Mathematical Structures in Computer Science (2022), **32**, pp. 760–776
doi:10.1017/S0960129521000505

PAPER

Implicit computation complexity in higher-order programming languages

A Survey in Memory of Martin Hofmann

U. Dal Lago

Department of Computer Science and Engineering, University of Bologna, Bologna, Italy
Email: ugo.dallago@unibo.it

(Received 1 November 2020; revised 15 December 2021; accepted 21 December 2021; first published online 15 March 2022)

Abstract
This paper is meant to be a survey about implicit characterizations of complexity classes by fragments of higher-order programming languages, with a special focus on type systems and subsystems of linear logic. Particular emphasis will be put on Martin Hofmann's contributions to the subject, which very much helped in shaping the field.

Keywords: Lambda calculus; linear logic; type systems; computational complexity

1. Introduction

While *computability* theory studies the boundary between what can and cannot be computed effectively *without* putting any specific constraint on the underlying algorithm, *complexity* theory refines the analysis by classifying the computable functions based on the amount of *resources* algorithms computing those functions require when executed by paradigmatic machines like Turing machines. Resources of interest can be computation time, space, or communication, while bounds are not expressed as fixed, absolute constants but rather parametrically – and asymptotically – on the size of the input. This way algorithms are allowed to consume larger and larger amounts of resources when the size of the input increases, and what matters is not the absolute value, but the rate of growth.

A complexity class can thus be defined as the collection of all those functions which can be computed by an algorithm working within resource bounds of a certain kind. As an example, we can form the class FP of those functions which can be computed in *polynomial time*, that is, within an amount of computation time bounded by a polynomial on the size of the input. As another example, we can form the class FLOGSPACE of those functions which can be computed in logarithmic space. Complexity theory has developed grandly since its inception (see Cobham 1965; Hartmanis and Stearns 1965), and nowadays, many complexity classes are studied, even if much remains to be discovered about their being distinct or not. Traditionally, complexity classes are studied by carefully analyzing the combinatorial behavior of machines rather than looking at the structural and morphological aspects of the underlying algorithm. (For an introduction to computational complexity theory, we recommend the textbooks by Goldreich 2008 and Arora and Barak 2009).

Starting from the early nineties, researchers in mathematical logic and theoretical computer science have introduced *implicit* characterizations of (some of) the various complexity classes. By implicit, we here mean that classes are not given by explicitly constraining the amount of resources a *machine* is allowed to use, but rather by imposing linguistic constraints on the way *algorithms* can be formulated, that is, on their *structure*. This idea has developed into an area called *implicit computational complexity* (ICC in the following), which at the time of writing is very active, with annual thematic workshops specifically devoted to it.

Many areas within logic and computer science have been affected: from *recursion theory*, within which the first attempts at characterizing the feasible functions have been given, to the field of *type theory*, in which restrictions on algorithms are given as formal systems of types, to *proof theory*, in which the structure of proofs and the dynamics of cut-elimination is leveraged when defining appropriate restrictions.

The purpose of this survey is to give the reader the possibility of understanding *what* implicit computational complexity is, the typical *flavor* of the results obtained in it, and *how* these results are proved. We will focus our attention on characterizations of complexity classes in terms of proof theory and higher-order programming languages, an area of ICC which Martin Hofmann has contributed to shaping and in which he had a fundamental role since the very beginning.

2. A Bird's Eye View on Complexity and Higher-Order Programs

Most programming languages are designed to be expressive: not only many alternative constructs are usually available, but the class of computational tasks programs can solve is kept as large as possible. In fact, one of the first theorems any programming language designer proves about a new idiom is its being *Turing powerful*, namely that any computable function can be encoded into it. Any such programming language, for obvious reasons, is simply too powerful to guarantee bounds on the amount of resources programs need, that is, to guarantee complexity bounds. But are all constructs any given programming language offers *equally harmful*, in this respect? In other words, could we remove or refine them, this way obtaining a reasonably expressive programming language in which programs consume *by construction* a bounded amount of computational resources? This can be seen as the challenge from which one starts when defining ICC systems.

One could for example consider Kleene's algebra of general recursive functions as a rudimentary programming language in which programs are built from some basic building blocks, the initial functions, by way of composition, primitive recursion, and minimization. As is well-known, dropping minimization makes this formalism strictly less expressive: the functions one represents (the so-called primitive recursive functions) are, at least, all *total*. Primitive recursive functions are however well beyond any reasonable complexity class: they are after all the class of functions which can be computed in time and space bounded by primitive recursive functions, the latter being strictly larger than Kalmar's elementary functions. Essentially, this is due to the possibility of having *nested* recursive definitions. As an example, one can *first* define addition from the initial functions

$$\begin{aligned} add(0, x) &= x; \\ add(n+1, x) &= add(n, x) + 1; \end{aligned}$$

and *then* define exponentiation $n \mapsto 2^n$ from addition as follows:

$$\begin{aligned} exp(0) &= 1; \\ exp(n+1) &= add(exp(n), exp(n)). \end{aligned}$$

Composing *exp* with itself allows us to form towers of exponentials of arbitrary size, thus saturating the Kalmar elementary functions. A recursion on *exp* leads us even beyond the aforementioned class, already too broad complexity-wise. Rather than removing primitive recursion from the algebra up-front (which would have too strong an impact on the expressive power of the underlying function algebra), we could somehow *refine* the algebra by distinguishing arguments that matter for complexity from arguments that do not: the arguments of any function f are dubbed either *normal*, when their value can have a relatively big impact on f's complexity, or *safe* when their value can only influence f's behavior very mildly, combinatorially speaking. If $\vec{x}$ are the normal parameters and $\vec{y}$ are the safe ones, we indicate the value of f on $\vec{x}$ and $\vec{y}$ as $f(\vec{x};\vec{y})$, where the semicolon serves to separate the normal from the safe. This way, the usual scheme of primitive recursion can be restricted as follows:

$$f(0,\vec{x};\vec{y}) = h(\vec{x};\vec{y});$$
$$f(n+1,\vec{x};\vec{y}) = g(n,\vec{x};\vec{y},f(n,\vec{x};\vec{y})).$$

Please notice that the way we define f from g and h is morphologically identical to what happens in the usual primitive recursive scheme. That it is a *restriction* is a consequence of the distinction between normal and safe arguments: indeed, the recursive call $f(n,\vec{x};\vec{y})$ is required to be forwarded to one of g's *safe* argument, while f's first parameter must be *normal*. This makes *exp* not expressible anymore and gives rise to new function algebra called *safe recursion* and due to Bellantoni and Cook (1992) (**BC** in the following). Is one getting close to any reasonably small complexity class, this way? Surprisingly, the answer is positive:

Theorem (Bellantoni and Cook 1992). *The class* **BC** *equals the class* FP *of polynomial time computable functions.*

This breakthrough result, discovered independently by Leivant (1993) in a slightly different setting, can be seen as the starting point of ICC and has given rise to a variety of characterizations of different complexity classes along the same lines, from polynomial space (Leivant and Marion 1994; Oitavem 2008) to nondeterministic polynomial time (Oitavem 2011), from circuit classes (Leivant 1998) to logarithmic space (Neergaard 2004).

