Linear Types in Haskell

Luís Brandão
Departamento de Ciências de Computação
Faculdade de Ciências Universidade do Porto, Portugal
U.C. Introdução à investigação científica 2017/2018
up201504440@fc.up.pt

October 29, 2018

Abstract

Linear types are actively being worked on and will soon be available in Haskell's GHC. They show great potential to improve the way resources are controlled by allowing us to restrict their usage through the type system. This report aims to introduce the reader to the concept of Linear Types, with a focus on functional programming. In it, we study the implementation of linear type systems, formulate typing rules for algorithmic type checking and write a type checker for lambda calculus in Haskell.

1 Introduction

Linear Types in programming languages allow finer control over the usage of storage. Although language agnostic, they have particularly interesting applications in functional programming languages where they provide new ways to fix old problems that functional programming struggles with, such as efficient mutable array implementations. By enabling usage restrictions on certain function values via the type signature, we can ensure that certain objects are not duplicated, or that an array is safe to update destructively because there is only one reference to it.
This report will in particular focus on Linear Haskell as described in *Linear Haskell: practical linearity in a higher-order polymorphic language* [3] and it's proposed implementation of Linear Types in Haskell. Ee will use *Linear Haskell* to refer to that paper's proposal to extend the GHC type system with linear types.
The main goal of this work is to introduce the reader to the topic of Linear Types, to enable grasping the broader literature that exists on this subject. It consists of:

- A survey of linear type systems.
- A new linear type system designed for type checking.
- An implementation of the previous system in Haskell

2 Background

2.1 Linear Logic

Linear logic [6], devised by logician Jean-Yves Girard is a *resource aware* formal logic system. Unlike propositional logic, it acknowledges the existence of a cost associated with using an assumption, stating that each assumption can be used exactly once. They may not be discarded, or duplicated. In propositional logic, it's trivial to say that given $A \to B$ and A, we can deduce both A and B. This is false under linear logic, since we *consumed* the value A to obtain B. We can have A *or* B, but never both.
When applied to the context of a type system in a programming language, we can see that, for instance these functions

```
copy :: a -> (a,a)
discard :: (a,b) -> a
```

do not admit a linear type. Just like objects in the real world, we cannot duplicate them, or arbitrarily discard them. In the context of a programming language, these are desirable properties for certain objects. We might want certain data structures not to be able to be thrown away, or to make sure a file handler has only a single reference. Linear Types allow us impose these restrictions. Furthermore, the guarantee that a value may not be duplicated has important implications when it comes to encouraging efficient implementation of programming logic. It would allow us to better reason about code that handles things like mutable arrays, or file IO, by imposing design constraints that not only avoid overhead but also ensure correctness.
While efficient resource management can naturally already be achieved with traditional type systems [4], linear types create a restriction that the compiler can check, therefore shifting that responsibility from the programmer to the compiler. This helps novice programmers to avoid common pitfalls, and reduces the difficulty of reasoning about more complicated systems that are often

too hard to program, such as computing directly with serialized data. This is described in full in the original GHC proposal paper [3].
The guarantee of no discarding is also very important due to the way it allows to optimize the use of garbage collecting. Linear values are explicitly allocated an deallocated in the code itself. This allows us to signal the garbage collector to release storage when we now a value will never be used again.
Once again, this could already be achieved through manual memory management, but linear types give us a high level way to gain greater control about how we want to manage memory. and only when we want to. This means we can choose to neglect memory management when performance impact is negligible, and take control only for code that has performance impact.

2.2 Linear Haskell

The GHC extension proposal is a conservative extension of Haskell. Rather than dividing values into linear values and non linear values, this extension uses linear type arrows to denote that a value is bound linearly to that type. This means that functions utilizing linear arrows can still function with non-linear Haskell code, since a linear function can still be fed non-linear arguments. This backwards-compatibility is crucial for the practicality of using linear types in any project.
To achieve this, a new function type is introduced, named the *linear arrow*.

$$A \multimap B$$

Which guarantees that the function's argument must be consumed *exactly once*. But what does consumption actually mean withing this context. As seen in [3], the definition of consumption is:

Definition 1 *(Value consumption)*

- *To consume a value of atomic base type (like Int or Ptr) exactly once, just evaluate it*
- *To consume a pair exactly once, pattern-match on it, and consume each component exactly*
- *In general, to consume a value of an algebraic datatype exactly once, pattern-match on it, and consume all its linear components exactly once.*

We can generalize the concept of the linear arrow to allow the same notation to represent both linear and non linear functions, by introducing multiplicity to function arrows. Multiplicities can have a value of one, meaning they can be consumed exactly once, or a value of ω meaning they can be used any number of times.

$$A \to_q B, \quad q ::= 1 \mid \omega$$

This generalization also highlights the possibility of introducing different levels of multiplicity. It might be interesting for instance, to introduce the multiplicity zero, or to introduce *affine* types (can be used *at most* once).

