2020-05-15

An Intuitive Look at FP Soundness

Rose R. Silver

Worcester Polytechnic Institute

An Intuitive Look at FP Soundness

by

Rose Rensselaer Silver

A Major Qualifying Project

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

in Computer Science

by

May 2020

APPROVED:

Therese Mary Smith
Joshua M Cuneo

Abstract

What is efficiently computable? What can these programs describe? These questions are at the focal point of complexity theory, and find theoretical roots in implicit computational complexity theory. It is widely established that the program complexity class of functions whose runtimes are polynomial with respect to their input is considered tractable or efficient. This thesis establishes an intuitive look at pattern expansion, runtime expansion, and an architecture agnostic programming language sound in FP. This language is constrasted with logics known to be both sound and complete for FP and finally the idea of the all-encompassing or universal algorithm is considered in FP over an FP bounded language. Is there a program which can compute every problem solvable in polynomial time in polynomial time?

Acknowledgements

Jessica Greenleaf

The love of my life and the most patient human on earth. Thank you for supporting me through far too many cups of coffee.

Contents

List of Figures

0.1 Polynomial Time Complexity (FP)

Poly-Time is the complexity class containing all programs whose runtimes can be described, or upper-bounded, by a polynomial. [Ric08] Commonly referred to as “tractable” or “efficient”, this complexity class is considered the class of problems which can feasibly be solved on some deterministic computational device. FP is formally described as:

> A binary relation $P(x, y)$ is in FP if and only if there is a deterministic polynomial time algorithm that, given x, can find some y such that $P(x, y)$ holds.

A good understanding of what FP describes in terms of what programs are allowed to do and stay within FP is important to understanding the transformations covered by the complexity class. To begin with, computer architecture seems to play a role. A computer only capable of incrementing a number by one every time step would take an exponential amount of time to compute the function $f(x) = 2^x$. However, a much more powerful computer which has a multiply instruction could compute this function in linear time. It appears that two different computers can solve the same problem in different time bounds. Without the ability to divorce computation from architecture, a unified concept of what functions are reachable in FP time is impossible to create.

0.2 Implicit Computational Complexity (ICC)

The fundamental question that must be solved is ”What takes time?” The computer with a multiply instruction takes less time than the computer with only the successor function, but under the surface there surely must be an at least comparable concept of an atomic time step [DL19]. A natural way to measure computational complexity without the consideration of technology is to use idealized models of computation like turing machines or lambda calculus. Lambda calculus is turing complete, capable of computing every computable function [Pol11]. This provides an extremely strong starting point for describing the atomic units of computation in a turing-complete manner, while never actually touching the architecture. This is because lambda-calculus describes transformations and idealized functions. The actual resources available to the machine are irrelevant. While this framework describes a complete view of computation, its unrestrictive structure allows programs whose runtimes are exponential with respect to the independent variable and programs that do not halt. A common example is the omega combinator:

$$(\lambda x.(xx) \quad \lambda x.(xx))$$

To illustrate the nature of this combinator it is commonly rewritten:

$$\Omega := (\omega \quad \omega)$$

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

This divergent combinator applies its inner term ω to itself. In turn ω creates the same shape as before (Ω), creating an infinite loop of replacement. This illustrates the existence of dangerous, and non-polynomial, programs which are still considered syntactically correct. It would be useful to create an idealized system of computation whose bounds of execution are already polynomial, restricting the runtime of all syntactically correct programs while still defining the idealized system of computation in which to root the question of "what takes time during computation?".

The most basic typed system of lambda calculus, aptly named the Simply Typed

Lambda Calculus (STLC), holds the same reasoning power and operations as intuitionistic propositional logic [BDPR01]. Again, through the Curry-Howard isomorphism, not only are these two concepts similar, they are identical to each other [BDPR01, Ili13]. A comparison of STLC terms and logic rules are below:

$$\text{(Var)} \qquad \Gamma, x{:}\tau \vdash x : \tau$$

$$\text{(Abs)} \qquad \frac{\Gamma, x{:}\sigma \vdash M : \tau}{\Gamma \vdash (\lambda x{:}\sigma\, M) : \sigma \to \tau}$$

$$\text{(App)} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau}$$

Figure 1: Isomorphism between STLC and IPL

STLC is also strongly normalizing, meaning that all valid programs halt eventually [Ter12]. While exponentially complex programs still exist within this structural framework, programs are at least guaranteed to end. While logics exist specifically for the purpose of describing ICC [DL11, Gir98, Laf04], these will be discussed after building an equivalent language structural restriction, and will be used to describe the equivalency between the two constructs, and to prove soundness. For now, STLC's grammar rules will be used to loosely describe the concept of ICC to be used from here on.

