Slide 1 - Programming languages

As many of you know, there exists many different programming languages, each with their own strengths and weaknesses, being more suitable for certain type of tasks or uses. Among these, common distinction: imperative languages and functional ones. Imperative languages are the most know, and are based on a sequence of instructions that change the memory state of the program. In these, mutations happens frequently, and side effects, that is, interaction with the outside world (e.g. file read) can happen at any time. In functional languages, instead, the main building blocks are functions and expression. Often, they describe what to do/obtain, rather than how to do obtain it. Commonly, they tend to avoid mutations and untracked side effects, in favor of immutability and pure functions (side effects are pushed to the boundaries of the program as much as possible)

Slide 2 - Functional languages - why?

During my thesis, I focused on functional programming languages, because albeit a bit less known than imperative ones, they present interesting properties.

Among those, they are often quite close to their mathematical foundations, which makes reasoning about programs easier, especially when doing static analysis, aka. studying the properties of a program without running it.

One consequence of their proximity with pure mathematical models is that we often don't want to care about memory management explicitly, and prefer that the language runtime handles it for us, through automatic garbage collection.

Slide 3 - Functional data structures

Let's see what the memory representation of functional data structures looks like. Here we take Haskell as the reference language, but the same concepts apply to many other functional languages as well.

In Haskell, OCaml, etc, a data structure is made of blocks, called *heap objects*, that are linked together through pointers. Each heap object represent a data constructor, and a given field of this constructor can only contains a primitive type (e.g. `Int#`) or a pointer to another heap object.

The first bytes of a heap object are reserved for a pointer to a header, aka. info table, which is the identity card of the corresponding data constructor.

Here we see on screen how a list of integers is represented in memory. The colon operator is cons, that is, the data constructor corresponding to a single link of the linked list. The bracket aka. nil operator is the empty list.

Slide 4 - Building order in functional languages - current

This memory organisation of data structure as a tree of heap objects, and the fact that data structures are immutable, has some consequences on how data structures are built in these languages.

Notably, in the case of lists, we can only join together a front value with an existing list to create a new list through the cons data constructor. So to build the list [1,2], we have to start by creating the list containing just 2 (by joining the value 2 and nil), and then we can join the value 1 with this list to obtain the list [1,2], as displayed on screen.

As you might imagine, this is quite restrictive, and might not always be the order we want to build our data structures in.

Slide 5 - Building order in functional languages - goal

So what about relaxing this restriction, and allowing to build data structures in any order, by joining our blocks however we want?

E.g, we could decide to first create the list containing the value 1, with no tail yet, and only later connect it to the tail containing the value 2.

But doing so means our data structures, at some point during their construction, will be in an invalid state, as they will contain holes, displayed as red squares on screen, that is, some fields of the data constructors won't be initialized until later.

Some of this has been pioneered by Minamide in his work on functional data structures with holes, but we will take a step further towards flexibility in structure building.

Slide 6 – Challenges

Having holes in our data structures is of course creating new challenges for functional languages: - First, it means our memory model is not purely immutable anymore, as we will need to mutate the unitialized fields later on to give them a value - Second, it means that our structures are no longer safe to read at any time. Indeed, it would be highly problematic to try to read the value of a field that is not initialized yet, as it would lead to unpredictable behaviour/segfault.

Hence, we will need to encapsulate these appearingly impure and unsafe operations in a functional interface, that ensures that the user won't suffer from the oddities happening under the hood.

Slide TOC - Destination passing

This interface will be in destination-passing style, that we will now describe

Slide 8 - Here comes destination-passing

Destination passing at its core its quite simple: instead of returning a value, a function in DPS will take a write pointer as a parameter, and write the result of its computation in the memory cell pointed to by this pointer.

The write pointer is called a destination. This technique actually takes his roots in systems programming, where it is used to avoid unnecessary allocations and copies, e.g. by reusing already allocated memory cells. It can also serve to avoid multiple calls to the allocator, by allocating a big chunk of memory at once, and passing pointers to sub regions of this chunk as several different functions. A function in DPS is always more general than its counterpart returning a value, as we can easily recover the latter with a simple wrapper allocating a fresh memory cell and passing a pointer to it as destination.