The lesson by Bellantoni and Cook is that *refining* a closure operator rather than *removing* it up-front can be a very effective strategy to *tune* the expressive power of the underlying algebra and turn it into a characterization of a complexity class. This can also form the basis for the definition of type systems so as to go towards programming languages and calculi capable of handling higher-order functions, and we will give an overview of the results one can obtain this way in Section 3 below. Are function algebras the only kind of computational model amenable to such a refinement? Is it possible that other programming languages and logical formalisms can be treated similarly? We will give an answer to this question in the rest of this section.

Suppose, for example, that one wants to follow the same kind of path Bellantoni and Cook followed, but from a very different starting point, namely looking at the λ-calculus rather than at recursive functions. The pure, untyped λ-calculus is a minimal programming language whose terms are those generated by the following grammar:

$$M ::= x \;\big|\; \lambda x.M \;\big|\; MM$$

and whose dynamics is captured by rewriting, and in particular by so-called β-reduction:

$$(\lambda x.M)N \to_{\beta} M\{x/N\}$$

The obtained language is well-known to be Turing-powerful: there are various ways of representing the natural numbers and functions between them which make this very simple formal system capable of capturing *all* computable functions. In doing so, the λ-calculus shows its capability of simulating many different operators (e.g., conditionals, recursion, and iteration) within a very simple framework in which the basic computation step, namely β, is so simple and apparently innocuous. What is it that makes β so powerful, computationally speaking? The answer lies in the possibility for a λ-abstraction $\lambda x.M$ to not only forward the argument N to M when applied to the former, but also to *duplicate it*. Take, for example, the paradigmatic divergent term $\Omega = \Delta\Delta$, where $\Delta = \lambda x.xx$. The fact that Ω rewrites to itself (thus giving rise to divergence, and ultimately rendering the pure λ-calculus inadequate for ICC) fundamentally relies on the fact that *two* occurrences of the abstracted variable occur in the body of Δ. Similarly, Turing's fixed-point combinator

$$\Theta = (\lambda x.\lambda y.y(xxy))((\lambda x.\lambda y.y(xxy))$$

can be used to simulate recursion, since for every M it holds that ΘM rewrites in two β-steps to $M(\Theta M)$. And, again, all this crucially relies on the possibility of duplicating, since both the argument and part of the combinator itself are duplicated along the reduction process.

When looking for a *subrecursive* variation on the pure λ-calculus, many roads have been followed, the most explored one being the use of type systems, namely of formal systems which single out certain λ-terms as typable, guaranteeing at the same time that all typable programs satisfy certain properties, typical examples being safety and reachability properties. In particular, many of the introduced type systems guarantee that β-reduction is strongly normalizing on all typable λ-terms, namely that divergence is simply impossible. Sometimes, one can also go beyond that and introduce systems of types which guarantee various forms of *bounded* termination. We will give a survey on all this in Section 3 below.

But there is another, more fundamental, way of keeping the complexity of β-reduction under control, namely acting directly on duplication, as suggested by Girard (1987) in his work on linear logic. There, the main intuition is precisely the one of decomposing intuitionistic implication (thus β-reduction) into a linear implication and an exponential modality. Without delving into the proof-theoretical details of all this, we can already grasp the essence by considering a refinement $\Lambda_!$ of the pure λ-calculus:

$$M ::= x \;\big|\; \lambda\ell x.M \;\big|\; \lambda! x.M \;\big|\; MM \;\big|\; !M$$

Besides the usual operators, we find new ones, namely the so-called *boxes*, that is, terms of the form $!M$, and two kinds of abstractions, namely nonlinear ones, of the form $\lambda! x.M$, and linear ones, of the form $\lambda\ell x.M$. The latter is dubbed *well-formed* only if x occurs free in M *at most* once and *outside* the scope of any box. The β-reduction rule has to be refined itself into a linear rule *and* a nonlinear rule:

$$(\lambda\ell x.M)N \rightarrow_{\ell\beta} M\{x/N\} \qquad\qquad (\lambda! x.M)!N \rightarrow_{!\beta} M\{x/N\}$$

This refinement *by itself* does not allow us to tame the expressive power of the underlying computational model: the pure λ-calculus can be retrieved in all its strength and wildness thanks to nonlinear abstractions, that is, the following embedding allows to simulate $\rightarrow_\beta$ by way of $\rightarrow_{!\beta}$:

$$(x)_! = x \qquad (\lambda x.M)_! = \lambda! x.(M)_! \qquad (MN)_! = (M)_!!(N)_!$$

Indeed:

$$((\lambda x.M)N)_! = (\lambda! x.(M)_!)!(N)_! \rightarrow_{!\beta} (M)_!\{x/(N)_!\} = (M\{x/N\})_!$$

(where the last step is nothing more than a straightforward substitution lemma).

How should one proceed, then? A simple idea consists in dropping nonlinearity altogether, thus getting rid of boxes and nonlinear abstractions in the underlying calculus:

$$M ::= x \;\big|\; \lambda\ell x.M \;\big|\; MM$$

This way, one obtains a calculus, called the *affine* λ-calculus and indicated as Λ_ℓ, in which divergence is simply absent, for very good reasons: along any $\rightarrow_{\ell\beta}$ step, the *size* (i.e., the total number of symbols) in the underlying term strictly decreases. How about the expressive power of the obtained calculus? Apparently, this is quite limited: even though some of the number schemes in the literature (e.g., so-called Scott numerals) can indeed be seen as terms in Λ_ℓ, the class of functions which are representable in the calculus is very small, well below any complexity class. Things change, however, if one consider Λ_ℓ as a computational model for *boolean* functions. If this is the case, a nice, and tight correspondence between terms and boolean circuits starts to show up, leading to the following result:

Theorem (Mairson and Terui 2003). *Given a term M in Λ_ℓ, testing whether it reduces via $\rightarrow_{\ell\beta}$ to a specific normal form is a* P*-complete problem.*

This does not mean, however, that every polynomial time problem can somehow be represented as *a term* in Λ_ℓ. Instead, *instances* of any problem in P can very cheaply be turned into terms in such a way that solving the problem boils down to reducing the term. More results along these lines will be discussed in Section 4.

If one is rather interested in *uniform* encodings, that is, in encodings of *problems* (and not problem *instances*) into terms, the natural way to proceed consists in allowing nonlinear abstractions to occur in the calculus, at the same time constraining, in various ways, the way x can occur in M whenever forming the term $\lambda!x.M$. Indeed, some such constraints have been analyzed in the literature, giving rise to characterizations of different complexity classes. We will review some of them in Section 4, but it is instructive to see how this turns out to be possible by way of a simple example, namely the so-called *soft* λ-calculus $\Lambda_{S\ell}$, due to Baillot and Mogbil (2004), and inspired by soft linear logic (Lafont 2004).

Suppose, as an example, that whenever one forms $\lambda!x.M$, the variable x is allowed to occur free in M:

(1) either at most *once* and in the scope of *a single* occurrence of the ! operator;
(2) or possibly more than once, but *outside* the scope of any ! operator.