0.3 Poly-Time Bounds

The first step to successfully describing Poly-Time restriction is to examine what, exactly, a polynomial runtime means, what it entails, and the minimum actions required to describe it. It needs to be deconstructed in its simplest sense and then reconstructed to a full generalization of the valid problem space. The first simplification that can be made is found by first examining the structure of a functional program. The process towards normalization of an STLC term is through repeated application until no more application rules can be executed [DL19]. Lambda abstractions, constants and variables don't take any time of their own accord, only the act of application transforms or moves the expression in any way. Expectedly, application correlates directly to the cut rule of linear logics [Gir98]. With this observation, the simplest way of creating an ICC framework is to count the number of applications a term requires before it becomes normalized [DL19].

A polynomial is a series of terms of the form:

$$Ax^n \pm Bx^{n-1} ... \pm Zx^0$$

Any program which normalizes in poly-time must contain a polynomial number of applications to reach normalization. STLC is, as discussed, too unrestricted for the uses of limiting programs to only poly-time. Instead of attempting to find and remove every case which violates the runtime constraint, an empty framework will be slowly enriched in STLC-like fashion until only polynomial time complexity is allowed by constructs of the language. In its most basic state, this new language $L1$ requires only two constructs:

$$L1 := c \mid (e : \tau_2 \rightarrow \tau_1 \quad e : \tau_2) : \tau_1$$

Notice that lambda abstractions do not exist, and neither do variables. Additionally, the rule for application has been restricted to applying an expression to a function type expression which accepts the argument's type. This rule, Typed-Application (TA), restricts valid terms to those which normalize to a value, also known as well-typed terms [Ter12]. TA removes absurd applications like ($true$ 3) from the language. While there is nothing syntactically wrong with the term $(true3)$ within STLC, and in fact it is already in its normalized state, terms of such a form are rarely useful because they don't describe values or results of a relationship, they instead exist as incompressible terms frozen in time, seemingly part way through their computation. Removing these normalized but application-including terms isn't strictly necessary; it is a clarity choice which does not affect the complexity of the language [Ter12]. c describes the constants of the language. This includes primitive functions and value constants. For a first attempt at picking apart the shape and relationship between program and runtime, $c := ZERO : Int \mid INC : Int \to Int$.

0.3.1 syntax

Before building on this toy language and experimenting with its capabilities, a brief overview of its syntax is required, as it is modified slightly from pure lambda calculus.

Atomic instructions are shown as a labeled box. If they are functions, they consume the value to their right and return the transformed value to the left. This action is equivalent to TA and takes one time step. When written vertically, this flow is from top to bottom. Arrows may be included for clarity in the direction of argument to function. A string of atoms is called a block.

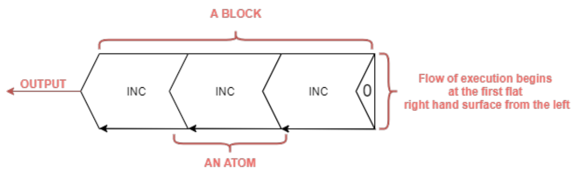
A BLOCK
OUTPUT
INC
INC
INC
0
Flow of execution begins at the first flat right hand surface from the left
AN ATOM

Figure 2: A block of atoms

It is easy to see that the number of function-type atoms in a program is equal to the time the program will take to run, as every atom resolves in one time step and the well-typed guarantee proves that the final normalized form will be a value output from the last function-type atom to resolve.

Theorem 0.3.1 (length-runtime relation) *The runtime of a program is equal to the number of function-type atoms it contains. This is always upper-bounded by the number of atoms comprising the program.*

0.4 Playing with the Toy Language

Programs are treated as simulations of functions. For now, only binary relations will be explored, leaving the generalization to multiple dimensions as an expansion on the final language. To examine the expressive power of the language $L1$, a sample program is stepped through below:

