Extra Problems, 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 · a month ago

Here are some extra problems to work on after you have completed this assignment, and want even more challenge. Feel free to discuss them, but please don't give out full answers.

For all of these problems, it is worth-while to study the function types first. In fact the types should tell you what the function does.

- Write a function `compose_opt: ('b -> 'c option) -> ('a -> 'b option) -> 'a -> c' option` that composes two functions with "optional" values. If any of the functions returns `NONE` on its "step", the whole thing would be `NONE`.
- Write a function `do_until: ('a -> 'a) -> ('a -> bool) -> 'a -> 'a`. `do_until f p x` will apply `f` to `x` and result in a "new" `x`, and will continue to do so until `p x` is true. It will then return that `x`. Example: `do_until divBy2 isOdd n` (function names made up) will divide the number `n` by `2` until it reaches an odd number. In effect, it will remove all factors of `\` 2 `from` n`.
- Using the previous function, write a function `fixed_point: (''a -> ''a) -> ''a -> ''a` that given a function `f` and an initial value `x` applies `f` to `x` until such a point where `f x = x`. (Notice the use of `''` to indicate equality types)
- Write a function `map2: ('a -> 'b) -> 'a * 'a -> 'b * 'b` that given a function that takes `'a` values to `'b` values and a pair of `\` 'a `values returns the corresponding pair of` `'b` values. (This is fairly easy)
- Write a function `app_all: ('b -> 'c list) -> ('a -> 'b list) -> 'a -> 'c list`, so that: `app_all f g x` will apply `f` to every element of the list `g x` then concatenate the resulting `'c list`s. For example if `f n = [n, 2 * n, 3 * n]`, then `app_all f f 1 = [1, 2, 3, 2, 4, 6, 3, 6, 9]`.
- Implement `List.foldr` (http://sml-family.org/Basis/list.html\\\\\\\\\\\\\\#SIG:LIST.foldr:VAL).
- Write the function `partition: ('a -> bool) -> 'a list -> 'a list * 'a list`. It should not be done via using two `filter`s, as that would require traversing the list twice. Make sure to only traverse the list once.
- Write a function `unfold: ('a -> ('b * 'a) option) -> 'a -> 'b list` that produces a list of `'b` values given a "seed" of type `'a` and a function that given a seed produces `SOME` of a pair of a `'b` value and a new seed, or `NONE` if it is done seeding. For example here is an elaborate way to count down from 5: `unfold (fn n => if n = 0 then NONE else SOME(n, n-1)) 5 = [5, 4, 3, 2, 1]`.

- You can implement `foldl` using an appropriate `foldr` on functions. (This is fairly challenging)

Difference Lists

Many might find this one challenging.

A "difference list" is a function that "represents" a list `xs` in the following sense: Given a list `ys` as input, it returns the list `xs @ ys`, i.e. it prepends `xs` to its input. This can provide a somewhat efficient way of concatenating multiple lists, when the new lists would need to be appended on the right. Difference lists can help "delay" the computation, so that the appends can happen in their more efficient order, from right to left, instead of from left to right.

If some of this did not make sense, don't worry about it. Just think of it as practice in higher-order-functions.

First we need this new type:

```
type 'a dlist = 'a list -> 'a list
```

So a dlist is just a function that given a list returns another list. Think of the dlist as denoting the "difference" between those two lists, in the above sense that there is some list `xs` so no matter what the input list `ys` is, the output would be the list `xs @ ys`.

Here's some functions to write:

- `toDlist: 'a list -> 'a dlist`. Takes an ordinary list, and turns it into a "dlist". So `toDlist xs` should be a function that when given a list `ys` returns `xs @ ys`.
- `append: 'a dlist -> 'a dlist -> 'a dlist`. Takes two dlists, and creates a new dlist. It should be the case, that if `f` represents the list `xs` and `g` represents the list `ys`, then `append f g` ends up being the dlist that represents the list `xs @ ys`. (But the idea is to do it without computing `xs @ ys`. You should arrange things so that the computation that would end up taking place if we did `append f g zs` would be `xs @ (ys @ zs)`.)
- `cons: 'a -> 'a dlist -> 'a dlist`. So if `f` represents the list `xs`, then `cons x f` would be the `dlist` representing `x :: xs`.
- `toList: 'a dlist -> 'a list`. Given a dlist, obtain the list it represents. This is easy.
- `concat: 'a dlist list -> 'a dlist`.

