Church Meets Turing

Eric Bond, Matthew Keenan, Yuxuan Xia

Fall 2023

1 Introduction

Can a compiler check that a function in a functional language runs in polynomial time? Compilers can use type systems to statically enforce a wide variety of properties from security level non-interference[VS97][OLEI19], memory safety[Rey02][Rus], resource usage[OLEI19], and correctness[LPR$^+$20]; so how about time and space complexity? It turns out that that they can, and we will demonstrate programming in some of these langauges in this report, but there are many complications and it will lead us into the field of ***Implicit Computational Complexity Theory***(ICC)[DL22].

Implicit Computational Complexity is an important bridge between type theory and computation complexity, the ICC conference puts it like this: *"implicit computational complexity provides a framework for a principled incorporation of computational complexity into areas such as formal methods in software development, the study of programming languages, and database theory."* [icc]

In this mini survey, we will cover the practical results of ICC as a framework for designing programming languages which are sound for a certain complexity class and the theoretical results of ICC as a branch of complexity theory.

2 Lambda Calculus and Complexity

Unlike regular complexity theory, ICC is heavily inspired by researches in programming languages and formal logic, which assumes a computational model different than the usual Turing machine. In this section, we introduce the *lambda calculus*, and briefly explain why lambda calculus in its simplest form (untyped lambda calculus) is not that suitable for studying complexity theory.

2.1 Lambda Calculus

The simplest form of lambda calculus is *untyped lambda calculus*, where the basic units are *lambda terms*. A lambda term is one of the following:

1. x: a **variable** that is any identifier representing a parameter.

2. $\lambda x.M$: a **lambda abstraction** defines a function that takes in a parameter x and returns the function body M.
3. MN: an **application** of M on N, which evaluates M as a function with N as a parameter by substituting the parameter of M with N. (This is called *substitution* or *β-reduction*)

For example, consider the following "add" function in lambda term:

$$add := \lambda x.\lambda y.x + y$$

To see lambda term evaluation in action, we can apply *add* to any two numbers:

$$\begin{aligned} add\ 4\ 5 &\to (\lambda x.\lambda y.x + y)\ 4\ 5 \\ &\to (\lambda y.4 + y)\ 5 && \text{(substitute } x \text{ with 4)} \\ &\to 4 + 5 \to 9 && \text{(substitute } y \text{ with 5)} \end{aligned}$$

The simple syntax of untyped lambda calculus gives it very strong flexibility, and allows simple terms to have unexpected behaviours. For example, consider this term ω:

$$\omega := (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$$

If we try to evaluate this term:

$$\begin{aligned} &(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \\ \to &(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \\ \to &(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \\ \to &\dots \end{aligned}$$

In each step we substitute x in the left $(\lambda x.\ x\ x)$ with $(\lambda x.\ x\ x)$, which would give back the same term! This is the easiest case of a term that diverges, or simply loops forever.

In fact, using a similar construction, we can construct a term that not only loops forever, but even grows in size:

$$\begin{aligned} &(\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x) \\ \to &(\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x) \\ \to &(\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x) \\ \to &\dots \end{aligned}$$

These kind of behaviours suggest that untyped lambda calculus might sometimes be too flexible and hard to use. In fact, because function application in lambda calculus is just substitution, it cannot specify what inputs to take that would make sense. Consider back the *add* example:

$$\begin{aligned} add\ True\ 5 &\to (\lambda x.\lambda y.x + y)\ True\ 5 \\ &\to (\lambda y.True + y)\ 5 \\ &\to True + 5 \to ? \end{aligned}$$

This motivates the use of *type systems* to limit inputs to terms. This leads to what's called *simply typed lambda calculus*, and constitutes the basics of many of todays function programming languages.

In simply typed lambda calculus, each lambda term would have an associated type, such as base types int, $bool$, or function types $int \to bool$, etc. A lambda term only evaluates when the input's type matches the type the function accepts. We'll see later in this report that we can use type systems to regulate semantic behaviours of programs, including running time.

