Notes and Tips, week 3

Subscribe for email updates. PINNED

No tags yet. + Add Tag Sort replies by: Oldest first Newest first Most popular

Charilaos Skiadas COMMUNITY TA · 19 days ago

Here's some personal notes and tips for this week.

Notes for week 3

- Functions are **values**, they can be stored in pairs, in lists, passed as arguments to functions, returned as values from functions.
- Anonymous function syntax very useful for passing functions or returning functions.
- In the absense of recursion issues, `fun f x = e` is almost the same as `val f = fn x => e` (but the former is prefered style)
- Avoid 'unnecessary function wrapping'. (Prefer `val g = f` rather than `fun g x = f x`, and `fun f x = g` rather than `fun f x y = g y`)
- Make sure to clearly understand the type signature of the functions `map` and `filter`.
- Study and understand the mechanics of `Lexical Scope`. It is important.
- Very important to understand the difference between *functions* and *closures*.
- Think about why lexical scope is so important when dealing with "higher-order-functions".
- Always look at the type signatures of your functions, and think about how they make sense.
- A type signature like `'a -> 'b -> 'c` should be read as `'a -> ('b -> 'c)`.
- Functions come with closures. Those can be used to avoid recomputation of expressions that don't change with each call.
- Study and understand the type of `fold`.
- `fold` allows separating recursive traversal from the data processing.
- We can create our own `infix` operators.
- Currying allows for function that "apply their arguments one at a time"
- Study the `curry` and `uncurry` methods.
- SML's foldl method has slightly different form than the one presented in class.
- Some times `val f1 = f2` leads to "value-restriction" errors. Turning it into a `fun` form usually fixes this.

Notes on assignment 3

- It would be a great idea, while they are fresh in your mind, to review hw1 and hw2 and find ways to use higher order functions to simplify them. `map/filter/fold` can turn a lot of the previous problems into 1-liners:

    - `number_in_month` can be done via a `fold` or via `filter` together with `length`.
    - `number_in_months` can be done via a `fold` on the months list.
    - `dates_in_month` can be done via `filter`.
    - `dates_in_months` can be done via `fold`.
    - `month_range` can be done by creating a list of the days, followed by a `map` of `what_month` on that list.
    - `oldest` can be done via `fold`.
    - `get_substitutions` can be done via `fold`.
    - `similar_names` can be done via `map`.
    - `all_same_color` can be done via `fold`, but takes a bit of thinking, and requires treating the empty list separately.
    - `sum_cards` can be done via `fold`.
    - `officiate` can probably be modeled around a `fold`, where the list is the moves and the "accumulator" is a tuple describing the state of the system.
- You are required to use certain library functions. Do NOT create your own instead.
- Some questions ask that you use certain functions. Make sure you do!
- Use pattern matching. Avoid `#` and `hd`.
- Start with the provided file, you need what is in there. Add your code to it.
- Almost all answers in the first part are very short, many are a simple val binding.
- Study the relevant library documentation
- Don't forget about the function composition operator `o`: If you find yourself doing `fun f x = h (g x)`, this can be written as `val f = h o g`.
- It is often easier to first start with writing the function in the full `fun f x y = ...` form, then seeing if some of the arguments can be removed (see "unnecessary function wrapping").
- Take the time to write down the types of the functions you want to combine in a piece of paper, before trying to code. You can avoid a lot of typechecking errors by thinking about what makes sense.
- For problem 9, make sure to understand what the provided function `g` does. Its goal is to traverse a pattern structure, applying the functions `f1` and `f2` that you give it at key parts. Make sure to understand the types of those two functions. The solutions to this problem will all be very short, mostly partial applications of `g`.
- Typing `f;` in the REPL is a great way to find out about the type of a function `f`.
- For `longest_string1` note that if multiple strings have the same length, you need to return the earlier one. The opposite for `longest_string2`.
- Make sure you understand what `longest_string_helper` does. Its first argument is a function that compares two integers (for instance `<` or `>` are such functions, though see the next point for more on this). It then takes in curried form a second argument of a string list. Finally it returns a single string. Your code should be such, that when you use `>` as the first argument you get behavior identical to `longest_string1`. This helper will in effect be carrying out the common parts of `longest_string1` and `longest_string2`.
- Infix operators like `>`, `<>` etc cannot be used just like that as functions wherever you would have used say an `f`. However, you can obtain the "function" corresponding to an infix operator via `op`, like so: `(op <>)` is the function `f(x,y) = x <> y`.
- `longest_string3` and `longest_string4` are both extremely short and defined via `val` bindings. They should be just a partial application of `longest_string_helper`.

