A Python Implementation of the Linear Lambda Calculus

Vedaant Shah, Peter Feng
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
`{vedaant.shah, pfeng32}@gatech.edu`

Abstract

In this work, we create a Python implementation of the Linear Lambda Calculus (LLC). While there has been work formalizing LLC, no such implementation exists in Python. By doing so, we create a typed version of lambda calculus enforcing linear logic in Python. By doing so, we build a compact yet effective model of linear logic within Python that can be potentially used for future reasoning related to linearity. This is similar to the purpose of STLC, which was built as a lightweight model for traditional logic .

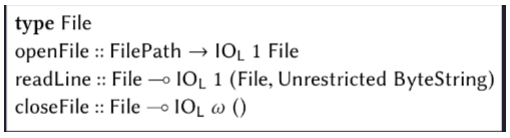
We start by examining LLC and proposing a version mirroring the version of simply typed lambda calculus (STLC) from this course. Then, we discuss our implementation and provide examples illustrating how the linear behavior is indeed achieved.

1 Overview - Linear Logic

As motivation for a linear lambda calculus, let us consider the broader topic of linear logic first. Linear logic is a branch of logic which emphasizes casual implication. Specifically, when dealing with an implication $A- > B$, linear logic provides a way to say that A is consumed in the attempt and cannot be used again. The original paper by Jean-Yves Girard which proposed the system details the difference between reactions(something is consumed) and situations(nothing is consumed) in an implication [5].

In the context of programming languages, linear logic is the idea that each variable in an expression must be used exactly once. For example, a statement like $f(x)$: return 1 would not be allowed since the function f does not use the variable x. Similarly, $f(x)$: return $x + x$ is also not allowed as x is used twice. While this may seem highly restrictive, linear logic emphasizes the use of resources since we cannot underuse or overuse each variable.

Today, linear logic is employed in modern programming languages as well. Specifically, both Linear Haskell and Clean enforce this requirement, and an example of Linear Haskell code can be seen in Figure 1. Note that the file is opened, read exactly once, and then closed, thus meeting the linear logic criteria.



```
type File
openFile :: FilePath → IO_L 1 File
readLine :: File ⊸ IO_L 1 (File, Unrestricted ByteString)
closeFile :: File ⊸ IO_L ω ()
```

Figure 1: Example of Linear Haskell Code for File IO

Furthermore, it turns out that this entire concept of linear logic can be emphasized via the type checker. This again makes sense due to the Curry-Howard correspondence, which states that type checking and logic directly map to one another.

2 Linear Type Checking

As mentioned, one way of enforcing linear logic in a programming language is to have a linear type system. Linear types systems behave similarly to standard type systems, however there is the additional requirement that for an expression to be well typed, every property of the environment must be used exactly once. In this regard, the linear type system is mimicking the concept that each variable is used exactly once in linear logic, hence why the two are equivalent.

2.1 Substructural Types

The typing rules seen so far in this course have included three implicit rules taken as fact: weakening, contraction, and exchange.

Weakening is the idea that a property of the environment can be removed for free, which is clearly not allowed with linear typing since every property must be used once. Contraction is the idea that a property can be duplicated in the environment, which is again not allowed with linear types since every property cannot be used more than once. Lastly, exchange allows one to swap the order of properties in the environment, and this is the only such rule allowed with linear types as it does not affect amount of the use of the properties.

Before proceeding, we remark upon the fact that linear types inherently are not novel and actually belong to a family of types called substructural types, which include other type systems such as affine types or ordered types([1]), all of which place various restrictions on which of these three rules are allowed.

3 Linear Lambda Calculus

Now, we introduce the linear lambda calculus (LLC), which is similar to simply typed lambda calculus (STLC) except with the restrictions of linear types and hence linear logic. While multiple works ([2], [3]) have formalized LLC, we propose our own version designed to be analogous to the features of the STLC version seen in this course.

At a high level, our version of LLC includes both linear type checking rules for all the features we have seen in class (unit type, sum type, product type, abstraction, and application), along with small step semantics for reducing a linearly type safe expression.

