COMP 321 Homework 4, Fall 2021: Pattern Matching

In this homework, you will implement *nested pattern matching*, as you are probably familiar with from SML. This means that you can `case` on types like booleans, numbers, strings, lists, tuples, and datatypes in an arbitrarily "deep" way. Many languages have adopted this kind of pattern matching — even Python is getting in on the fun.[1].

For example, the code for merging two sorted lists into a sorted list (which you might have seen in mergesort)

```
fun merge (l1 : int list , l2 : int list) : int list =
    case (l1 , l2) of
        ([] , l2) => l2
      | (l1 , []) => l1
      | (x :: xs , y :: ys) =>
            (case x < y of
                 true => x :: (merge (xs , l2))
               | false => y :: (merge (l1 , ys)))
```

pattern-matches on a pair of lists. We call each `p => e` separated by a vertical bar a "branch". Each branch consists of a pattern `p` and a "right-hand side" expression `e`. In this code, the three branches of the case on `(l1,l2)` distinguish the cases where

1. `l1` is empty and `l2` is any list
2. `l1` is any list and `l2` is empty
3. `l1` has at least one element and `l2` has at least one element

The patterns *bind* the variables in them in the right-hand side, so in the final branch, the right-hand side can use `x` to refer to the first element of `l1`, and `xs` to refer to the rest of `l1`, etc.

Patterns can "open up" as much of a data structure as you want them to; e.g. we could refine the match to

```
    case (l1 , l2) of
        ([] , l2) => e1
      | (l1 , []) => e2
      | (x :: [] , y :: ys) => e3
      | (x :: x' :: xs , y :: ys) => e4
```

[1] `https://www.python.org/dev/peps/pep-0636/`

to distinguish the cases

1. `l1` is empty and `l2` is anything
2. `l1` is anything and `l2` is empty
3. `l1` has one element and `l2` has at least one element
4. `l1` has at least two elements and `l2` has at least one element

etc.

Intuitively, a case expression steps by finding a pattern that *matches* the value being cased on. A value matches a pattern when the value has the same structure as the pattern up to the spots marked by variables. The case then steps to the corresponding right-hand side with the parts of the value corresponding to the variables in the pattern substituted in. Patterns can overlap, in the sense that two different patterns can match the same value. For example in

```
case ([] , []) of
    ([] , l2) => 1
  | (l1 , []) => 2
  | (x :: xs , y :: ys) => 3
```

the value `([],[])` matches both the first branch and the second branch, so without some further stipulation it is unclear which branch should be selected. We adopt a *first-match* semantics, which means that the first/topmost matching branch is selected — so in this case, the value is 1 and not 2.

In addition to understanding pattern matching, there are a few auxiliary goals for this homework. First, you will learn how datatypes can be encoded using products, sums, and recursive types. This encoding will shorten your implementation of pattern matching: you will only need to write code for constructors for product/sum/recursive types, not for arbitrary datatypes. Second, you will practice using fixed points to define recursive functions, and implement a type checker and operational semantics for them. Third, you will practice using exceptions for control flow in SML (see Lecture 15/16). This will simplify the code for your type checker and pattern matcher.

1 Syntax

The language you will implement in this homework is called FPC in PFPL Chapter 20, but we extend it with nested pattern matching.

This section describes the code provided in `syntax.sml`.

1.1 Types

For this assignment, the types are mostly the same as in your midterm code, except we also add recursive types and type variables:

```
datatype typ =
    ...
  | Rec of name * typ
  | TVar of name
```

In concrete syntax, we write `rec a is tau` for the abstract syntax `Rec(a,tau)`. The type variable name `a` is bound in `tau` and stands for a type. In concrete syntax, we write just `a` for the type variable, which is translated to the abstract syntax `TVar a`. Note that the concrete syntax for product/sum/function types requires parentheses around them, e.g. you have to write `(nat -> nat)` and not `nat -> nat`.

For example, the type `nat` of natural numbers is written

```
rec a is (Unit + a)
```

in concrete syntax, and as (for some variable name `a:name`)

```
Rec(a, Sum(Unit, TVar a))
```

in SML. This recursive type satisfies the type equation that `nat` is bijective with `unit + nat`, where we think of `inleft <>` as zero and `inright e` as the successor of `e`.