- For `first_answer` and `all_answers`, make sure you do some cases manually to get a feel for them.
- A nested pattern match for the function `match` is probably a good idea.
- Note that the arguments to `first_match` are curried.
- To help you think of the pattern matching problem, here's some patterns in "normal SML speak" and their equivalents in the assignment's setting.
    - `(_, 5)` ------> `TupleP [Wildcard, ConstP 5]`
    - `SOME (x, 3)` ------> `ConstructorP ("SOME", TupleP [Variable "x", ConstP 3])`
    - `(s, (t, _))` ------> `TupleP [Variable "s", TupleP [Variable "t", Wildcard]]`
- For the purposes of the assignment, an empty `Tuple` is different from `Unit`, and a `Tuple` with one element is different from that element by itself. So for example the pattern `UnitP` should not match the value `Tuple []`.

Challenge Problem

Edwin Dalorzo made an excellent post about the challenge problem a bit further down the page. I would only ruin it by saying anything more.

⬆ 16 ⬇ · flag

Ezra Schroeder Signature Track · 18 days ago 🔗

Charilaos Skiadas you said

- Make sure you understand what `longest_string_helper` does. Its first argument is a function that compares two integers (for instance `<` or `>` are such functions). It then takes in curried form a second argument of a string list. Finally it returns a single string. Your code should be such, that when you use `>` as the first argument you get behavior identical to `longest_string1`. This helper will in effect be carrying out the common parts of `longest_string1` and `longest_string2`.

When I type "longest_string_helper >" in my sml file which has my solution to homework 3, the compiler complains at me (something about an infix operator). If I say "longest_string_helper (fn (x,y) => x > y)" it is perfectly happy. Based on that do you think I could be doing the problem right, or definitely wrong? Thanks.

⬆ 0 ⬇ · flag

Pavel Lepin COMMUNITY TA · 18 days ago 🔗

That might be worth clarifying, but there's an item down the list that does explain that operators can be converted to proper functions using `op`. (Which does pretty much the same thing you've done by stuffing it into an anonymous function.)

⬆ 1 ⬇ · flag

Ezra Schroeder Signature Track · 18 days ago 🔗

Okay so I think you're saying my answer might be right, just less elegant. Thanks.

↑ 0 ↓ · flag

Ezra Schroeder Signature Track · 18 days ago

Thanks Pavel Lepin, you're right, "longest_string_helper (op >)" looks better than "longest_string_helper (fn (x,y) => x > y)". How come I have to say (op >) when I call longest_string_helper and I can't just say op > without the enclosing parens?

↑ 0 ↓ · flag

Pavel Lepin COMMUNITY TA · 18 days ago

...and "less elegant" would be a strong way to put it -- `op` wasn't mentioned in the lectures, unless I'm very much mistaken, just here in the hints. :-)

↑ 1 ↓ · flag

Pavel Lepin COMMUNITY TA · 18 days ago

Remember that function application associates to the left, `a b c` means `(a b) c`, not `a (b c)`. This is needed to conveniently call curried functions.

↑ 1 ↓ · flag

Charilaos Skiadas COMMUNITY TA · 18 days ago

Thanks guys, I tried to clarify it a bit more to avoid the confusion.

↑ 1 ↓ · flag

+ Comment

Edwin Dalorzo · 18 days ago ★ APPROVED

This is my understanding of the challenge exercise. I think I am not revealing any implementation details.

Annotations About Challenge Exercise

My understanding of the challenge exercise is as follows:

Let's imagine that we have a case expression with different patterns, like

```
case x of p1 | p2 | ... | pn
```

Where `x` is of certain type `t` that we could infer out of the patterns `p1`, `p2`, ..., `pn`.

In summary the objective of the challenge exercise is to create an algorithm that (alike the SML compiler), is capable of inferring what is the type `t` of `x` based on the patterns `p1`, `p2`, ..., `pn`.

These patterns are provided as the second argument of the challenge exercise function and they represent every one of the branches in a case expression.

If all the patterns in the case expression are compatible with some type `t` then the answer is `SOME t`, otherwise `NONE`.