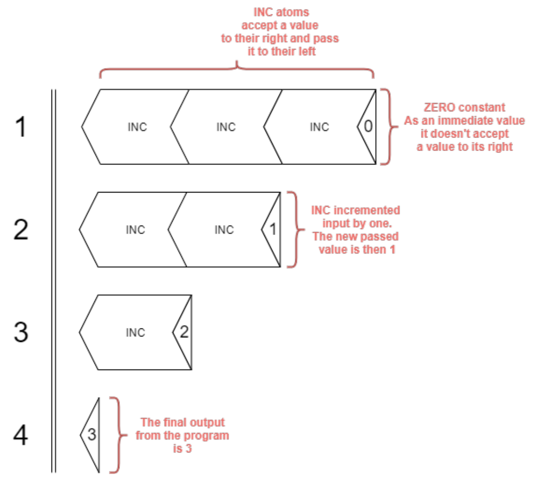
INC atoms accept a value to their right and pass it to their left
1
INC
INC
INC
0
ZERO constant As an immediate value it doesn't accept a value to its right
2
INC
INC
1
INC incremented input by one. The new passed value is then 1
3
INC
2
4
3
The final output from the program is 3

Figure 3: Step-through of THREE := [INC][INC][INC][ZERO]

The program $THREE$ normalizes in three time steps and expresses the value 3 itself. This supports the length-runtime theorem. The property of equality between the number a program describes and the runtime of the program is intentional, and particularly useful, as runtime and value are conflated at least in this very minimal language.

Theorem 0.4.1 (proportional computation) *The expression $e(n)$, computing the natural number n can be computed in $O(n)$ time by $L1$ for any positive n.*

Sketch Proof 0.4.1 *For all natural numbers n:*

$e(0) \Longrightarrow ZERO$

this is a normalized, well-typed expression. No computation takes place and the halting state is equal to the desired output. No time has been taken, therefore n = 0 = O(n)

$e(n > 0) \Longrightarrow (INC \quad e(n-1))$

this expression requires n steps of computation, as the base case for the recursive formula e is $ZERO$ when $n = 0$ and the inductive step $e(n-1)$ adds one typed application of INC for every recursive replacement. Therefore the starting expression will contain n calls to INC, n applications, and run in $O(n)$ time. This expression is well-typed as it is the application of n $Int \to Int$ function-type constants to the constant $ZERO$, which has type Int. It is obvious to see that at every step the expression is still well formed, as the application of $Int \to Int$ to Int results in type Int, which is in turn accepted by the next call to INC.

This can be thought of as a stronger version of the length-runtime relation for integers. This theorem only holds when the language's one atomic construct for which time steps are counted is INC, as every time step is equivalent to one step on the proverbial "number line".

Theorem 0.4.2 (runtime-length-value equivalency) *Using language $L1$: Due to the proportional computation theorem, n can be computed in $O(n)$ time. Conversely, if a program runs in $O(n)$ time, it must compute n.*

Sketch Proof 0.4.2 *This theorem is true due mostly to the extreme restriction of $L1$. As shown by the proportional runtime theorem, $L1$ can only compute positive in-*

tegers and does so in time equal to their magnitude. The strength of this relationship makes it reversible.

0.4.1 C

So far, the language can only describe finite natural numbers, but computes them in time equal to the magnitude of the number. In other words it computes n in $O(n)$. Without capturing the independent variable of the binary relation this simulates, constant time computation is expectedly the only form of program that can take place. Constants are a subsection of polynomials, meaning the language is still bounded by polynomial time, however it lacks the power to describe all of FP. Programs can only simulate relations of the form $f(x) = c$.

0.4.2 $x + C$

The next step is to capture the argument of the binary relation. This atom can be thought of as a restricted version of variables introduced by lambda abstractions. By including the atom $X : Int$, a value set before execution to a concrete number equal to the argument of the relation, the language can not only compute n as shown above, but also $x + n$ in $O(n)$ time. This is trivially proven by the fact that X is precomputed to a value, all integers are processed the same way by INC, and don't require time-steps to resolve to a value – they already are a value.

In terms of time complexity, all programs still execute in constant time. This is a break from the proportional computation theorem, as values can be computed in time less than their magnitude. To keep with the proportional computation theorem the computation of x must take x time. This can't happen yet because even with the inclusion of the independent variable, programs cannot alter their own structure or length. As given by the length-runtime relation theorem, the length of a program is directly related to its execution time. This expanded language can simulate functions of the form $f(x) = x + C$ and normalizes the program in $O(C)$ time. Ideally, keeping the proportional computation theorem true lets the distinction between described function classes and runtime disappear again. $f(x) = x + C$ should be computed in $O(x + C)$ time.