We have provided

```
    (* substTyp(tau, (sigma,a)) is the substitution tau[sigma/a] *)
    val substTyp : typ * (typ * name) -> typ

    (* check if two types are alpha-equivalent *)
    val alphaEquivTyp : typ * typ -> bool
```

which perform substitution of types for type variables, and test alpha-equivalence of types. Because types have bound variables now, it is no longer enough to use SML = to test if two types are equal, because e.g. `Rec(b, Sum(Unit, TVar b))` also represents the natural numbers.

1.2 Patterns

We represent patterns by the following datatype:

```
    datatype pat =
        TrivPat
      | PairPat of pat * pat
      | InLeftPat of pat
      | InRightPat of pat
      | FoldPat of pat
      | VarPat of name
      | WildPat
      | AsPat of pat * name
```

These correspond to the following concrete syntax:

`TrivPat`	`<>`
`PairPat(p1,p2)`	`< p1 , p2 >`
`InLeftPat(p)`	`inleft p`
`InRightPat(p)`	`inright p`
`FoldPat(p)`	`fold p`
`VarPat(x)`	`x`
`WildPat`	`_`
`AsPat(p,x)`	`name p as x`

1.3 Expressions

The new expressions that we haven't seen before are

```
datatype exp =
        ...
      | Case of exp * (pat * exp) list
      | Fold of (name * typ) * exp
      | Fix of typ * (name * exp)
```

These correspond to concrete syntax as follow:

```
Case(e, [(p1,e1),...,(pn,en)])      case e { p1 => e1 | ... | pn => en }
Fold((a,t), e)                      fold[rec a is t] e
Fix(t,(x,e))                        fix x:t.e
```

For case analysis, we represent the branches by a list of pairs of a pattern and an expression.

Recursion via Fix Unlike in previous assignments, we do not build structural recursion into case analysis (a la `rec` in Gödel's T). Instead, we have general recursion via `fix x:t.e`. For example, the double function is written like this:

```
let
    double be fix d:(nat -> nat).
              lam x : nat.
                 case x { 0 => 0 | s y => s s (d @ y) }
in
    double @ 4
end ;
```

and this expression evaluates to 8. At first, it may seem odd that there are two names for `double`, the name `double` used in the body of the let and the name `d` used for the recursive calls. This is because, in an SML function declaration

```
fun double(x) = case x of 0 => 0 | _ => 2 + double(x-1)
val z = double 4
```

there are really two different scopes for `double`: it is bound in the body of the function itself (for recursive calls) and in the subsequent code. The above syntax separates these into the let-bound name `double` (the subsequent code) and the `fix`-bound name `d` (the recursive calls). Of course, we can always alpha-convert these to be the same

```
let
    double be fix double:(nat -> nat).
              lam x : nat.
                 case x { 0 => 0 | s y => s s (double @ y) }
in
    double @ 4
end ;
```

Datatypes as recursive types Above, we said that we will use the type `rec a is (Unit + a)` to represent natural numbers. This means the numbers are defined as follows:

```
0      fold[rec a is (Unit + a)](inleft[rec a is (Unit + a)] <>)

1      fold[rec a is (Unit + a)](inright[Unit](
          fold[rec a is (Unit + a)](inleft[rec a is (Unit + a)]<>)))

2      fold[rec a is (Unit + a)](inright[Unit](
          fold[rec a is (Unit + a)](inright[Unit](
            fold[rec a is (Unit + a)](inleft[rec a is (Unit + a)]<>)))))

...
```

Roughly, `inleft <>` corresponds to 0, and `inright e` corresponds to `1 + e`. However, each sum constructor occurs with a `fold` constructor for the recursive type, so leaving off the type annotations we get

```
0      fold(inleft <>)

1      fold(inright(fold(inleft <>)))

2      fold(inright(fold(inright(fold(inleft <>)))))

...
```

By default, the type annotations are not printed out when expressions are displayed, but there is a flag in `print.sml` that you can change if you want to see them.

