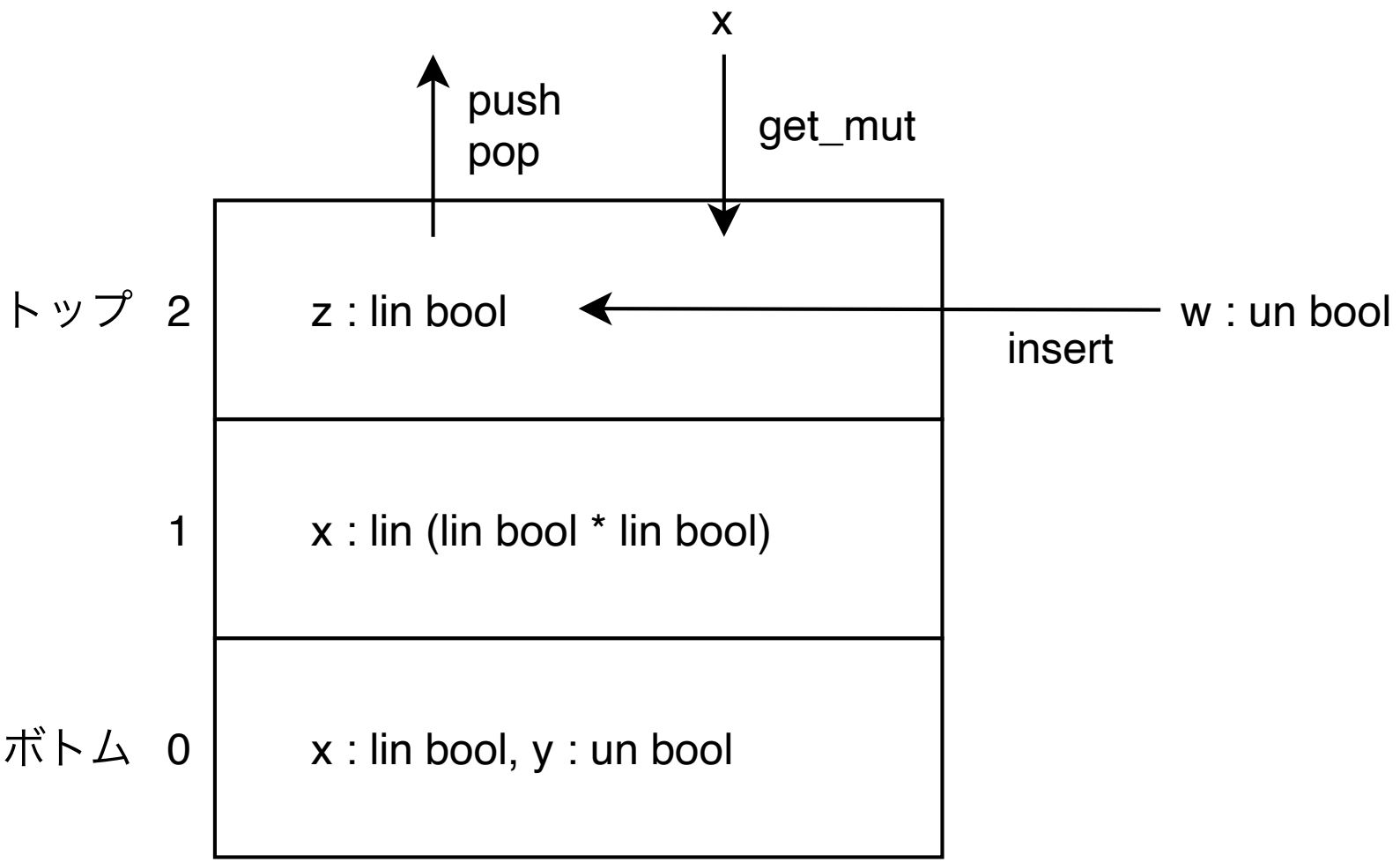
```
$ cargo run 入力ファイル
```

スタックによる型環境

型環境

- 型環境とは、変数と型の対応を保存するマップであった
- 型環境は概念的には、マップのスタックと実装可能
- スタックで管理することで、変数のスコープとシャドーイングを表現可能