3.1 Linear Type System

In regards to the linear type system, the typing rules for our version of LLC can be seen below:

$$\frac{}{\vdash \langle\rangle : \mathbf{1}}\ \text{ty-unit} \qquad\qquad \frac{}{x : \tau \vdash x : \tau}\ \text{ty-var}$$

$$\frac{\Gamma, a : A \vdash E : B}{\Gamma \vdash (\lambda a : A \,.\, E) : A \to B}\ \text{ty-abs}$$

$$\frac{\Gamma_1 \vdash y : A \qquad \Gamma_2 \vdash x : A \to B}{\Gamma_1, \Gamma_2 \vdash x\ y : B}\ \text{ty-app}$$

$$\frac{\Gamma_1 \vdash x : A \qquad \Gamma_2 \vdash y : B}{\Gamma_1, \Gamma_2 \vdash \langle x,\ y\rangle : A \times B}\ \text{ty-product}$$

$$\frac{\Gamma \vdash x : A}{\Gamma \vdash \mathsf{inl}_{A+B}\ x : A + B}\ \text{ty-sum-left} \qquad\qquad \frac{\Gamma \vdash x : B}{\Gamma \vdash \mathsf{inr}_{A+B}\ x : A + B}\ \text{ty-sum-right}$$

$$\frac{\Gamma_1 \vdash x : A + B \quad \Gamma_2 \vdash a : A \to C \quad \Gamma_2 \vdash b : B \to C}{\Gamma_1, \Gamma_2 \vdash \textsf{case}\ x\ a\ b : C}\ \text{ty-case}$$

$$\frac{\Gamma \vdash x : A \quad \Gamma \vdash y : B}{\Gamma \vdash \langle\langle x, y\rangle\rangle : A \& B}\ \text{ty-and} \qquad \frac{\Gamma \vdash x : A \& B}{\Gamma \vdash \textsf{fst}\ x : A}\ \text{ty-first} \qquad \frac{\Gamma \vdash x : A \& B}{\Gamma \vdash \textsf{snd}\ x : B}\ \text{ty-second}$$

$$\frac{\Gamma_1 \vdash \langle x, y\rangle : A \times B \quad \Gamma_2, a : A, b : B \vdash e : C}{\Gamma_1, \Gamma_2 \vdash \textsf{destruct}\ \langle x, y\rangle\ \textsf{in}\ (\lambda a : A. \lambda b : B. e) : C}\ \text{ty-destruct}$$

Now, consider each rule and how it achieves linear typing. In the case of ty-unit, the unit $\langle\rangle$ is known to be of the unit type, meaning that no additional assumptions are required. Hence, the environment must be empty as a non-empty environment implies we are illegally not using some assumptions. The same logic applies to ty-var, where x being of type τ only needs knowledge of $x : \tau$ in our environment.

For ty-abs, we have the same type rule as STLC since linearity does not change the requirement of adding the input variable's type to the environment and then showing the function body is of the output type.

For ty-app, we again follow a similar rule to STLC of showing that the left expression is a function from $A \to B$ and the right expression is of type A. However, now we invoke linearity: each property of the environment must be used exactly once, meaning that each property can either be used to show the type of the left expression or the type of the right expression, hence causing us to partition the environment in Γ_1, Γ_2. However, our type checker may not know in advance which environment properties to use for which of the two hypotheses. As such, when discussing our implementation later, we more concretely state this issue and how we overcome it.

Now, let's consider product ty-product. Again, we define the product construction in the same way as STLC, but to show something is of a product type, we must show that each component is of the appropriate type. Once again, this requires splitting the environment to satisfy linearity.

For sum types, we once again construct a sum type in the same manner as STLC via inl or inr, and the type checking rules for the constructor are the same as well. For destructing a sum type via case, the syntax is again the same, but now linearity comes into play. Like STLC, we show the first argument is of the sum type $A + B$, the second argument is a function from $A \to C$, and the third argument is a function from $B \to C$. However, we will only end up using one of the two functions during execution, so rather than needing to split the environment into three parts, we have one part for showing the first argument, and a second part for both the functions.

3.1.1 Linear Products

Note the analysis of the linear type rules above has not mentioned how to destruct a product type. In STLC, this was done using fst and snd, however this leads to the question of what it means to consume a product type in linear logic. One may define consumption of a product as both elements of the product are used exactly once, or that only one element is used exactly once.

