Linear Type Systems

What Are They, and How Are They Used in Programming Languages

Olivier Lischer
OST Eastern Switzerland University of Applied Sciences
MSE Seminar "Programming Languages"
Supervisor: Farhad Mehta
Spring 2025

Abstract

In every programming language, resource management, including memory management, is an essential part of the language. In C/C++, resource management is done manually and therefore the responsibility lies with the developer. This gives you the most control, but it can also lead to errors such as user-after-free and double-free. In contrast, many so-called high-level programming languages use an automatic system with garbage collectors to clean up unused memory. With Rust, there is a system programming language that offers many advantages of manual resource management without sacrificing resource safety. The goal of this paper is to present a type system for a functional programming language that enables a runtime system to perform resource management without a garbage collector. To study the type system for such a functional programming language, the simply typed lambda calculus is used and then extended by a linear type system. This linear type system already enables a certain resource safety. By adding ownership typing, the type system enables a garbage collector free runtime system. A simply typed lambda calculus with a linear type system was presented, which also features a borrowing type system known from Rust. With the type system presented, it is possible to implement a functional programming language that does not require a garbage collector and shifts many errors in resource management to compile time.

Keywords: Linear Types, Type Systems, Memory Management

1 Introduction into Memory Management

Every application must manage memory in some way. In many imperative programming languages like Java, C# and Python, as well in many functional programming languages, memory management is abstracted away. Many of them achieved this by using a garbage collector which removes memory that is no longer required by the application. However, a garbage collector is no-deterministic and may interrupt the normal flow of the application. This is not acceptable to all applications. The alternative is manual memory management as is done in C/C++. While manual memory management gives the programmer full control over when and how memory is allocated or freed, it is also a very difficult task to implement correctly. Many bugs are caused by memory related errors like *use-after-free* or *double-free* that are caused by doing manual memory management incorrectly. With Rust there is a programming language that uses automatic memory management with no runtime overhead. This is achieved with the help of the type system and by carefully tracking the usage of variables. The theoretical foundation for this system builds upon a linear type system, that again is backed by linear logic.

As Rust demonstrates, it is possible to get safety by automatic memory management and at the same time no runtime overhead. However, many functional programming languages have some type of garbage collector in its runtime system. The question arise: is it possible to design a functional programming language that no longer requires a garbage collector? By proposing a lambda calculus with a linear type system, it should be demonstrated, that it is possible to also implement functional programming languages without a garbage collector for automatic memory management.

This paper is structured as follows: first there is a theoretical introduction into and linear types. Next, motivating examples are given as to why linear types can also be helpful in programming languages that also have a garbage collector.

Afterwards, existing linear type systems are presented. Rust is used as an example for an imperative language, while Haskell for the functional programming language example. Followed by a proposal how a lambda calculus can be extended, so that the memory management could be done without a garbage collector.

2 Introduction into Linear Type Systems

This section introduces the concept of linear type systems. First, a simply typed lambda calculus without a linear type system is established. From there, the existing lambda calculus is extended with linear types. Linear logic, as introduced by Jean-Yves Girard[Gir87], forms the basis for linear type systems. However, the discussion of linear logic is beyond the scope of this paper.

2.1 The simply typed lambda calculus

First the syntax for a simply typed lambda calculus (STLC) with sum and product types is established. A *type* is a type variable, function, product or a sum. T, U, V are types, while X, Y, Z are type variables.

$$T, U, V := X \mid (U \to V) \mid (U \times V) \mid (U + V)$$

A *term* is variable, introduction or elimination of a function, product or sum type. t, u, v are terms and x, y, z are individual variables.

$$
\begin{aligned}
&t, u, v := \\
&\quad \mid x : T \\
&\quad \mid (\lambda x : T.\, v) \mid (t\ u) \\
&\quad \mid (u, v) : (U \times V) \mid (\text{case } t : (U \times V) \text{ of } \{(x, y) \to w\}) \\
&\quad \mid (inl\ u) : (U + V) \mid (inr\ v) : (U + V) \\
&\quad \mid (\text{case } t : (U + V) \text{ of } \{inl\ x \to u; inr\ y \to v\})
\end{aligned}
$$