Slide 9 - Functional DPS API - Types (1/2)

In functional programming, we will use DPS to a slightly different extent. We will use destination as first-class objects that reference a yet unitialized field of a data structure under construction, aka a hole. A destination will be the only way to reference and write to a hole.

We will also need a wrapper to hide an incomplete structure while it's still under construction. This is the role of the Ampar type: it's an opaque pair with the structure under construction on the left, and the destinations pointing to holes of the structure on the right, tying everything together nicely.

The fact that destinations are first-class in our approach is a key difference to existing system from Minamide, in which we interact with holes through the incomplete wrapper (Ampar) as a whole.

As you can see on the diagram on the right of the slide, objects on top of the grey line are the objects that the programming can interact with, that is, the safe functional API objects, while everything under the grey line are the blocks we want to connect, but that the user can't access or read directly.

Slides 10 - Function DPS API - Types (2/2)

Indeed, the user will only be interacting through the `Ampar` and `Dest` types.

`Ampar` has two types parameters, the first one is the type of the structure under construction, and the other is an arbitrary type that must at least contain all the destinations pointing to the holes of the structure. This type can be a single dest, a tuple of dests, or even more complex containers with dests and auxiliary data too.

`Dest t` is just typed wrapper for a raw pointer, represented by type `Addr#` in Haskell. The type indicates which can be written to the corresponding hole.

Slide 11 - Example: Breadth-first tree relabeling in DPS (1/2)

The fact that destinations are first-class and can be stored in arbitrary data structures is a very interesting feature, contrasting with other existing approaches.

It can be used to implement algorithms that are not usually very natural to write in functional style, i.e. with pure functions and immutable data structures. One such example is breadth-first tree relabeling, where we take a binary tree with no values on nodes, and want to construct an output tree of the same shape, but with an integer on each node, with integers assigned in breadth-first order.

There is one obvious imperative algorithm, where we store the next nodes to process in a queue, and each time we process a node (at the front of the queue), we append at the end of the queue its children subtrees to be processed later; achieving breadth-first traversal this way.

Slide 12 - Example: Breadth-first tree relabeling in DPS (2/2)

In functional programming, one usually has to resort to clever artifices to achieve the same result. Here however, thanks to destinations, we can mimic the imperative algorithm quite closely, but storing the destinations to the holes of the output tree in a queue, and processing them as we would do in the imperative case.

We see on screen that we have one destination per hole in the output tree, and we store them in a queue, in breadth-first order. Each time we write a new hollow subtree to a destination, we add the destinations to the new uninitialized fields of this subtree at the end of the queue.

Slide 13 - Functional DPS API - Functions (1/2)

This makes a good transition to see which operations exactly we can do on destinations and ampars.

The first operation is adding a new block, aka a hollow data constructor, into a hole of an existing structure. At the top, we add a new cons cell to the existing incomplete list, while at the bottom, we write the empty list into the hole, completing the list.

Let's see now how this operation is typed in our API, as the `fill @Ctor` function

What happens behind the scenes hasn't changed at all. We only added the API stuff on top of the grey line. We see that `fill @'(:)` takes as a parameter a destination pointing to the hole we want to fill, and returns a pair of destinations pointing to the new holes that were created when adding the new cons cell.

For `fill @'[]`, it takes a destination as a parameter in the same way, but returns unit because there is no newly created hole when adding the empty list.

The second operation is writing an already complete value into a hole of the corresponding type. This is done with `fillLeaf`, as you can see here: we write the integer 1 to the hole of the list.

We see that `fillLeaf` takes as parameters a destination and a value of the corresponding type, and returns unit, as no new holes are created when doing such an operation.

The last operation on destinations is when you want to merge two incomplete structures together, by connecting the root of the second structure into a hole of the first one, here merging two incomplete lists into one. This is done with the `fillComp` function. The API stuff here is a bit more complex, but again, under the hood, it still the same simple merging operation we just detailed.

Slide 14 - Functional DPS API - Functions (2/2)

Now, let's focus on `Ampar`s.

First, we can create a new ampar, with just a hole, and a destination pointing to this hole (so no data yet inside). This is done with the `newAmpar` function

The next function is the one that allows to access and operate on destinations, that are always contained in Ampar, which is an opaque wrapper so the programmer can't access them in any other way.