スタックによる型環境


x
push
pop
get_mut
トップ 2
z : lin bool
w : un bool
insert
1
x : lin (lin bool * lin bool)
ボトム 0
x : lin bool, y : un bool

スタックによる型環境のインターフェース

- push/pop：スタック操作と同じ
- get_mut：変数をキーとして、上から順にマップを検索していき、一番はじめに見つけたものを取得（一番はじめに見つけたものを返すことでシャドーイングを表現）
- insert：最も上に位置するマップに、変数と型の対応づけを追加

スタックによる型環境の実装

```
type VarToType = BTreeMap<String, Option<parser::TypeExpr>>;

#[derive(Debug, Clone, Eq, PartialEq, Default)]
struct TypeEnvStack {
    vars: BTreeMap<usize, VarToType>,
}

impl TypeEnvStack {
    fn new() -> TypeEnvStack { }

    // 型環境をpush
    fn push(&mut self, depth: usize) {  }

    // 型環境をpop
    fn pop(&mut self, depth: usize) -> Option<VarToType> { }

    // スタックの最も上にある型環境に変数名と型を追加
    fn insert(&mut self, key: String, value: parser::TypeExpr) { }

    // スタックを上からたどっていき、はじめに見つかる変数の型を取得
    fn get_mut(&mut self, key: &str) -> Option<(usize, &mut Option<parser::TypeExpr>)> { }
}
```

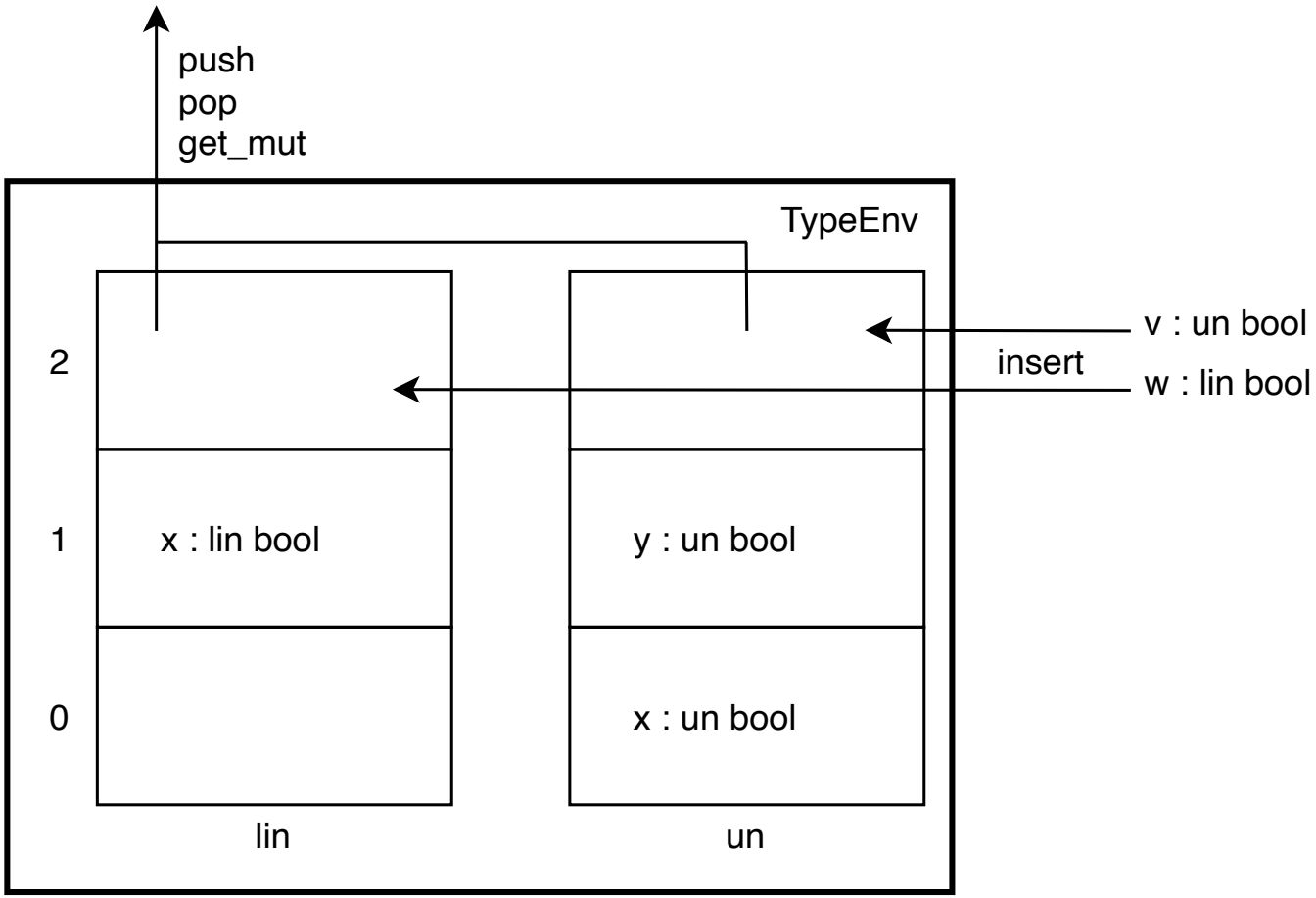
- 実装の詳細は省略
- スタックはBTreeMapで実装
  - キーがスタックの高さで、値がマップ
  - ふつう、スタックはリストかベクタで実装するが、linの制約の実装上BTreeMapを利用
- pop/pushのdepthには、スタックの高さを指定
  - はじめは0
  - 変数スコープが増えるとdepth + 1