Simultaneous substitution To implement pattern matching, it is convenient to use *simulatenous substitution* $e[e_1/x_1...e_n/x_n]$. The idea here is that the variables $x_1...x_n$ are all free in e, but not in $e_1...e_n$. The simultaneous substitution replaces each x_i with the corresponding e_i. This could be implemented as an iterated substitution $e[e_1/x_1][e_2/x_2]...[e_n/x_n]$ (generalizing the two-binding `let` problem from before) but it is easier and more efficient to do the simultaneous substitution in one pass. This is implemented for you in

```
    type ssubst = (exp * name) list
    val subst : exp * ssubst -> exp
```

2 Type Checker

Note: as usual, the type checker and operational semantics can be implemented in either order, so feel free to work on the next section first if you want.

To simplify the code for type errors, we will use exceptions rather than the `typOrError` type from previous assignments. This means your type checker has the following type:

```
    exception TypeError of string
```

```
(* Given a context gamma and expression e,
   return tau if there is a tau such that gamma |- e : tau, or
   raise TypeError otherwise *)
val check : Syntax.exp -> Syntax.typ
```

The basic idea is to raise the exception `TypeError` (ideally with an informative error message) whenever you would have returned `IllTyped`. Using exceptions simplifies the code because you don't need to explicitly propagate the type errors. E.g. for `let`, instead of

```
        | Let (e1, (x,e2)) =>
                     (case (checkOpen (g, e1)) of
                          WellTyped t1 => checkOpen (((x,t1) :: g), e2 )
                        | IllTyped => IllTyped)
```

we can just write

```
        | Let (e1, (x,e2)) => checkOpen (((x,checkOpen (g, e1)) :: g), e2 )
```

and any type errors that arise while checking `e1` will be automatically raised.

Task 1 (5%). Write the cases of `checkOpen` for `Var`, `Lam`, `App`, `Let` (as above), `Triv`, `Pair`, `InLeft`, `InRight`. Hint: start with your midterm solution code for these and modify it to use exceptions. Hint 2: because a program is well-typed only if all of its sub-programs are well-typed, you should never need to `handle` an exception.

Task 2 (10%). Write the cases of `checkOpen` for `fix` (rule 19.1g on page 169 of PFPL) and `fold` (rule 20.2a on page 20.1 of PFPL).

2.1 Type checking Pattern Matching

In words, we type check a case expression $\mathsf{case}(e)\{p_1 \Rightarrow e_1 \mid \ldots \mid p_b \Rightarrow e_b\}$ as follows:

1. Type check e to see that it has some type τ.
2. For each branch i, check that the pattern p_i is a pattern for that type τ, binding some context Δ_i, and then check that using the variables in Δ_i, each right-hand side e_i has some type σ.
3. The case itself has type σ.

Formally:

$$\frac{\Gamma \vdash e : \tau \qquad \forall i,\ (p_i : \tau\ \mathsf{binding}\ \Delta_i)\ \text{and}\ \Gamma, \Delta_i \vdash e_i : \sigma}{\Gamma \vdash \mathsf{case}(e) p_1 \Rightarrow e_1 \mid \ldots \mid p_b \Rightarrow e_b : \sigma}$$

Intuitively, $p : \tau\ \mathsf{binding}\ \Delta$ is defined by

- `x` is a pattern of type τ for any τ, binding $x : \tau$.
- _ is a pattern of type τ for any τ, binding nothing.
- <> is a pattern of type `Unit`, binding nothing

- `inleft p` is a pattern of type `t1 + t2` if `p` is a pattern of type `t1`, and it binds what `p` binds
- `inright p` is a pattern of type `t1 + t2` if `p` is a pattern of type `t2`, and it binds what `p` binds
- `< p1, p2 >` is a pattern of type `t1 * t2` if `p1` is a pattern of type `t1` and `p2` is a pattern of type `t2`, and it binds all the variables bound by both of them.
- `fold p` is a pattern of type `rec a is t` if `p` is a pattern of type `t [ rec a is t / a ]`, and it binds what `p` binds.

