5 Other uses for linear types

Linear types haven't really found a place in mainstream commercial application of software engineering. For this reason, while this section does not provide any concrete contributions, I thought it was warranted to list some new, innovative and unexpected uses for linear types.

Session types

Permutations

During my time on this Master program I was also working for a commercial company using Idris for their business: Statebox.

One of their project is a validator for petri-nets and petri-net executions: `FSM-oracle`. While the technical details of this projects are outside the scope of this text, there is one aspect of it that is fundamentally linked with linear types, and that is the concept of permutation.

FSM-Oracle describes petri-nets using *`hypergraphs`* those hypergraphs have a concept of *`permutation`* that allows to move wires around. This concept is key in a correct and proven implementation of hypergraphs. However, they also turn out to be extremely complex to implement as can attest the files `trying to fit` their definition into a `Category`.

Linear types can thankfully ease the pain by providing a very simple representation of permutations:

```
Permutation : Type -> Type
Premutation a = (1 ls : List a) -> List a
```

That is, a `Permutation` parameterised over a type `a` is a linear function from `List a` to `List a`.

This definition works because no elements from the input list can be omited or reused for the output list. *Every single element* from the argument has to find a new spot in the output list.

If we update this definition to use Vectors instead of lists we get

```
Permutation : Nat -> Type -> Type
Permutation n a = (1 ls : Vect n a) -> Vect n a
```

Which allows us to recover definitions like `swap`

```
swap : (1 n : Nat) -> Permutation (n + m) a
swap n xs = let (b, e) = splitAt' n xs in
        rewrite plusCommutative n m in e ++ b
```

And the categorical semantics are simply the ones from Idris types and functions.

Compile-time string concatenation

Strings are ubiquitous in programming. That is why a lot of programming languages have spent a considerable effort in optimising string usage and string API ergonomics. Most famously Perl is notoriou for is extensive and powerful string manipulation API including the much dreaded and beloved first-class regex support (with more recent additions including built-in support for grammars).

One very popular feature to ease the ergonomics of string literals is *string interpolation*. String interpolation allows you to avoid this situation

```
show (MyData arg1 arg2 arg3 arg4) = "MyData (" ++ show arg1 ++ " " ++ show arg2 ++ "
    " ++ show arg3 ++ ++ show arg4 ++ ")"
```

by allowing string literal to include expressions *inline* and leave the compiler to build the expected string concatenation. One example of string interpolation syntax would look like this

```
show (MyData arg1 arg2 arg3 arg4) = "MyData ({arg1} {arg2} {arg3} {arg4})"
```

The benefits are numerous but I won't dwell on them here. One of them however is quite unexpected: Predict compile-time concatenation with linear types.

As mentioned before, one intuition to understand the *erased linearity* `0` is to consider those terms absent at runtime but available at compile-time. In the case of string interpolation, this intuition becomes useful in informing the programmer of the intention of the compiler while using the feature. Indeed, in the following program we declare a variable and use it inside a string interpolation statement.

```
let name = "Susan"
    greeting = "hello {name}" in
    putStrLn greeting
```

However, it would be reasonable to expect the compiler to notice that the variable is also a string literals and that, because it is only used in a string interpolation statement, it can be concatenated at compile time. Effectively being equivalent to the following:

```
let greeting = "hello Susan" in
    putStrLn greeting
```

But those kind of translations can lead to very misleading beliefs about String interpolation and its performance implications. In this following example the compiler would *not* be able to perform the concatenation at compile time:

```
do name <- readLine
   putStrLn "hello {name}"
```

Because the string comes from the *runtime*.

Runtime you say? Wait a minute

Yes, we've already established this intuition that *erased* linearity is absent at runtime but allowed unrestricted use at compile-time. This intuition stays true here and allows us to explore the possibility of allowing the following program to compile

```
let 0 name = "Susan"
    1 greeting = "hello {name}" in
    putStrLn greeting
```

Since the variable `name` has linearity `0`, it cannot appear at runtime, which means it cannot be concatenated with the string `"hello "`, which means the only way this program compiles is if the string `"Susan"` is inlined with the string `"hello "`at compile-time.

Using holes we can describe exactly what would happen in different circumstances. As a rule, string interpolation would do its best to avoid allocating memory and performing operations at runtime. Much like our previous optimisation, it would look for values which are constructed in scope and simply inline the string without counting it as a use.

```
let 1 name = "Susan"
    1 greeting = "hello {name}" in
    putStrLn greeting
```

Would result in the compile error

```
There are 0 uses of linear variable name
```

Adding a hole at the end would show.

```
let 1 name = "Susan"
    1 greeting = "hello {name}" in
    ?interpolation
```

```
1 name : String
1 greeting : String
---------------------------
interpolation : String
```

As you can see, the variable `name` has not been consumed by the string interpolation since this transformation happens at compile time.

