直和型の代わりにユニオン型を持つ
静的型付け関数型プログラミング言語の開発
- Cotton -

1　背景

動的型付け言語に型注釈を付ける際の型システムなどで広く用いられているユニオン型は、静的型付け関数型言語の世界でも便利な機能となると考えられるが、静的型付け関数型言語の中でユニオン型をもつ言語は少ない。

2　目的

静的型付け関数型言語の世界でユニオン型がまだ人気でない理由として、ユニオン型を採用する静的型付け関数型言語がまだ少ないことや、あったとしても、ユニオン型は直和型（Rust の enum や、Haskell や OCaml のデータ型、F#の discriminated union のような機能）やシールドクラスなどの他の機能の補助的な役割を担うのみに留まっている点などがあると考えられる。

そこで本プロジェクトでは、網羅性チェックつきのパターンマッチや型推論、`flat_map` 関数に対するシンタックスシュガー（Haskell の do 記法や、Scala の for のような機能）などの、関数型言語によくある機能を取り入れつつも、直和型やシールドクラスなどではなくユニオン型を採用した型システムを持つシンプルな静的型付けプログラミング言語を開発することで、静的型付け関数型言語にユニオン型を採用する有用性を多くの人に知ってもらうことを目指した。

3　開発の内容

本プロジェクトでは静的型付けプログラミング言語のコンパイラとランゲージサーバーを開発した。言語の名前は Cotton という。次のような機能を実装した。

- ユニオン型
- 再帰的な型エイリアス
- 網羅性チェック付きのパターンマッチ
- 型推論
- 演算子定義
- 高階多相
- オーバーロード
- インターフェース
- モジュールシステム

ランゲージサーバーではシンタックスハイライトと型推論結果の表示の機能を実装した。

コンパイラ、ランゲージサーバーは共に MIT ライセンスで公開している。
以下サンプルコードである。

```
fib : I64 -> I64 =
    // 網羅性チェックつきのパターンマッチ
    | 0 => 0
    | 1 => 1    // `.`は左辺の値に右辺の関数を適用する演算子
    | n => (n - 1).fib + (n - 2).fib

main : () -> () =
    | () => 10.fib.println
```

ソースコード 1: fibonacci 数列の第 10 項目を出力する例

```
// 演算子定義
(..) : I64 -> I64 -> List[I64] =
    | a, b => (a < b).
        | True => a /\ (a + 1..b) // `/\`はリストのコンストラクタ
        | False => Nil

// 演算子の結合の方向・優先順位の宣言
infixl 4 ..

fizzbuzz : I64 -> String =
    // `/\`はタプルのコンストラクタでもある
    | n => (n % 3 /\ n % 5).
        | 0 /\ 0 => "FizzBuzz"
        | 0 /\ _ => "Fizz"
        | _ /\ 0 => "Buzz"
        | _ /\ _ => n.to_string

main : () -> () =
    | () => do
        (1..101).map(
            | i => i.fizzbuzz.println
        )
        ()
```

ソースコード 2: fizzbuzz の例

```
// 数字の0を表すデータ型
data O
// +1を表すデータ型
data S(A) forall { A } // `forall`は型引数の宣言

// 型エイリアス
type Nat = O | S[Nat]
type Even = O | S[Odd]
type Odd = S[Even]

// 偶数を引数にとって2で割った結果を言語組込みの整数として返す関数
div2 : Even -> I64 =
    | O => 0
    | S(S(n)) => 1 + n.div2
```

ソースコード 3: 型エイリアスの例

詳しい言語仕様やこのプロジェクトで使っている「ユニオン型」と「直和型」の意味については言語の紹介記事で解説している: https://zenn.dev/nanikamado/articles/e352e024a17b3f

4 従来の技術との相違

4.1 直和型との比較

ユニオン型はシンプルでありながら直和型の代替となることができ、さらにいくつかの面において直和型よりも表現力が高い。

直和型と比較したユニオン型のメリットについての詳しい説明は言語の紹介記事で解説している: https://zenn.dev/nanikamado/articles/e352e024a17b3f#直和型-vs-ユニオン型

4.2 Scala 3 との比較

Cotton と同じようにユニオン型を持っていて網羅性チェック付きのパターンマッチができる言語として、Scala 3 がある。Cotton が Scala 3 より優れている点としては、再帰的な型エイリアスを定義できることが挙げられる。

Cotton では型エイリアスを再帰的に定義することができるため、タプルを用いてリストを定義することによりタプルをリストの部分型としてみなすことができるようにするといったことが可能であるが、Scala 3 ではできない。

また、Scala 3 にはシールドクラス/シールドトレイとといった、ユニオン型と一部役割のかぶる機能があり、ユニオン型はシールドクラスの補助としてのみ使われることも多い。Cotton はシールドクラスなどを持っていないため、Scala 3 よりもユニオン型の表現力の高さを実例によって示す目的に適している。

4.3 OCaml との比較

OCaml の多相バリアントは Cotton のユニオン型にかなり近く、ほとんど同じことができる。

しかし、多相バリアントにも次のような欠点がありる。

- 多相バリアントのタグは宣言することなく使うため、タイポしたときや、中に入れる値の型を間違えたときのエラーが分かりづらいかもしれない。
- 組み込み型や、普通の直和型として定義してある型を多相バリアントに混ぜて使うときにも多相バリアントのタグをつける必要がある。
- OCaml には多相バリアントも直和型もあるので、どちらを使うべきか迷ってしまうかもしれない。

5　期待される効果

従来の関数型言語において直和型が担ってきた役割をユニオン型に担わせた Cotton の言語仕様は、関数型言語の開発者に新たな気づきを与えると考えている。今後ユニオン型の表現力を活かした書きやすいプログラミング言語が増えていくことが期待できる。

6　普及の見通し

実用的なプログラミング言語としての普及のためにはまだ不足している機能があるため、組み込み関数の追加や実行速度の改善、ランゲージサーバーの機能追加、フォーマッターなどの周辺ツールの整備などをし、本プロジェクトの開発期間終了後も引き続き開発に取り組んでいく。

7　クリエータ名（所属）

- 伊藤 謙太朗（電気通信大学 情報理工学域 I 類コンピュータサイエンスプログラム）
- 福間 遼太郎（慶應義塾大学 理工学部情報工学科）

（参考）関連 URL

- コンパイラ、ランゲージサーバーのリポジトリ: https://github.com/nanikamado/cotton
- VSCode 向け拡張機能のリポジトリ: https://github.com/nanikamado/cotton-language-server
- Cotton の紹介記事: https://zenn.dev/nanikamado/articles/e352e024a17b3f
- Playground: https://gitpod.io/#https://github.com/nanikamado/cotton-playground