With this in place, the syntax for typing rules can be established. An *assumption list* is a list of pairs of variables to types. Γ and Δ are assumption lists and $n \geq 0$.

$$\Gamma, \Delta := x_1 : T_1,\ \ldots,\ x_n : T_n$$

A *typing judgment* has the form

$$\Gamma \vdash t : T$$

and can be read as "Under the assumption Γ the term t has type T".

Starting with the structural rules. The simplest rule is *Id*, which creates a tautology. It states that given that term x has type U, it is true that x has type U.

$$\text{Id}\ \frac{}{x : U \vdash x : U}$$

The next rule is *Exchange*. It states that the order of the variables in an assumption list is not relevant.

$$\text{Exchange}\ \frac{\Gamma, x : U, y : V, B \vdash w : W}{\Gamma, y : V, x : U, B \vdash w : W}$$

By adding more elements to the context, the type of term does not change. An alternative interpretation is that it is allowed to discard variables, when they are not used. This is expressed by *Weakening*.

$$\text{Weakening}\ \frac{\Gamma \vdash v : V}{\Gamma, x : U \vdash v : V}$$

That a variable can be used multiple times in a term is expressed by *Contraction*. A term v is build using the two variables x and y of type U. By replacing x and y with z, the type of v does not change.

$$\text{Contraction}\ \frac{\Gamma, x : U, y : U \vdash v : V}{\Gamma, z : U \vdash v[x := z; y := z] : V}$$

The remaining 7 rules are *logical* rules and always come in an introduction and elimination form. Starting with the rules for the function type the introduction rule $\to$-I derives the type for a lambda abstraction respectively functions.

$$\to\text{-I}\ \frac{\Gamma, x : U \vdash v : V}{\Gamma \vdash (\lambda x.v) : (U \to V)}$$

The elimination rule for function types is the application $\to$-E. By providing the required input, the lambda abstraction resolves to a final type.

$$\to\text{-E}\ \frac{\Gamma_1 \vdash t : (U \to V) \qquad \Gamma_2 \vdash u : U}{\Gamma_1, \Gamma_2 \vdash (t\ u) : V}$$

The second group of logical rules targets product types. First the introduction rule $\times$-I, that states how to build a product type from two single values.

$$\times\text{-I}\ \frac{\Gamma \vdash u : U \qquad \Delta \vdash v : V}{\Gamma, \Delta \vdash (u, v) : (U \times V)}$$

The elimination rule $\times$-E stats that it is possible to extract the values from the product type and use them for a new value.

$$\times\text{-E}\ \frac{\Gamma \vdash t : (U \times V) \qquad \Delta, x : U, y : V \vdash w : W}{\Gamma, \Delta \vdash (\text{case } t \text{ of } \{(x, y) \to w\}) : W}$$

The last rule set targets the sum types and consists of two introduction rule, one for each variant, and one elimination rule. The introduction rules are used to create a sum type with the left or the right variant using *inl* respectively *inr*.

$$+\text{-I}\ \frac{\Gamma \vdash u : U}{\Gamma \vdash (inl\ u) : (U + V)} \qquad \frac{\Gamma \vdash v : V}{\Gamma \vdash (inr\ v) : (U + V)}$$

The last rule states that regardless of the inner value of the sum type, the result of the extraction must be always the same.

$$+\text{-E}\ \frac{\Gamma \vdash t : (U + V) \qquad \Delta, x : U \vdash u : W \qquad \Delta, y : V \vdash v : W}{\Gamma, \Delta, \vdash (\text{case } t \text{ of } \{inl\ x \to u; inr\ y \to v\}) : W}$$

All rules for a STLC are summarized in Figure 1.