Thus, in LLC, we must have two product types to handle each view. For our traditional product constructed using $\langle\rangle$, we require consumption to use both arguments. Thus, we introduce the destruct syntax in ty-destruct, where one needs to show the argument is indeed of the product type and that the resulting expression is of the output type. Once again, the environment is split into two to achieve linearity.

We now also introduce a second product type, constructed with $\langle\langle\rangle\rangle$. This product type is consumed by only using one argument, hence we consume it via fst and snd in ty-first, ty-second respectively. The constructor rule ty-and is similar to ty-product except that we won't need to split the environment since we will only end up using 1 of the arguments.

As a remark on notation, this new product type is known as an additive conjunction, while our traditional product type is known as a multiplicative conjunction. The additive/multiplication term refers to whether we consume using one or both arguments. Similarly, the conjunction term refers to the fact that we construct using both arguments, while a disjunction means that we construct using both arguments. Thus, using this notation, our sum type is also known as an additive disjunction. Note that a multiplicative disjunction does not exist within LLC since it makes no sense to consume both arguments while only being able to construct with 1 argument. These conjunctions and disjunctions directly correspond to constructs within linear logic ([2]).

3.2 Small Step Reduction Semantics

With the linear type checking rules in mind, let's consider small step reduction semantics for a type safe expression. We propose applicative order reduction rules, meaning that an argument is substituted into a function without fully reducing the argument first. The complete set of rules for our reduction syntax is given below:

$$(\lambda x : A.\ e_1)\ e_2 \to e_1[e_2/x]$$
$$(\lambda x : A.e_1) \to (\lambda y : A.e_1)[y/x] \text{ provided y is not free in } e_1$$
$$\lambda x.(e_1\ x) \to e_1 \text{ provided x is not free in } e_1$$
$$\text{case (inl } e_1)\ e_2\ e_3 \to e_2\ e_1$$
$$\text{case (inr } e_1)\ e_2\ e_3 \to e_3\ e_1$$
$$\text{fst } \langle\langle e_1, e_2\rangle\rangle \to e_1$$
$$\text{snd } \langle\langle e_1, e_2\rangle\rangle \to e_2$$
$$\text{destruct } \langle e_1, e_2\rangle \text{ in } \lambda a : A.\lambda b : B.e \to e[e_1/a][e_2/b]$$

We remark that these rules appear the exact same as the rules for STLC seen in this class, with the exception of our new rule (destruct), which for all intents and purposes is just the same semantics as the case operator except requiring a function of two arguments. This is true because once an expression is linearly type safe, linearity plays no role in the reduction process itself. Therefore, the reduction semantics are the same as STLC, and for that reason, the focus of our projection is on the linear type checking.

3.3 Progress and Preservation Theorems

In regards to progress and preservation, note that with the exception of destruct, the set of type safe expressions in LLC is clearly a subset of the type-safe expressions in STLC. Additionally, the reduction semantics are the same. Also note that in the case of destruct, it is essentially analagous to the case operator from STLC, with the only difference being that it consumes a multiplicative conjunction by providing a function with two arguments.

From this, while we do not provide a formal proof of progress and preservation, one could reuse the arguments from STLC if desired.

4 Implementation

As our implementation, we take in a desired expression and type. These are run through the linear type checker containing the rules from section 3.1. If this fails, we return that the expression is not linearly type safe; else, we proceed with the small step semantics for reduction. This is all done through the abstract syntax tree (AST) form, where the user's input expression and type are given as ASTs, allowing our model to easily perform the type checking and reduction as everything is already parsed into the internal format used by our code. While a more practical use case is for users to submit input strings which would then be parsed into ASTs, creating a parser would require substantially more work and is beyond the scope of this class.

Additionally, the reduction semantics are the exact same as STLC, so we place our efforts primarily in linear type checking. Thus, our current implementation's reduction semantics only support standard abstraction and application (not products, sums, etc.) although we provided the full reduction semantics in this report earlier, while our linear type checker indeed implements all of the rules discussed earlier. This was discussed with the professor and determined as the appropriate scope of the project since there is no meaningful difference between STLC and LLC in reduction semantics, so we focused the vast majority of our efforts into the linear type checking.

4.1 Novelty

We remark that there currently exists no implementation of STLC in Python, let alone LLC. Thus, while the concept of LLC is not novel in itself, our work is the first to implement it in Python. Additionally, our proposed typing rules are not the exact same as the previous works ([2], [3]), but rather inspired by these works while being analogous to the STLC version seen in class.