2.2 Measuring Complexity

An important first question, therefore, is how do we even measure the time complexity of a functional programming language? A Naive answer is to suggest that you can measure the number of times you apply a function (often called a β-reduction). This falls down, however, because the amount of time it takes a computer to perform all the substitutions of a β-reduction could depends on the size of the expression. Thankfully [ADL16] showed as recently as 2016, that a polynomial number of β-reductions does indeed imply that a function is taking a polynomial number of steps. Since this result was only available very recently, much of the other prior work on polynomial-time functional programming relies on a different strategy, namely interpreting the functional language on a polynomial-time Turing machine.

This result doesn't, however, generalize to space complexity classes and smaller time complexity classes. In 2022, [ADLV22a][ADLV22b] provide the first *reasonable* result for studying logarithmic space using the pure, untyped, lambda calculus. There had been many previous results for logarithmic space under various conditions and using alternative tools like graph rewriting, and the Krivine abstract machine. This recent work addresses their deficiencies and simplifies the results.

For the full details on measuring the complexity of lambda calculus, we will refer the reader to the previously mentioned papers. Our focus will be on syntax level language features that allow programming languages to capture polynomial time.

3 LFPL

LFPL is a type system that guarantees polynomial runtime, and when awkwardly extended with non-iterable versions of iterable data structures can compute all polynomial-time functions.

There are two main principles that keep LFPL programs to polynomial time. Firstly, lambda variables can only be used at most once within a function body. Secondly, iterable data structures require an object of type $\diamond$ to "pay" for the

construction of the data type. These diamonds can't be created except for by deconstructing other data structures.

LFPL has the following programming constructs available:

Function Abstraction and Application

Functions introduce the forms $\boldsymbol{\lambda x : T.\ e}$ and $\boldsymbol{e\ e}$. Notably x can only be used at most once within the body of e because the functions in LFPL are *affine linear*. We use $\boldsymbol{T \multimap T}$ to denote the type of linear functions.

Products (Tuples)

There are two different types of product, which we will write with $\boldsymbol{e \times e}$ and $\boldsymbol{e \otimes e}$. We use the same notation for their types, $\boldsymbol{T \times T}$ and $\boldsymbol{T \otimes T}$ The difference is that the first product allows you to use the same variable on both sides of the product (i.e. $\boldsymbol{x \times x}$ is allowed, as an exception to the "variables are only used once" rule), but only allows you to isolate either the left or the right hand side later. You must choose whether you want $\texttt{fst}\ \boldsymbol{(1 \times 2)}$ (which evaluates to 1) or $\texttt{snd}\ \boldsymbol{\langle 1, 2 \rangle}$ (which evaluates to 2). In contrast, the second kind of product does not let you use the same variable on both sides, but you can then deconstruct the tuple using $\texttt{let}\ \boldsymbol{x \otimes y = e}\ \texttt{in}\ \boldsymbol{e}$.

Iterable Datatypes

The canonical example of iterable data types is binary numbers, which we will give type $\mathbf{N}$ have constructors $\mathbf{Nil} : \mathbf{N}$, $\mathbf{S_0} : \diamond \multimap \mathbf{N} \multimap \mathbf{N}$, and $\mathbf{S_1} : \diamond \multimap \mathbf{N} \multimap \mathbf{N}$, where Nil constructs an empty number and S_0 and S_1 append either a zero or a one to the right end of the number. Notice that both successor constructs have a "diamond" argument. These diamonds cannot be created during execution, they must have been provided in the input.

The iterator for the binary numbers takes the form $\mathbf{iter_N(e, f, g)} : \mathbf{N} \multimap \mathbf{T}$ where e has type $\mathbf{T}$ and f and g both have type $\diamond \multimap \mathbf{T} \multimap \mathbf{T}$ for some type $\mathbf{T}$. **It is also worth noting that e, f and g cannot use variables from outside the iterator**. This iterator begins with e and then traverses the given number from right to left, calling f on its result if it sees a 0 and calling g if it sees a 1. Notice that the constructors that required a diamond give that diamond back to us in the iterator so we can build up another data structure.

We can extend this idea to lists with constructors $\mathbf{Emp} : \mathbf{ListT}$ and $\mathbf{Cons} : \diamond \multimap \mathbf{ListT} \multimap \mathbf{ListT}$.

Non-Iterable Datatypes

Finally we add non-iterable datatypes to LFPL. The key idea here is that these datatypes do not require a diamond to build, but to prevent them increasing the runtime to exponential, we cannot use them to run a function multiple times.