In other words, terms like $\lambda!x.yxx$ or $\lambda!x.y!x$ would be considered legal, while $\lambda!x.x!x$ would not be. To realize why this is a sensible restriction, observe how, whenever firing a nonlinear β-step, a term in the form $(\lambda!x.M)!N$ becomes $M\{x/N\}$, and either multiple copies of N are produced along the substitution process, all of them lying outside of any ! operator, or N is not copied at all, staying at the same nesting depth it was before the rewriting step. Summing up, while reducing any term of $\Lambda_{S\ell}$, the actual size of the term can definitely increase, but it does so in a very controlled and predictable way. Observe how allowing multiple occurrences of *x at different depths* when forming nonlinear abstractions would allow to type divergent terms, like

$$(\Omega)_! = (\lambda!x.x!x)!(\lambda!x.x!x).$$

As one can easily check, Ω is *not* a term of $\Lambda_{S\ell}$.

Summing up, we described two ways of refining and restricting universal computational models so as to make them capable of capturing relatively small complexity classes. Both of them rely on finding the right parameters to act on, namely recursion depth (in recursion theory), and duplication (in the λ-calculus). Are there other ways of achieving the same goal? The answer is affirmative, and we will say a few words about that in Section 5 below. The next two sections, however, are meant to be surveys on works about type systems and linear logic fragments, respectively.

3. Type Systems

Type systems are among the most successful lightweight formal methods and are particularly fit for languages with higher-order functions, namely the ones we are interested in here. Most type systems automatically ensure *safety* properties: well-typed programs *cannot* go wrong, even if the converse does not hold. Besides safety, some type systems also guarantee *reachability*: well-typed programs *cannot* diverge, that is, they are guaranteed to reach a final state or normal form. Again, the converse is not guaranteed to hold, except for peculiar forms of type systems (e.g., intersection types, by Coppo and Dezani-Ciancaglini 1980). A refinement of reachability, namely termination *in a bounded number of steps,* could give further guarantees as for the feasibility of the underlying algorithm, rather than mere totality.

Traditionally, one very natural way of defining type systems guaranteeing termination properties consists in deriving them from proof systems through the so-called Curry-Howard correspondence (see the textbooks by Girard et al. 1989 and Sørensen and Urzyczyn 2006 for an introduction). This way, termination is a byproduct of the cut-elimination or normalization properties for the underlying logical system, precisely because β-reduction mimics them. This is the case in the so-called simply-typed λ-calculus, which corresponds to intuitionistic propositional logic, or in the polymorphic λ-calculus, itself a sibling of second-order intuitionistic logic. In the aforementioned systems, however, the obtained class of terms is not fit for a characterization of complexity classes in the spirit of ICC. For example, the class of polymorphically typable λ-terms is so huge that it represents all functions on the natural numbers which are *provably total* in second-order arithmetic. As another example, simply-typed λ-terms require hyper-exponential time to be reduced to their normal forms (see Fortune et al. 1983), but the functions which can be encoded as simply-typed terms do not include some, like the identity, which are part of all reasonable complexity classes, see the work by Zaionc (1990, 1991).

If one is really interested in giving implicit characterizations of complexity classes by way of type systems, one is thus forced to follow different routes, some of which we will describe in the rest of this section.

3.1 Higher-order primitive recursion

One way to enrich the simply-typed λ-calculus so as to turn it into a language with a very high expressive power is to endow it with constants for the natural numbers, and operators for primitive recursion at all finite types. As is well-known, the obtained system, the so-called Gödel's System **T**, is capable of representing all functions on the natural numbers which are provably total in *first-order* Peano's Arithmetic. As such, then, **T** cannot be close to the characterization of *any* complexity classes.

System **T**, however, can be seen as nothing more than a generalization of Kleene's primitive recursion to higher-order types. It is thus very natural to wonder whether one could turn safe recursion into a sub-recursive type system guaranteeing polynomial time computability. A naive idea is that the underlying function space can be refined into two:

- The space $\square A \to B$ of functions depending on a *safe* argument of type A and returning an element of type B.
- The space $\blacksquare A \to B$ of functions depending on a *normal* argument of type A and returning an element of a type B.

This allows us to define a type system very much in the style of safe recursion. This, however, does not lead us to a characterization of the polynomial computable functions. Consider the following higher-order function $HOexp$, whose type is $\blacksquare NAT \to \square NAT \to NAT$

$$HOexp(0) = \lambda x.succ(x);$$

$$HOexp(n+1) = \lambda x.HOexp(n)(HOexp(n)(x)).$$

The behavior of this function is intrinsically exponential, $HOexp(n, m) = m + 2^n$. Actually, the obtained system, call it **HOSR**, can be proved to precisely characterize Kalmar's elementary time:

Theorem (Leivant 1999*b*). **HOSR** *precisely captures the class* E *of Kalmar's elementary functions.*

If one wants to trim the complexity of the calculus down to complexity classes capturing notions of feasibility, the key idea consists in imposing linearity restrictions. The exponential behavior *HOexp* exhibits is essentially due to the fact that *two* recursive calls take place in the inductive case. If one imposes linearity constraints on higher-order variables, one indeed obtains a calculus, called **HOSLR** which has the desired property:

Theorem (Hofmann 1997*b*). **HOSLR** *precisely captures* FP.

Hofmann's result has been proved by way of a realizability interpretation, which has been later turned into a more general and flexible tool by Dal Lago and Hofmann (2011). An earlier and very influential work by Hofmann (2000*b*) proved – by way of linear combinatory algebras – that an extension of **HOSLR** to a larger class of datatypes remain sound. A system essentially equivalent to **HOSLR** has been analyzed from a more syntactical viewpoint, obtaining results analogous to Theorem 3.1 but having a more operational flavor, by Bellantoni et al. (2000). Probabilistic variations on **HOSLR** have been considered and proved to be useful tools in the study of cryptographic constructions, by Mitchell et al. (1998), Zhang (2009), and Cappai and Dal Lago (2015). Contrary to what happens in the realm of first-order function algebras (see, e.g., the work by Avanzini and Dal Lago 2018), higher-order safe recursion is very fragile as a characterization of the polytime computable functions: switching from word algebras to tree algebras makes the class of representable functions go up to E, as proved by Dal Lago (2009*b*).

3.2 Non-size increasing computation

Systems like **HOSLR** are implicit, thus providing simple characterizations of the complexity class FP. As such, they can be seen as compositional verification methodologies for the complexity analysis of programs. This is true, of course, if the fact that all typable programs represent polynomial time functions is proved operationally, that is, by showing that those programs can be reduced *themselves* within some polynomial time bounds. This observation has led many researchers to work on the use of systems based on predicative recursion, and ICC in general, as static analysis techniques.

Quite soon, however, strong limitations to these ideas emerged, as observed by Hofmann (1999):

> *Although these systems allow one to express all polynomial time functions, they reject many natural formulations of obviously polynomial time algorithms. The reason is that under the predicativity regime, a recursively defined function is not allowed to serve as step function of a subsequent recursive definition. However, in most functional programs involving inductive data structures such iterated recursion does occur.*

In other words, even if *extensionally* complete, these systems tend to be *intentionally* very weak, ruling out many useful and popular algorithmic schemes which, although giving rise to polynomial time functions, requires forms of *nested recursion*. Examples include the so-called `InsertionSort` algorithm, which was observed *not* to be expressible in, say, **HOSLR** (see again Hofmann 1999).