2.2 Linear Type Systems

After the STLC is established, the type system can be extended to be linear. To build a linear type system, four operations from linear logic are needed: $\multimap$, $\otimes$, $\oplus$ and !. In the following, the new linear operators are introduced before modifying the rules for the STLC.

2.2.1 Linear logic operations. For a non-linear function $f : U \to V$, the argument of type U can be used any number of times to construct a V. In addition, the resulting value of type V can be used without restriction. To create a linear function $\multimap$ is used. In a linear function $f' : U \multimap V$, the argument, of type U can only be used once and the resulting value of type V must be used exactly once as well. Therefore, $\multimap$ is the linear counterpart of $\to$. To develop an intuition, think about a vending machine. Let

$$
\begin{aligned}
C &:= \text{One Coin} \\
W &:= \text{Water} \\
S &:= \text{Soda}
\end{aligned}
$$

and some axioms for a vending machine:

$$
\begin{aligned}
C &\implies W \\
C &\implies S
\end{aligned}
$$

The linear function $water : C \multimap W$ shows that the coin can only be used once to buy water and is then no longer available. In contrast, the non-linear function $water' : C \to W$ shows that the coin can also be used to buy a soda after the vending machine has dispensed the water.

$$\text{Id}\ \frac{}{x : U \vdash x : U} \qquad \text{Exchange}\ \frac{\Gamma, x : U, y : V, B \vdash w : W}{\Gamma, y : V, x : U, B \vdash w : W}$$

$$\text{Weakening}\ \frac{\Gamma \vdash v : V}{\Gamma, x : U \vdash v : V} \qquad \text{Contraction}\ \frac{\Gamma, x : U, y : U \vdash v : V}{\Gamma, z : U \vdash v[x := z; y := z] : V}$$

$$\to\text{-I}\ \frac{\Gamma, x : U \vdash v : V}{\Gamma \vdash (\lambda x.v) : (U \to V)} \qquad \to\text{-E}\ \frac{\Gamma \vdash t : (U \to V) \qquad \Delta \vdash u : U}{\Gamma, \Delta \vdash (t\ u) : V}$$

$$\times\text{-I}\ \frac{\Gamma \vdash u : U \qquad \Delta \vdash v : V}{\Gamma, \Delta \vdash (u, v) : (U \times V)} \qquad \times\text{-E}\ \frac{\Gamma \vdash t : (U \times V) \qquad \Delta, x : U, y : V \vdash w : W}{\Gamma, \Delta \vdash (\text{case } t \text{ of } \{(x, y) \to w\}) : W}$$

$$+\text{-I}\ \frac{\Gamma \vdash u : U}{\Gamma \vdash (inl\ u) : (U + V)} \qquad \frac{\Gamma \vdash v : V}{\Gamma \vdash (inr\ v) : (U + V)}$$

$$+\text{-E}\ \frac{\Gamma \vdash t : (U + V) \qquad \Delta, x : U \vdash u : W \qquad \Delta, y : V \vdash v : W}{\Gamma, \Delta, \vdash (\text{case } t \text{ of } \{inl\ x \to u; inr\ y \to v\}) : W}$$

Figure 1. Rules for the simply typed lambda calculus

The $\otimes$ is the linear product type. For a product $(u, v) : (U \otimes V)$, both u and v can be used, but only once each. The function $waterSoda : C \multimap C \multimap (W \otimes S)$ can be constructed using the vending machine example. It says that one can buy a water and a soda with two coins.

The linear sum type is formed with $\oplus$. Let $t : (U \oplus V)$ and similar to non-linear sum types, a case distinction is necessary to proceed further. The intrinsic value of t can only be used once. In the case of a broken vending machine, there is the function $waterOrSoda : C \multimap (W \oplus S)$. When a coin is inserted, the vending machine randomly returns either a water or a soda, but never both at the same time.