More fun with callbacks

This requires having seen/read the "callbacks" section. In there an interface was described, to work with callbacks:

```
val cbs : (int -> unit) list ref
val onKeyEvent (int -> unit) -> unit
val onEvent int -> unit
```

Where `cbs` holds a list of the callbacks to be called, `onKeyEvent` allows you to register a new callback (I will call them handlers), and `onEvent` calls all the registered callbacks. We will do various modifications to this.

The first modification, is that we will allow the handlers to return a boolean instead of `unit`. The return value is meant to indicate whether we should "stop" the propagation of the event. If a handler returns `true`, then any remaining handlers in the list must not be called for that event. Newly registered handlers must take precedence. The new signatures would be:

```
val cbs : (int -> bool) list ref
val onKeyEvent (int -> bool) -> unit
val onEvent int -> unit
```

You should implement this before moving on.

For the second modification, we will allow the ability to un-register handlers. This will be done as follows: `onKeyEvent` will, instead of returning nothing, return a value that its caller could then use to remove this function from consideration. In order to do this, we need to be able to "compare" two functions, which is not possible as functions are not equality types. To get around this, we'll use a little trick: Two references are equal exactly if they are the *same* reference. So a simple way to attach a "tag" to something to make sure you can identify it uniquely later, is to use a `ref ()`, which is a value of type `unit ref`. These don't hold any useful information, but you can test them for equality and they will only agree if they are trully the same. So with this in mind, let's define a "handler" datatype and modify our interface:

```
datatype handler = Handler of (int -> bool) * unit ref
val cbs : handler list ref
val onKeyEvent (int -> bool) -> handler
val removeHandler: handler -> unit
val onEvent int -> unit
```

The method `removeHandler` takes a handler `Handler (f, r)` and looks through the `cbs` list looking for a handler `Handler (f', r')` whose reference is identical to `r` (i.e. `r = r'`). It then removes that handler from the list. You can decide what to do if such a handler cannot be found. You can fail silently or raise some sort of exception. Alternatively, you can have a `bool` type as the return value, an report the success or failure that way.

You should implement this before moving on.

This next part is definitely a bit more advanced. We will move on to adding "topics" to our events, essentially build a simple PubSub system. The "data' we transmit will still be single integers. Users of our system can "subscribe" their handler to either a specific topic or to all topics. Users can also use the "publish" function to publish their data on a specific topic, or all topics. Then handlers who have subscribed to those topics will be called.

We keep the callbacks for each topic in a simple list of `(topic, callbacks)`.

So here is the new interface to implement (you may find it useful to create other helper functions):

```
datatype topic = AllTopics | Topic of string
datatype handler = Handler of (int -> bool) * unit ref
val cbs : (topic * (handler list ref)) list
val subscribe: topic -> (int -> bool) -> handler
val unsubscribe: handler -> unit
val publish: topic -> int -> unit
```

Publishing on "AllTopics" should make all handlers fire, regardless of which specific topic they had registered for or if they had registered for "AllTopics" instead. Publishing on any topic should always fire any handlers that have subscribed to "AllTopics".

Binomial Heaps

In this section we will build Binomial Heaps. Some familiarity with heaps in general will be helpful. We will follow closely section 3.2 of Okasaki's book.

This will be fairly lengthy, you have been warned.

We will build them as minheaps. The elements they will hold would be pairs, of an integer and some other element, and the integers are used for the ordering.

```
type 'a elem = int * 'a
```

The integer may be thought of as the "weight" of the element, and it would have to be provided via some sort of weight function `weight: 'a -> int`. Given such a function, we can create elements:

```
toElem: ('a -> int) -> 'a -> 'a elem
toElem weight x = (weight x, x)
```

From now on we will almost always be talking about elements. You might want to create a function:

```
leq: 'a elem -> 'a elem -> bool
leq (i, _) (j, _) = i <= j
```

Binomial trees

We start with the notion of a binomial tree. A *binomial tree of rank k* is defined recursively as follows:

1. A rank 0 binomial tree consists of a single element.
2. A rank k+1 binomial tree is formed from two rank k binomial trees by making one tree a left-most child of the other.

