Pattern Matching in Pizarnik

V. E. McHale

March 13, 2025

Contents

1 Introduction

Pizarnik's approach to pattern-matching is original, based on commitment to the idea of pattern-matching as inverse[1] and suggestive notation from linear logic (&, "with", is dual to disjunction, $\oplus$).

2 Extensible Pattern Matches

Tags such as `` `true `` are constructors, inspired by symbols (`` `blue ``) in k (preceded by symbols `'red` in Lisp). `` `true `` has type `-- `true`. It can also be said to have type $--\{\texttt{`true} \oplus \texttt{`false}\}$. To see why we would wish to admit `` `true `` $\oplus$ `` `false ``, consider

```
type Bool = { `true ⊕ `false };

not : Bool -- Bool
    := [ { `false⁻¹ `true & `true⁻¹ `false } ]
```

`` `false⁻¹ `` has type `` `false -- `` and hence `` `false⁻¹ `true `` has type `` `false -- `true ``; `` `true⁻¹ `false `` has type `` `true -- `false ``.

Perhaps counterintuitively, we get pattern-match exhaustiveness checking for free. $\oplus$ is a disjunction and thus we may generalize as we see fit, but & is a conjunction; it restricts.

To invert a sum type, we supply a product of its inverses (pattern match clauses). This is precisely the de Morgan laws.

2.1 Reusing Pattern-Match Arms

We can define a function

```
if : a a `true -- a
   := [ { `true⁻¹ drop } ]
```

This is not so useful on its own—it requires the programmer to supply exactly one value and then discards it—but see how it unifies:

```
else : a a `false -- a
     := [ { `false⁻¹ nip } ]

choice : a a Bool -- a
       := [ { if & else } ]
```

Pattern-match arms are defined just as functions and can be reused.

2.2 Wildcards

`drop : a --` is one of the building blocks of stack programming. We denote it as `_` as it functions like a wildcard in pattern-matching.

```
isZero : Int -- Bool
       := [ 0⁻¹ `true & _ `false ]
```

2.3 Typesafe Head

```
type List a = { `nil ⊕ List(a) a `cons };

type NE a = { List(a) a `cons };

head : NE a -- a
     := [ { `cons⁻¹ nip } ]
```

Any nonempty list can be supplied as an argument to a function on lists.

2.4 Or-Patterns

Suppose we have

```
type Ord = { `lt ⊕ `eq ⊕ `gt };
```

Then we can define

```
lte : { `lt ⊕ `eq } --
    := [ { `lt⁻¹ & `eq⁻¹ } ]

gt : Ord -- Bool
      := [ { lte `false & `gt⁻¹ `true } ]
```

So we get or-patterns for free and they are reusable as functions.

3 Error Hierarchies

References

[1] Daniel Ehrenberg. Pattern matching in concatenative programming languages. 2009.