The last operation is called "of course" and is represented by !. The ! establishes the connection between a non-linear type and a linear type by allowing it to be used zero, once or several times. If $u : (!U)$, then u can be used like a non-linear type without restrictions. In the vending machine example, a function $waterSodaOfCourse : (!C) \multimap (W \otimes S)$ can be found that tells us that one has as many coins as necessary to buy a water and a soda.

2.2.2 Extending the Simply Typed Lambda Calculus.

First the function rules can be modified, simply by replacing $\to$ with $\multimap$ signalling, that the argument must only be used once. The same can be done for the product and sum types by replacing + with $\oplus$ and $\times$ with $\otimes$. If the rules for *weakening* and *contraction* are removed, a linear type system is constructed. However, with this type system no useful software can be written: a functions like `double` cannot be written as the input argument is used twice[Wad91].:

$$(\lambda x.(x, x)) : U \multimap (U \otimes U)$$

Therefore, the rules *Weakening* and *Contraction* must be modified:

$$\text{Weak}\ \frac{\Gamma \vdash v : V}{\Gamma, x : (!U) \vdash v : V}$$

$$\text{Cont}\ \frac{\Gamma, x : (!U), y : (!U) \vdash v : V}{\Gamma, z : (!U) \vdash v[x := z; y := z] : V}$$

The ! gives the possibility to use an input zero, once or multiple times. Therefore, the type of the `double` function in a linear type system would be:

$$(\lambda x.(x, x)) : (!U) \multimap (U \otimes U)$$

With all the rules in place, the STLC is extended with a working linear type system.

Since the lambda calculus is pure and resources such as memory or files do not exist, linear types are not as useful. However, since the lambda calculus is the theoretical basis for all functional programming languages, it still makes sense to introduce linear types in order to build on them for more sophisticated programming languages.

3 Motivation for Linear Type Systems

As already written in section 1, memory management is a difficult task that can lead to various safety-critical problems. This section describes two common errors in memory management and how linear types can solve these problems.

$$\text{Id}\ \frac{}{x : U \vdash x : U} \qquad \text{Exchange}\ \frac{\Gamma, x : U, y : V, B \vdash w : W}{\Gamma, y : V, x : U, B \vdash w : W}$$

$$\text{Weakening}\ \frac{\Gamma \vdash v : V}{\Gamma, x : (!U) \vdash v : V} \qquad \text{Contraction}\ \frac{\Gamma, x : (!U), y : (!U) \vdash v : V}{\Gamma, z : (!U) \vdash v[x := z; y := z] : V}$$

$$\multimap\text{-I}\ \frac{\Gamma, x : U \vdash v : V}{\Gamma \vdash (\lambda x.v) : (U \multimap V)} \qquad \multimap\text{-E}\ \frac{\Gamma_1 \vdash t : (U \multimap V) \qquad \Gamma_2 \vdash u : U}{\Gamma_1, \Gamma_2 \vdash (t\ u) : V}$$

$$\otimes\text{-I}\ \frac{\Gamma_1 \vdash u : U \qquad \Gamma_2 \vdash v : V}{\Gamma_1, \Gamma_2 \vdash (u, v) : (U \otimes V)} \qquad \otimes\text{-E}\ \frac{\Gamma_1 \vdash t : (U \otimes V) \qquad \Gamma_2, x : U, y : V \vdash w : W}{\Gamma_1, \Gamma_2 \vdash (\text{case } t \text{ of } \{(x, y) \to w\}) : W}$$

$$\oplus\text{-I}\ \frac{\Gamma \vdash u : U}{\Gamma \vdash (inl\ u) : (U \oplus V)} \qquad \frac{\Gamma \vdash v : V}{\Gamma \vdash (inr\ v) : (U \oplus V)}$$

$$\oplus\text{-E}\ \frac{\Gamma_1 \vdash t : (U \oplus V) \qquad \Gamma_2, x : U \vdash u : W \qquad \Gamma_2, y : V \vdash v : W}{\Gamma_1, \Gamma_2, \vdash (\text{case } t \text{ of } \{inl\ x \to u; inr\ y \to v\}) : W}$$