To modify runtime with respect to x, and the program length must change to reflect this on each instance of the function. A new language construct must be included to capture this idea of programs whose lengths can vary to be able to compute x in proportional time. This construct is $LOOP$, and it is one of the two required structural constructs. The syntax of $LOOP$ is:

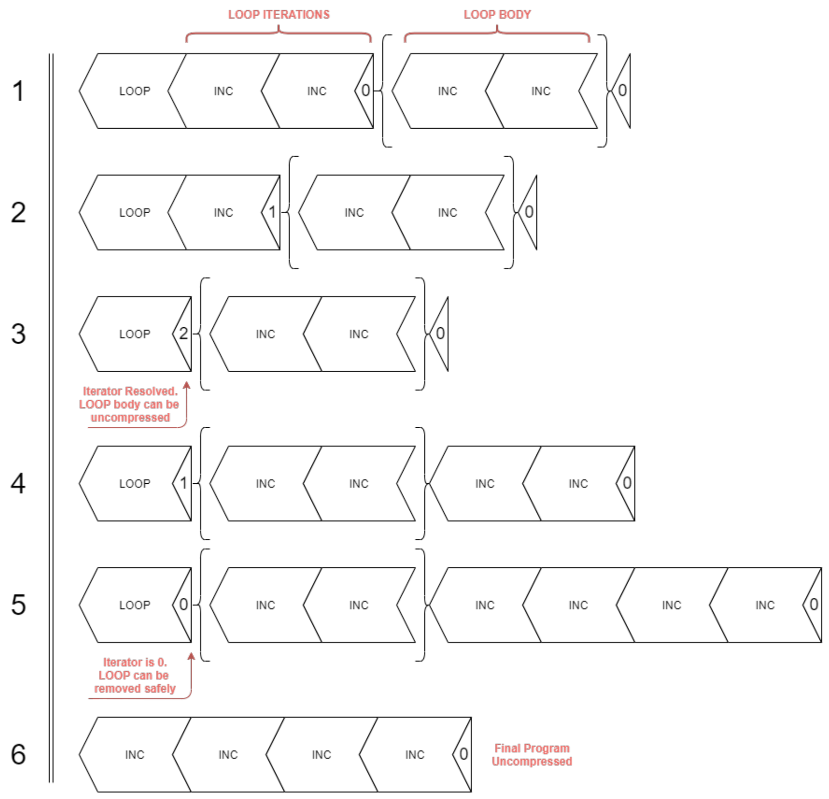

Figure 4: LOOP Decompression step-through

An interesting discovery from the $LOOP$ construct is the idea of concatenating blocks. Since the block within the $LOOP$ is applied into itself the number of times called for by the iteration count, the runtime of the resulting program is equal to the iteration count multiplied by the runtime of the captured block. Not only is this intuitive, the length-runtime relation shows that the concatenation of two blocks into one is equal in runtime to the sum of the runtimes of each block independently. Succinctly, $LOOP(A)\{B\}$ runs in $O(A * B)$ time.

By re-restricting the X atom to use only within the iteration count of $LOOP$, x can be simulated to be computed in $O(x)$ time by the program:

$$LOOP(X)\{[INC]\}[ZERO]$$

Now relations of the form $f(x) = x+C$ can still be simulated, the only difference is that these relations run in $O(x+C)$ time, re-unifying the proportional computation theorem. This new language is:

$$L2 := c \mid (e : \tau_2 \to \tau_1 \quad e : \tau_2) : \tau_1$$

$$c := \{\, m \mid x := LOOP(X)\{INC\} \mid INC \mid ZERO \,\}$$

$$m := \{\, LOOP(n)\{\_\} \mid LOOP(X)\{\_\} \,\}$$

$L2$ enjoys the same runtime-length-value equivalency as $L1$, as $LOOP$ is simply a pattern compression meta-construct and x is computed in $O(X)$ time. All meta-constructs (denoted by m) are expanded before runtime and create a constant slowdown for any given program. Due to the constant nature of the slowdown, it can be ignored in asymptotic calculations.