線形型システムの型環境

線形型システムの型環境

- スタックによる型環境を2つもつ
- linとun用の型環境

線形型システムの型環境



線形型システムの型環境のインターフェース

- push/pop：2つの型環境に対してpush/pop
- insert：最も上のマップに対してinsert。ただし、lin型の変数はlinの型環境へ、un型の変数はunの型環境へinsert
- get_mut：両方の型環境からgetし、よりスタックの高い位置にある方を返す

線形型システムの型環境の実装

```
#[derive(Debug, Clone, Eq, PartialEq)]
pub struct TypeEnv {
    env_lin: TypeEnvStack, // lin用
    env_un: TypeEnvStack,  // un用
}

impl TypeEnv {
    pub fn new() -> TypeEnv {  }

    /// 型環境をpush
    fn push(&mut self, depth: usize) {  }

    /// 型環境をpop
    fn pop(&mut self, depth: usize) -> (Option<VarToType>, Option<VarToType>) {  }

    /// 型環境へ変数と型をpush
    fn insert(&mut self, key: String, value: parser::TypeExpr) {  }

    /// linとunの型環境からgetし、depthが大きい方を返す
    fn get_mut(&mut self, key: &str) -> Option<&mut Option<parser::TypeExpr>> {  }
}
```

- 実装の詳細は省略
- スタックによる型環境を2つ持つ

抽象構文木(AST)

構文定義

```
//! <VAR>   := 1文字以上のアルファベットから成り立つ変数
//!
//! <E>     := <LET> | <IF> | <SPLIT> | <FREE> | <APP> | <VAR> | <QVAL>
//!
//! <LET>   := let <VAR> : <T> = <E>; <E>
//! <IF>    := if <E> { <E> } else { <E> }
//! <SPLIT> := split <E> as <VAR>, <VAR> { <E> }
//! <FREE>  := free <E>; <E>
//! <APP>   := ( <E> <E> )
//!
//! <Q>     := lin | un
//!
//! 値
//! <QVAL>  := <Q> <VAL>
//! <VAL>   := <B> | <PAIR> | <FN>
//! <B>     := true | false
//! <PAIR>  := < <E> , <E> >
//! <FN>    := fn <VAR> : <T> { <E> }
//!
//! 型
//! <T>     := <Q> <P>
//! <P>     := bool |
//!            ( <T> * <T> )
//!            ( <T> -> <T> )
```