Figure 2. Rules for a linear type system

Two of the most common errors in memory management are *use-after-free* and *double-free*. When a resource is allocated, e.g. memory, one can use it freely. After the resource is no longer needed, it should be free. A use-after-free error can occur after the resource has been released by using a pointer or reference to the resource even though the resource is already released. What happens now depends on the type of resource, the runtime system and many other things. The situation is similar with double-free error. After a resource has been released without an error, the application can attempt to release the resource again. Once again, the outcome is not entirely predictable. Both result from the difficulty of determining what part in the application is responsible for releasing the assigned resource and invalidating all remaining pointers and references to the resource.

Even in a purely functional programming language like Haskell, linear types have advantages. The Linear Haskell paper[BBN+18] lists two reasons why linear types are useful in Haskell:

- *In-place updates*: In the case of mutable arrays, ensuring that the value can be updated in place, allows for a more efficient implementation of $O(n)$ to $O(1)$.
- *Ensuring correct protocol usage*: In the file IO example, it is enforced that all IO operations only take place on valid file descriptors.

A concrete example of where *Ensuring correct protocol usage* could be violated is reading the first line of a given file and printing it to the standard output. This task is solved Listing 1 with Haskell. The function `printFirstLine` implements the functionality without bugs. The second function `printFirstLineError` uses the same function calls, but closes the file handle before reading the file, which violates the protocol and leads to an error at runtime. In a linear type system, the function `hClose` could *consume* the file handle. By consuming the file handle, the rest of the application is no longer allowed to use this file handle, thereby preventing a *use-after-free* error at compile time.

```
1  printFirstLine :: FilePath -> IO ()
2  printFirstLine fpath = do
3    fileHandle <- openFile fpath ReadMode
4    firstLine <- hGetLine fileHandle
5    putStrLn firstLine
6    hClose fileHandle
7
8  printFirstLineError :: FilePath -> IO ()
9  printFirstLineError fpath = do
10   fileHandle <- openFile fpath ReadMode
11   hClose fileHandle
12   firstLine <- hGetLine fileHandle
13   putStrLn firstLine
```

Listing 1. File IO Motivation

The following section 4 presents two programming languages that offer a linear type system. It is shown that errors such as those in Listing 1 can be prevented with the help of linear types.

4 Existing Linear Type System Implementations

This section describes two different programming languages that already support a linear type system. In subsection 4.1 the functional programming language Haskell is described how it utilizes linear types. In the section that follows after, the imperative programming language Rust is described. Rust is positioned as a systems programming language with a safe memory management thanks to its type system. A novelty of Rust is its ownership model to enable automatic memory management without any runtime overhead.

4.1 Haskell

Linear types are not part of standard Haskell, but an experimental language extension in GHC, first proposed in Linear Haskell[BBN+18]. After the paper, a GHC proposal was created in which the work is now being pursued[Ghc].

Because Haskell is a purely functional programming language the ideas from subsection 2.2 can be used without any major changes. To allow linear functions, the Haskell syntax must be extended. This was done by adding a new kind of function type as described in Listing 2.

```
1  data Multiplicity = One | Many
2  type a %1 -> b = a %One  -> b
3  type a    -> b = a %Many -> b
```

Listing 2. Function type definition

A function of type `a \%1 -> b` is a linear function and must uphold the criteria: "when its result is consumed exactly once, then its argument is consumed exactly once". More informally written: In every branch, the argument of the function must be used exactly once. This corresponds to the rules $\multimap$-I and $\multimap$-E from Figure 2 and would be written as $a \multimap b$ with the syntax from subsection 2.2. The normal function type `a -> b` can be still used and is called an unrestricted function, as its argument and results can be used without any restrictions. To allow uniform handling of linear and unrestricted functions the multiplicity-polymorphic arrow `a \%m -> b` was introduced.