Let us first consider the booleans of type $\mathbf{B}$. We have two constants $\mathbf{true}$

and **false**, and an if construct **if** : $\mathbf{B} \multimap (\mathbf{T} \times \mathbf{T}) \multimap \mathbf{T}$. Notice that we used the parallel product $\times$ since we know the if will always discard one of the branches.

We can also make a non-iterable version of the binary numbers. Let's call this $\mathbf{N'}$. This data type has perhaps more expected types on its constructors: $\mathbf{Nil'} : \mathbf{N'}$, $\mathbf{S'_0} : \mathbf{N'} \multimap \mathbf{N'}$, and $\mathbf{S'_1} : \mathbf{N'} \multimap \mathbf{N'}$. In the absence of an iterator, we add some functions for querying bits of the number: $\mathbf{iszero} : \mathbf{N'} \multimap \mathbf{B} \otimes \mathbf{N'}$ to check if a number is zero (and return the number itself, so we can continue working with it), $\mathbf{head} : \mathbf{N'} \multimap \mathbf{B} \otimes \mathbf{N'}$ to check the final digit of a number, and $\mathbf{tail} : \mathbf{N'} \multimap \mathbf{N'}$ that removes the last digit.

3.1 Programming in LFPL

Programming in LFPL is quite restrictive, which makes it awkward to even program simple functions. Let's start with a simple list append function. In a normal language, we might expect this function to look something like this:

$$\mathsf{append}(L_1, L_2) \triangleq \mathsf{iter}(L_2,\ (\lambda x, ys.\ \mathsf{Cons}\ x\ ys))\ L_1$$

There are a couple reasons, however, that this is not valid in LFPL. The first is that we have not paid a diamond to build the list constructor, nor have we received a diamond for iterating through it. This one is easy to fix, we just add a diamond to the loop body: $(d, x, ys \to \mathsf{Cons}\ d\ x\ ys)$

The second, harder, issue is that inside the arguments to iter we do not have access to the variable L_2, (we cannot use variables from outside the iterator inside the iterator). To fix this, instead of using the iterator to build up the list, we instead have to use the iterator to build up a function that appends something to L_1:

$$\mathsf{append}(L_1) \triangleq \mathsf{iter}(\lambda L_2.\ L_2,\ \lambda d\ x\ f.\ \lambda L_2.\ \mathsf{Cons}\ d\ x\ (f\ L_2))\ L_1$$

3.2 Completeness: Running a Turing machine

While this language feels very restrictive, it is in fact able to simulate any polynomial-time Turing machine. This is important because it means we can run any polynomial-time algorithm in this system. The non-iterable datatypes are important: it was originally shown [Hof99] that any non-space-increasing polynomial-time calculation can be done in LFPL, but it was later [AS02] [Hof03] extended to any polynomial-time calculation by adding non-iterable datatypes. We will show that this language is 'complete' for polynomial time by showing that we can simulate any Turing machine for any polynomial number of steps.

First we need to build up a data structure for storing the state of a Turing machine. For the head, we can use a boolean to store the state currently under

the head of the machine, and then two non-iterable natural numbers to store the tape to the left and to the right of the head. For the machine's internal state, we can just use a series of booleans. If we product all this together using $\otimes$, we get a Turing machine state, S.

$$\mathsf{S} \triangleq (\mathsf{N}' \otimes B \otimes \mathsf{N}') \otimes (B \otimes B \otimes B \dots)$$

From this we can build a state transition function $t : \mathsf{S} \multimap \mathsf{S}$ for our Turing machine, that transitions one state of the Turing machine. We won't give an example t here because it would be quite long.

Once we have a transition function t, all we need to do now is to copy the input onto the tape, and then find a way to run the transition function polynomially many times.

Hofmann [Hof99] uses a # operation that takes a $f(n) : \mathsf{N} \to \mathsf{N}$, and turns it into another function $f^{\#}(n) : \mathsf{N} \to \mathsf{N}$ which is the result of applying f $\mathsf{length}(n)$ times.

$$\begin{aligned} f^{\#}(n) = \mathsf{iter}(& \\ & m \to m, \\ & d \to g \to m \to f(g(m@[0]_d)), \\ & d \to g \to m \to f(g(m@[1]_d)) \\ &)\ n\ \mathsf{Nil} \end{aligned}$$

We've used the notation $m@[1]_d$ to indicate that we want to add a 1 as the most significant digit to m, paid for using d. This function will look similar to append from above.

