1 Literature review

This literature review, or context review, will enumerate and comment on the existing literature about linear types and related topics. In order to give context to this research project I will present it through three lenses: The first aims to tell the origin story of linear types and their youthful promises. The second will focus on the current understanding of their application for real-world use. And the last one will focus on the latest theoretical developments that linear types spun up.

Origins

Linear types were first introduced by J-Y. Girard in his 1987 publication simply named *Linear logic* . In this text he introduces the idea of restricting the application of the weakening rule and contraction rule from intuitionistic logic in order to allow to statement to be managed as *resources*. Linear terms once used cannot be referred again, premises cannot be duplicated and contexts cannot be extended. This restriction was informed by the necessity real-world computational restriction, in particular accessing information concurrently.

One of the pain points mentioned was the inability to restrict usages to something more sophisticated than "used exactly once". Linear variables could be promoted to their unrestricted variants with the exponential operator (!) but that removes any benefit we get from linearity. A limitation that will be revisited in the follow-up paper: Bounded linear logic.

It is worth noting that, already at this stage, memory implication were considered, typically the exponential operator was understood as being similar to "long term storage" of a variable such that it could be reused in the future.

Bounded Linear Logic, Girard 1991

Bounded linear logic improves the expressivity of linear logic while keeping its benefits: intuitionnistic-compatible logic that is computationally relevant. The key difference with linear logic is that weakening rules are *bounded* by a finite value such that each value

can be used as many time as the bound allows. In addition, some typing rules might allow it to *waste* resources by*underusing* the variable, hinting that affine types might bring some concrete benefits to our programming model.

As before, there is no practical application of this in terms of programming language, at least not that I could find. However this brings up the first step toward a managing *quantities* and complexity in the language. An idea that will be explored again later with Granule and Quantitative Type Theory.

NOTE: (I should re-read this one to find more about the expected uses at the time)

Applications

Soon after the development of linear types, they appeared in a paper aimed at optimising away redundant allocations when manipulating lists: The deforestation algorithm.

Deforestation (Wadler ref) is a small algorithm proposed to avoid extranious allocation when performing list operations in a programming language close to System-F. The assumption that operations on lists must be linear was made to avoid duplicating operations. If a program was non-linear, the optimisation would duplicate each of the computation associated with the non-linear variable, making the resulting program less efficient.

While deforestation itself might not be the algorithm that we want to implement today, it is likely we can come up with a similar, or even better, set of optimisation rules in idris2 by relying on linearity. In this case linearity avoid duplicating computation, this idea was again investigated in "Once upon a Type" which formalises the detection of linear variables and uses this information for safe inlining. Indeed arbitrarily inlining functions might result in duplicated computation (just like in the deforestation algorithm). Beside inlining and mutation, another way to use linear types for performance is memory space mutation.

Linear types ca change the world (Walder 1991) show that Linear types can be used for in-place update and mutation instead of relying on copying. And they both provide

programming API that make use of linear defintions and linear data in order to showcase where and how the code differ in both performance and API.

However the weakness of this result is that the API exposed to the programmer relies on a continuation, which is largely seen as unacceptable user experience (ask your local javascript developer what they think of "callback hell"). However, we can probably reuse the ideas proposed there and rephrase them in the context of Idris2 in order to provide a more user-friendly API for this feature, maybe even make it transparent for the user. This API problem carries over to another way linear types can be useful: Memory management and reference counting.

It turns out that linear types can also be used to replace entirely the memory management system, this paper shows that a simple calculus augmented with memory management primitives can make use of linearity in order to control memory allocation and deallocation using linear types.

This breakthrough is not without compromises either. The calculus is greatly simplified for modern standards and the amount of manual labour required from the developper to explicitly share a value is jarring in this day and age. What's more, it is not clear how to merge this approach with modern implementation of linearity (such a Quantitative Type Theory). While this paper seems quite far removed from our end goal of a transparent but powerful memory optimisation it suggest some interesting relation between data/codata and resource management (linear infinite streams?).

Invertible functions

More recently, linear types and linear functions have been useful in order to describe and study *invertible functions*, that is, functions that have an inverse such that their action can be undone. For example the function `+ 1` can is invertible and its effect is to subtract `1` from a value that was modified by `+1`.

Practical affine types

What does it mean to have access to linear and affine types *in practice*? Indeed, most the results we've talked about develop a theory for linear types using a variant of linear logic, and then present a toy language to showcase their contribution. However this does not teach us how they would interact and manifest in existing programs or in existing software engineering workflows. Do we see emerging new programming patterns? Is the user experience improved or diminished? In what regards is the code different to read and write? All those questions can only be answered by a fully fledged implementation of a progrmming language equiped to interact with existing systems.

Practical affine types show that their implementation for linear+affine types allow to express common operations in concurent programs without any risk of data races. They note that typical stateful protocols should also be implementatble since their language is a strict superset of other which already provided protocol implementations. Those two results hint at us that linear types in a commercially-relevant programming language would provide us with additional guarantees without impeding on the existing writing or reading experience of programs. A result that we well certainly attempt to reproduce in Idris2.

Linear Haskell 2017

Haskell already benefits from a plethora of big and small extensions, they are so prevalent that they became a meme in the community: every file must begin with a page of language extension declarations. Linear Haskell is notable in that it extends the type system to allow linear functions to be defined. It introduces the linear arrow `-o` which declares a function to be linear. Because of Haskell's laziness, linearity doesn't mean "will be used exactly once" but rather "*if* it is used, then it will be used exactly once".