ASTの実装

```rust
#[derive(Debug)]
pub enum Expr {
    Let(LetExpr),      // let式
    If(IfExpr),        // if式
    Split(SplitExpr), // split式
    Free(FreeExpr),    // free文
    App(AppExpr),      // 関数適用
    Var(String),       // 変数
    QVal(QValExpr),    // 値
}

#[derive(Debug)]
pub struct AppExpr {
    pub expr1: Box<Expr>,
    pub expr2: Box<Expr>,
}


#[derive(Debug)]
pub struct IfExpr {
    pub cond_expr: Box<Expr>,
    pub then_expr: Box<Expr>,
    pub else_expr: Box<Expr>,
}
```

```rust
#[derive(Debug)]
pub struct SplitExpr {
    pub expr: Box<Expr>,
    pub left: String,
    pub right: String,
    pub body: Box<Expr>,
}

#[derive(Debug)]
pub struct LetExpr {
    pub var: String,
    pub ty: TypeExpr,
    pub expr1: Box<Expr>,
    pub expr2: Box<Expr>,
}


#[derive(Debug)]
pub enum ValExpr {
    Bool(bool),                    // 真偽値リテラル
    Pair(Box<Expr>, Box<Expr>), // ペア
    Fun(FnExpr),                   // 関数（λ抽象）
}
```

ASTの実装

```rust
#[derive(Debug, Eq, PartialEq, Clone, Copy)]
pub enum Qual {
    Lin, // 線形型
    Un,  // 制約のない一般的な型
}
```

```rust
#[derive(Debug)]
pub struct QValExpr {
    pub qual: Qual,
    pub val: ValExpr,
}
```

```rust
#[derive(Debug)]
pub struct FnExpr {
    pub var: String,
    pub ty: TypeExpr,
    pub expr: Box<Expr>,
}
```

