Pattern Matching in a Stack-Based Concatenative Language

V. E. McHale

September 26, 2021

Abstract

Pattern matching works in an elegant but idiosyncratic way in stack-based languages. Thus we present canonical explanation and details for implementation.

Contents

1 Typing

The exposition here, including algebraic data types, depends heavily on the program of statically typing stack-based languages, see especially Diggins [1, 2] and also McHale [6].

As an example, let us consider

```
type Maybe a { Just a | Nothing }
```

Then the `Just` constructor and has type `a -- (Maybe a)` and `Nothing` has type `-- (Maybe a)`.

1.1 Patterns

Now let us consider types in the context of pattern matching;

```
isJust : (Maybe a) -- Bool
       =: [
    { case
        | Just    -> drop True
        | Nothing -> False
    }
]
```

The `Just` pattern has type `(Maybe a) -- a` while the `Nothing` pattern has type `(Maybe a) --`. No variables are bound, instead the constructor tag is stripped and remaining data is left on the stack; the inverse [3] of that particular constructor is applied when there is a match. Thus patterns for different constructors may have different types, but each arm must have the same type; the above will unify.

This scheme does pose some problems; we cannot use wildcards in some cases, for instance a wildcard cannot cover cases where two constructors (and thus patterns) have different type.

```
type These a { This a | That b | These a b }

pick : ((These Int) Int) -- Int
     =: [ {case | This -> | _ ->} ]
```

The above would be nonsensical: the `case` is missing a `That` and a `These` arm, but a `That` pattern has type `((These Int) Int) -- Int` while a `These` pattern has type `((These Int) Int) -- Int Int`; a wildcard pattern could not cover both!

But the advantage of this approach is inarguable: we can still use the concatenative style over the applicative style. This means there are no bound values, nothing named, and no need for environments; substitution is not refractory [8, 9]. Programs remain pointfree.

1.1.1 Or-Patterns

In the case when two constructors have the same type, we can use or-patterns [4], viz.

```
type OS { Linux | Darwin | Windows | FreeBSD | NetBSD }

isBSD : OS -- Bool
      =: [
    { case
        | (FreeBSD|NetBSD) -> True
        | _                -> False
    }
```

2 Implementation

2.1 Byte-Level Representations

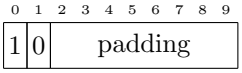
To see how `case` atoms are implemented, we must know how values are represented.

In Kempe [7], one begins with a tag for the constructor, then the data in question, in the order of the fields in question. Padding is added if needed so that all values of the same type has the same size; this depends on sizing for Kempe types.

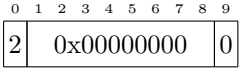
As an example, consider:

```
type These a { This a | That b | These a b }
```

Then a value of type `((These Int) Bool)` would be 9 bytes. A `False That` of such type would be encoded like so:



while `False 0 These` would be:



Kempe does not allow dynamically sized data (and hence recursive data types). For dynamically sized recursive data types, one would not have padding.

Let us return to the `isJust` example:

```
isJustInt : (Maybe Int) -- Bool
          =: [
    { case
        | Just    -> drop True
        | Nothing -> False
    }
]
```

Then, for the `Just` arm:

- Check the tag. If it is equal to 0, execute `drop True`.
- Otherwise, execute the next arm.

For the `Nothing` arm:

- We do not need to check the tag since it is the last arm in the `case`.
- Discard all padding. A `Nothing` of type `(Just Int)` will have 8 bytes of padding.
- Execute `False`.

Kempe does not do anything more advanced, but this approach can be extended to decision trees [5] or jump tables.

2.2 Exhaustiveness Checking

Exhaustiveness checking is relatively easy: we simply check that every constructor is present.

Exhaustiveness checking allows us to implement everything without any chance of runtime errors; this is desirable in embedded environments.

References

[1] Christopher Diggins. Typing functional stack-based languages, 2007.

[2] Christopher Diggins. Simple type inference for higher-order stack-oriented languages, 2008.

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

[4] Luc Maranget. Warnings for pattern matching. *Journal of Functional Programming*, 17(3):387–421, 2007.

[5] Luc Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ML '08, page 35–46, New York, NY, USA, 2008. Association for Computing Machinery.

[6] Vanessa McHale. A compiler for a stack-based language. `http://vmchale.com/static/original/compiler.pdf`, 2021.

[7] Vanessa McHale. kempe. `https://hackage.haskell.org/package/kempe`, 2021.

[8] Manfred von Thun. Frequently asked questions about joy for programmers. `http://joy-lang.org/faq/`.

[9] Manfred von Thun. Rationale for joy, a functional language. `http://joy-lang.org/rationale-for-joy/`.

A Examples

```
nip : a b -- b
    =: [ dip(drop) ]

fromMaybe : a (Maybe a) -- a
          =: [
    { case
        | Just    -> nip
        | Nothing ->
    }
]
```

A.1 Kleinsche Vierergruppe

```
type Element { E | A | B | C }

mult : Element Element -- Element
     =: [
        { case
            | E ->
            | A -> { case | E -> A | A -> E | B -> C | C -> B }
            | B -> { case | E -> B | A -> C | B -> E | C -> A }
            | C -> { case | E -> C | A -> B | B -> A | C -> E }
        }
]
```