Task 3 (20%). Implement the case of `checkOpen` for `case`, with its helper function `checkPat(p,tau)` implementing the relation $p : \tau$ binding Δ. You might want to write out inference rules for $p : \tau$ binding Δ formalizing the above informal description, like

$$\frac{}{x : \tau \text{ binding } (x : \tau)} \qquad \frac{}{\langle\rangle : \text{unit binding } \cdot} \qquad \frac{e : \tau_1 \text{ binding } \Delta}{\text{inleft}(e) : \tau_1 + \tau_2 \text{ binding } \Delta} \qquad \ldots$$

to guide your implementation of `checkPat`. Hint: the flow of information in `checkPat` is very different than `checkOpen`: for `checkOpen`, the context and expression are inputs, and the type is an output. For `checkPat`, the pattern and its type are inputs, and the context that it binds is an output.

For testing, there are examples in `examples.fpc` and the `Top.file_type` etc. commands are the same as in the previous homework.

3 Operational Semantics

We will *not* convert the `progress` function to use exceptions — this wouldn't help much, because most cases of it need to check whether the recursive calls return done or a step. So, many of the cases of `progress` for this language are unchanged from the midterm code.

Task 1 (10%). Implement the cases of `progress` for `fold`(20.3a, 20.3b on page 178 of PFPL) and `fix` (19.3h on page 170 of PFPL).

3.1 Stepping pattern matching

Intuitively, $\text{case}(e)\{p_1 \Rightarrow e_1 \mid \ldots \mid p_b \Rightarrow e_b\}$ steps by first stepping e until it is done, and then finding the first p_i that matches e. Matching a pattern p against an expression e produces a simultaneous substitution $[e_1/x_1, \ldots, e_k/x_k]$ for the variables $\Delta = (x_1 : \tau_1 \ldots x_k : \tau_k)$ bound by the pattern p, where each e_j has type τ_j. In words,

- `x` matches any `e`
- _ matches anything
- <> matches <>
- `inright p` matches `inright[...] e` if `p` matches `e`
- `< p1, p2 >` matches `< e1 , e2 >` if `p1` matches `e1` and `p2` matches `e2`

- `fold p` matches `fold[...] e` if `p` matches `e`

It's up to you to write out how to calculate the simultaneous substitution for each match. If you like, you can write out inference rules for a relation

$$p \,\mathsf{matches}\, e \,\mathsf{with} \theta$$

If $p : \tau \mathsf{binding} \Delta$ and $e : \tau$ then $p \,\mathsf{matches}\, e \,\mathsf{with}\, \theta$ should mean that θ is a simultaneous substitution for the variables Δ such $p[\theta] = e$. I.e. if $\Delta = x_1 : \tau_1 \ldots x_n : \tau_n$, then $\theta = e_1/x_1 \ldots e_n/x_n$ is a list of expressions where each $e_i : \tau_i$, and substituting these into p produces e.

Task 2 (20%). Implement the function `match (p : pat, e : exp) : ssubst` that determines if a pattern matches an expression, and returns the simultaneous substitution for all variables bound by the pattern — i.e. it implements $p \,\mathsf{matches}\, e \,\mathsf{with}\, \theta$. If the pattern does not match the expression, raise the `DoesntMatch` exception.

The operational semantics for case is:

$$\frac{e \mapsto e'}{\mathsf{case}(e)\{p_1 \Rightarrow e_1 \ldots p_n \Rightarrow e_n\} \mapsto \mathsf{case}(e')\{p_1 \Rightarrow e_1 \ldots p_n \Rightarrow e_n\}}$$

$$\frac{e \,\mathsf{value} \qquad p_i \,\mathsf{matches}\, e \,\mathsf{with}\, \theta \qquad (\forall j < i. \neg(p_j \,\mathsf{matches}\, e))}{\mathsf{case}(e)\{p_1 \Rightarrow e_1 \ldots p_n \Rightarrow e_n\} \mapsto e_i[\theta]}$$

Task 3 (10%). Implement the case of `progress` for `case`. Hint: at some point you will want to use the syntax `e1 handle DoesntMatch => e2` to handle the `DoesntMatch` exception. You can assume that the patterns are *exhaustive*, which means that every value matches at least one pattern p_i.

For testing, there are examples in `examples.fpc` and the `Top.file_eval` etc. commands are the same as in the previous homework.