Linear and multiplicity-polymorphic arrows are never inferred and must always be declared. If the type of a function is not declared, the type of the unrestricted function `a -> b` will be inferred. However, the multiplicity of a variable can be inferred from the context.

```
1  f :: A %1 -> B
2  g :: B %1 -> C
3
4  h :: A %1 -> C
5  h x = g y
6    where
7      y = f x
8
9  hAnnoted :: A %1 -> C
10 hAnnoted x = g y
11   where
12     %1 y = f x
```

Listing 3. Linear function examples

The multiplicity of bindings is inferred by the compiler as follows:

- Top level bindings having multiplicity "Many"
- Recursive bindings having multiplicity "Many"
- Lazy non-variable pattern bindings having multiplicity Many
- In all other cases, the multiplicity is inferred from the term

So to make a non-variable pattern binding with multiplicity 1, the pattern must be strict, like `let !(x, y) = rhs`, whereas the `let (x, y) = rhs` has always multiplicity Many. Caution is advised, because the "!" in `let !(x, y) = rhs` is not the same as in linear logic! While in linear logic the "!" allows the variable to be used arbitrarily, in Haskell it indicates a strict evaluation and therefore infers multiplicity 1.

4.1.1 From linear logic to Haskell.

Next the connection between the operations in Haskell and the SLTC in subsection 2.2 is established. Haskell's unrestricted function types correspond to the function type $U \to V$ and are covered by the $\to$-I and $\to$-E rules, while the linear function type $U \multimap V$ is represented by the rules $\multimap$-I and $\multimap$-E. In Haskell, all fields in an algebraic data type are linear by default and are represented by the rules $\otimes$-I and $\otimes$-E for product types, respectively $\oplus$-I and $\oplus$-E for sum types. Using the generalized algebraic data type (GADT) or the record type syntax, the multiplicity can be customized. However, the syntax has no effect on record selectors. For examples of using data types with linear types, see Listing 4.

```
1  data T1 a = MkT1 a
2
3  construct :: a %1 -> T1 a
4  construct x = MkT1 x
5
6  deconstruct :: T1 a %1 -> a
7  deconstruct (MkT1 x) = x
8
9  data T2 a b c where
10     MkT2 :: a -> b %1 -> c %1 -> T2 a b c
11
12 data T3 a b c = MkT3 { x %'Many :: a, y :: b, z :: c
       }
13
14 x :: T3 a b c -> a
15 y :: T3 a b c -> b
```

Listing 4. Data types with Linear types in Haskell

4.1.2 Examples in Haskell.

In Listing 1 a file IO function was written using non-linear Haskell. In Listing 5 the same functionality is implemented again but with linear types enable and using the linear-base library. The `openFile` function creates the file handle. The `getLine` function takes the file handle as a linear type, therefore must be used exactly once. To enable further readings with the same file handle,

the function must return the file handle as a return value. Finally, the `hClose` consumes the file handle and returns unit.

```
linearGetFirstLine :: FilePath -> RIO (Ur Text)
linearGetFirstLine fp = Control.do
  handle <- Linear.openFile fp System.ReadMode
  (t, handle') <- Linear.hGetLine handle
  Linear.hClose handle'
  Control.return t

linearPrintFirstLine :: FilePath -> System.IO ()
linearPrintFirstLine fp = do
  text <- Linear.run (linearGetFirstLine fp)
  System.putStrLn (unpack text)

-- from linear-base
openFile :: FilePath -> IOMode -> RIO Handle
hGetLine :: Handle %1 -> RIO (Ur Text, Handle)
hClose :: Handle %1 -> RIO ()
```

Listing 5. Linear File IO in Haskell

The usage of `handle` after `hGetLine` or `handle'` after `hClose` results in a compiler error and therefore preventing possible use-after-free errors.

4.2 Rust

The Rust compiler tracks where a variable goes out of scope. As soon as the variable goes out of scope, the compiler knows the memory bound to the variable can be freed, as no one uses the memory any more. This tracking is the ownership model and is based on linear types theory.