3 Type System

3.1 Linearly typed λ calculus

Here we define the syntax for linearly typed λ calculus. This will be the language we will use for the terms of the type system.

$$\begin{array}{rrll} p, q & ::= \omega \mid 1 & & \text{Multiplicities} \\ A, B & ::= A \to_q B \mid \mathsf{Boolean} & & \text{Types} \\ t, u & ::= x \mid t\ u \mid \lambda x :_q A.t & & \text{Terms} \\ \Gamma & ::= \varnothing \mid \Gamma, x :_q T & & \text{Contexts} \end{array}$$

3.2 Multiplicities and contexts

Some preliminary definitions from Linear Haskell [3]

Definition 2 (Multiplicity properties) *The addition and multiplication operators follow the following properties:*

- $+$ *and* $\cdot$ *are associative and commutative*
- *1 is the unit of* $\cdot$
- $\cdot$ *distributes over* $+$
- $\omega \cdot \omega = q$
- $1 + 1 = 1 + \omega = \omega + \omega = \omega$

Definition 3 (Context Scaling) *The multiplication property of the multiplicity can be distributed over a context to scale it by that multiplicity*

$$p(x :_q A, \Gamma) = x :_{p \cdot q} A,\ p\Gamma$$

Definition 4 (Context Addition) *Context addition is simply the merging of two context adding multiplicities to common bindings.*

$$\begin{array}{ll} (x :_p A, \Gamma) + (x :_q A, \Delta) & \stackrel{\text{def}}{=} x :_{p+q} A, \Gamma + \Delta \\ (x :_p A, \Gamma) + \Delta & \stackrel{\text{def}}{=} x :_p A, \Gamma + \Delta \quad (x \notin \Delta) \\ () + \Delta & \stackrel{\text{def}}{=} \Delta \end{array}$$

3.3 Typing Rules

Figure 1 presents the typing rules for Linear Haskell [3]. The typing judgments follow the standard form $\Gamma \vdash t : A$ where Γ is the variable context, t is a term and A is the type that is admitted for this term.

$$\frac{}{\omega\Gamma + x :_1 A \vdash x : A}\ \text{var}$$

$$\frac{\Gamma, x :_q A \vdash t : B}{\Gamma \vdash \lambda(x :_q A).\, t : A \to_q B}\ \text{abs}$$

$$\frac{\Gamma \vdash t : A \to_q B \qquad \Delta \vdash u : A}{\Gamma + q\Delta \vdash tu : B}\ \text{app}$$

Figure 1: Linear Haskell type checking rules

In the var rule, we scale Γ by ω, since any bindings that would be present in the context were not used, and must have their multiplicities adjusted in order to not violate linearity.
In the case of the app rule we need to make sure that if an argument was declared with multiplicity q then we required the context Δ is scaled accordingly.

3.4 Algorithmic typing rules

The rules deduction rules provided for Linear Haskell are sufficient to type any linear expression. They are also not deterministic, and therefore challenging to implement as a computer program. This difficulty comes from the application rule, where there is no way to know how to split context between the two expressions. To fix this, we can restructure the type system described in [3] so as to pass as input the most general admissible context, and output the actual used context. This output is then fed into the next subterm, as described in [4]. The revised type system is shown in figure 2.

$$\frac{\Gamma = x :_q A, \Gamma'}{\Gamma \vdash_{\mathsf{c}} x : A\ ;\ x :_1 A}\ \text{varTc}$$

$$\frac{\Gamma, x :_q A \vdash_{\mathsf{c}} t : B; \Gamma' \qquad x :_q A \text{ compatible } \Gamma'}{\Gamma \vdash_{\mathsf{c}} \lambda(x :_q A)t : A \to_q B\ ;\ \Gamma' - x}\ \text{absTc}$$

$$\frac{\Gamma \vdash_{\mathsf{c}} t : A \to_q B\ ; \Gamma' \qquad \Gamma \vdash u : B\ ;\ \Delta}{\Gamma \vdash_{\mathsf{c}} t\ u : B\ ;\ \Gamma' + q\Delta}\ \text{appTc}$$

Figure 2: Algorithmic type rules

We define a compatibility relation between a binding and a context and use this relation to ensure that each variable is used in accordance with the multiplicity it was initially declared with.