4 Derived Forms/Examples

4.1 Derived Forms

Above, you implemented pattern matching only for sums, products, recursive types, which is theoretically sufficient to express all datatypes but practically a little hard to work with. We can very quickly get pattern matching for familiar types like booleans, natural numbers, lists by *defining* these types to be certain sum/product/recursive types, which means you don't need to extend the type checker or the operational semantics for them. For this assignment, we will hardcode a few such definitions in the SML implementation (a better solution would be for the language to have a general datatype mechanism, so that the programmer could define these in the language, rather than you defining them in the implementation).

For example, in `syntax.sml`, booleans are defined by

```
val boolTyp = Sum(Unit,Unit)
val trueExp = InLeft(Unit,Triv)
val falseExp = InRight(Unit,Triv)
val truePat = InLeftPat(TrivPat)
val falsePat = InRightPat(TrivPat)
```

That parser translates the concrete syntax `bool` and `true` and `false` to these, so, for example, you can write

```
case true {true => false | false => true};
```

instead of the expanded version

```
case inleft[unit]<> { inleft <> => inright[unit]<>
                    | inright<> => inleft[unit]<>};
```

(Note that these definitions are *transparent* in the sense that the definitions of `true` and `false` are exposed; you can also write

```
case true { inleft <> => inright[unit]<>
          | inright<> => inleft[unit]<>};
```

One other thing that the SML datatype mechanism does is that it *hides* the implementation of the datatype from the programmer, using something called *existential types*, which we haven't covered yet.)

Task 1 (10%). Define `nat` and `list` (lists of natural numbers) and their constructors and patterns similarly in `syntax.sml`. `nat` should be `rec t is (unit + t)` while `list` should be `rec t is (unit + (nat * t))` (see Lecture 13/14).

The concrete syntax is `nat`, `z` (zero), `s` (successor), `list`, `nil`, `cons`.

4.2 Programming in FPC with pattern matching

Task 2 (5%). We did not include product projections `projleft` and `projright`, and unfolding of recursive types `unfold` in FPC-with-pattern-matching, because they are special cases of pattern matching. In `tasks.fpc`, show how they are defined (your code should work for any product/recursive types, but you can type it as `((nat * nat) -> nat)` for the projections and `(nat -> (unit + nat))` for the unfold – we don't have *polymorphism* in FPC, so we can't yet state that these work for all types).

Task 3 (10%). In `tasks.fpc` define less-than (at type `((nat * nat) -> bool)`)[2] by recursion, and then translate the code for `merge` from the beginning of this handout from SML to FPC-with-pattern-matching.

5 Extra credit

Task 1 (a little%). The parser and syntax datatypes include a constructor for "as patterns" `AsPat(p,x)`, written in concrete syntax `name p as x`. The idea is that this matches the same things as `p` matches, but in addition to binding what `p` binds, it binds `x` to the whole value that `p` matches. For example,

```
case p { cons <x, name cons<y,xs> as tail > => ...}
```

[2]In an older version this said to return nat instead of bool; that is fine too. Before adding booleans to the assignment I was thinking that the function would return 0 or 1 as false and true.

matches a list with at least 2 elements, and binds `x` to the first element, binds `y` to the second element, binds `xs` to all of the other elements, and `tail` to all but the first element (i.e. to `cons<y,xs>`). Extend your type checker and operational semantics to this.

Task 2 (a little%). In FPC, `fix x:t.e` is actually definable from recursive types, as in Section 20.3 of PFPL. In `tasks.fpc`, code the fixed point of any functional `f : ((nat -> nat) -> (nat -> nat))`, and show how to use it to define the `double : (nat -> nat)` function without using the built-in `fix`. Hint: the construction in the book needs to be modified a little to work for a function `f` instead of a expression with a free-variable $x.e$, because the definition in the book uses a call-by-name substitution for the variable.

Task 3 (a lot%). Above, we assumed that every $\mathsf{case}(e)\{p_1 \Rightarrow e_1 \mid \ldots \mid p_n \Rightarrow e_n\}$ was exhaustive: if $e : \tau$, then every value of type τ matches some pattern p_i. Implement an *exhaustiveness checker* that takes a list of patterns and returns true if they are exhaustive and false otherwise.