`updWith` takes an ampar, and a function, and applies this function on the right of the ampar, so on the structure that contains the destinations of the structure. Applying the function on the destinations will often have effects on the underlying structure under construction, as shown on the diagram.

Finally, when we are done building our structure, we can extract the completed structure from the ampar with `fromAmpar'`

For that, the ampar must not contain any more destinations on the right, as destinations act as the witness for remaining holes. So `fromAmpar'` can only be called when the ampar contains a unit on the right, and if so, it returns the completed structure on the left so that the user can now use it and read it.

Slide 15 - Functional DPS API - Simplified version

Here we have a recap of the API we just saw.

We might spend a minute on the signature of `fill`. It takes the name of the data constructor to allocate as a type parameter, and must return a bunch of destinations of the right type depending on how many fields the chosen constructor has. This is done with the `DestsOf` type family, that maps a data constructor to the corresponding type of destinations for its fields.

I must admit that the API in its current form is still quite unsafe, as we will now see. But all basic operations are in place, in the sense that anything we do

under the carpet on our building blocks won't change at all in future iterations.

Slide 16 - TOC - Safety concerns for functional DPS

The rest of this talk will be dedicated to the safety aspects of our approach.

Slide 17 - Unrestricted use of destinations

As hinted before, the API in its current form is still quite unsafe, and we will now see why.

We update an ampar with `updWith`, applying a function on the right of the ampar, that is, on the destinations packed by the ampar. But nothing prevents us from applying a function that just drops the destination, and returns unit, so that we can next call `fromAmpar'` to extract the structure.

As we see here on the memory representation, doing so let us extract a hole into the normal world, meaning that we risk a segfault or undefined behaviour, depending on the value of this unitialized memory cell.

We ought to control the use of destinations more strictly to prevent this from happening.

Slide 18 - TOC - Linear types

To do so, we will mainly use linear types, that we will now briefly introduce.

Slide 19 - Linearity to the rescue

Linear types are a type system extension that let us track how many times a function uses its arguments. More precisely, a linear function has to use its argument exactly once.

The intuition is that we want destinations to be used linearly. Indeed, if they are dropped, as we just saw, a hole won't be filled and might be read by the user. If destinations are reused multiple times, we might run into complications when the compiler tries to optimize execution by changing the order of computation of expressions, which is usually safe to do in a pure language such as Haskell.

With a linear discipline on destinations, they can really act as the witness for holes, meaning that once consumed, we have the guarantee that the corresponding hole has been filled.

Slide 20 - Linear Haskell Overview (1/2)

Fortunately for us, linear types are already part of GHC, the Haskell compiler we use. In Linear Haskell, we have a new function type, s lolly t, which means that to produce t, the function will consume s exactly once.

Slide 21 - Linear Haskell Overview (2/2)

The definition of what consuming a value means is quite intuitive: for a data structure, it means consuming every part of the structure once. For a function, it means applying the function once, and consuming the result produced by this application once.

Let's see a few examples.

<...>

Slide 22 - Linear scopes and resources

One important aspect is that linearity on function arrows is a chains of consumption requirements, but we have no immediate way to say that a value of a given type must be only used linearly. To do so, we use the scoping function trick.

If we want the Resource type to be only ever used linearly, we make it so that the only way to obtain a Resource is to call the function `withResource`. The user don't receive a `Resource` directly, but rather, must provide a linear callback to do what they want with the resource. And this callback must consume the resource in the scope of the callback, because the return type must be `Ur t`, which is an infranchissable barrier for any linear argument. So the resource cannot leak.

Slide 23 - Updating the API with linearity

We can now update our API to use linearity on destinations.

First, we will use a token type as a source of exchange currency to create new ampars. Tokens can be duplicated, but cannot leak out of the scope of `withToken`. But once a token has been exchanged for an ampar, the corresponding ampar cannot be duplicated.

`newAmpar` and `toAmpar` in this new version of the API take both a token as an extra parameter. And we also update `updWith` to take a linear function instead of a regular function to apply on the destinations, so that the function must use the destinations it receives linearly. Here we have no need to guard the output of the linear callback with `Ur`, as the result of the callback is placed back inside the ampar, and not given to the user.

Slide 24 - Functional DPS API - with linearity

There aren't many changes in the whole API; we mostly changed most function arrows to linear ones, so that both Ampars and Destinations are linear resources now.

