先進情報セキュリティとアルゴリズム 王先生第四回

28G23027 川原尚己

typing_split, typing_app, typing_let, typing_free の実装内容を以下に示す.

```rust
fn typing_split(expr: &parser::SplitExpr, env: &mut TypeEnv, depth: usize) -> TResult {
    let t1 = typing(&expr.expr, env, depth)?;
    let mut depth = depth;
    safe_add(&mut depth, &1, || "変数スコープのネストが深すぎる")?;

    match t1.prim {
        parser::PrimType::Pair(p1, p2) => {
            env.push(depth);
            // ローカル変数の型を追加
            env.insert(expr.left.clone(), *p1);
            env.insert(expr.right.clone(), *p2);
        }
        _ => {
            return Err("splitの引数がペア型でない".into());
        }
    }

    let ret = typing(&expr.body, env, depth);

    let (elin, _) = env.pop(depth);

    for (k, v) in elin.unwrap().iter() {
        if v.is_some() {
            return Err(format!("splitの式内でlin型の変数\"{k}\"を消費していない").into());
        }
    }

    ret
}
```

expr は split を表す変数であり，split される変数の左右の変数の型が格納されている．
まず，expr を typing 関数により型付けを行い，t1 に格納する．その後 t1 の左右の型を型環境 env に挿入する．最後に，split の結果として，元のペアの左右の型を return する．

```rust
fn typing_app(expr: &parser::AppExpr, env: &mut TypeEnv, depth: usize) -> TResult {
    let t1 = typing(&expr.expr1, env, depth)?;
    let t_arg;
    let t_ret;
    match t1.prim {
        parser::PrimType::Arrow(t1_l, t1_r) => {
            t_arg = t1_l; // 引数の型
            t_ret = t1_r; // 返り値の型
        }
        _ => return Err("関数型でない".into()),
    }

    let t2 = typing(&expr.expr2, env, depth)?;

    if *t_arg == t2 {
        Ok(*t_ret)
    } else {
        Err("関数適用時における引数の型が異なる".into())
    }
}
```

引数 expr は関数を表す変数であり，expr1 は関数部分を，expr2 は引数部分を表す．
まず，expr1 を typing 関数により，型付けを行い，それを t1 に格納する．関数型 t1 の引数型を t_arg に，返り値型を t_ret に格納する．expr2 を typing 関数により型付けを行い，それを t2 に格納する．最後に，t_arg と t2 で格納されている値が等しいか確認を行う．

```rust
fn typing_let(expr: &parser::LetExpr, env: &mut TypeEnv, depth: usize) -> TResult {
    let t1 = typing(&expr.expr1, env, depth)?;
    if t1 != expr.ty {
        return Err(format!("変数\"{}\"の型が異なる", expr.var).into());
    }

    let mut depth = depth;
    safe_add(&mut depth, &1, || "変数スコープのネストが深すぎる")?;
    env.push(depth);
    env.insert(expr.var.clone(), t1); // 変数の型をinsert
    let t2 = typing(&expr.expr2, env, depth)?;

    let (elin, _) = env.pop(depth);
    for (k, v) in elin.unwrap().iter() {
        if v.is_some() {
            return Err(format!("let式内でlin型の変数\"{k}\"を消費していない").into());
        }
    }

    Ok(t2)
}
```

expr1 を typing 関数により型付けを行い，let 式の型を t1 に格納する．t1 の型を型環境に追

加した後，let 式の本体部分である expr2 の型を t2 に格納する．最後に，t2 を return し，let 式の本体部分の型を返すことができる．

```rust
fn typing_free(expr: &parser::FreeExpr, env: &mut TypeEnv, depth: usize) -> TResult {
    if let Some((_, t)) = env.env_lin.get_mut(&expr.var) {
        if t.is_some() {
            *t = None;
            return typing(&expr.expr, env, depth);
        }
    }
    Err(format!(
        "既にfreeしたか、lin型ではない変数\"{}\"をfreeしている",
        expr.var
    )
    .into())
}
```

free 式内の変数に対応する型を型環境より取得する．その後，その型を None に書き換えることで削除する．最後に，型を free したことを反映するため，typing 関数を適用し，新たな型環境を返す．

以下に，各種エラー例及び実行例を示す．ただし，煩雑であるため抽象構文木は省略している．

```
式:
un <lin true, lin false>

型付けエラー: un型のペア内でlin型を利用している
Error: Typing
```

err1.lin

```
式:
lin fn x : lin bool {
    free x;
    x
}

型付けエラー: "x"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
Error: Typing
```

err2.lin

```
式:
lin fn x : lin bool {
    free x;
    free x;
    lin true
}

型付けエラー: 既にfreeしたか、lin型ではない変数"x"をfreeしている
Error: Typing
```

err3.lin

```
式:
split lin <lin true, lin false> as x, y {
    x
}

型付けエラー: splitの式内でlin型の変数"y"を消費していない
Error: Typing
```

err4.lin

```
式:
lin fn x : lin (lin bool * lin bool) {
    split x as a, b {
        if a {
            b
        } else {
            lin true
        }
    }
}

型付けエラー: splitの式内でlin型の変数"b"を消費していない
Error: Typing
```

err5.lin

```
式:
un fn x : lin bool {
    un fn y : un bool {
        lin fn z : un bool {
            lin <x, y>
        }
    }
}

型付けエラー: "x"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
Error: Typing
```

err6.lin

```
式:
let x : lin bool = lin true;
un false

型付けエラー: let式内でlin型の変数"x"を消費していない
Error: Typing
```

err7.lin

```
式:
lin fn x : lin bool {
    if x {
        lin false
    } else {
        lin true
    }
}

の型は
lin (lin bool -> lin bool)
です。
```

ex1.lin

```
式:
let x : un bool = un true;
if x {
    un false
} else {
    un true
}

の型は
un bool
です。
```

ex2.lin

```
式:
lin fn x : lin bool {
    free x;
    lin false
}

の型は
lin (lin bool -> lin bool)
です。
```

ex3.lin

```
式:
split lin <lin true, lin false> as x, y {
    free x;
    free y;
    un true
}

の型は
un bool
です。
```

ex4.lin

```
式:
(lin fn x : lin bool {
    if x {
        un <un true, un false>
    } else {
        un <un false, un true>
    }
} lin true)

の型は
un (un bool * un bool)
です。
```

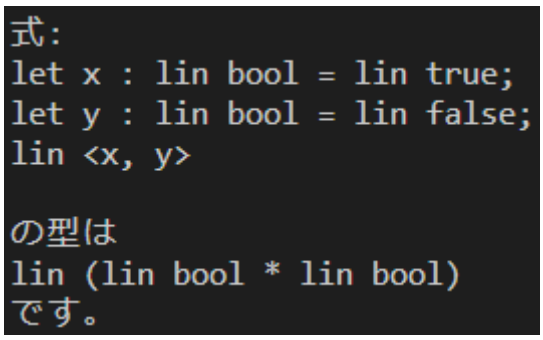
ex5.lin

```
式:
un <un true, un false>

の型は
un (un bool * un bool)
です。
```

ex6.lin

```
式:
lin fn x : lin (lin bool * lin bool) {
    split x as a, b {
        if a {
            b
        } else {
            b
        }
    }
}

の型は
lin (lin (lin bool * lin bool) -> lin bool)
です。
```

ex7.lin



ex8.lin

いずれの場合でも，error コードの場合は対応するメッセージを，正しく型が与えられているコードの場合は，その型を表示している．

参考文献：
https://github.com/ytakano/rust_zero/tree/master