0.4.3 Positive Coefficient Polynomials

In fact, upon closer inspection of the $LOOP$ meta-construct, all positive coefficient polynomial functions can be described. In their abstract form, positive coefficient polynomials are of the form:

$$Ax^n + Bx^{n-1} ... + Zx^0$$

Through the runtime-length-value equivalency theorem, addition is analogous to block concatenation. All that needs to be described is each term. Stitching them together into a positive polynomial is as simple as concatenating programs together. Each term is a coefficient multiplied by the independent variable raised to a finite, constant power. $a := LOOP(A)\{[INC]\}$ computes a block of INC A long and of value A when given the constant $ZERO$ as the rightmost atom for the application string. $ax := LOOP(X)\{[a]\}$ then computes a block of INC $A * X$ long. Following this logic, $LOOP(X)\{[ax]\}$ computes a block of length $A * X^2$. Clearly any finite, constant power can be constructed via sinking the coefficient within a nesting of $LOOP(X)$ equal in depth to the power. By restricting $LOOP$ depth to a constant level, and without a way for loop depth to be written in an independent variable dependent sense, exponential functions cannot be described. It is, in essence, the restriction on the depth of recursion that is so crucial to confining programs to FP-complexity [Gir98].

0.4.4 General Polynomials

It is at this moment that many of the grounding theorems must sadly become invalidated. To describe all polynomials, subtraction must finally be implemented. A polynomial takes the general form:

$$Ax^n \pm Bx^{n-1} ... \pm Zx^0$$

Each term is either positive or negative, and positive terms have already been constructed; all that remains are the negative terms. These can be constructed by including an atom to $L2$'s constants, $DEC : Int \rightarrow Int$. DEC decreases the input value by one. By substituting a positive term construction's INC within its $LOOP$ nesting with DEC, this term expands to be a decrement to a value of the size called for by the term. Unfortunately, programs' runtimes are now not always equal to the value they report, but their runtime can be found by turning the general polynomial into a positive polynomial via substitution of subtraction with addition. Since the only programmatic difference between a general polynomial and its positive counterpart is whether certain blocks are constructed of INC or DEC, polynomial runtime is not jeopardized.

No program can execute in time equal to a polynomial with a negative term, because that would imply the ability to remove atoms from the program, an ability not present in any of the languages presented here. If this were possible, DEC would not be needed, the proportional runtime theorem and the length-runtime-value relation would still hold true. The inclusion of the DEC atom changed the expressiveness of the language but not the runtime constraint. Investigating this further, it becomes evident that the actual atoms themselves do not matter. The meta-constructs change the shape of the program, and they can only change the shape in polynomial-length expansions. As long as every included atom is guaranteed to run in polynomial time with respect to the independent variable, every constructable program must admit FP-complexity. The question that remains is one of completeness. Can every FP program be written in an atom-generic version of $L2$, or are there unreachable programs which would run within the time-complexity restraint?

0.5 Handling Randomness

With a generic language containing an arbitrary collection of atomic combinators, it is no longer true that the functions describable in polynomial runtime are polynomials themselves. But within the constraint of $LOOP$, every program only describes its length expansion in terms of concatenations of blocks with lengths equal to the coefficient. Completely random blocks of polynomial length cannot be constructed. A certain function might be described as:

> for any x, $f(x)$ is computable by a random, valid, and unchanging string of atoms of length $p(x)$, where p is a polynomial function

This piece-wise function admits polynomial runtime as is evident from the length-runtime relation. p outputs a series of atoms of polynomial length and polynomial length means polynomial runtime. This function is not compressible into a collection of $LOOP$s, because there is no pattern between instances of x. Creating a program which describes f is, as the language structure stands, impossible. The solution lies in the ability to modify the content of a program based on the value of x. $SWITCH(x)\{K_1, K_2...K_{x_{max}}\}$ is the last meta-construct and allows this. $SWITCH$ takes in a number x, and replaces itself with the content of the x^{th} block within its braces (K_x). This is currently an unsafe operation that allows the polynomial time restriction to be broken. Just as this solves the issue for the example function, consider the following function:

> for any x, $f(x)$ is computable by a random, valid, and unchanging string of atoms of length $e(x)$, where e is an exponential function

A program structured as:

$$SWITCH(x)\{1 : \text{random string length } e(1), ...n : \text{random string length } e(n)\}$$

is legal within the constraint of the language structure, but grows exponentially in runtime.