About the Type List

We would not need the first argument of the challenge exercise if it weren't because constructor patterns do not tell us of what type they are. For instance, consider a case expression like

```
case c of Red => ... | Green => ... | _ => ...
```

We cannot tell what is the type of `Red` or `Green` here. Likewise, in the challenge exercise if we found a constructor like

```
Constructor("Red",UnitP)
```

How could we possibly infer the type of this constructor unless we had some metadata?.

And so this explains why we need a first argument of the challenge function containing a type list. It is nothing but our definition of datatypes.

```
datatype color = Red | Green | Blue
```

Would become somewhat like:

```
[
    ("Red", "color", UnitT),
    ("Green", "color", UnitT),
    ("Blue", "color", UnitT)
]
```

Type Inference Examples

Example 1:

Supposing we had this function

```
fun b(x) =
   case x of
       (10) => 1
      | a => 3
```

The compiler would determine that `x` is an integer. How? Easy, one of the patterns is a integer constant. Thus, the other pattern named `a` must be an integer as well. And there you have it, we just inferred the type of `x`.

In our challenge exercise this pattern would be expressed as

```
[ConstP 10, Variable "a"]
```

And our algorithm should say that the answer is `SOME IntT` which corresponds with the type the compiler would infer.

Example 2:

A piece of code like the following would not even compile, because we cannot infer a common type for all patterns. The types in the different patterns are conflicting. We cannot tell if `x` is an integer or an option.

```
fun b(x) =
   case x of
      (10) => 1
      | SOME x => 3
      | a => 3
```

Thus, consider the following pattern, corresponding with the code above:

```
[ConstP 10, Variable "a", ConstructorP("SOME",Variable “x”)]
```

This cannot produce a common type and the answer our algorithm yields should be `NONE`, equivalent with the compiler throwing an error due to incapacity to determine a common type.

Example 3:

Let's do a more complicated example now:

```
fun c(x) =
    case x of
      (a,10,_) => 1
      | (b,_,11) => 2
      | _ => 3
```

What is the type of `x`?

Well, we can easily infer it's a tuple of three elements. Based on the patterns we know the second and third elements of this tuple are integers. The first one, on the other hand, can be "anything".

This would correspond with:

```
[TupleP[Variable "a", ConstP 10, Wildcard], TupleP[Variable "b", Wildcard, ConstP 11],
 Wildcard]
```

And the answer given by our algorithm should be: `SOME TupleT[Anything,IntT,IntT]`.

Example 4:

Let's use a datatype now.

```
datatype color = Red | Green | Blue
```

Then we need to define the first (metadata) argument of our challenge function as:

```
[("Red","color",UnitT),("Green","color",UnitT),("Blue","color",UnitT)]
```

Let's say now that we have a function like this:

```
fun f(x) =
   case x of
     Red => 0
     | _ => 1
```

Corresponding with something like:

```
[ConstructorP("Red", UnitP), Wildcard]
```

Our algorithm should go over the patterns and say this is of type:

```
SOME (Datatype "color")
```

Example 5:

Let's use now a more complex datatype

```
datatype auto =  Sedan of color
               | Truck of int * color
               | SUV
```

This would correspond to a metadata argument as follows:

```
[("Sedan","auto", Datatype "color"),("Truck","auto",TupleT[IntT, Datatype "color"),("S
UV","auto",UnitT)]
```

Let's say now that we had a function like this:

```
fun g(x) =
   case x of
```

```
      Sedan(a) => 1
      | Truck(b,_) => 2
      | _ => 3
```

What is the type of `x`? Well, we can easily infer they are all of type `auto`.

So, the following argument:

```
[ConstructorP("Sedan", Variable "a"), ConstructorP("Truck", TupleP[Variable "b", Wildcard]), Wildcard]
```

Should yield `SOME (Datatype "auto")`.

Example 6:

Let's now define a polymorphic type to make this interesting

```
datatype 'a list = Empty | List of 'a * 'a list
```

So, we must first define our metadata argument:

```
[("Empty","list",UnitT),("List","list",TupleT[Anything, Datatype "list"])]
```

The trick is to notice that the polymorphic type 'a corresponds to anything here, and so the type inference becomes a bit trickier later on.