This addition to the language was motivated by a concern for safe APIs, typically when dealing with unsafe or low-level code. Linear types allow to expose an API that cannot be misused while keeping the same level of expressivity and being completely backwards compatible. This backward compatibility is in part allowed thanks to parametric linearity, the ability to abstract over linearity annotations.

Cutting edge linear types

Granule is a language that features *quantitative reasoning via graded modal types*. They even have indexed types to boot! This effort is the result of years of research in the domain of effect, co-effect, resource-aware calculus and co-monadic computation. Granule itself makes heavy use of *graded monads* (Katsuma, Orchard et al. 2016) which allow to precisely annotate co-effects in the type system. This enables the program to model *resource management* at the type-level. What's more, graded monads provide an algebraic structure to *combine and compose* those co-effects. This way, linearity can not only be modelled but also *mixed-in* with other interpretations of resources. While this multiplies the opportunities in resource tracking, this approach hasn't received the treatment it deserves with regards to performance and tracking runtime complexity.

Quantitative type theory, Atkey 2016

Up until now we have not addressed the main requirement of our programming language: We intend to use *both* dependent types *and* linear types within the same language. However, such a theory was left unexplored until*I got plentty o nuttin* from McBride and its descendant, *Quantitative type theory*, came to fill the gap. While other proposal talked about the subject, they mostly implement *indexed* types instead of *fully dependent* types. In order to allow full dependent types, two changes were made:

- Dependent typing can only use *erased* variables
- Multiplicities are tracked on the *binder* rather than being a feature of each value or of the function arrow (our beloved lollipop arrow $\multimap$)

While this elegantly merges the capabilities of a Martin-Löf-style type theory (intuitionistic type theory, Per Martin-Löf, 1984) and Linear Logic, the proposed result does not touch upon potential performance improvement that such a language could feature. However it has the potential to bring together Linear types and dependent types in a way that allows precise resource tracking and strong performance guarantees.

Counting immutable beans

As we've seen, linearity has strong ties with resource and memory management, including reference counting. Though *Counting immutable beans* does not concern itself with linearity per se, it mentions the benefits of *reference counting* as a memory management strategy for purely functional programs. Indeed, while reference counting has, for a long time, been disregarded in favor of runtime garbage collectors, it now has proven to be commercially viable in languages like Swift or Python. The specific results presented here are focused on the performance benefits in avoiding unnecessary copies and reducing the amount of increment and decrement operation when manipulating the reference count at runtime. It turns out the concept of "borrowing" a value without changing it reference count closely matches a linear type system with graded modalities. Indeed as long as the modality is finite and greater than 1 there is no need to decrement the reference count. Here is an illustration of this idea

```
f : (2 v : Int) -> Int
f v = let 1 val1 = operation v -- operation borrow v, no need for dec
          1 val2 = operation v -- we ran out of ses for v, dec here
       in val1 + val2
```

In our example, since `v`could be shared prior to the calling of `f` we cannot prove that v can be freed, we can only decrement its reference count. However, by inspecting the reference count we could in addition reuse our assumption about "mutating unique linear variables" and either reclaim the space or reuse it in-place.

Linear uses

As we've seen linear types have lots of promise regarding uses in modern programming practices. They allow to model common patterns that are notorious for being error-prone and the source of important security flaws. Indeed a protocol might have been proven safe but it's implementation might not be. Linear types allow stateful protocols to make use of an additional layer of guarantee.

However those efforts have not been very successful in penetrating the mainstream of programming languages. While we will not discuss the reasons *why* we will note that linear types can actually *help* overcoming a common criticism of purely functional programming: That they are slow and do not/cannot provide any performance guarantee. Indeed, as we've seen in the review, linear types show a lot of promise regarding performance but have not realised that promise in practice. The hope is that Idris2 will provide the necessary infrastructure to demonstrate those ideas and finally legitimize linear types as a valid typing discipline for commercial software.

Indeed, Idris2 features both linear and dependent types, providing programmers with the tools to write elegant and correct software (Type Driven Development, Brady 2017), while ensuring performance. In this thesis I am going to reuse the intuition originally suggested by Wadler 1991 and rephrase it as

> If you own a linear variable and it is not shared, you can feely mutate it

This somewhat mimics Rust's (Nicholas D Matsakis and Felix S Klock II. 2014) model of ownership where variables are free to be mutated as long as they are uniquely owned. But differs in that it does not make use of *uniqueness* but rather uses linearity as a proxy for it.

This idea can be illustrated with the following example:

```
let 1 v : Int = 3
    1 v' : Int = v + 17
 in print v'
```

The bound variable `v` was created locally and is *linear* advertising a use of `1`. Then, the function `+ 17` makes use of it and puts the result in a variable `v'`. This new variable does not need to allocate more memory and can simply reuse the one allocated for `v`, effectively mutating the memory space occupied by `v'` without losing the immutable semantics we expect.

As one can see this does away with the continuation while reaping the benefits of in-place mutation awarded by our linear property. In addition, this innocuous could be expanded to more complex examples like

```
let 1 array = …
    1 array' = map f array
    1 array'' = map g array'
 in sum array''
```

Which could be executed in *constant space complexity* effectively duplicating the results from deforestation, but in a more general setting since this approach does not make any assumption about the type of the data handled, only that operations are linear and mutations are constant in space.