Now, using this operation, we can run t any polynomial number of times. We can run it a scalar number of times using function composition. For example, $(t(t(t(x)))$ applies t three times. We can extend this to polynomials with the operator, for example f runs the transition function $|n^3|$ times.

3.3 Soundness: how LFPL guarantees polynomial runtime

Let us briefly address the question of how we know that LFPL programs take polynomial time to run. The full proof was given in the original LFPL paper [Hof99], we'll just sketch out some intuition here.

The first observation to make is that every step reduction in this language, with the exception of iteration, makes the expression smaller. This is why we are using linear functions, normal function application can make the expression grow but copying its input. Because in linear functions, the input is only used

once, we simply move the application argument to its new position within the function body and remove the λ guaranteeing that the new expression is smaller.

The hard part of this is dealing with the iteration case, how do we know that the iteration can only produce polynomial-sized expressions? This is where the diamonds become useful. We know that there is a fixed number of diamonds n in the input, and we know that this number will never grow because there is no way to create a new diamond. This number of diamonds bounds the number of iterations. The body of an iterator can only therefore be run n times (or $m \times n$ times if the iterator itself has been run m times). This create a polynomial limit on the number of times any part of the program can be run.

4 Bounded Linear Logic & Graded Modal Types

In Linear Logic, lambda variables must be used exactly once within a function body. (We should note that we are using the term logic because these type systems are typically derived from logical reasoning systems using the Curry-Howard correspondence[Wik23]). LFPL is a bit more flexible because it uses affine linear logic, where variables can be used at most once in functions. This draconian limitation guaranteed PTIME soundness at the cost of having an inexpressive programming language. For instance, linearity disallows natural presentations of functions like taking the square of a number or performing safe division. Is it possible to allow more flexibility while preserving PTIME soundness?

$$\text{✗}square \triangleq (\lambda x : int.\ x * x)$$
$$\text{✗}safeDiv \triangleq (\lambda x : int.\ \lambda y : int.\ \mathsf{if}\ (y = 0)\ \mathsf{then\ none\ else\ some}(x/y))$$

Thankfully, the answer is yes and it begins with Girard et al's introduction of bounded linear logic[GSS92] in the 90s. Bounded linear logic defines a logic where variable usage is marked by a natural number denoting the number of times a variable is allowed to be used. This number is bound by a polynomial. There have been many variations on this idea: [Laf04] tries to remove the polynomials from the syntax of the language, [DLH09] extendes bounded linear logic by allowing quantification over resource bounds, and [GS14] reformulates bounded linear logic as a graded type theory. We will present the modern take on this idea which uses graded modal type theory [OLEI19] [1][HMWO21].

Graded types allow us to annotate types with a grade, so *int* might become *int*[2] where the [2] indicates that this is a variable that can be used twice. For example, this is what the square and safe division programs could look like:

[1]Granule does not currently enjoy PTIME soundness. We have reached out to the authors of [Atk23] and [OLEI19] to discuss the feasibility of designing a sound and complete PTIME language using graded modal types.

✓$square \triangleq (\lambda x : int[2].\ x * x)$

✓$safeDiv \triangleq (\lambda x : int[1].\ \lambda y : int[2].\ \mathsf{if}\ (y = 0)\ \mathsf{then\ none\ else\ some}(x/y))$

These grades can be generalized beyond integers to any *set equipped with a semiring and a preorder structure* that is, it must have operations that act like $+, \times, \leq$ and values that act like 0 and 1. Swapping out this semiring allows us to capture different complexity classes as we will see below.

Two examples of polynomial-time type systems that use types bounded by grades are soft linear logics [Laf04], in which types are bound by a natural number, and bounded linear logic [DLH09], in which types are bound by resource polynomials.

The $\mathsf{d}\ell\mathsf{PCF}$[DLG11] and Geometry of Types[DlP13] line of research does this more generally. In those systems, terms and types of a higher order language are indexed by a first order language. The choice of first order language determines the complexity class of the higher order terms. This allows for a framework approach to designing and studying ICC systems.