4.2 Environment Partitions

As discussed earlier, some of the LLC typing rules require splitting the current environment into two disjoint parts in order to prove both hypotheses. However, a natural challenge is that the type checker may not know which environment properties should be used for which hypotheses.

Thus, our initial implementation worked by simply trying all possible splits. If one split resulted in a successful type check, then the expression is type-safe, and it is not type-safe if not such splits are successful. While this method indeed works, it takes exponential time with respect to the size of the environment and will not be feasible for large programs.

Instead, we propose our own method of attempting to use all of the environment on the first hypothesis. Then, if we are able to successfully reach the end of the proof (at either ty-unit or ty-var), we use modified versions of those rules where we do not require the environment to be empty for ty-unit, or that only the single variable is in the environment for ty-var. Thus, if this first step is successful, there will be some portion of the environment remaining, and then we check that the second hypothesis can be proved using all of the remaining environment. Note that this procedure is done recursively since it's possible for the first hypothesis to also include environment splits - in other words, only at the root of the induction do we require both hypotheses to use up everything in the environment.

4.3 Examples

Now, let us examine some examples of our implementation successfully type checking input expressions. First, consider the input expression $\lambda x : \mathbf{1}.\ \langle\rangle$ and input type $\mathbf{1} \to \mathbf{1}$. Our type checker deems this invalid, which is indeed correct since x is not used within the function, so the expression is not linear. From an implementation perspective, our type checker will reach a point where $x : \mathbf{1}$ is in the environment, but the given expression is only $\langle\rangle$, which fails ty-unit.

Similar to the above, consider $\lambda x : \mathbf{1}.\ x$ with type $\mathbf{1} \to \mathbf{1}$. Our type checker deems this correct since now x is used within the function. Specifically, we will reach a point where the environment is $x : \mathbf{1}$ and the expression is $x : \mathbf{1}$, so we can use ty-var to deem it valid.

As an example of a variable being used illegally more than once, consider $\lambda x : \mathbf{1}.\ \langle x, x\rangle$ with type $\mathbf{1} \to \mathbf{1} \times \mathbf{1}$. Our type checker correctly deems this invalid, and this is because we first add $x : \mathbf{1}$ to our environment and then need to show $\langle x, x\rangle : \mathbf{1} \times \mathbf{1}$. However, we know from ty-product, that this involves splitting the environment, so one of the two splits will not have access to $x : \mathbf{1}$ and thus this split will fail.

Lastly, consider $((\lambda x : \mathbf{1}.\ \langle\langle x, x\rangle\rangle)\ \langle\rangle) : \mathbf{1}\&\mathbf{1}$. Our type checker deems this as valid, which is true because we are using the additive conjunction here. Specifically, since this product type is consumed by only using 1 of the arguments, we do not need to split the environment as mentioned in ty-and. So, both hypotheses can use the fact that $x : \mathbf{1}$, making the expression type-safe.

We remark that while these examples are quite simple, they best illustrate the concept of linearity in an understandable manner and show how our implementation is indeed able to achieve this behavior.

5 Next Steps

In regards to future steps, our implementation can be further enhanced. First, we could create a parser so that users may write their own text strings as input rather than needing to use our AST format directly.

Additionally, we could complete the reduction semantics implementation to include support for the two product types, sum type, and destructors of both. This would be the same behavior in STLC, but as mentioned earlier, no Python implementation of STLC exists, thus complicating this step and forcing us to leave it out of scope.

Lastly, we plan to explore the addition of inductive types within LLC. While inductive types are not part of the traditional LLC (they are not part of standard STLC either), there has been some new work formalizing inductive linear types ([4]).

6 Conclusion

In conclusion, our work is the first to create a Python implementation of LLC. While our implementation can be improved as discussed earlier, our current work is a strong starting point and clearly achieves the desired linear behavior as seen through our discussed examples. Thus, we have created a lightweight model in Python that could be used to reason about linear logic, similar to how STLC is a model for reasoning about standard programming languages.

7 References

[1]Substructural Types
[2] Linear Lambda Calculus
[3] A taste of linear logic
[4] A Calculus of Inductive Linear Constructions
[5] LINEAR LOGIC: ITS SYNTAX AND SEMANTICS