Now if we had this function

```
fun j(x) =
   case x of
       Empty => 0
     | List(10,Empty) => 1
     | _ => 3
```

Evidently the patterns are of type `list`, but not just that, but a list of integers.

So, the following argument corresponding to the patterns in the function:

```
[ConstructorP("Empty",UnitP),ConstructorP("List",TupleP[ConstP 10, ConstructorP("Empty",UnitP)]), Wildcard]
```

Should yield: `SOME (Datatype "list")`.

This case is tricky, because `ConstP(10)` needs to correspond with `Anything` in the constructors metadata as you can see above.

Example 7:

Let's consider this variation of the previous case

```
fun h(x) =
   case x of
      Empty => 0
    | List(k,_) => 1
```

In this case `k` could be anything. So, we represent these branches as:

```
[ConstructorP("Empty",UnitP),ConstructorP("List",TupleP[Variable "k", Wildcard])]
```

And the answer should be `Datatype "list"`.

And once more, notice how `Variable "k"` needs to correspond with `Anything` in the constructor metadata.

So, in the previous example `ConstP(10)` and now `Variable "x"` can be considered "compatible with" `Anything`.

Example 8:

Just another example

```
fun g(x) =
   case x of
      Empty => 0
    | List(Sedan(c),_) => 1
```

Corresponding with:

```
[ConstructorP("Empty",UnitP),ConstructorP("List",TupleP[ConstructorP("Sedan", Variable
 "c"), Wildcard])]
```

Should evidently yield `SOME (Datatype "list")`.

Example 9:

What I *could not fully understand* was the part of the "most lenient" pattern. In the assignment we get two examples, not at all clear for me.

The first one suggest that we have two patterns of the form:

```
TupleP[Variable “x”, Variable “y”]
TupleP[Wildcard, Wildcard].
```

This would correspond to something like

```
fun m(w) =
    case w of
          (x,y) => 0
```

```
         | (_,_) => 1
```

Interestingly this would not compile, since the patterns are redundant, namely, we would alway go out throught the first branch. My conclusion was that this statement was simply used with illustration purposes.

We can infer that `w` is a tuple with two elements that can be of anything. So the answer to this type of patterns should be:

```
TupleT[Anything, Anything].
```

Maybe a TA can help me understand better this part.

Example 10:

The second example of the "most lenient" requirement is likewise rather ambiguous for me. The example given would not compile either because once more the patterns are redundant.

The second example suggest a list of patterns like this:

```
TupleP[Wildcard, Wildcard]
TupleP[Wildcard, TupleP[Wildcard,Wildcard]]
```

Which would correspond with

```
fun m(w) =
    case w of
      (_,(_,_)) => 0
    | (_,_) => 1
```

We can infer that `w` is a tuple of two elements, the first one can be anything, the second one is evidently a tuple of other two elements, which in turn can be anything.

So, if we had to infer this we had to say the type of this is

```
TupleT[Anything, TupleT[Anything, Anything]]
```

Which is the expected answer by the challenge exercise. But yet again, the compiler would not handle this type of expression without errors.

Somebody with a broader understanding of the so called "most lenient" requirement please enlighten me.

↑ 74 ↓ · flag

Charilaos Skiadas COMMUNITY TA · 18 days ago

Amazing work Edwin! Wish I could upvote you more than once, but this darned system keeps taking my vote away if I click multiple times.

↑ 3 ↓ · flag

Pavel Lepin COMMUNITY TA · 18 days ago 🔗

You could hit the "approve" button on top of that. :-)

↑ 0 ↓ · flag

Charilaos Skiadas COMMUNITY TA · 18 days ago 🔗

Good point!

Hm, still, only puts one asterisk next to the word APPROVED, wish I could add 2-3 more.

It will have to do I suppose ;)

↑ 0 ↓ · flag

Charilaos Skiadas COMMUNITY TA · 17 days ago 🔗

Edwin just a minor correction. You have in numerous places something analogous to: `SOME f x`, and most of not all should be `SOME (f x)`.

↑ 0 ↓ · flag

Charilaos Skiadas COMMUNITY TA · 17 days ago 🔗

I think there's also a "Camion" where it should say "Truck".

↑ 0 ↓ · flag

Edwin Dalorzo · 17 days ago 🔗

Awesome! Thanks for the feedback, Charilaos.

I have addressed the comment about some `SOME` and fixed the "Camion" thing :-)

I originally wrote all this in Spanish since I was on a study group of Spanish speakers, but I translated to English this week and I missed that word.