A way out consists in taking *another* property of programs as the one on which potentially problematic programs (i.e., programs with too high a complexity) are ruled out. Rather than saying that nested recursions are simply not allowed, it is possible to define a type system in such a way that all typable programs are by construction *non-size-increasing*. This way, since recurring or iterating over a non-size-increasing (polytime) function is always safe, iteration and recursion can be made more liberal, and nesting is indeed allowed. This idea, introduced by Hofmann (1999), has been explored in various directions by Hofmann (2000*c*, 2002, 2003) and ultimately gave rise to the fruitful line of work on amortized resource analysis (see, as an example, the work by Jost et al. 2010 or the many recent contributions on the subject).

But how can all this be made concrete? Actually, the ingredients are relatively simple:

- On the one hand, the underlying system of types must enforce an affine regime where every free variable occurs at most once, even when it is of ground type. In other words, there is just *one* way of typing functions, namely by linear types in the form $A \multimap B$.
- On the other hand, every function inducing a strict increase on the input length must "pay the price" by taking an additional input of type $\diamond$, itself providing the "missing size." As an example, the successor function has type $\diamond \multimap NAT \multimap NAT$. It is of course important to guarantee that no closed term of type $\diamond$ can be built.

This way, a type system called **NSI** is defined and proved to characterize the polynomial time computable problems:

Theorem (Hofmann 1999). *The type system* **NSI** *characterizes the class* P.

Variations of **NSI** in which the language of terms is made to vary lead to characterizations of larger complexity classes, like PSPACE and EXP, as shown in the work by Hofmann (2002, 2003), witnessing the flexibility and interest of the approach.

4. Linear Logic

As explained in Section 2, one natural way of taming the expressive power of systems like the λ-calculus is to control copying by decomposing *functions* in the sense of functional programming into *copiers* and *forwarders*, similarly to what we did by going from Λ to $\Lambda_!$. This enables the definition of restrictions to $\Lambda_!$ obtained by tuning the way nonlinear abstractions can manipulate duplicable terms, that is, to which extent boxes can be copied, opened, and thrown inside other boxes.

There is a natural way of interpreting all this in terms of logic and types, which is precisely how this line of research comes into being. We are referring here to Linear Logic as introduced by Girard (1987). Among many other things, Linear Logic can be seen as a way of decomposing the type of functions $A \to B$ as traditionally found in type systems into $!A \multimap B$ where the ! operator, called the *bang*, is the type of duplicable versions of objects of type A, while $\multimap$ is a binary type constructor allowing to build spaces of *linear* functions, which in this context can be taken as functions using their argument at most once. This way, restricting the copying process can be done by acting on the axioms the bang operator is required to satisfy, which in full linear logic are the following ones:

$$!A \multimap !A \otimes !A$$

$$!A \multimap 1$$

$$!A \multimap A$$

$$!A \multimap !!A$$

The first two axioms (contraction and weakening, respectively) state that an object of type $!A$ can be duplicated and discarded, while the following two (dereliction and digging, respectively) allow for such a duplicable object to be turned into one of type A or $!!A$ (respectively) by either forgetting its nature as a duplicable object or by turning it into a *duplicable* duplicable object. These axioms can be seen as saying that ! is categorically a *comonad*, but can also be seen as the types of appropriate combinators written in $\Lambda_!$:

$$CONTRACTION \triangleq \lambda!x.\langle x, x\rangle : !A \multimap !A \otimes !A$$

$$WEAKENING \triangleq \lambda!x.* : A \multimap 1$$

$$DERELICTION \triangleq \lambda!x.x : !A \multimap A$$

$$DIGGING \triangleq \lambda!x.!!x : !A \multimap !!A$$

By *dropping* some of the above principles, one could get meaningful fragments of linear logic, with the hope of capturing complexity classes this way. This has materialized in many different forms, which we are going to examine in the rest of this section, somehow by reverse chronological order (this way facilitating the nonexpert reader).

4.1 Soft linear logic

One drastic way of taming the expressive power of linear logic is the one we have already considered in Section 2, namely going to systems like $\Lambda_{S\ell}$ in which boxes are *opened* whenever copied. This, correspond to collapsing the four comonadic axioms into just one:

$$MULTIPLEXOR^n \triangleq \lambda!x.\langle \underbrace{x, \ldots, x}_{n \text{ times}} \rangle : !A \multimap \underbrace{A \otimes \cdots \otimes A}_{n \text{ times}}$$

Logically speaking, this amounts to switching to *soft linear logic* (**SLL**, introduced by Lafont 2004), which can indeed be proved to be a characterization of the polynomial time computable problems:

Theorem (Lafont 2004). *The logic* **SLL** *characterizes the class* P *of polynomial time computable problems.*

Soft linear logic has proved to be amenable to many adaptations and variations. First of all, changing the way strings are represented leads to a characterization of polytime computable *functions*, as proved by Gaboardi and Ronchi Della Rocca (2007). Then, other complexity classes can be proved to be captured, of course by a careful variation of the underlying language of terms, an example being PSPACE (see Gaboardi et al. 2012). The work by Redmond (2007, 2015, 2016) is remarkable in its way of exploring the possible ramifications of the ideas behind soft linear logic from a categorical and rewriting perspective. The calculus $\Lambda_{S\ell}$ has been introduced and studied by Baillot and Mogbil (2004). The robustness of the paradigm is witnessed by works generalizing it to higher-order processes (by Dal Lago et al. 2016), session types (by Dal Lago and Di Giamberardino 2016), and quantum computation (by Dal Lago et al. 2010).

4.2 Elementary linear logic

Rather than forcing boxes to be opened whenever copied, one could simply rule out dereliction and digging, this way enabling a form of stratification, obtaining a system in which contraction and weakening are the only two valid axioms. The obtained system, called *elementary linear logic* (**ELL** in the following), is due to Girard (1998), but has been studied in a slightly different form by Danos and Joinet (2003). It characterizes the Kalmar elementary functions, hence the name:

Theorem (Danos and Joinet 2003; Girard 1998). *The logic* **ELL** *characterizes the Kalmar elementary functions.*

Elementary linear logic has also been studied from other viewpoints, namely as a type system for the pure λ-calculus enabling efficient optimal reduction algorithms, by Asperti et al. (2004), Coppola and Martini (2006) and later by Baillot et al. (2011). Even if the expressive power of the system can be seen as admittedly too large, Baillot (2015) showed how it can be tamed by considering programs having specific types, this way obtaining characterizations of P and kEXPTIME (for every $k \in \mathbb{N}$).

4.3 Light linear logic

The idea of seeing boxes as rigid entities as embodied by **ELL** can also be turned into a system characterizing the *polynomial* time computable functions, thus trimming the expressive power of the underlying logic down to the realm of feasibility. This, however, requires a significant change in the structure of formulas and rules. More specifically, not one but *two* kinds of boxes are necessary, namely the usual one, supporting weakening and contraction, and a new one, indicated as §. The syntax of λ-terms has to be appropriately tailored:

$$M ::= x \;\big|\; \lambda\ell x.M \;\big|\; \lambda! x.M \;\big|\; \lambda\S x.M \;\big|\; MM \;\big|\; !M \;\big|\; \S M.$$

Besides the two β-rules we already know, there is a third one, namely the following:

$$(\lambda\S x.M)\S N \rightarrow_{\beta} M\{x/N\}.$$

Whenever forming a nonlinear abstraction $\lambda! x.M$, the variable x can appear more than once in M, and this occurrence needs to be in the scope of exactly one ! or § operator. On the other hand, for $\lambda\S x.M$ to be well-formed it must be that x occurs at most once in M *and* in the scope of exactly one § operator. Finally, an additional constraint has to be enforced, namely that whenever forming $!M$, the subterm M has to have at most one free variable. From an axiomatic viewpoint, this corresponds to switching to the following schemes

$$!A \multimap !A \otimes !A;$$
$$!A \multimap 1;$$
$$!A \multimap \S A.$$

The constraint about boxes having at most one free variable corresponds to the failure of the following axiom

$$!A \multimap !B \multimap !(A \otimes B)$$

which instead holds in **SLL** and **ELL**. The obtained logical system characterizes the polynomial time computable functions:

Theorem (Asperti and Roversi 2002; Girard 1998). *The logic* **LLL** *characterizes the class* FP.