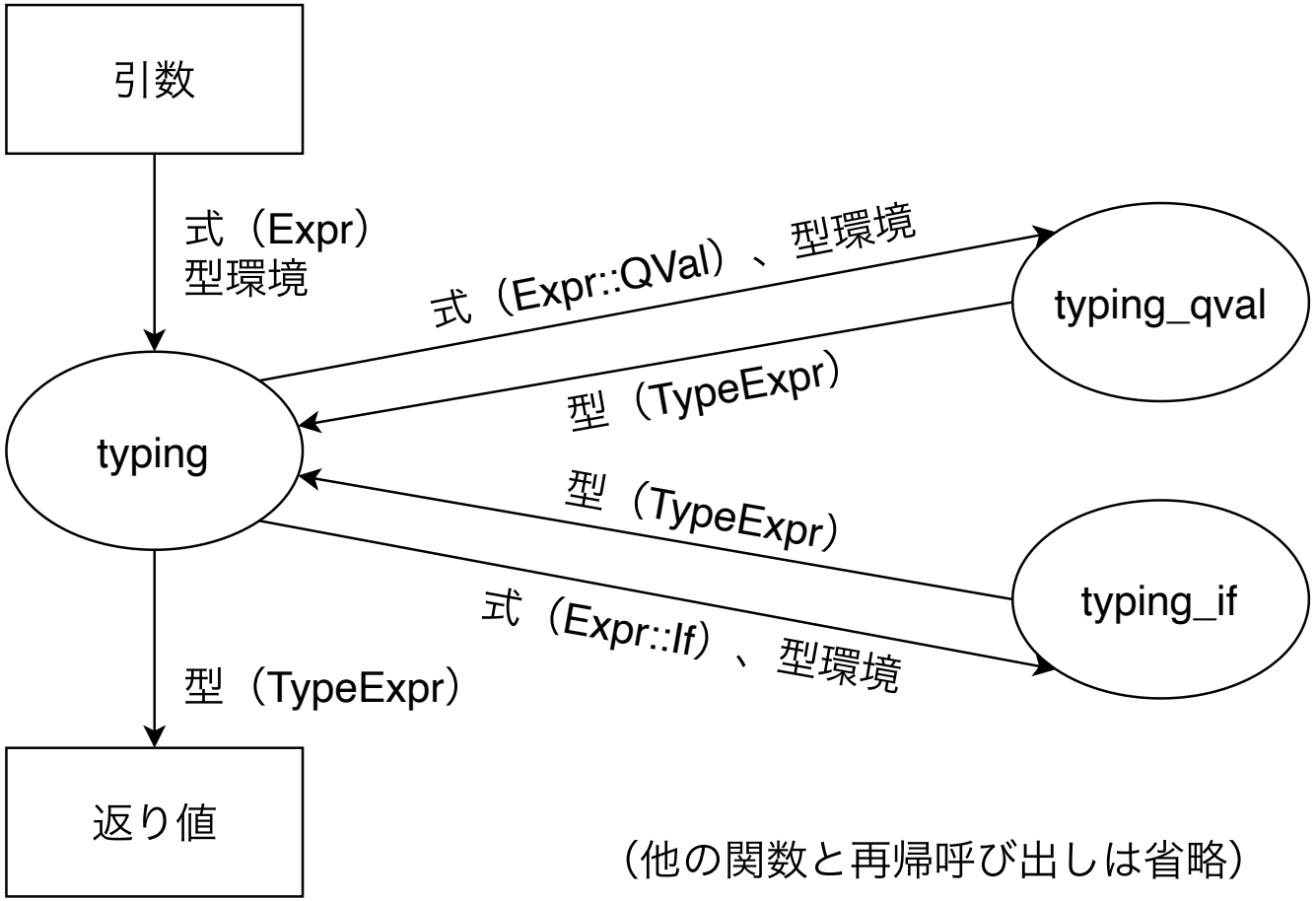
```rust
#[derive(Debug)]
pub struct FreeExpr {
    pub var: String,
    pub expr: Box<Expr>,
}
```

```rust
#[derive(Debug, Eq, PartialEq, Clone)]
pub struct TypeExpr {
    pub qual: Qual,
    pub prim: PrimType,
}
```

```rust
#[derive(Debug, Eq, PartialEq, Clone)]
pub enum PrimType {
    Bool,                                    // 真偽値型
    Pair(Box<TypeExpr>, Box<TypeExpr>),      // ペア型
    Arrow(Box<TypeExpr>, Box<TypeExpr>), // 関数型
}
```

線形型システム

型検査器の設計概要



式を受け取って、型を返す関数を再帰的に呼び出す。

（他の関数と再帰呼び出しは省略）

typing関数

```rust
type TResult = Result<parser::TypeExpr, String>;

/// 型付け関数
/// 式を受け取り、型を返す
pub fn typing(expr: &parser::Expr, env: &mut TypeEnv, depth: usize) -> TResult {
    match expr {
        parser::Expr::App(e) => typing_app(e, env, depth),
        parser::Expr::QVal(e) => typing_qval(e, env, depth),
        parser::Expr::Free(e) => typing_free(e, env, depth),
        parser::Expr::If(e) => typing_if(e, env, depth),
        parser::Expr::Split(e) => typing_split(e, env, depth),
        parser::Expr::Var(e) => typing_var(e, env),
        parser::Expr::Let(e) => typing_let(e, env, depth),
    }
}
```

- TResultが返り値の型
- typing関数は、式をうけとり、その式に応じて適切な関数を呼び出すのみ

型環境の操作

- 新しく変数が定義される場合は、スタックをpush
- 変数スコープから抜ける場合は、スタックをpop
- こうすることで、変数のスコープやシャドーイングを表現可能

値の型付け

- 値の型付けのまえに、もう一度制約を確認しよう
- 問題となるのは、ペアと関数

再訪T-Pair規則

$$\frac{\Gamma_1 \vdash e_1 : T_1 \quad \Gamma_2 \vdash e_2 : T_2 \quad \mathrm{un}(T_1) \quad \mathrm{un}(T_2)}{\Gamma_1 \circ \Gamma_2 \vdash \mathrm{un} < e_1, e_2 > : \mathrm{un}\ (T_1 * T_2)}$$