But is that enough?

Slide 25 - TOC - Safe yet?

Unfortunately, not quite yet.

Slide 26 - Scope escape - Example

There is one issue that has troubled me for many months, namely, scope escape.

Here is a program in which scope escape happens.

The essence of it is that if destinations are first-class resources, then we can store them through other destinations in the same way we would do with a regular value.

But as fillLeaf consumes its right hand side linearly now; if we use a destination on the right of fillLeaf, it means the destination on the right (here named d) will appear to be consumed linearly, although the correspond hole has not been filled.

Slide 27 - Scope escape - Bigger picture

It might appear as a very niche edge case, but it isn't really. In fact, scope escape is a fundamental issue regarding linear storage, let me explain.

Many linear APIs, and ours is no exception, rely on the fact that a given linear resource must be consumed in the scope it is given to the user.

But if we also have a function that can take a linear resource and store it away linearly, then it makes the linear value disappear from the scope it has been made available, and hence, we break the invariant our API relies on.

And unfortunately, there is no difference type-wise, with linear types, between an actual consumption of the value in the given scope, and a storage for future consumption through a function like `storeAway`.

Slide 28 - TOC - Avoiding scope escape in Haskell

Let's see what our options are to avoid scope escape with Haskell type system only.

Slide 29 - With just Haskell type system (1/2)

The first solution, and probably the simplest one, is to not provide a linear storage function like `fillLeaf`. In other words, making fillLeaf consume a destination linearly, but not consume the value to fill into the dest linearly. But if `fillLeaf` is no longer consuming the value in a linear fashion, it means that leaves of our data trees will be non-linear values. And because the spine of a data structure is neither inherently linear or not, it means we end up with a data structure that is in practice unrestricted.

So in this first approach, we replace the Ampar type by UAmpar, which is a wrapper to build an unrestricted structure, that is, a structure that can be duplicated at will (but cannot contain linear stuff).

This is the approach detailed in my first paper and practical implementation linear-dest, corresponding to chapter 3 of the manuscript. One consequence is that we cannot store destinations in ampars anymore, and that might be desirable in a few instances, e.g. for efficient breadth-first tree traversal.

Slide 30 - With just Haskell type system (2/2)

A second option is to distinguish destinations that will host linear values, and the ones that will host unrestricted values.

We can continue interacting with destinations for unrestricted values in the same way as in solution 1, but we must develop new safe primitives to interact with destinations for linear values, so that scope escape through linear storage is not possible. This is detailed in chapter 4 of the manuscript, and won't be presented here because it's quite complex and time is short.

Slide 31 - TOC - A more general solution: age control

But maybe we can do better, if we get away from Haskell limitations?

Slide 32 - Age system

One idea to avoid scope escape is to keep track of the scope from which a resource is originating, and to prevent storing away a resource in a scope that is older than the scope it comes from.

We will only track the scopes that are of interest for us, that is, how many nested `updWith` call we are in. We call this the age of a resource.

So when entering a new `updWith`, the new variable mapping for the destinations of the ampar will have age arrow0, and every pre-existing variable will see its age multiplied by arrow1, meaning that the scope integer is increased by one.

Slide 33 - Age system - Example

In this complex example, we see how the age of each variable evolves through nested `updWith` calls. Colors indicate in which scope objects live. Destinations are fundamentally living in a scope "one level more inner" than the structure they are pointing to. This is because when we use `updWith` on an ampar, the destination must be of a different level than everything existing before, including the left of the ampar.

Conversely, when we use fillLeaf, the right argument of fillLeaf must be one level older than the dest, to match the age of the structure it will end up into.

Slide 34 - Age control in fill functions

Such a rule for fillLeaf has been formalized in chapter 2 of the PhD manuscript, here adapted with Haskell notations. The core rule is that the value being fed into a destination must always be older than the destination, because we want to rule out the case of young things leaking into older scopes (aka scope escape).

Slide 35 - Age control prevents scope escape

And now, as we can see in updated representation of scope escape example, if we track ages of destinations, we end up with a younger d : arrow 0 being fed into older dest dd : arrow 1, which is rejected by the type system rule we just presented.

Slide 36 - TOC Formalization

Now, let's see how we formalized this age system.