0.5.1 Finite Height Programs

This issue's origin is the lack of restriction placed on the form of atomic concatenation blocks. When these blocks can be infinite in length, then simply creating any function from piece-wise instances of programs with the $SWITCH$ construct is possible. By restricting all blocks to be of finite length, all super-polynomial runtime classes are unreachable [DL11]. Using the previous exponential function as an example, it is obvious that the n^{th} term will be of length $e(n)$. The function is also known to be incompressible because of its randomness, meaning no use of $LOOP$ is possible for compression or pattern repetition. Even if the function was not completely random but still exponential, any use of $LOOP$ would still only become uncompressed when considering runtime. As $LOOP$ only compresses program length by a polynomial factor, the exponential function still wouldn't be constructable in finite length because exponentials divided by polynomials still trend towards infinity for large n . If this function's domain is bounded by a finite maximum number, the longest possible string would be of length $e(DOMAIN\_MAX)$, making the program computable in $O(e(DOMAIN\_MAX))$; a known, constant, and therefore polynomial, runtime. However, assuming the nontrivial case wherein the function's domain is nonfinite, the length of the blocks tends towards infinity at a rate of $e(n)/p$ where p is some polynomial best-guess compression. By removing the possibility of block-length trending towards infinity, this, and any other super-polynomial function cannot be simulated.

Perhaps it would appear that the random polynomial function shown previously would be impossible to simulate as it too is incompressible. This is not true. For any x, the string of length $p(x)$ is known to be polynomial in length with respect to x. A $LOOP(X)$ command compresses the length of the block created by a factor of x. For every term of $p(x)$, wrapping the simulated term within the program in a number of $LOOP$s equal to the degree of the term results in a finite, constant number, unrelated to the magnitude of x. In fact, it is equal in length to the coefficient of the term. In this way a finite block length can be obtained for polynomial programs.

This is only half the answer, as $LOOP$s impose order on the resulting program in the form of tiled concatenations of their internal blocks. In some sense, the ability to "choose" a different block of code every iteration is necessary to maintain the randomness property of the original function. Exposing the iteration of a $LOOP$ as a variable is the last key element of the language. With this variable, a $SWITCH$ can nest within a $LOOP$, switching on every iteration. The randomness feature of the function is now able to be described, and every block is of finite length. It should be noted that to maintain finite program length, in addition to finite block length, the random polynomial function is not completely computable. A finite subset of instances can be captured in a finite program, but because there is no way to predict the string computing the next instance, each instance of the function sits in its own branch of a $SWITCH$ statement, preparing to unroll to the correct polynomial length string for that instance. No compression exists to make this $SWITCH$ statement have finite branches as each branch holds no relation to any other branch.

0.5.2 The Program Tree

To answer the problem of completeness, the program space itself must be well understood. A naive approach might be to catalog every possible string of atoms and meta-constructs. Random strings of this nature can be created in $O(|L|^d)$ time, where $|L|$ is the number of atoms and meta-constructs in the language and d is the length of the string [Lev73]. This captures every possible program, but it also captures a lot of garbage that is unexecutable. To remove the unexecutable programs, a syntax graph can be used to direct flow of the random strings, only ever choosing the next symbol from a list of viable symbols at the point of program construction [Sch04]. Every valid program starts with a beginning symbol, or atom in its language. For the binary relation $f(x : A) = y : B$, these are the atoms whose return types are B. From this entrance symbol, any symbol whose return type matches the input type of the start symbol can be concatenated with it. The natural description of this data structure is a tree, where leaves symbolize the terminal symbols (value-typed nodes) and the branch is a complete program which can be executed from the leaf towards the root, and internal nodes are atoms which accept the return type of all the nodes branching from it and return a type coherent with the node it stems from.

This program tree is the beginning program state tree (T-META). Because it captures meta-constructs, each program in it containing one or more meta-constructs transforms into a program without meta-constructs at execution and when all independent variables are defined before execution time. This uncompressed version of programs can also be constructed into a tree (T-BARE). All programs in T-META, when uncompressed and considered ranging over the domain on which they operate, are tree slices of T-BARE. A branch of T-BARE is a program capable of computing an instance of a binary relation, much like $L1$, and T-META contains programs which compute whole functions. For a given branch of T-META, the branches of T-BARE it describes must be polynomial in length to the independent variable, otherwise the poly-time constraint is violated as given by the length-runtime relation.

Theorem 0.5.1 (FP Soundness) *Any finite branch of T-META decompresses to polynomial length branches of T-BARE.*

Sketch Proof 0.5.1 *The meta-constructs of T-META are* $m := \{LOOP|SWITCH\}$

$LOOP$ can only perform multiplicative expansion. Polynomials are closed under multiplication.