- 問題となるのは、修飾子qはunのとき
- この規則はつまり、unのペアを生成するときは、内包するe1、e2の型はどちらもlin型でないということになる

再訪T-Abs規則

$$\frac{\mathrm{un}(\Gamma) \quad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \mathrm{un\ fn}\ x : T_1\ \{\ e_2\ \} : \mathrm{un}\ (T_1 \to T_2)}$$

- こちらも、問題となるのはunの場合
- これはつまり、e2の型付けに使えるのは、xか、un型の自由変数のみといいうこと

typing_qval関数でのペア

```
fn typing_qval(expr: &parser::QValExpr, env: &mut TypeEnv, depth: usize) -> TResult {
    // プリミティブ型を計算
    let p = match &expr.val {
        parser::ValExpr::Bool(_) => parser::PrimType::Bool,
        parser::ValExpr::Pair(e1, e2) => {
            // 式e1とe2をtypingにより型付け
            let t1 = typing(e1, env, depth)?;
            let t2 = typing(e2, env, depth)?;
            // expr.qualがUnであり、
            // e1か、e2の型にlinが含まれていた場合、型付けエラー
            if expr.qual == parser::Qual::Un
                && (t1.qual == parser::Qual::Lin || t2.qual == parser::Qual::Lin)
            {
                return Err("un型のペア内でlin型を利用している".to_string());
            }
            // ペア型を返す
            parser::PrimType::Pair(Box::new(t1), Box::new(t2))
        }
```

e1とe2の型はそれぞれt1とt2で返してくれる

un型のペアがlin型の値を持つ場合エラー

typing_qval関数での関数（1/2）

```
fn typing_qval(expr: &parser::QValExpr, env: &mut TypeEnv, depth: usize) -> TResult
    // 省略
        parser::ValExpr::Fun(e) => {
            // 関数の型付け
```

1. un型の関数内でlin型の自由変数を使えないようにする

```
// un型の関数内では、lin型の自由変数をキャプチャできないため
// lin用の型環境を置き換え
let env_prev = if expr.qual == parser::Qual::Un {
    Some(mem::take(&mut env.env_lin))
} else {
    None
};
```

```
// depthをインクリメントしてpush
let mut depth = depth;
safe_add(&mut depth, &1, || {
    "変数スコープのネストが深すぎる".to_string()
})?;
env.push(depth);
env.insert(e.var.clone(), e.ty.clone());
```

2. 型環境のスタックをプッシュし、引数の型を追加

typing_qval関数での関数（2/2）

3. 関数内の式を型付け

4. 型環境のスタックをポップ。ポップした中にlin型が含まれていればエラー

```
// 関数中の式を型付け
let t = typing(&e.expr, env, depth)?;

// スタックをpopし、popした型環境の中にlin型が含まれていた場合、型付けエラー
let (elin, _) = env.pop(depth);
for (k, v) in elin.unwrap().iter() {
    if v.is_some() {
        return Err(format!("関数定義内でlin型の変数\"{}\"を消費していない", k));
    }
}
```

```
// lin用の型環境を復元
if let Some(ep) = env_prev {
    env.env_lin = ep;
}
```

5. 必要な場合はlinの型環境を復元

```
// 関数型を返す
parser::PrimType::Arrow(Box::new(e.ty.clone()), Box::new(t))
```

変数の型付け

- lin型の変数を利用した場合は、型環境のValueをNoneに設定
- 参考： `type VarToType = BTreeMap<String, Option<parser::TypeExpr>>;`

typing_var関数

```
fn typing_var(expr: &str, env: &mut TypeEnv) -> TResult {
    let ret = env.get_mut(expr);
    if let Some(it) = ret {
        // 変数の型が定義されている場合
        if let Some(t) = it {
            // 消費されていない
            if t.qual == parser::Qual::Lin {
                // lin型
                let eret = t.clone();
                *it = None; // linを消費    ←注目
                return Ok(eret);
            } else {
                return Ok(t.clone());
            }
        }
    }

    Err(format!(
        "\"{}\"という変数は定義されていないか、利用済みか、キャプチャできない",
        expr
    ))
}
```

if式の型付け

$$\frac{\Gamma_1 \vdash e_1 : q\ \mathrm{bool} \quad \Gamma_2 \vdash e_2 : T \quad \Gamma_2 \vdash e_3 : T}{\Gamma_1 \circ \Gamma_2 \vdash \mathrm{if}\ e_1\ \{\ \mathrm{e}_2\ \}\ \mathrm{else}\ \{\ e_3\ \} : T}$$