This is a very specific format. Let's have a look at a couple of these (and you can see more in the wikipedia entry):

1. A rank 1 binomial tree is formed from two rank 0 binomial trees, i.e. elements. So it has exactly two elements, one is the root and the other a child.
2. A rank 2 binomial tree is formed by taking two rank 1 binomial trees. The rule is that we use the root of one tree, and just add the root of the other as a new leftmost child. So that root now will

have a left child that forms a rank 1 binomial tree of itself, and also will have a single element right child (a 0 rank tree). So a rank 2 binomial tree consists of a root, whose children are roots of a 1-rank and a 0-rank tree respectively. We will see this pattern continue.
3. For a rank 3 binomial tree, we need to start with two rank 2 binomial trees. Then the root of one of them will remain a root, but the root of the other will be added as a leftmost child of the first tree's root. So a rank 3 binomial tree has a root whose children are in order a 2-rank tree, a 1-tank tree and a 0-rank tree.

This holds in general: A rank r+1 binomial tree consists of a root, with r+1 children, each of which is a root for a rank r, rank r-1, rank r-2, etc binomial tree.

The number of items in a binomial tree or rank r+1 is 2^r (2 to the power r).

All the trees we will be dealing with will be heap-ordered: Every node has a value no larger than that of its children.

In SML, we can represent a tree with a triple `(rank, root, children)`:

```
datatype 'a tree = T of int * 'a elem * 'a tree list
```

Make sure you understand how this type makes sense before moving on.

Some simple helper functions for trees, that you might want to implement (though depending on how much you use pattern matching for the rest, they may not be needed):

```
rank: 'a tree -> int
root: 'a tree -> 'a elem
```

We will discuss operations on such trees after we talk about heaps.

Binomial Heap

A *binomial heap* is a list of binomial trees, in increasing order of rank, where no rank may be repeated (but ranks may be skipped):

```
type 'a heap = 'a tree list
```

So for example a heap could consist of the list `[tree1, tree3]` of a rank 1 tree and a rank 3 tree. This heap will have exactly `2^3 + 2^1 = 10` elements. In general, since each tree of rank k holds 2^k elements, it follows that for every integer `N` the ranks of the trees to be used to form a heap of size N are forced upon us: They depend on the binary representation of N. For example for a heap of 21 (=16+4+1) elements we would end up using a tree of rank 0, a tree of rank 2 and a tree of rank 4, in that order. There is no other way to do it.

When a new element is to be added, these trees need to restructure themselves, and we will look at it in a moment. For the most part, all that is needed is to add lower rank trees into the mix, but some times merging needs to occur. For example if we had this heap of 21 elements, and we wanted to add one more element, then the two rank 0 trees would merge together to form a rank 1 tree.

Notice that upon seeing a binomial heap, it is not immediate which element is the smallest. You can count however on the fact that it is going to be in one of the roots that form the tree list. So it can be found in time logarithmic to the size of the heap.

We will now start building a number of useful operations. I will only describe the algorithms here, your task would be to implement them.

Preliminary operations

We start with operations to be used in testing that the invariants we are after are maintained. First we need two fundamental operations for building trees:

```
toTree: 'a elem -> 'a tree
treeLink: 'a tree -> 'a tree -> 'a tree
```

The first operation takes an element and turns it into the corresponding rank-0 tree. The second operation, `treeLink`, takes two rank-k trees and creates a rank-k+1 tree, preserving the heap ordering. So it must compare the two roots of the trees, and whichever has the largest value, it will add that tree as a leftmost child of the other tree's root. To add it as a leftmost child you simply need to cons it (`::`) to the child list. If `treeLink` is asked to link two trees of unequal ranks, it should throw an error.