5 Implicit Complexity in Context

In Section 3 and Section 4 we introduced two type systems which are sound for polynomial time, both of which are variants of linear logic [Gir87]. Many of the results in ICC are based off of linear or substructural logics. The reason for this can partially be attributed to the fact that marking usage constraints on variables was the key insight in what is considered to be a seminal ICC paper[BC92] by Bellantoni and Cook.

While their work used a model of computation based on Kleene's algebra of recursive functions [Kle81], others noticed that the variable usage restriction could also be captured in typed linear lambda calculus.

$$\frac{\Gamma \vdash \mathbf{M} : \mathbf{A}}{\Gamma, \mathbf{x} : \mathbf{B} \vdash \mathbf{M} : \mathbf{A}}\ \text{Weakening}$$

$$\frac{\Gamma, \mathbf{x} : \mathbf{B}, \mathbf{x} : \mathbf{B} \vdash \mathbf{M} : \mathbf{A}}{\Gamma, \mathbf{x} : \mathbf{B} \vdash \mathbf{M} : \mathbf{A}}\ \text{Contraction}$$

$$\frac{\Gamma, \mathbf{x} : \mathbf{B}, \mathbf{y} : \mathbf{C} \vdash \mathbf{M} : \mathbf{A}}{\Gamma, \mathbf{y} : \mathbf{C}, \mathbf{x} : \mathbf{B} \vdash \mathbf{M} : \mathbf{A}}\ \text{Exchange}$$

Structure rules in a type system dictate how variables in a context can be used. The three main structure rules are weakening, contraction, and exchange. Weakening allows for unnecessary variables to be added to the context which is a form of duplicating data. Contraction allows unused/duplicated variables to be removed from a context. Exchange allows for variables in a context to

be used in any order. A substructural type system is a type system that uses a subset of the structural rules. A *linear* type system permits the exchange rule which results in a programming language in which variables must be used *exactly once*. An *affine* type system permits both exchange and contraction which yields a programming language in which variables must be used *at most once*. From these components, various substructural type theories are formed including LFPL, Elementary Affine Logic, Light Affine Logic, and Light Linear Logic. All of which are sound for PTIME [LH05].

5.1 Limitations

A type system which is purely linear or purely affine is extremely limited in terms of its expressiveness as a programming language. That is why linear type systems for real programming languages[LPR$^+$20] usually add an additional construct called the *bang modality*(denoted **!**). The bang modality reintroduces a notion of unrestricted variable usage by permitting all the structural rules only for types annotated with *bang*(**!**). For example, the following program terms are permitted

$$\begin{aligned} \mathbf{!}\mathit{Weakening} &\triangleq \boldsymbol{\lambda}\mathbf{!x.(x,x)} : \mathbf{!A} \multimap \mathbf{!A} \otimes \mathbf{!A} \\ \mathbf{!}\mathit{Contraction} &\triangleq \boldsymbol{\lambda}\mathbf{!x.}* : \mathbf{!A} \multimap \mathbf{1} \\ \mathit{Dereliction} &\triangleq \boldsymbol{\lambda}\mathbf{!x.x} : \mathbf{!A} \multimap \mathbf{A} \\ \mathit{Digging} &\triangleq \boldsymbol{\lambda}\mathbf{!x.!!x} : \mathbf{!A} \multimap \mathbf{!!A} \end{aligned} \tag{1}$$

The reintroduction of unrestricted usage breaks soundness w.r.t. any program running in polynomial time. There is an underlying tension in design space of ICC type systems between the usability of the language and its soundness.

19

Results on LFPL

LFPL captures the complexity class $E = TIME(2^{O(n)})$.

LFPL with higher-order functions captures the class $EXP = TIME(2^{n^{O(1)}})$.

LFPL with structural recursion and higher-order functions captures Polytime.

LFPL with structural recursion and first-order functions captures polytime and simultaneously linear space.

LFPL with first-order functions and tail recursion only captures linear space.

LFPL with structural recursion and multiple use of variables in boolean context captures Polyspace.

Figure 1: LFPL with various extensions [eve]

5.2 Other Approaches and Classes

There is more to the expressiveness of a programming language than bean counting variable usage. Hofmann and Dal Lago have a series of works where they consider LFPL as a core calculus which they then extend with various programming language features and observe the resulting implicit complexity bounds. Some of their results are summarized in Figure 1. Take note that the LFPL we mention in Section 3 is just one of the results in Figure 1.