Having the string come from a function call however means we do not know if it has been shared before or not, which means we cannot guarantee (unless we restrict our

programming language) that the string was not shared before, therefore the string cannot be replaced at compile time.

```
greet : (1 n : String) -> String
greet name = let 1 greeting = "hello {name}" in ?consumed
```

```
0 name : String
1 greeting : String
---------------------------
consumed : String
```

The string `name` has been consumed and the core will therefore perform a runtime concatenation.

Invertible functions

Yet another use of linearity appears when trying to define invertible functions, that is function that have a counterpart that can undo their actions. Such functions are extremely common in practice but aren't usually described in terms of their ability to be undone. Here are a couple example

- Addition and substraction
- `::` and `tail`
- serialisation/deserialisation

The paper about `sparcl` goes into details about how to implement a language that features invertible functions, they introduce a new (postscript) type constructor `• : Type -> Type` that indicate that the type in argument is invertible. Invertible functions are declared as linear functions `A• -o B•`. Invertible functions can be called to make progress one way or the other given some data using the `fwd` and `bwd` primitives:

```
fwd : (A• -> B•) -> A -> B
bwd : (A• -> B•) -> B -> A
```

Invertible functions aren't necessarily total, For example `bwd (+ 1) Z` will result in a runtime error. This is because of the nature of invertible functions: the `+ 1` functions

effectively adds a `S` layer to the given data. In order to undo this operation we need to *peel off* a `S` from the data. But `Z` doesn't have a `S` constructor surrounding it, resulting in an error.

Those type of runtime errors can be avoided in Idris by adding a new implicit predicate that ensure the data is of the correct format:

```
bwd : (f : (1 _ : A•) -> B•) -> (v : B) -> {prf : v = fwd f x)} -> A
```

This ensures that we only take values of `B` that come from a `fwd` operation, that is, it only accepts data that has been correctly build instead of abitrary data. If we were to translate this into our nat example it would look like this

```
undo+1 : (n : Nat) -> {prf : n = S k} -> Nat
```

which ensures that the argument is a `S` of `k` for any `k`.

Support for non-computational theories

Agda features cubical type theory which allows to define a lot of theorems as a proof in the language, rather than as a postulate. However a notorious limitation of cubical Agda is the inability to be computed to runnable machine code. Efforts are going into inserting cubical theories into running programs, by restricting their uses to erased types, but this is still a work in progress.

However if the same effort existed in idris2, cubical theorems could all be annotated with linearity 0 such that the compiler trivially checks that they are never instantiated at runtime while providing the theorems and proofs we need at compile time.

Levitation improvements

The gentle art of levitation shows that a dependently typed language has enough resources to describe all indexed data types with only a few constructors. The ability to define types as a language library rather than a language features allows a treat deal of introspection given those data definitions. Indeed it is now possible to inspect them and derive interface definitions for them, as well as optimise some of them that are convertible to primitive types without loss of semantics.

Those features are plagued by a terrible shortcoming: the verbosity of the definitions not only make the data declaration hard to write and read but also makes the compiler spend a lot of time constructing and checking those terms. Generating those definitions is linear in therms of constructors and constructing a value is quadratic in its number of arguments.

This performance hit can be alleviated by erasing the proofs of well-formedness as mentioned in Ahmad Salim's thesis. Reducing the complexity from quadratic to linear. His implementation in Idris1 relies on erased terms with the `.` syntax. Ours can be strongly enforced by using the `0` linearity and the `Exists` data type.

Our linear version of levitation shows how linearity polymorphism can help simplify the code by removing the need for our custom index and relying on linear variables.

Levitation is now also incomplete with respects to data definitions in Idris2, indeed, the constructor

```
(::) : {n : Nat} -> {0 a : Type} -> a -> Vect n a -> Vect (S n) a
```

There is no way to define `a` to be erased with traditional levitation, it must be augmented with new constructors to reflect that the arguments are erased.

Coincidentally, this might actually *help* our use of levitated declarations since we can now assume every index with usage `0` to be a type parameter and every other parameter to be an index.

In addition, idris2 now supports type matching thanks to explicit linearity declarations for types. This is necessary in order to implement automatic derivation of interfaces with methods that use higher order functions like Functor and Applicative.