Next, let us write a method that will test for a given tree if it is heap-ordered and binomial:

```
isValidTree: 'a tree -> bool
```

It will need to test a number of things:

1. That it has the correct number of children, and they have the correct ranks.
2. That the children themselves are valid (recursive calls). (`List.all` could come in handy)
3. That the values at the roots of the children are all no less than the root of the tree (the recursive calls will take care of making sure that the elements further down the trees are even bigger).

Now we will write some functions for heaps.

```
size: 'a heap -> int
isValidHeap: 'a heap -> bool
```

The first function counts the number of elements in the heap. (You can either use the rank of each tree to know how many elements are in it, or you can write a size function for trees as well). The second tests if the binary heap is valid: This means that all the trees in it are "valid" according to `isValidTree`, and further that they appear in increasing rank (and no ranks are repeating).

The fundamental function you will need to write is the following:

```
insertTree: 'a tree  -> 'a heap -> 'a heap
```

This function takes a (valid) binomial tree and a (valid) binomial heap, and merges the tree into the heap. It essentially has to walk throw the heap, and find the right spot to insert the tree. If the next tree in the list is of higher rank, then the new tree can be inserted before it, if it is of higher rank then it has to skip over it. If they are of equal rank, then the two trees need to merge (`treeLink`), producing a tree of rank one higher, and now we continue trying to insert this new tree.

Now we can discuss the main functions of a heap, which you should implement:

```
emptyHeap: 'a heap        (* This is actually just a value, not a function *)
insert: 'a elem -> 'a heap -> 'a heap
removeMinTree: 'a heap -> 'a tree * 'a heap
findMin: 'a heap -> 'a elem
removeMin: 'a heap -> 'a elem * 'a heap
heapToList: 'a heap -> 'a elem list
heapSort: 'a elem list -> 'a elem list
merge: 'a heap -> 'a heap -> 'a heap
```

The first is a value representing the empty heap (just an empty list). The function insert is given an element and a heap, in curried form, and returns the heap resulting from inserting the item into the heap. You insert an item by turning it into a rank-0 tree, then inserting that tree.

The function `removeMinTree` starts with a heap, and finds the tree whose root is the minimum element of the heap. Then it removes that tree from the list and returns a pair of this removed tree and the remaining of the heap.

The function `removeMin` takes a heap, and returns a pair of the minimum element along with the heap of the remaining elements. You need to implement this by using `removeMinTree` to get hold of the tree with the minimum element and the rest of the heap with that tree removed. Then the root of that tree is your minimum element, and the children of that root are binomial trees that need to be reinserted into the "rest of the heap" (likely via a fold and insertTree). The function `findMin` is a bit more economical, as it simply needs to return the smallest element.

The function `heapToList` forms a list of the elements in the heap in increasing order, but successively removing the smallest element. The function `heapSort` starts with a list of elements, inserts them into a heap, then calls `heapToList`. Kudos for using folds for this.

You can do variations of these last functions (and `findMin`), that work with and return `'a list`s instead of `'a elem list`s. They may need to be provided a "weight" function as an argument.

Lastly, the function `merge` merges two heaps, an operation that binomial heaps are well suited for.

0 · flag

Pavel Lepin COMMUNITY TA · a month ago

I'll "borrow" a couple more of these, if you don't mind. :-)

0 · flag

Charilaos Skiadas COMMUNITY TA · a month ago

No need to ask ;). Borrow away.

Though they are probably short on examples of using folds.

0 · flag

Pavel Lepin COMMUNITY TA · a month ago

I hijacked `do_until` and `fixed_point`, as I felt I was, um, fixating on folds a little, and wanted to add some other HOFs to the list.

0 · flag

Charilaos Skiadas COMMUNITY TA · a month ago

I pretty much "borrowed" those from Odersky's course ;).

I'm personally very partial to `compose_opt`.

0 · flag

Pavel Lepin COMMUNITY TA · a month ago

I figured out as much. :-)

I do like `compose_opt` naturally, but I feel that it really should be a part of a series of problems developing monadic computations, and since we're not covering that here, and we already have a ton of problems for people to work on, I decided to leave it out.

0 · flag

+ Comment

Pierre Barbier de Reuille · a month ago

Just a question: for the difference list, my solution is about 10 lines including everything. And every function is really simple. Does that sound reasonable, or am I probably missing something?