↑ 0 ↓ · flag

Charilaos Skiadas COMMUNITY TA · 17 days ago 🔗

As per the "most lenient" part, I think the best example is really your Example 3. The kind of answer you get there is the intent of "most lenient" I think. It's just the idea that beyond whatever's locked in place because of other restrictions, whatever's left needs to be flexible. The term is perhaps not the most accurate.

↑ 1 ↓ · flag

Charilaos Skiadas COMMUNITY TA · 17 days ago 🔗

Here's one interesting case to look at, in relation to Example 6 with the lists. The following set of patterns should ideally be failing (i.e. NONE), as it can infer that the `'a` is not viable:

```
[ConstructorP("Empty",UnitP),
 ConstructorP("List",TupleP[ConstP 10, ConstructorP("Empty",UnitP)]),
 ConstructorP("List",TupleP[TupleP[Wildcard, Wildcard], ConstructorP("Empty",U
nitP)]
```

I am not sure if the autograder tests for it. I do know my code fails it, but I haven't submitted yet.

↑ 1 ↓ · flag

Pierre Barbier de Reuille · 17 days ago 🔗

Isn't the problem here that `Anything` shouldn't be allowed in the type of a constructor? As this is the role of parametric types to allow for some form of `Anything` and we don't have parametric types here ...

↑ 0 ↓ · flag

Peng Sun · 17 days ago 🔗

I just want to thank you Edwin, for helping me fix the bug in my code. :)

↑ 1 ↓ · flag

Charilaos Skiadas COMMUNITY TA · 17 days ago 🔗

I think you are correct Pierre, I think to do it properly we would need to manage a type environment with type variables, and essentially run the type inference algorithm. I don't see an easy way around it. Even if I could detect the specific example I gave above by sort of testing all n*n combinations of patterns for inconsistencies, rather than my folding over the list and only compare one pair at a time, I don't see how I would work with a case like `datatype 'a foo = Foo of 'a | Bar of 'a`.

So I might have to let that case go by.

↑ 0 ↓ · flag

David Starks-Browning · 16 days ago 🔗

I don't understand how to apply the rule "You must return the “most lenient” type that all the patterns can have." Won't everything match "Anything"? In particular, for Example 1 above, why isn't the answer "SOME Anything" rather than "SOME IntT"? Does the “most lenient” type rule only apply if you can't otherwise match against IntT?

⬆ 0 ⬇ · flag

Pavel Lepin COMMUNITY TA · 16 days ago 🔗

Certainly not. `Anything` means that *all* possible values can be matched against this set of patterns, but matching `UnitP` against `Const 2` is definitely a type error. The only patterns that match `Anything` are `Wildcard` and `Variable of string`.

⬆ 0 ⬇ · flag

Ariën J. Molenaar · 16 days ago 🔗

Thanks a lot Edwin, your explanation helped a lot in understanding the exercise.

⬆ 1 ⬇ · flag

Liemin Ma · 15 days ago 🔗

Is the answer must be "SOME (Datatype "someDatatype")" ?

⬆ 1 ⬇ · flag

Philip Diessner · 15 days ago 🔗

Very nice post, Thank you. Made it much clearer what the actual question and wanted syntax was.

⬆ 0 ⬇ · flag

Hung-Kun Chen · 14 days ago 🔗

My code seems to pass the test cases of the grader.
But it does not pass Edwin's example 6 and 8.
It could pass a modified version of int list or auto list, where Anything is replaced by a specific type.

It is said in the lecture that list is not a type, but "int list" is a type.
So I think it probably okay to support only non-polymorphic type.

⬆ 0 ⬇ · flag

Michael Kashin · 13 days ago

So can we assume that the first argument can be empty i case there are no constructor patterns in the pattern list (2nd argument)?

↑ 0 ↓ · flag

Charilaos Skiadas COMMUNITY TA · 13 days ago

It should not matter to your code what the first argument is, until it hits a constructor pattern.

↑ 1 ↓ · flag

A post was deleted

Ilya Taranov · 11 days ago

For example 10 I somehow get

```
SOME (TupleT [Anything,TupleT [#,#]])
```

in repl. What would that mean?
However, it seems to be deep equal to the right answer.