$SWITCH$ performs block substitution, and is runtime upper-bounded by the length of the longest potential substitution. It can be substituted from the program by its longest block in an asymptotic calculation. A potential simplification on this idea could be to include a passthrough atom NOP for no-operation, which takes a timestep but simply passes the value received through itself. By padding the short branches of a $SWITCH$ with NOP all branches are the same length.

The implicit meta-construct of concatenation affects runtime in an additive sense. Polynomials are closed under addition.

By restricting programs to be of finite length, all T-META branches are guaranteed to expand to polynomial length (runtime) branches of T-BARE because exponential and super-polynomial length sequences cannot be compressed to a finite length by operations polynomials are closed under.

0.5.3 FP Completeness

As T-META stands, the author is unaware of a way to prove its completeness within the complexity class of FP. The format of $LOOP$ and $SWITCH$ as pattern constructors and manipulators is too alien to lambda calculus to form a coherent equivalency between the FP describing logics such as light logics. The largest issue is the unconvertability from $LOOP$ style pattern expansion to lambda style pattern expansion. $SWITCH$ and $LOOP$ were based on the fundamental basic operations of primitive recursion (general recursive functions without minimization) [Col91], as an expansion of the projection function and primitive recursive function, respectively. Because $SWTICH$ is more general than the projection function Π, it is also not known to the author if this language reduces to the time class $ELEMENTARY$ is PR does [Sch16].

Interestingly, the flexibility of $SWITCH$ allows random piece-wise polynomial functions to be described. These functions are not computable, as they are not able to be contained in a finite program; their algorithmic complexity is infinite because there is no finite length program capable of computing any instance of the function. By the finite program length restriction these can't be described in a finite sense in T-META, but they are describable in an infinite-allowing construction of T-META that still disallows super-polynomial runtimes. By looking at a T-META program in two dimensions, block length and branch length, it is clear that the polynomial restraint lies only on the block length axis and algorithmic complexity lies on both axes; linearly increasing on both. The random polynomial function used in the Handling Randomness section is a good example of how thinking of T-META programs in two dimensions allows some incomputable but FP-instance relations to be described.

0.6 Discussion

0.6.1 Comparison to Light Linear/Affine Logic

Logics are another form of computation useful in categorizing implicit computational complexities. These forms of math look at resource use as a description of the time bound taken by a program. The simplest logic is classical logic. Unfortunately this formalism is easily proven to be too weak to describe FP. In classical logic the implication $A \to B$ doesn't consume A. This is referred to as contraction or the idempotency of entailment. To translate logical connectives to programming, one considers truths to be resources. Resources are functionally equivalent to blocks as they have been described in this paper. Such unrestricted introduction of resources allows the program to grow in completely unrestricted bounds. To combat this issue linear logics were created, and further refined by light logics, discussed here. The following is the language for the intuitionistic fragment of Light Linear Logic, without additive connectives [Gir98]. The sequent calculus interpretation of the

$$A ::= X \mid A \otimes A \mid A \multimap A \mid !A \mid \S A \mid \forall X A$$

Figure 5: Language of Intuitionistic Light Logic [Gir98]

introduction rules are the same as the inuitionistic fragment of linear logic, except for the handling of the exponential modalities ! and §.

$$\frac{\Gamma \vdash A}{!\Gamma \vdash !A}\,!\,\mathrm{f} \qquad \frac{\Gamma, \Delta \vdash A}{!\Gamma, \S\Delta \vdash \S A}\,\S \qquad \frac{\Gamma, !A, !A \vdash C}{\Gamma, !A \vdash C}\,!cL \qquad \frac{\Gamma \vdash C}{\Gamma, !A \vdash C}\,!wL$$

Figure 6: Light Linear Logic Rules as Sequent Calculus

It is known that the class of functions on binary lists representable in LLL is exactly FP [Gir98]. By this fact, dissecting the available operations in LLL helps characterize the operations allowable in FP. The dual fragment of LLL doesn't contain §. All of LLL, LAL, and the dual classes of both of these fragments are FP

[Laf04]. The crucial operations made available by these fragments are the exponential modalities which can be explained succinctly as soft lambda calculus:

$$M := x \mid \lambda x.M \mid \lambda !x.M \mid MM \mid !M$$

Figure 7: the language of Soft Lambda Calculus

Modalities control copying [DL19]. a variable appears linearly in the body it is bound in or can be copied by loss of a modality. The following are legal Soft Lambda Calculus terms:

$$\lambda !x.yxx$$

$$\lambda !x.y!x$$

$$\lambda !!x.y!x!x!x$$

Figure 8: Soft Lambda Calculus Modalities

The maximum depth a variable is nested in an exponential modality bounds the maximum polynomial degree a LLL term takes to normalize [DL11]. This runtime-bounding system is reminiscent of $LOOP$, but allows the pattern re-organization available to lambda abstractions, where $LOOP$ only concatenates blocks at their end.