Note that, as far as I can tell, I am fulfilling all the criteria. If you want, I can give a set of tests that my program pass.

0 · flag

Charilaos Skiadas COMMUNITY TA · a month ago

Correct, all of mine are one-liners, most with less than 10 characters. Kudos if you did `concat` with a fold.

You're welcome to post some tests.

Basically the hard part of the problem is wrapping your head around the definitions. Once you get past that it's simple.

0 · flag

+ Comment

Pierre Barbier de Reuille · a month ago

I didn't think about using fold (shame on me). But while doing so, I had an interesting problem. My current definition is on the form:

```
fun concat lst = (* something simple *) lst
```

So I tried to reduce it to:

```
val concat = (* something simple *)
```

But then, the signature of the second form is `(?.X1 -> ?.X1) list -> ?.X1 -> ?.X1`, and if I try to explicitely specify the type with:

```
val concat : 'a dlist list -> 'a dlist = (* ... *)
```

I get the following error: `Error: explicit type variable cannot be generalized at its binding declaration: 'a`

I don't understand what is going on there.

0 · flag

Charilaos Skiadas COMMUNITY TA · a month ago

It is the "value restriction" problem. You can read a bit about it in week 3's notes, and we'll revisit it in week 4. It is somewhat mysterious, but there for a good reason.

The very last error is easy to understand though. `val`s are actual values, they can't have parametric types in them.

1 · flag

Pavel Lepin COMMUNITY TA · a month ago

The current thinking appears to be that the reasons are not so good (after oh so many years!) -- the equivalent keeps tripping up newcomers to Haskell, so they recently made `-XNoMonomorphismRestriction` the default in GHCi (but only in GHCi!) Once again, a historical perspective helps -- ML is an old language that explored much of the territory that was terra incognita back then.

0 · flag

+ Comment

Pierre Barbier de Reuille · a month ago

Here we're asked to implement `foldr` and then `foldl` in term of `foldr`. But wouldn't that be more efficient to implement the tail recursive `foldl` and then `foldr` in term of `foldl`?

This way, we end up with one tail-recursive function, and a function that, although it looks tail-recursive, is in effect re-implementing a call stack in a different way. As with the current version we have two call stacks: the one of `foldr` and the one we need to emulate `foldl` from `foldr`.

0 · flag

Charilaos Skiadas COMMUNITY TA · a month ago

I'm not sure I know how to do foldr in terms of foldl. As far as I can recall it only goes one way, but it's been a while since I've thought about it.

0 · flag

Pavel Lepin COMMUNITY TA · a month ago

Hmmm, that's a nice problem. I'm pretty sure that's true in some cases, at the very least - you *cannot* implement right fold in terms of left fold on infinite lazy streams, as the latter never terminates. But for finite strict lists it can be done by reversing the list (which is `List.foldl (op ::) []`), then applying the left fold once again. This is inefficient, however, and I believe that inefficiency can't be rid of, though I'm not up to proving that.

```
Standard ML of New Jersey v110.72 [built: Wed May 12 15:29:00 2010]
- val xs = [1, 2, 3, 4, 5];
val xs = [1,2,3,4,5] : int list
- fun f (x, (sum, acc)) = (x + sum, x + sum :: acc);
val f = fn : int * (int * int list) -> int * int list
- List.foldr f (0, []) xs;
[autoloading]
[library $SMLNJ-BASIS/basis.cm is stable]
[autoloading done]
val it = (15,[15,14,12,9,5]) : int * int list
- fun frev xs = List.foldl (op ::) [] xs;
val frev = fn : 'a list -> 'a list
- frev xs;
val it = [5,4,3,2,1] : int list
- fun ffoldr f z = (List.foldl f z) o frev;
```

```
val ffoldr = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
- ffoldr f (0, []) xs;
val it = (15,[15,14,12,9,5]) : int * int list
```

0 · flag

Pierre Barbier de Reuille · a month ago

You can absolutely write `foldr` in term of `foldl`. What you are doing is basically use composition to create your call stack:

```
fun foldr' f init lst =
    let fun curried_f v acc = f (v, acc)
        fun rev_comp (v, acc) =
            acc o curried_f v
    in
        List.foldl rev_comp (fn x => x) lst init
    end
```

However, as Pavel noted, then you cannot use it on an infinite stream. But as SML seems to be strict, then `List.foldr` wouldn't finish either in this case.