↑ 0 ↓ · flag

Peter Eriksen COMMUNITY TA · 10 days ago

@Ilya Taranov: It means that the REPL has truncated the output. Search the forums for "printDepth".

↑ 0 ↓ · flag

Cosimo Stufano · 9 days ago

Thanks Edwin. I used your examples as my test cases. My code passed all tests except for example 8. However since the deadline was approaching I decided to submit to the grader anyway and I got 105. Apparently your examples are more strict than the grader tests. I followed the scheme that Charilaos proposed in another thread. Perhaps after the hard deadline we can exchange code. (I assume that your code passes all the examples)

↑ 0 ↓ · flag

+ Comment

HANG HANG · 17 days ago

For the challenge, could I assume that the first argument of typecheck_patterns contains all

constructor informations that used in all given patterns?
For example, will there be any case like:

```
ConstructorP("a", UnitP)
```

in some pattern but

```
("a","xxx",UnitP)
```

is not in the first argument

↑ 2 ↓ · flag

Pierre Barbier de Reuille · 17 days ago 🔗

I decided to handle this as an error in the matching, so if this happens no types can fit the pattern. To be more precise:

```
typecheck_patterns ([], [ConstructorP ("S", UnitP)]) = NONE
```

This is similar to having a constructor that exist, but with the wrong type as argument ...

↑ 5 ↓ · flag

HANG HANG · 17 days ago 🔗

Thanks! I just submit the file, with the same logic as you stated. It is accepted.

↑ 2 ↓ · flag

+ Comment

Abhinav Kumar · 13 days ago 🔗

Question about Problem 10: **match**

1. To compare the lengths of Tuple and TupleP, are we allowed to use List.length?
2. Are patterns with more than one variable with same name allowed. If no, are we allowed to use the method **check_pat** to verify the variable names?
3. If a **Variable "x"** matches **Unit**, is it correct to return SOME **[("x", Unit)]**

↑ 1 ↓ · flag

Charilaos Skiadas COMMUNITY TA · 13 days ago 🔗

> 1. To compare the lengths of Tuple and TupleP, are we allowed to use List.length?

Yes

> 1. Are patterns with more than one variable with same name allowed. If no, are we allowed to use the method check_pat to verify the variable names?

I did not check for them. I think it falls under the category of "you do not need to worry about that case"

> 1. If a Variable "x" matches Unit, is it correct to return SOME [("x", Unit)]

`("x", Unit)` should be one of the resulting bindings, yes.

↑ 1 ↓ · flag

+ Comment

Brian Bi · 13 days ago

How many points is the challenge problem worth this time?

↑ 0 ↓ · flag

Pavel Lepin COMMUNITY TA · 13 days ago

Five points.

↑ 0 ↓ · flag

Edwin Dalorzo · 12 days ago

Oh, you do not do a challenge problem for the points, you do it for the sake of your engineering curiosity and to prove yourself you can. Many would do it even if it was worth 0 points, and many would not surrender for days, and when finally solve it, would shout "I MADE FIRE!" and hit their chest repeatedly with their fists :-)

↑ 8 ↓ · flag

Peter Eriksen COMMUNITY TA · 12 days ago

> *...and when finally solve it, would shout "I MADE FIRE!" and hit their chest repeatedly with their fists.*

Those moments are priceless.

↑ 4 ↓ · flag

Brian Bi · 12 days ago

I was just asking because I wanted to make sure I had gotten the full score :-P

↑ 2 ↓ · flag

+ Comment

Abhinav Kumar · 11 days ago 🔗

With the challenge problem, is it possible for input patterns to have a constructor name that is not present in the input datatype list? On a similar note, do we have to check for tuple lengths in case of the patterns list having TupleP's?

↑ 0 ↓ · flag

Peter Eriksen COMMUNITY TA · 10 days ago 🔗

> *...is it possible for input patterns to have a constructor name that is not present in the input datatype list?*

Yes, it's possible.

> *...do we have to check for tuple lengths in case of the patterns list having TupleP's?*

Yes, because tuples of different lengths have different types like in ML.

↑ 0 ↓ · flag

+ Comment

Mikhail Maltsev · 10 days ago 🔗

>Somebody with a broader understanding of the so called "most lenient" requirement please enlighten me.