0.6.2 Universal Searchers

Universal searchers are programs which instead of operating on functions operate on program space itself. They are used in an attempt to build solutions to relations in an unsupervised manner. They can be useful in helping to create programs which solve relations previously not known to be solvable. Particular work has been done in this field by Jürgen Schmidhuber. Universal searchers such as LSEARCH search the program space interleaving every possible program by running the programs in time inversely proportional to their length [Lev73]. LSEARCH works by running one step of a program every $|L|^{|d|}$ time steps, where $|d|$ is the length of the program and $|L|$ is the number of symbols in the language. Through this interleaving method LSEARCH runs in polynomial time on an exponential problem space [Lev73].

Much work has been done to refine universal searchers to run faster, learn from their failures, and have some form of bias optimality. The largest weakness of LSEARCH is its inability to describe more than instances, except by random chance. OOPS (Optimal Ordered Problem Solver) is an attempt to improve on this weakness by creating two partial solutions. One tries to solve the current instance and the other attempts to find a program which solves all instances up to the current one [Sch04]. The first solution behaves like the piecewise function discussed in Handling Randomness, where each instance is handled by a different solver, and the second solution attempts to find the general solution. OOPS additionally exploits its previous knowledge when creating further solutions, a feature absent from LSEARCH [Sch04].

0.6.3 Poly-time FP Universal Function

Of course the question must be asked, since universal searchers operate in a simulated FP time bound, if they are fed a language constrained to FP; is there any problem within FP unsolvable by this meta-program? Well, no. But this isn't the holy grail of tractable programs either. While this program will solve the posed problem in a polynomial time, the polynomial time is directly proportional to the length of the program. Even short programs quickly begin to require resources and time growing to magnitudes outliving the lifetime of the universe [Sch03]. This has always been the practical curse of the universal searcher. In the words of the immortal Leonid Levin, inventor of LSEARCH:

"Only math nerds consider 2^{500} finite"

Bibliography

[BDPR01] Gianluigi Bellin, Valeria De Paiva, and Eike Ritter. Extended curry-howard correspondence for a basic constructive modal logic. In *Proceedings of methods for modalities*, volume 2, 2001.

[Col91] Loïc Colson. About primitive recursive algorithms. *Theoretical Computer Science*, 83(1):57–69, 1991.

[DL11] Ugo Dal Lago. A short introduction to implicit computational complexity. In *Lectures on Logic and Computation*, pages 89–109. Springer, 2011.

[DL19] Ugo Dal Lago. Machine-free complexity: Implicit complexity. University Lecture, 2019.

[Gir98] Jean-Yves Girard. Light linear logic. *Information and Computation*, 143(2):175–204, 1998.

[Ili13] Danko Ilik. Continuation-passing style models complete for intuitionistic logic. *Annals of Pure and Applied Logic*, 164(6):651–662, 2013.

[Laf04] Yves Lafont. Soft linear logic and polynomial time. *Theoretical computer science*, 318(1-2):163–180, 2004.

[Lev73] Leonid Anatolevich Levin. Universal sequential search problems. *Problemy peredachi informatsii*, 9(3):115–116, 1973.

[Pol11] Andrew Polonsky. *Proofs, types, and lambda calculus.* PhD thesis, PhD thesis, University of Bergen, 2011.

[Ric08] Elaine Rich. *Automata, computability and complexity: theory and applications.* Pearson Prentice Hall Upper Saddle River, 2008.

[Sch03] Jürgen Schmidhuber. Godel machines: Self-referential universal problem solvers making provably optimal self-improvements. *arXiv preprint cs.LO/0309048*, 2003.

[Sch04] Jürgen Schmidhuber. Optimal ordered problem solver. *Machine Learning*, 54(3):211–254, 2004.

[Sch16] Sylvain Schmitz. Complexity hierarchies beyond elementary. *ACM Transactions on Computation Theory (TOCT)*, 8(1):1–36, 2016.

[Ter12] Kazushige Terui. Semantic evaluation, intersection types and complexity of simply typed lambda calculus. In *23rd International Conference on Rewriting Techniques and Applications (RTA'12).* Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012.