Definition 5 *The compatibility relation is defined as:*

$$x :_\pi A \text{ compatible } \Gamma, x :_\rho A, \Delta \iff \rho = \omega \ \vee\ \rho = \pi$$

We also define other auxiliary operations for binding removal, presented in the rules above by a subtraction operator, which simply removes the occurrence of a bind from a context.

Definition 6 (Bind removal)

$$\begin{aligned}(x :_\pi A, \Gamma) - x &\stackrel{\text{def}}{=} \Gamma \\ (y :_\pi A, \Gamma) &\stackrel{\text{def}}{=} y :_\pi A, (\Gamma - x) \quad (x \neq y) \\ () - x &\stackrel{\text{def}}{=} ()\end{aligned}$$

3.5 Correctness

Soundness of the Linear Haskell was proven in the original paper [3]. To prove the correctness of the revised typing rules, we prove equivalence between the Linear Haskell typing rules and the Algorithmic Typing Rules.

$$\Gamma \vdash t : A \iff \Gamma \vdash_{\mathsf{c}} t : A\ ;\ \Delta,\ \wedge\ \Delta \subseteq \Gamma$$

Proof: Follows by induction on the length of the type derivation.

4 Implementation

We will now implement the algorithmic type checking rules into a an actual type checker, written in Haskell. The code can has been made available in full on github at https://github.com/Luis-Brandao/-Haskell-Linear-Types-Typechecker.

4.1 Preliminary Definitions

We begin by declaring the Haskell type that will define the types for the linear lambda calculus, as well as the multiplicities.

```
data Type = TypeArrow Multiplicity Type Type
| TBoolean deriving (Eq,Show)
data Multiplicity = One | Omega deriving Eq
```

We also define *aliases* to represent a context as a list of binds, and a bind as a tuple of a string, a multiplicity and a type.

```
type Bind = (Name, Multiplicity, Type)
type Context = [Bind]
```

We define Multiplicity as an instance of *Num*, and specify it's instances of addition, multiplication and equality.

```
data Multiplicity = One | Omega deriving Eq

instance Num Multiplicity where
  _ + b = Omega
  One * b = b
  b * One = b
  _ * _ = Omega
  fromInteger 1 = One
  fromInteger _ = Omega
```

The following auxiliary functions implement the operations we will require for the *check* function. Their implementations are very straight-foward. For brevity, we present only the type signatures.

```
scaleCtx :: Multiplicity -> Context -> Context
addCtx :: Context -> Context -> Context
look :: String -> Context -> Maybe Bind
deleteBind :: String -> Context -> Context
```

The *look* function is an auxiliary function that simply searches for a bind in a context that matches a given variable name. If found, the function return that bind, otherwise it return nothing.

4.2 Type Checker

Type checkers are, by nature, expected to fail. To capture this, we make use of a type checking monad [4] *Tc*. In cases where the type checker fails due to an *ill-typed* term, we return an error message that details what caused failure.

```
type Tc a = Either String a
```

We use a top level function *checkTop* that calls the check function with an empty context. The check function will recursively go through the term, returning the context and type of each sub term.

The main check function can be seen in figure 3. Each pattern corresponds to a typing rule as described in section 3.4.

The first pattern $ctxIn\ (Var\ x)$ corresponds to the variable typing rule, and is the *base-case* of recursion. To find the type and context we simply have to *look* in the input context for a bind that matches that variable's name. If the variable was declared before being used, we will find it's bind in the context and return that same type and a context that contains this variable's bind, with the type we found in the context and a multiplicity of one. Otherwise, this variable wasn't declared, and the type checker fails.

The second pattern $ctxIn\ (Lambda\ (x, q, a)\ t)$ corresponds to the abstraction rule. In order to type these terms we need to call the checker on the subterm t to obtain it's type and context. We then check if the bind of the variable we are abstracting is compatible with the context we obtained. If it is, then we can return that context subtracting the variable from it, and the type $TypeArrow\ q\ a\ b$.