Complexity Class	Type Theory
PTIME/FP	[Hof03] [DLH09]
PSPACE	[Hof03]
2k-EXP/2k-FEXP	[BG20]
P/Poly	[MT15]
LOGSPACE	[Maz15] [ADLV22b]
PP	[DLKO21]
P#	[DLKO22]
EQP, BQP, ZQP	[DMZ10]

Figure 2: Some ICC results

Over roughly three decades since the seminal Bellantoni and Cook paper, many complexity classes have been characterized by type theory. Figure 2 contains

just a small sampling to demonstrate the variety of classes that have been studied. These include time and space bound computation, non uniform computation, probabilistic computation, and quantum computation. ICC has proven to be a useful field of research leading to an understanding of how to write programming languages which are sound for some resource bound, even if they aren't user friendly. But with all these results, has ICC helped us learn any more about the essence of complexity theory?

5.3 Implications for Complexity Theory

We look towards the Habilitation thesis[Maz17] of Damiano Mazza for answers. While Scott Aaronson jokes[Aar] about a world in which P v.s. NP is solved using operads and higher topos theory (mathematics used by modern type theory), Mazza's thesis embodies what such an approach could look like. Mazza relies heavily on tools from categorial logic to explore ICC from a more mathematical standpoint. From this perspective, he is able to give a 2-operad definition of lambda calculus and reductions, a framework for representing non-uniform computation using his parsimonious lambda calculus, and a proof of the Cook Levin theorem using lambda calculus as the model of computation. While these results are impressive, section 4.2 of his thesis provides a somber analysis of the utility of this approach with respect to studying complexity. Mazza claims:

> "[ICC] seems to be worthless when it comes to lower bound techniques. ... Not only do we not gain any intuition on the limits of computation with bounded resources but, more often than not, we actually lose the ability to prove that such limits exist!".

Additionally

> "We must be realistic and admit that, for the time being, there is no hope of making progress in structural complexity theory by use of tools from logic and programming languages theory. This is particularly frustrating because such tools, like types, categories and rewriting, seem to be at least as good as the combinatorial ones in describing computation as a dynamic phenomenon"

In conclusion, Implicit Computational Complexity is a rich sub field of Complexity Theory which provides a foundation for the design of resource bound programming languages. While there exists multiple languages which are sound for complexity classes, the usability of these languages is typically poor and insufficient for daily programming tasks.

6 Project Video Link

https://drive.google.com/file/d/193op9NxbXBDJg-P3z6zDraOUXl9e0daj/view?usp=sharing

References

[Aar] Scott Aaronson. https://scottaaronson.blog/?p=1293.

[ADL16] Beniamino Accattoli and Ugo Dal Lago. (leftmost-outermost) beta reduction is invariant, indeed. *Logical Methods in Computer Science*, 12, 01 2016.

[ADLV22a] Beniamino Accattoli, Ugo Dal Lago, and Gabriele Vanoni. Multi types and reasonable space. *Proc. ACM Program. Lang.*, 6(ICFP), aug 2022.

[ADLV22b] Beniamino Accattoli, Ugo Dal Lago, and Gabriele Vanoni. Reasonable space for the -calculus, logarithmically. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '22, New York, NY, USA, 2022. Association for Computing Machinery.

[AS02] Klaus Aehlig and Helmut Schwichtenberg. A syntactical analysis of non-size-increasing polynomial time computation. *ACM Trans. Comput. Logic*, 3(3):383–401, jul 2002.

[Atk23] Robert Atkey. Polynomial time and dependent types, 2023.

[BC92] Stephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the polytime functions (extended abstract). In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing*, STOC '92, page 283–293, New York, NY, USA, 1992. Association for Computing Machinery.

[BG20] Patrick Baillot and Alexis Ghyselen. Combining linear logic and size types for implicit complexity. *Theoretical Computer Science*, 813:70–99, 2020.

[DL22] U. Dal Lago. Implicit computation complexity in higher-order programming languages: A survey in memory of martin hofmann. *Mathematical Structures in Computer Science*, 32(6):760–776, 2022.

[DLG11] Ugo Dal Lago and Marco Gaboardi. Linear dependent types and relative completeness. In *2011 IEEE 26th Annual Symposium on Logic in Computer Science*, pages 133–142, 2011.