0 · flag

Pavel Lepin COMMUNITY TA · a month ago

We can define lazy streams in Standard ML, however. :-) We'll be discussing lazy streams in first section using Racket (which is also call-by-name), and exactly the same techniques can be used in SML.

0 · flag

Erik Colban · a month ago

I can confirm that it is possible to implement `foldr` in terms of `foldl`. There is a very nice one-liner. What you end up doing is building a function `(fn z => f (a, f(b, ...f(u, z)...)))` as your accumulator, which when applied to the zero of the `foldr` returns the desired result. One way to look at it, is that you're building a call stack on the heap. It is in no way any more efficient than a regular `foldr`.

0 · flag

Charilaos Skiadas COMMUNITY TA · a month ago

Hm now I wonder what gave me the impression that you could only do it one way and not the other. It was a while back, I can't recall where I saw the problem in the first place.

Fun problem to think about either way.

0 · flag

+ Comment

Erik Colban · a month ago

I liked the difference list problems but I am curious about some functions that are not mentioned.

1. `null: 'a dlist -> bool`. Returns true if the underlying list is empty
2. `hd: 'a dlist -> 'a`. Returns the head of the underlying list if not empty; raises an exception otherwise
3. `tl: 'a dlist -> 'a dlist`. Returns the `dlist` corresponding to the tail of the underlying list's tail, if not empty; raises an exception otherwise.

These functions (except for `tl`, which has a nice direct definition, if one can determine that the dlist is not empty) could be easily defined by using the `toList` function, but is that really needed? The underlying `'a list` is available in the closure of the `dlist` yet not accessible to other functions than the `dlist`. Is it possible to have several functions sharing a common closure? The OOP equivalent would be an object with several methods. What is the best approach to defining the 3 methods above?

1 · flag

Pierre Barbier de Reuille · a month ago

Of course you can have many functions sharing a dynamic environment. This was even in one of the example of the course. Look at the video "Optional: Abstract Data Types With Closures", which has set as examples implemented exactly like this. That being said, difference lists are optimised for concatenation, and I don't think you will be able to have `hd` and `tl` be simple here.

2 · flag

Peter Eriksen COMMUNITY TA · a month ago

> *Is it possible to have several functions sharing a common closure?*

The above comment pointed to some course material. I would like to add, that one of the simplest ways to achieve it would be to change `toDlist` to something like:

```
val toDlist = fn : 'a list ->
                   'a dlist *              (* the dlist *)
                   unit -> bool *          (* null of the above *)
                   unit -> 'a *            (* hd of the above *)
                   unit -> 'a dlist        (* tl of the above *)
```

```
val (x, nullx, hdx, tlx) = toDlist [1,2,3,4]
val hd_of_x = hdx()
```

Each of the returned functions have access to the same closure as `x`, and in particular access to the underlying list.

I haven't implemented it, so I might have made a mistake, but does the overall idea make sense?

2 · flag

Erik Colban · a month ago

Yes, thanks. I'll have to try it to get the hang of it. The type of hd should be `unit->'a` rather than `unit-> 'a dlist`? A more efficient implementation of `toList` is to provide the underlying `'a list`, rather than creating a new copy, which I guess it's what was originally intended, which means adding `unit->'a list` to the tuple.

0 · flag

Peter Eriksen COMMUNITY TA · a month ago

> *The type of hd should be unit->'a rather than unit-> 'a dlist?*

Correct, I've edited my comment.

Do you know whether `xs @ []` will return a copy of `xs`? Maybe it will return an alias?

0 · flag

Pierre Barbier de Reuille · a month ago

xs has to be copied, as you need to change the link to the next element in all of them. However, in `xs @ ys`, `ys` should be used as is. Maybe to explain, If `xs = [1,2,3]` we can see it as `(1 -> (2 -> (3 -> [])))`, where the brackets mark the pairs forming the links of the list. If you append `ys`, you will have `(1 -> (2 -> (3 -> ys)))`. As you can see, all the pairs are different, so they need to be created from the original `xs`.