Light linear logic has been the first truly implicit characterization of the polytime computable functions as a fragment of linear logic, and as such a breakthrough contribution to the field. The original formulation by Girard (1998) was actually different and more complex than the one we have sketched, essentially due to the presence of so-called additive connectives, which were initially thought to be necessary to encode polytime machines. A much simpler system, called *light affine logic* (**LAL** in the following, and due to Asperti 1998; Asperti and Roversi 2002), has been proved complete for polytime functions by Roversi (1999) relying on the presence of free weakening, namely the axiom $A \multimap 1$. In other words, any object can be discarded, even if not lying inside a box.

Light linear logic and light affine logic have been analyzed from many different perspectives. Stemming from the already mentioned work on type systems for **ELL**, various type systems have been introduced being inspired by **LAL** but whose underlying language is the pure untyped λ-calculus. This includes works by Baillot (2002), Atassi et al. (2007), Coppola et al. (2008), and Baillot and Terui (2009). The difficulties here come from the fact that the information about duplicability of subterms is lost if considering the ordinary λ-calculus as the underlying language of programs. This renders type *inference*, but even subject *reduction*, problematic. The ways out vary from switching to a dual version of the underlying logic to considering call-by-value rather than call-by-name reduction. Interestingly, Murawski and Ong (2004) proved that light affine logic and safe recursion are, although equivalent from an extensional perspective, incomparable as for their intensional expressive power.

4.4 Bounded linear logic

Light linear logic and soft linear logic are both extensionally complete for the polynomial time computable functions, but are very poor as for the class of *algorithms* they can capture. Their type languages, indeed, are very simple and only reflect some qualitative information about the iterative structure of the underlying term. As a consequence, many algorithmic patterns (including, again, certain forms of nested recursion) are simply ruled out. In other words, the situation is quite similar to the one we described in Section 3.2, when talking about **HOSLR**. How could one catch more algorithms staying within the realm of linear logic? How could one get the kind of quantitative refinement needed for a more powerful methodology?

An earlier system, introduced by Girard et al. (1992), is Bounded Linear Logic, **BLL**. As we will see soon, **BLL** formulas contain some genuinely quantitative information about the input-output behavior of programs and as such can be seen as being fundamentally different from light or soft linear logic. After its inception, however, **BLL** has remained in oblivion for at least ten years, as observed by Hofmann (2000*a*):

> *[Bounded Linear Logic] allows for the definition of all polynomial time computable functions [...], where enforcement of these bounds is intrinsically guaranteed by the type system and does neither rely on external reasoning nor an ad hoc solution [...]. Unfortunately, BLL has received very little attention since its publication; a further elaboration might prove worthwhile*

Indeed, when seen as a type system for the λ-calculus, **BLL** guarantees that typable terms *both* satisfy some quantitative constraints about the relation between the input and output size *and* can be computed in a polynomial amount of time. The former is explicit in formulas, while the latter is somehow hidden, but emerges in a natural way. Indeed, the two play well together when proving that polynomial time computability is compositional.

But how is **BLL** actually defined? Rather than relying on the comonad !, it is based on a graded version of it parameterized on the natural numbers, for which the four familiar axioms from linear logic become the following ones:

$$!_{n+m}A \multimap !_nA \otimes !_mA;$$

$$!_nA \multimap 1;$$

$$!_{1+n}A \multimap A;$$

$$!_{nm}A \multimap !_n!_mA;$$

where n and m are natural numbers. In other words, duplicable objects are handled *without* dropping any of the four axioms, but taking them in quantitative form. To reach completeness, a slightly more general formulation needs to be considered, namely one in which n and m are multivariate polynomials (i.e., the so-called resource polynomials, inspired by Girard's earlier work on finite dilators) rather than plain natural numbers. This way, for example, contraction turns out to have the following form:

$$!_{x<p+q}A \multimap !_{x<p}A \otimes !_{y<q}A[x \leftarrow y+q].$$

Intuitively, the formula $!_{x<p}A$ can thus be read as follows:

$$A[x \leftarrow 0] \otimes A[x \leftarrow 1] \otimes \ldots \otimes A[x \leftarrow p-1].$$

This allows for an alternative and somehow richer representation of data. As an example, the usual second-order impredicative encoding of natural number

$$NAT \triangleq \forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha,$$

becomes the following one

$$NAT_p \triangleq \forall\alpha.!_{x<p}(\alpha(x) \multimap \alpha(x+1)) \multimap \alpha(0) \multimap \alpha(p),$$

which is parameterized by a polynomial p and as such can be assigned only to (Church representation) of natural numbers of bounded value. This, by the way, allows for a very liberal form of fold combinator. This is indeed one of the main ingredients towards the following:

Theorem (Girard et al. 1992). *The logic* **BLL** *characterizes* FP.

Subsequently to the inception of **BLL**, Pitt (1994) introduced a term calculus in which the computational and programming aspects of it were very thoroughly analyzed. But after that, not so much happened for some years, except for the alternative, realizability-based proof of polytime soundness, due to Hofmann and Scott (2004). After a ten year hiatus in which research was rather directed towards simpler characterizations of complexity classes like light or soft linear logic, the interest around **BLL** has been rising again for ten years now, due to the higher *intensional* expressive power this system offers. In particular, **BLL** has been shown by Dal Lago and Hofmann (2010*a*) to subsume many heterogeneous ICC systems, while a system of linear dependent types derived from **BLL** has been proved by Dal Lago and Gaboardi (2011) and Dal Lago and Petit (2013, 2014) to be relatively complete as a way to analyze the complexity of evaluation of programs.

4.5 Some further lines of work

Apart from the subsystems of linear logic we have mentioned so far, others have been the subject of some investigation by the ICC community.

What happens, as an example, if duplication and erasure are simply forbidden? As already mentioned in Section 2, one ends up in a purely affine λ-calculus in which only the linear abstraction is available:

$$M ::= x \;\big|\; \lambda\ell x.M \;\big|\; MM$$

From a purely proof-theoretical perspective, this system is known as Multiplicative Linear Logic, **MLL**. As we mentioned above, the size of the underlying term (or proof) strictly decreases along any (linear) β-reduction sequence, and thus, reduction cannot take time significantly higher than the size of the starting term, thus of the input. This of course prevents complexity classes like FP to be captured by the logic the usual way, that is, by stipulating that any function $f \in$ FP has to be represented uniformly as a single term M_f. On the other hand, a deep connection between **MLL** proofs and boolean *circuits* have been discovered by Mairson (2004), and Mairson and Terui (2003), and later strengthened by Terui (2004*b*), whose work highlights a deep connection between the *depth* in boolean circuits and the *logical depth* of cut-formulas in proofs.