Edwin, I really enjoyed reading your post, but I think your understanding of "most lenient" type in terms of case expressions is actually a bit misleading. The inferred type indeed must be the most lenient, i.e. the most general one, and it is required to be accepted by each of the patterns. The semantics of case expression is somewhat different. So I think, that we should substitute this code:

```
case x of p1 | p2 | ... | pn
```

By some other variant, which would produce a type, which will be matched by every pattern. The type system in SML is, of course, much more complex, than the one we have in the example, so my analogy is not so straightforward: Imagine, that we have a set of functions and every pattern p1, p2, ... corresponds to constraints, that are put on their argument. Now, we infer a type that is accepted by each of these functions. I have chosen `bool` as a return type; the function calls are joined using `andalso`. This is not relevant to the problem, and this could have been done in a different way (for example, return `unit` and use `;` instead of `andalso`) Here is my example for patterns `Wildcard`, `Tuple[Variable("x"), Wildcard]` and `Tuple[ConstP(10), Variable("y")]`

```
fun p1 _=
    true
fun p2 (x, _) =
    true
fun p3 (x, y) =
    x = 10
test x =
    p1 x
    andalso p2 x
    andalso p3 x
```

When we run this in SML, the typechecker infers that x is `int * 'a` which corresponds to expected answer `SOME TupleT[IntT, Anything]`. Note: we are not dealing with values in this problem (I mean the type `valu` from problems 10 and 11) so we don't actually use the value of our constant.

Let's try this: `UnitP` and `TupleP[ConstP(42)]`.

```
fun p4 () =
    true
fun p5 x =
    x = 42
test2 x =
   p4 x
   andalso p5 x
```

Unsurprisingly, we get an error from typechecker. And our function should also output `NONE`.

Finally, let's try datatypes. For example, `[("T1Int", "t1", IntT), ("T1Unit", "t1", UnitT), ("T2T1", "t2", Datatype "t1")]` corresponds to:

```
datatype t1 =
    T1Int of int
    | T1Unit
datatype t2 = T2T1 of t1
```

Now we define the patterns: `[ConstructorP("T2T1", ConstructorP("T1Int", Wildcard)), ConstructorP("T2T1", Variable("x"))]`

```
fun p5 (T2T1(T1Int _)) =
    true
fun p6 (T2T1 x) =
    true
fun test3 x =
    p5 x
    andalso p6 x
```

We get back the type `val test3 = fn : t2 -> bool` (i.e. `t2`), so it should correspond to `SOME (Datatype "t2")`

↑ 3 ↓ · flag

+ Comment

VISHAL DEEPAK AWATHARE · 5 days ago

im having trouble understanding the first argument of the challenge problem, lets consider the example 6 of the given hints by edwin, let our first argument be

```
[("Empty","list",UnitT),("List","list",TupleT[IntT, Datatype "list"])]
```

I changed Anything to IntT
and our patterns are

```
[ConstructorP("Empty",UnitP),ConstructorP("List",TupleP[Variable "k", Wildcard])]
```

should this give NONE or SOME e1.

↑ 0 ↓ · flag

Peter Eriksen COMMUNITY TA · 4 days ago

In ML syntax this input would be:

```
datatype list = Empty of unit
              | List of int * list


case e of
    Empty () => e1
  | List (k, _) => e2
```

Do the patterns of the case-expressions typecheck? Let's examine each of the parts:

1. `Empty` takes a `()`, which is of the expected type (`unit`).
2. `List` takes, as expected, a pair. The pair is of a variable, and a wildcard, and they may therefore have the expected types `int`, and `list` respectively.
3. Both branches have type `list`, as we saw above, so the patterns typecheck.

The result of `typecheck_patterns` would therefore in this case be `SOME (Datatype "list")`. That's also what my solution returns, fortunately.

Does it make sense to you that the above ML code would typecheck?

↑ 1 ↓ · flag

VISHAL DEEPAK AWATHARE · 4 days ago

ok got it and yeah the typecheck does make sense to me more or less, but if instead of the Variable "a" it was an UnitT it would not type check right because the type checker required an IntT but instead it found an UnitT(Just for confirmation)

↑ 0 ↓ · flag

+ Comment

New post

To ensure a positive and productive discussion, please read our forum posting policies before posting.

B	*I*	☰	☰	Link	<code>	Pic	Math	Edit: Rich ▾	Preview

☐ Make this post anonymous to other students

☑ Subscribe to this thread at the same time

Add post