[DLH09] Ugo Dal Lago and Martin Hofmann. Bounded linear logic, revisited. In Pierre-Louis Curien, editor, *Typed Lambda Calculi and Applications*, pages 80–94, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[DLKO21] Ugo Dal Lago, Reinhard Kahle, and Isabel Oitavem. A Recursion-Theoretic Characterization of the Probabilistic Class PP. In *MFCS*

2021 - 46th International Symposium on Mathematical Foundations of Computer Science, Tallinn, Estonia, August 2021.

[DLKO22] Ugo Dal Lago, Reinhard Kahle, and Isabel Oitavem. Implicit recursion-theoretic characterizations of counting classes. *Arch. Math. Log.*, 61(7–8):1129–1144, nov 2022.

[DlP13] Ugo Dal lago and Barbara Petit. The geometry of types. *SIGPLAN Not.*, 48(1):167–178, jan 2013.

[DMZ10] Ugo Dal Lago, Andrea Masini, and Margherita Zorzi. Quantum implicit computational complexity. *Theoretical Computer Science*, 411(2):377–409, 2010.

[eve] https://www-lipn.univ-paris13.fr/~baillot/geocal06/slides/hofmann1302.pdf.

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

[GS14] Dan R. Ghica and Alex I. Smith. Bounded linear types in a resource semiring. In Zhong Shao, editor, *Programming Languages and Systems*, pages 331–350, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[GSS92] Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. Bounded linear logic: a modular approach to polynomial-time computability. *Theoretical Computer Science*, 97(1):1–66, 1992.

[HMWO21] Jack Hughes, Daniel Marshall, James Wood, and Dominic Orchard. Linear Exponentials as Graded Modal Types. In *5th International Workshop on Trends in Linear Logic and Applications (TLLA 2021)*, Rome (virtual), Italy, June 2021.

[Hof99] M. Hofmann. Linear types and non-size-increasing polynomial time computation. In *Proceedings. 14th Symposium on Logic in Computer Science (Cat. No. PR00158)*, pages 464–473, 1999.

[Hof03] Martin Hofmann. Linear types and non-size-increasing polynomial time computation. *Information and Computation*, 183(1):57–85, 2003. International Workshop on Implicit Computational Complexity (ICC'99).

[icc] https://web.ecs.syr.edu/~royer/icc/ICC03/call.html.

[Kle81] Stephen C. Kleene. Origins of recursive function theory. *Annals of the History of Computing*, 3(1):52–67, 1981.

[Laf04] Yves Lafont. Soft linear logic and polynomial time. *Theoretical Computer Science*, 318(1):163–180, 2004. Implicit Computational Complexity.

[LH05] Ugo Dal Lago and Martin Hofmann. Quantitative models and implicit complexity. *ArXiv*, abs/cs/0506079, 2005.

[LPR+20] Yiyun Liu, James Parker, Patrick Redmond, Lindsey Kuper, Michael Hicks, and Niki Vazou. Verifying replicated data types with typeclass refinements in liquid haskell. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[Maz15] Damiano Mazza. Simple Parsimonious Types and Logarithmic Space. In Stephan Kreutzer, editor, *24th EACSL Annual Conference on Computer Science Logic (CSL 2015)*, volume 41 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24–40, Dagstuhl, Germany, 2015. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[Maz17] Damiano Mazza. *Polyadic Approximations in Logic and Computation*. Habilitation à diriger des recherches, Université Paris 13, November 2017.

[MT15] Damiano Mazza and Kazushige Terui. Parsimonious types and non-uniform computation. In Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann, editors, *Automata, Languages, and Programming*, pages 350–361, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[OLEI19] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. Quantitative program reasoning with graded modal types. *Proc. ACM Program. Lang.*, 3(ICFP), jul 2019.

[Rey02] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[Rus] RustBelt. https://plv.mpi-sws.org/rustbelt/publications.

[VS97] Dennis Volpano and Geoffrey Smith. A type-based approach to program security. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development*, pages 607–621, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[Wik23] Wikipedia. Curry–Howard correspondence — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Curry%E2%80%93Howard%20correspondence-oldid=1185371391, 2023. [Online; accessed 01-December-2023].