A related line of work is about linear approximations and parsimony, and is due to Mazza and co-authors. The original idea by Mazza (2014) stems from the so-called Approximation Theorem (well-known since the original work by Girard 1987) and sees arbitrary computations being approximated by affine ones, namely by (families of) terms in the linear λ-calculus. This allows to capture non-uniform complexity classes like P/poly and L/poly, as proved by Mazza and Terui (2015), but also of uniform classes like LOGSPACE, (see Mazza 2015).

A somewhat different line of work which is certainly worth being mentioned is the one by Baillot and Mazza (2010) on linear logic *by levels*, in which the stratification constraint intrinsic to **LLL** and **ELL** is replaced by a more flexible mechanism which allows to capture more algorithmic schemes, while still allowing for complexity bounds. Remarkably, linear logic by level is permissive enough to capture all (propositional, multiplicative, and exponential) linear logic proofs, as shown by Gaboardi et al. (2009), thus going beyond systems like **ELL**.

One last series of results which is worth being mentioned is concerned with linear logic and infinitary and coinductive data. The paper by Gaboardi and Péchoux (2015) on algebras and coalgebras in the light affine λ-calculus shows that the kind of distributive laws which are necessary to model coalgebras are compatible with the polytime bounds which are the essence of **LAL**. Infinitary data structures can however be captured differently, namely by going towards something similar to the so-called infinitary λ-calculus, see Dal Lago (2016). Interestingly, the decomposition linear logic provides offers precisely the kind of tools one needs to guarantee properties like confluence and productivity. Remarkably, all this amounts to defining yet another fragment of linear logic, namely the one, called **4LL** in which only dereliction is forbidden.

5. Other Approaches and Results

Up to now, we have focused our attention on characterizations of complexity classes by way of type systems and fragments of linear logic. This, however, does not mean that no other approaches exist, even within the realm of higher-order programming languages or proof theory. This section is meant to somehow complete the picture by giving pointers to alternative approaches, all within logic and programming language theory.

First of all, one has certainly to mention arithmetic, which can of indeed be seen as a way to represent (recursive) functions, but also to prove them total. The provably total functions of Peano Arithmetic form a huge class (*mutatis mutandis*, the same captured by Gödel's **T**), but refinements of them capturing complexity classes, also known as systems of *bounded* arithmetic, have been known since the pioneering work by Parikh (1971) and have later been studied in depth especially by Buss (1986). Despite their being able to capture the various levels of the polynomial hierarchy, systems of bounded arithmetic cannot be said to be *truly implicit*, given that explicit polynomial bounds are present in their syntax, in particular at the level of quantifiers. This is also apparent in $\mathbf{PV}^{\omega}$, a typed λ-calculus whose terms are meant to be realizers of bounded arithmetic formulas, and which has been used by Cook and Urquhart (1993) to prove bounded arithmetic systems to be sound characterizations of complexity classes. Along the same lines, but somehow reverse-engineering some of the type systems for higher-order recursion we described in Section 3, one can define systems of feasible arithmetic, as shown by Bellantoni and Hofmann (2002), and Aehlig et al. (2004). Similarly, light affine logic has been the starting point for defining a system of feasible set theory by Terui (2004*a*), later adapted to soft linear logic by McKinley (2008).

Providing higher-order programming languages with satisfactory forms of denotational semantics has been an active research topic for many decades, starting from the late sixties (see the textbook by Amadio and Curien (1998) for a modern introduction to the subject). Given the elusive nature of complexity classes, about which very few separation results are currently known, it is natural to look for an appropriate mathematical framework in which languages characterizing mainstream classes can be interpreted. This observation has triggered many investigations about the denotational semantics of ICC systems, and in particular of those we have introduced in this paper. Coherent and relational semantics, the preferred ways of giving meaning to linear logic proofs, have been adapted to logical systems such as **LLL** by Baillot (2004), and **SLL** and **ELL** by Laurent and Tortora de Falco (2006). An abstract approach to stratification has been proposed by Boudes et al. (2015). Another semantic framework which provides useful insights on the interplay of duplication and complexity is certainly the Geometry of Interaction, due to Girard (1988), which can be seen both as a denotational semantics *and* as an implementation mechanism. In the former sense, it provides a fine-grain tool for the analysis of the complexity of reduction, this way allowing for alternative, more compact proofs of soundness, as shown by Dal Lago (2009*a*). In the latter sense, it has been proved to induce space-efficient implementations, thus enabling the characterization of sub-linear complexity classes, as shown by Schöpp (2007), Dal Lago and Schöpp (2010, 2016).

6. Conclusion

This paper is meant to be an introduction to implicit computational complexity and, in particular, to how complexity can be tamed in higher-order programs. Even if the scope of the survey is relatively restricted, there are certain areas which we have deliberately left out, the most prominent example being the one on characterizations of higher-*type* complexity classes: there seem to be intriguing relations between some of the characterizations we presented here and higher-type complexity classes, which are however not completely understood yet.

The present survey is by no means meant to be comprehensive, and it was written to be a memory of Martin Hofmann's contributions to ICC, and an ideal follow-up to his survey (Hofmann 2000*a*), twenty years after its appearance.

References

Aehlig, K., Berger, U., Hofmann, M. and Schwichtenberg, H. (2004). An arithmetic for non-size-increasing polynomial-time computation. *Theoretical Computer Science* **318** (1–2) 3–27.

Amadio, R. M. and Curien, P. (1998). *Domains and Lambda-Calculi*, Cambridge, Cambridge University Press.

Arora, S. and Barak, B. (2009). *Computational Complexity - A Modern Approach*, Cambridge, Cambridge University Press.

Asperti, A. (1998). Light affine logic. In: *Proceedings of LICS 1998*, IEEE Computer Society, 300–308.

Asperti, A., Coppola, P. and Martini, S. (2004). (optimal) duplication is not elementary recursive. *Information and Computation* **193** (1) 21–56.

Asperti, A. and Roversi, L. (2002). Intuitionistic light affine logic. *ACM Transactions on Computational Logic* **3** (1) 137–175.

Atassi, V., Baillot, P. and Terui, K. (2007). Verification of ptime reducibility for system F terms: Type inference in dual light affine logic. *Logical Methods in Computer Science* **3**(4) 1–32.

Avanzini, M. and Dal Lago, U. (2018). On sharing, memoization, and polynomial time. *Information and Computation* **261** (Part) 3–22.

Baillot, P. (2002). Checking polynomial time complexity with types. In: *Proceedings of IFIP TCS 2002*, vol. 223, Kluwer, 370–382.

Baillot, P. (2004). Stratified coherence spaces: A denotational semantics for light linear logic. *Theoretical Computer Science* **318** (1–2) 29–55.

Baillot, P. (2015). On the expressivity of elementary linear logic: Characterizing ptime and an exponential time hierarchy. *Information and Computation* **241** 3–31.

Baillot, P., Coppola, P. and Dal Lago, U. (2011). Light logics and optimal reduction: Completeness and complexity. *Information and Computation* **209** (2) 118–142.