- if e1 { e2 } else { e3 }という式

  1. e1をtyping関数で型付けし、bool型かチェック

  2. e2とe3をtyping関数で型付けし、同じ型かをチェック
     （ただし、e2とe3で同じ型環境を利用）

  3. e2とe3実行後の型環境が同じかをチェック

typing_if関数

```
/// if式の型付け
fn typing_if(expr: &parser::IfExpr, env: &mut TypeEnv, depth: usize) -> TResult {
    let t1 = typing(&expr.cond_expr, env, depth)?;
    // 条件の式の型はbool
    if t1.prim != parser::PrimType::Bool {
        return Err("ifの条件式がboolでない".to_string());
    }

    let mut e = env.clone();
    let t2 = typing(&expr.then_expr, &mut e, depth)?;
    let t3 = typing(&expr.else_expr, env, depth)?;

    // thenとelse部の型は同じで、
    // thenとelse部評価後の型環境は同じかをチェック
    if t2 != t3 || e != *env {
        return Err("ifのthenとelseの式の型が異なる".to_string());
    }

    Ok(t2)
}
```

1

2, 3

splitの型付けの流れ

1. 定義する 2 つの変数名が同じかをチェックし、同じならエラーにする(これは線形型システムの仕様とは異なるが、実装上の都合としてこうする)

2. 分割対象とする式の型付けを行う

3. 型環境をプッシュし、変数の型を挿入

4. 本体の式の型付けを行う

5. 型環境をポップし、ポップした中に lin 型の変数が残っていないかをチェック

関数適用の型付け

1. (左側の)関数部分の型付けを行い、結果が関数型かをチェック
2. (右側の)引数部分の型付けを行う
3. 関数部分の引数の型と、引数部分の型が同じかをチェック

free式の型付け

- 説明は省略
- レポート課題とする（後ほど説明）

let式と型付け

- let式は以下のような糖衣構文（syntax sugar）とする

  ```
  let x : T = e1; e2
  ```
  は
  ```
  (lin fn x : T { e2 } e1)
  ```
  と言う意味

- 型付けは、関数定義と呼び出しに変換して実装してもよいし、適切に意味を考えて実装しても良い

線形型言語のlet式の例

```
let x : lin bool = lin true;
let y : lin bool = lin false;
lin <x, y>
```

サンプルコード

サンプルコード

- ./codes/ のフォルダ以下に、線形型言語のサンプルコードがある
- exからはじまるファイルが正しくコンパイルできるコード
- errからはじまるファイルが型付けエラーとなるコード
- parse_err.linがパースに失敗し、構文エラーとなるコード

実行例

```
$ cargo run codes/ex1.lin
式:
lin fn x : lin bool {
    if x {
        lin false
    } else {
        lin true
    }
}
```

の型は

```
lin (lin bool -> lin bool)
```

です。

レポート課題

問題1：線形型システムの実装

- Src/typing.rsで、typing_split、typing_app、typing_let、typing_free を実装し、線形型システムの実装を完成させよ
- 実装した後は、サンプルコードや、自分で作成した線形型言語のコードを用いて動作を確認しレポートとしてまとめること

レポート課題

- 線形型の型検査アルゴリズムをRust言語を用いて実装せよ
- レポートには、実装した部分のソースコードと解説、実行結果を含めること
- ソースコードは
  https://drive.google.com/drive/folders/1MVd0OOfPO5U_fbizMl-JxeeRgA4Ssc7S?usp=sharing
  を利用すること
- 締切：2023年12月29日23時50分（JST）