0 · flag

Charilaos Skiadas COMMUNITY TA · a month ago

I think people are assuming a bit too much about dlist objects. Technically a dlist is just a function `'a list -> 'a list`, it does not need to represent appending a specific list `xs`, even though most of the use cases would be that way. For instance `fn ys => ys @ ys` would be a dlist value, but `toList()` on it would return `[]`. Or for that matter `fn ys => []` is also a dlist value, even though it most certainly cannot be thought of as a dlist in any reasonable way. So there may not even be a `xs` to try to preserve.

0 · flag

Erik Colban · a month ago

To answer Peter's question: I don't really know. I guess an implementation of `@` cold be such that if any of the operands is null, it returns an alias to the other. But if `dxs` is a dlist obtained through `toDlist xs` and `ys` is a list, the way I implemented `toDlist`, which I guess is how Charilaos expected me to implement it, resulted in `dxs ys` making a copy of `xs`, even in the case where `ys` is the empty list. Unless I create an ADT, as suggested by Pierre and Peter above, I don't see how I can avoid that.

0 · flag

Erik Colban · a month ago

I take that back. I could implement `toDlist` as follows:

```
fun toDlist xs ys = case ys of [] => xs
                             _ => ...
```

Sorry for all the confusion; I didn't think of that before just now!

0 · flag

Erik Colban · a month ago

Charilaos

You wrote in your original post: "A "difference list" is a function that "represents" a list `xs` in the following sense: ..."
I took that to mean that there is always an underlying `xs` . Your previous post seem to contradict that statement.

0 · flag

Charilaos Skiadas COMMUNITY TA · a month ago

Well the intent is to use them that way. But there is nothing in their definition as a type that enforces that.

0 · flag

+ Comment

Charilaos Skiadas COMMUNITY TA · a month ago

Added a section on implementing Binomial heaps, for those that feel particularly adventurous. It's actually easier than it sounds and a good practice on recursion.

0 · flag

Pavel Lepin COMMUNITY TA · a month ago

Now that's just evil! Pairing heaps win hands down on implementation complexity, and have better merge performance to boot. :-)

0 · flag

Charilaos Skiadas COMMUNITY TA · a month ago

Hm, I didn't know about those! That would be my homework I guess ;).

0 · flag

Pavel Lepin COMMUNITY TA · a month ago

I discovered those shortly after spending several hours banging my head against a binomial heap implementation in Scala. My feelings can be described as a strange mix of ecstatic happiness (those are *really* easy to implement) and resigned exhaustion (having inflicted all that suffering upon myself).

0 · flag

Pavel Lepin COMMUNITY TA · a month ago

Oh, in hindsight -- time well spent. The idea behind binomial heaps is profoundly beautiful.

0 · flag

Charilaos Skiadas COMMUNITY TA · a month ago

Agreed.

Hm, reading about Pairing Heaps now. Okasaki seems to suggest that they should only be used for applications that do not take advantage of persistence. Not quite sure why though.

0 · flag

Pavel Lepin COMMUNITY TA · a month ago

Damn, that's right. I can guess the reason for that -- the amortized bounds are for any chain of sequential actions, individual operations may be very expensive. If you keep repeating the same expensive operation over and over again, logarithmic del-min goes right out the window. Funnily enough I forgot about that and used those in a persistent setting in a contest recently. I suppose that worked because the sequence of actions wasn't designed to invoke pathological behavior.

0 · flag

Pierre Barbier de Reuille · a month ago

That may be a stupid question, but is this even valid in SML:

```
type 'a tree = int * 'a elem * 'a tree list
```

It seems to me that you cannot have recursive type aliases.

0 · flag

Pavel Lepin COMMUNITY TA · a month ago

That's right, recursive types should be defined using the `datatype` declaration. That shouldn't affect the gist of the problem.

0 · flag

Charilaos Skiadas COMMUNITY TA · a month ago

Hm good point. I'll try to fix it when I get some time. Today's piling up on me.

EDIT: Done.

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