The third pattern $ctxIn\ (App\ t\ u)$ corresponds to the application rule. Here, we recursively call the *check* function on each subterm, to obtain their type and context. We then apply context scaling and addition as described in section 3.3 to obtain the resulting context, and return that context and the type of term u.

4.3 Examples

In figure 4, we create a few examples to test the checking function. *identity, identity'* and *app* are all *well-typed* terms, since all variables are being used according to the type they were declared with. However, *discard* and *duplication* are *ill-typed*, because they declare variables as being linear, but try to duplicate or discard them, violating linearity.

In figure 5 we call use the GHCI to call the check function using these terms, and verify that the type checking algorithm is working according to expectations.

```
check :: Context -> Term -> Tc (Context, Type)

check ctxIn (Var x) = case (look x ctxIn) of
                      Just (_,_,ty) -> return ([(x,1,ty)],ty)
                      Nothing -> throw "Non␣defined␣var"

check ctxIn (Lambda (x,q,a) t ) = do
  (ctxOut,b) <- check ((x,q,a):ctxIn) t;
  let result = (TypeArrow q a b) in
  case look x ctxOut of
   Just (x',q',t') -> if (x',q',t') `compatible` ctxOut then
    return (( x `deleteBind` ctxOut),result) else throw "type␣error"
   Nothing -> if(q == Omega) then
    return (ctxOut,result) else throw (x ++ "␣was␣discarded")

check ctxIn (App t u) = do
  (_,(TypeArrow q a b)) <- check ctxIn t;
  (gamma,type2) <- check ctxIn u;
  let result = ctxIn `addCtx` (q `scaleCtx` gamma) in
    if(a == type2) then return (result, b) else throw "type␣error"

check _ _ = throw "pattern␣matching␣exhausted"
```

Figure 3: Main Haskell implementation

```
identity = (Lambda ("x",1,TBoolean) (Var "x"))

identity' = (Lambda ("x",Omega,TBoolean) (Var "x"))

app = (Lambda("f",Omega, (TypeArrow Omega TBoolean TBoolean)))
(Lambda("y",Omega, TBoolean)(App (Var "f") (Var "y")))

discard = Lambda("y",Omega, TBoolean)
(Lambda ("x",1,TBoolean) (Var "y"))

duplication = (Lambda("f",Omega,
(TypeArrow Omega TBoolean TBoolean))
(App (Var "f") (Var "f")))
```

Figure 4: Example expressions

```
*Main> checkTop identity
Right ([],TypeArrow One TBoolean TBoolean)

*Main> checkTop identity'
Right ([],TypeArrow Omega TBoolean TBoolean)

*Main> checkTop app
Right ([],TypeArrow Omega (TypeArrow Omega TBoolean TBoolean)(TypeArrow Omega TBoolean TBoolean))

*Main> checkTop discard
Left "x was discarded"

*Main> checkTop duplication
Left "type error"
```

Figure 5: Testing the examples in GHCI

5 Conclusion

This report presented a study on the implementations of linear types. We went over the theory behind linear type systems, and showed off how to extend a language's type system to support them. The type checker we developed in this work was designed for a very simple language without polymorphic types. Despite this, the process we described can be used to write a complete type checker for any full programming language.
The nature of this work is educational. The goal was to create a stepping ladder, something that could be used as a convergence point to all the literature that exists on linear logic, linear types and general type theory to anyone who wishes to learn more about linear types.

References

[1] Philip Wadler *Linear types can change the world!* , Programming Concepts and Methods, Sea of Galilee, Israel, April 1990. North Holland, Amsterdam, 1990.

[2] Philip Wadler *A taste of linear logic* Invited talk, Mathematical Foundations of Computing Science, Springer Verlag LNCS 711, Gdansk, August 1993.

[3] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, Arnaud Spiwack. *Linear Haskell: practical linearity in a higher-order polymorphic language.* Proceedings of the ACM on Programming Languages, ACM, 2017, 2 (POPL), pp.1-29. ¡10.1145/3158093¿. ¡hal-01673536¿

[4] Benjamin C. Pierce*Advanced Topics in Types and Programming Languages* Prentice-Hall 2005

[5] Philip Wadler *Advanced Functional Programming* Proceedings of the Båstad Spring School, May 1995, Springer Verlag Lecture Notes in Computer Science 925.

[6] J.-Y. Girard, Linear logic *Theoretical Computer Science, 50:1–102, 1987.*