There are four kinds of how to use a memory location. One of them is using pointers and is not considered here, as it is up to the developer to ensure that no guarantees are violated. A variable can have ownership of the location and has therefore the permission to read it, modify it, borrow out the location to someone else and give the ownership rights away. Second is a shared reference to the location. The holder of a shared reference can read from the memory location, but is not allowed to modify it. As the name implies, there can be many shared references to a single memory location. The last option is a mutable reference. With a mutable reference, the memory location can be read and also written. There can be only one mutable reference to a memory location at a given time. In Table 1 a summary of the different memory access type is listed.

```
fn ownership(a: A) -> B { /**/ }
fn shared_reference(a: &A) -> B { /**/ }
fn mutable_reference(a: &mut A) -> B { /**/ }

fn main() {
    let a = A::new();
    shared_reference(&a);
    mutable_reference(&mut a);
    ownership(a);
}
```

Listing 6. Memory access types examples

4.2.1 An intuition for the ownership model.

A child's toy can be used to build an intuition for the ownership model. The owner of the toy is free to use the toy as he wants. This corresponds to read and write access. The owner may want to lend out the toy to some other child, resulting in a *mutable reference*. The second child can once again play with the toy as its pleased, but must guarantee that the toy remain intact. While the toy is borrowed, the owner can no longer access it, as it is no longer in its possession. When the toy is returned, the owner could decide to make an exhibition for its toy. During the exhibition, the toy can be viewed by many other children, but no one can play with it, but only look at it. So every child has a *shared reference*. As the child grows older it decides that another child should become the new owner (*move*). The grown up child now has no rights any more to the toy.

4.2.2 From linear logic to Rust.

Because Rust is an imperative programming language it is required to perform additional steps so that the lambda calculus theory can be applied.

The linear function $f : U \multimap V$ from the lambda calculus can be mapped to `fn f(u: U): V` in Rust. While the value `u` can be used multiple times in the function `f`, from the outside, `u` is used once and can no longer be used after `f`. To duplicate a value, the `Clone` trait exists. When a type implements the `Clone` trait, the `clone` function can be called that allows for duplication of the current value. The `Copy` trait is an extension of the `Clone` trait by that it automatically calls `clone` when required.

Rust has sum and product types in form of `enum` respectively `struct`, therefore the rules +-I, +-E and ×-I and ×-E can be used. However, there is no mechanism to restricted their usage. So Rust has no way to enforce the rules ⊕-I, ⊕-E and ⊗-I and ⊗-E.

The different memory location access types are required, to overcome the shortcomings of having stateful computation. Rust's type system is not a pure adaptation of linear types, but rather an extension and is sometimes referred to as borrowing typing[BBN+18]. This borrowing typing is similar to the use-types introduced in the paper "Is there a use for linear logic?"[Wad91]

4.2.3 Examples in Rust.

Once again the example with file IO. The `BufReader` is a type that provides various function to read from a source like a file. The `BufReader::new` function takes ownership of the file, so `file` can no longer be used after the creation of the reader. The `BufReader::read_line` function takes a mutable reference to itself indicating that only one user must be there at this moment. Calling `drop` is not required in this function, as `reader` would be cleaned up automatically at the end of the function. Trying to read a second line after calling drop would result in a compiler error, as `reader` is consumed by `drop` and therefore preventing a use-after-free error.