Baillot, P. and Mazza, D. (2010). Linear logic by levels and bounded time complexity. *Theoretical Computer Science* **411** (2) 470–503.

Baillot, P. and Mogbil, V. (2004). Soft lambda-calculus: A language for polynomial time computation. In: *Proceedings of FOSSACS 2004*, LNCS, vol. 2987, Springer, 27–41.

Baillot, P. and Pedicini, M. (2001). Elementary complexity and geometry of interaction. *Fundamenta Informaticae* **45** (1–2) 1–31.

Baillot, P. and Terui, K. (2009). Light types for polynomial time computation in lambda calculus. *Information and Computation* **207** (1) 41–62.

Bellantoni, S. and Cook, S. A. (1992). A new recursion-theoretic characterization of the polytime functions. *Computational Complexity* **2** 97–110.

Bellantoni, S. and Hofmann, M. (2002). A new “feasible" arithmetic. *Journal of Symbolic Logic* **67** (1) 104–116.

Bellantoni, S. J., Niggl, K. and Schwichtenberg, H. (2000). Higher type recursion, ramification and polynomial time. *Annals of Pure and Applied Logic* **104** (1–3) 17–30.

Boudes, P., Mazza, D. and Tortora de Falco, L. (2015). An abstract approach to stratification in linear logic. *Information and Computation* **241** 32–61.

Buss, S. (1986). *Bounded Arithmetic*, Naples, Bibliopolis.

Cappai, A. and Dal Lago, U. (2015). On equivalences, metrics, and polynomial time. In: *Proceedings of FCT 2015*, LNCS, vol. 9210, Springer, 311–323.

Cobham, A. (1965). The intrinsic computational difficulty of functions. In: Bar-Hillel, Y. (ed.) *Logic, Methodology and Philosophy of Science, Proceedings of the 1964 International Congress*, North-Holland, 24–30.

Cook, S. A. and Urquhart, A. (1993). Functional interpretations of feasibly constructive arithmetic. *Annals of Pure and Applied Logic* **63** (2) 103–200.

Coppo, M. and Dezani-Ciancaglini, M. (1980). An extension of the basic functionality theory for the lambda-calculus. *Notre Dame Journal of Formal Logic* **21** (4) 685–693.

Coppola, P., Dal Lago, U. and Rocca, S. R. D. (2008). Light logics and the call-by-value lambda calculus. *Logical Methods in Computer Science* **4** (4) 1–28.

Coppola, P. and Martini, S. (2006). Optimizing optimal reduction: A type inference algorithm for elementary affine logic. *ACM Transactions on Computational Logic* **7** (2) 219–260.

Dal Lago, U. (2009a). Context semantics, linear logic, and computational complexity. *ACM Transactions on Computational Logic* **10** (4) 25:1–25:32.

Dal Lago, U. (2009b). The geometry of linear higher-order recursion. *ACM Transactions on Computational Logic* **10** (2) 8:1–8:38.

Dal Lago, U. (2016). Infinitary lambda calculi from a linear perspective. In: *Proceedings of LICS 2016*, ACM, 447–456.

Dal Lago, U. and Baillot, P. (2006). On light logics, uniform encodings and polynomial time. *Mathematical Structures in Computer Science* **16** (4) 713–733.

Dal Lago, U. and Di Giamberardino, P. (2016). On session types and polynomial time. *Mathematical Structures in Computer Science* **26** (8) 1433–1458.

Dal Lago, U. and Gaboardi, M. (2011). Linear dependent types and relative completeness. *Logical Methods in Computer Science* **8** (4) 1–44.

Dal Lago, U. and Hofmann, M. (2010a). Bounded linear logic, revisited. *Logical Methods in Computer Science* **6** (4) 1–31.

Dal Lago, U. and Hofmann, M. (2010b). A semantic proof of polytime soundness of light affine logic. *Theory of Computing Systems* **46** (4) 673–689.

Dal Lago, U. and Hofmann, M. (2011). Realizability models and implicit complexity. *Theoretical Computer Science* **412** (20) 2029–2047.

Dal Lago, U., Martini, S. and Sangiorgi, D. (2016). Light logics and higher-order processes. *Mathematical Structures in Computer Science* **26** (6) 969–992.

Dal Lago, U., Masini, A. and Zorzi, M. (2010). Quantum implicit computational complexity. *Theoretical Computer Science* **411** (2) 377–409.

Dal Lago, U. and Petit, B. (2013). The geometry of types. In: *Proceedings of POPL 2013*, ACM, 167–178.

Dal Lago, U. and Petit, B. (2014). Linear dependent types in a call-by-value scenario. *Science of Computer Programming* **84** 77–100.

Dal Lago, U. and Schöpp, U. (2010). Functional programming in sublinear space. In: *Proceedings of ESOP 2010*, LNCS, vol. 6012, Springer, 205–225.

Dal Lago, U. and Schöpp, U. (2016). Computation by interaction for space-bounded functional programming. *Information and Computation* **248** 150–194.

Danos, V. and Joinet, J. (2003). Linear logic and elementary time. *Information and Computation* **183** (1) 123–137.

Fortune, S., Leivant, D. and O’Donnell, M. (1983). The expressiveness of simple and second-order type structures. *Journal of the ACM* **30** (1) 151–185.

Gaboardi, M., Marion, J. and Ronchi Della Rocca, S. (2012). An implicit characterization of PSPACE. *ACM Transactions on Computational Logic* **13** (2) 18:1–18:36.

Gaboardi, M. and Péchoux, R. (2015). Algebras and coalgebras in the light affine lambda calculus. In: *Proceedings of ICFP 2015*, ACM, 114–126.

Gaboardi, M. and Ronchi Della Rocca, S. (2007). A soft type assignment system for *lambda* -calculus. In: *Proceedings of CSL 2007*, LNCS, vol. 4646, Springer, 253–267.

Gaboardi, M., Roversi, L. and Vercelli, L. (2009). A by-level analysis of multiplicative exponential linear logic. In: *Proceedings of MFCS 2009*, LNCS, vol. 5734, Springer, 344–355.

Girard, J. (1987). Linear logic. *Theoretical Computer Science* **50** 1–102.

Girard, J. (1998). Light linear logic. *Information and Computation* **143** (2) 175–204.

Girard, J., Scedrov, A. and Scott, P. J. (1992). Bounded linear logic: A modular approach to polynomial-time computability. *Theoretical Computer Science* **97** (1) 1–66.

Girard, J.-Y. (1988). Geometry of interaction I: An interpretation of system F. *Logic Colloquium* **127** 221–260.

Girard, J.-Y., Taylor, P. and Lafont, Y. (1989). *Proofs and Types*, New York, NY, USA, Cambridge University Press.

Goerdt, A. (1992a). Characterizing complexity classes by general recursive definitions in higher types. *Information and Computation* **101** (2) 202–218.

Goerdt, A. (1992b). Characterizing complexity classes by higher type primitive recursive definitions. *Theoretical Computer Science* **100** (1) 45–66.

Goldreich, O. (2008). *Computational Complexity - A Conceptual Perspective*, Cambridge, Cambridge University Press.

Hartmanis, J. and Stearns, R. (1965). On the computational complexity of algorithms. *Transactions of the American Mathematical Society* **117** 285–306.