Syntax	Name	Description	Occurrence
a	ownership	Read, Write, Move	exactly once
&a	shared reference	Read,	many times
&mut a	mutable reference	Read, Write	exactly once

Table 1. Memory Location access types

```
fn print_first_line(path: &Path) {
    let file = File::open(path).unwrap();
    let mut reader = BufReader::new(file);
    let mut line = String::new();
    reader.read_line(&mut line).unwrap();
    drop(reader); // Not required
    println!("{line}")
}

// BufReader::new(inner: R) -> BufReader<R>
// BufReader::read_line(&mut self, buf: &mut String)
      -> Result<usize>
// drop<T>(_x: T) -> ()
```

Listing 7. Linear File IO in Rust

5 Ingredients for a Functional Programming Language without Garbage Collection

Memory and resources play no role in the Lambda Calculus. Therefore, the x can be used freely in the following function:

$$(\lambda x.\ (f\ x,\ g\ x))$$

It is irrelevant whether f or g is applied first. It is also not important how often x is used to calculate the results. However, modern computers do not conform to this model. Depending on what x, f and g are and do, it may be relevant in which order and how many times they are used and evaluated. To address this problem, this section introduces the idea of a functional programming language based on the Lambda Calculus that uses a linear type system to eliminate the need for a garbage collector.

5.1 Linear Types combined Ownership typing

Using the example of Rust, a programming language with automatic memory management that does not have a garbage collector, it is known that one possibility is to track the usage of variables. Rust does this by checking whether a memory location has an owner. If a memory location no longer has an owner, it can be safely freed. In Rust, these checks are carried out by the so-called "Borrow Checker". The following describes an idea that relies solely on linear types and clever cloning under the hood.

The general idea is that data is moved to the function to be called by default. In the function $(\lambda x.\ fx)$, the argument x is moved directly into the function f. There are no allocation or deallocations required. In the case of a function of type $(!T) \multimap (U, V)$, where the argument can be used zero or many times, the original argument is deallocated when the function exited. If the argument is used multiple times, it should be cloned so that new individual copies of the value exists. An example of such a function could be the following:

$$(\lambda x.\ (f\ x,\ g\ x)) \ :\ (!T) \multimap (U, V)$$

Another type of functions is of type $(!T) \multimap (!U)$, where the input is unrestricted and the output is also unrestricted. The input can be handled as in the other cases before, and the output does require no special handling, as the output only indicates where the value may be used as an input argument.

As seen in this section, it is only necessary to track the lifetime of each variable to enable memory management similar to Rust. However, this is currently only an idea without a proof.

6 Conclusion and other work

The goal was to give an introduction to linear type systems and present how the lambda calculus can be extended to use a linear typed system. Further, programming languages should be studied to present how they use linear types to ensure that a selection of resource related bugs can be caught at compile time. Last, ingredients for a functional programming language that does not require a garbage collector were presented.

In section 2 the simply typed lambda calculus was introduced. After the basic operations of linear logic were presented, a linear type system for the simply typed lambda calculus was developed.

The connection between the theory and existing programming languages was then established in section 4 using Haskell and Rust. Further, it was presented, how a specific use-after-free bug can be caught at compile time. Once in Haskell and once in Rust.

In section 5 the idea of a pure language was presented that combines a linear type system with tracking the lifetime of variables. With this idea, it should be possible to replace a garbage collector with routines inserted by the compiler inserted free resources that are no longer used.

This paper provided an introduction to linear logic and liner types. However, certain operations from linear logic have been omitted, as they are not crucial for understanding linear type systems. For a more detailed consideration, the work of Jean-Yves Girard should be taken into account. Only a brief idea was presented in the last part. So far, this has not

been proven theoretically, nor is there any proof in the form of a prototype.

The best way to prove that the concept of a functional programming language works without a garbage collector would be to implement one.

There are other type system that may solve this problem as well. One such a type system is the unique type system used by the Clean programming language. It would be an interesting question whether, and how, the Clean runtime system could be modified to eliminate the garbage collector.

References

[BBN+18] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. Linear Haskell: Practical linearity in a higher-order polymorphic language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–29, January 2018.

[Ghc] Ghc-proposals/proposals/0111-linear-types.rst at master · ghc-proposals/ghc-proposals. https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0111-linear-types.rst.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, January 1987.

[Wad91] Philip Wadler. Is there a use for linear logic? *SIGPLAN Not.*, 26(9):255–273, May 1991.