Hofmann, M. (1997a). An application of category-theoretic semantics to the characterisation of complexity classes using higher-order function algebras. *Bulletin of Symbolic Logic* **3** (4) 469–486.

Hofmann, M. (1997b). A mixed modal/linear lambda calculus with applications to bellantoni-cook safe recursion. In: *Proceedings of CSL 1997*, LNCS, vol. 1414, Springer, 275–294.

Hofmann, M. (1999). Linear types and non-size-increasing polynomial time computation. In: *Proceedings of LICS 1999*, IEEE Computer Society, 464–473.

Hofmann, M. (2000a). Programming languages capturing complexity classes. *SIGACT News* **31** (1) 31–42.

Hofmann, M. (2000b). Safe recursion with higher types and bck-algebra. *Annals of Pure and Applied Logic* **104** (1) 113–166.

Hofmann, M. (2000c). A type system for bounded space and functional in-place update. *Nordic Journal of Computing* **7** (4) 258–289.

Hofmann, M. (2002). The strength of non-size increasing computation. In: *Proceedings of POPL 2002*, ACM, 260–269.

Hofmann, M. (2003). Linear types and non-size-increasing polynomial time computation. *Information and Computation* **183** (1) 57–85.

Hofmann, M. and Scott, P. J. (2004). Realizability models for BLL-like languages. *Theoretical Computer Science* **318** (1–2) 121–137.

Jost, S., Hammond, K., Loidl, H. and Hofmann, M. (2010). Static determination of quantitative resource usage for higher-order programs. In: *Proceedings of POPL 2010*, ACM, 223–236.

Lafont, Y. (2004). Soft linear logic and polynomial time. *Theoretical Computer Science* **318** (1–2) 163–180.

Laurent, O. and Tortora de Falco, L. (2006). Obsessional cliques: A semantic characterization of bounded time complexity. In: *Proceedings of LICS 2006*, IEEE Computer Society, 179–188.

Leivant, D. (1993). Stratified functional programs and computational complexity. In: *Proceedings of POPL 1993*, ACM Press, 325–333.

Leivant, D. (1998). A characterization of NC by tree recurrence. In: *Proceedings of FOCS 1998*, IEEE Computer Society, 716–724.

Leivant, D. (1999a). Applicative control and computational complexity. In: *Proceedings of CSL 1999*, LNCS, vol. 1683, 82–95, Springer.

Leivant, D. (1999b). Ramified recurrence and computational complexity III: Higher type recurrence and elementary complexity. *Annals of Pure and Applied Logic* **96** (1–3) 209–229.

Leivant, D. (2002). Calibrating computational feasibility by abstraction rank. In: *Proceedings of LICS 2002*, IEEE Computer Society, 345.

Leivant, D. and Marion, J. (1994). Ramified recurrence and computational complexity II: Substitution and poly-space. In: *Proceedings of CSL 1994*, LNCS, vol. 933, Springer, 486–500.

Mairson, H. G. (2004). Linear lambda calculus and ptime-completeness. *Journal of Functional Programming* **14** (6) 623–633.

Mairson, H. G. and Terui, K. (2003). On the computational complexity of cut-elimination in linear logic. In: *Proceedings of ICTCS 2003*, LNCS, vol. 2841, Springer, 23–36.

Mazza, D. (2006). Linear logic and polynomial time. *Mathematical Structures in Computer Science* **16** (6) 947–988.

Mazza, D. (2014). Non-uniform polytime computation in the infinitary affine lambda-calculus. In: *Proceedings of ICALP 2014*, LNCS, vol. 8573, Springer, 305–317.

Mazza, D. (2015). Simple parsimonious types and logarithmic space. In: *Proceedings of CSL 2015*, LIPIcs, 24–40.

Mazza, D. and Terui, K. (2015). Parsimonious types and non-uniform computation. In: *Proceedings of ICALP 2015*, LNCS, vol. 9135, Springer, 350–361.

McKinley, R. (2008). Soft linear set theory. *Journal of Logical and Algebraic Methods in Programming* **76** (2) 226–245.

Mitchell, J. C., Mitchell, M. and Scedrov, A. (1998). A linguistic characterization of bounded oracle computation and probabilistic polynomial time. In: *Proceedings of FOCS 1998*, IEEE Computer Society, 725–733.

Murawski, A. S. and Ong, C. L. (2004). On an interpretation of safe recursion in light affine logic. *Theoretical Computer Science* **318** (1–2) 197–223.

Neergaard, P. M. (2004). A functional language for logarithmic space. In: *Proceedings of APLAS 2004*, LNCS, vol. 3302, Springer, 311–326.

Oitavem, I. (2008). Characterizing PSPACE with pointers. *Mathematical Logic Quarterly* **54** (3) 323–329.

Oitavem, I. (2011). A recursion-theoretic approach to NP. *Annals of Pure and Applied Logic* **162** (8) 661–666.

Parikh, R. J. (1971). Existence and feasibility in arithmetic. *Journal of Symbolic Logic* **36** 494–508.

Pitt, F. (1994). *The Bounded Linear Calculus: A Characterization of the Class of Polynomial-Time Computable Functions Based on Bounded Linear Logic*. Master's thesis, University of Toronto, Department of Computer Science.

Redmond, B. F. (2007). Multiplexor categories and models of soft linear logic. In: *Proceedings of LFCS 2007*, vol. 4514, LNCS, Springer, 472–485.

Redmond, B. F. (2015). Polynomial time in the parametric lambda calculus. In: *Proceedings of TLCA 2015*, LIPIcs, vol. 38, 288–301.

Redmond, B. F. (2016). Bounded combinatory logic and lower complexity. *Information and Computation* **248** 215–226.

Roversi, L. (1999). A PTIME completeness proof for light logics. In: *Proceedings of CSL 1999*, LNCS, vol. 1683, Springer, 469–483.

Schöpp, U. (2007). Stratified bounded affine logic for logarithmic space. In: *Proceedings of LICS 2007, Proceedings*, 411–420.

Sørensen, M. H. and Urzyczyn, P. (2006). *Lectures on the Curry-Howard Isomorphism,* New York, NY, USA, Elsevier Science Inc.

Terui, K. (2004a). Light affine set theory: A naive set theory of polynomial time. *Studia Logica* **77** (1) 9–40.

Terui, K. (2004b). Proof nets and boolean circuits. In: *Proceedings of LICS 2004*, 182–191.

Terui, K. (2007). Light affine lambda calculus and polynomial time strong normalization. *Archive for Mathematical Logic* **46** (3–4) 253–280.

Terui, K. (2012). Semantic evaluation, intersection types and complexity of simply typed lambda calculus. In: *Proceedings of RTA 2012*, LIPIcs, 323–338.

Zaionc, M. (1990). A characterisation of lambda definable tree operations. *Information and Computation* **89** (1) 35–46.

Zaionc, M. (1991). lambda-definability on free algebras. *Annals of Pure and Applied Logic* **51** (3) 279–300.

Zhang, Y. (2009). The computational SLR: A logic for reasoning about computational indistinguishability. In: *Proceedings of TLCA 2009*, 401–415.

Cite this article: Dal Lago U (2022). Implicit computation complexity in higher-order programming languages. *Mathematical Structures in Computer Science* **32**, 760–776. https://doi.org/10.1017/S0960129521000505