Polynomial Time and Dependent Types

ROBERT ATKEY, University of Strathclyde, UK

We combine dependent types with linear type systems that soundly and completely capture polynomial time computation. We explore two systems for capturing polynomial time: one system that disallows construction of iterable data, and one, based on the LFPL system of Martin Hofmann, that controls construction via a payment method. Both of these are extended to full dependent types via Quantitative Type Theory, allowing for arbitrary computation in types alongside guaranteed polynomial time computation in terms. We prove the soundness of the systems using a realisability technique due to Dal Lago and Hofmann.

Our long-term goal is to combine the extensional reasoning of type theory with intensional reasoning about the resources intrinsically consumed by programs. This paper is a step along this path, which we hope will lead both to practical systems for reasoning about programs' resource usage, and to theoretical use as a form of *synthetic computational complexity theory*.

CCS Concepts: • **Theory of computation** → **Linear logic**; **Type theory**; **Complexity classes**; *Complexity theory and logic*; Categorical semantics.

Additional Key Words and Phrases: type theory, implicit computational complexity, linear logic

ACM Reference Format:
Robert Atkey. 2024. Polynomial Time and Dependent Types. *Proc. ACM Program. Lang.* 8, POPL, Article 76 (January 2024), 30 pages. https://doi.org/10.1145/3632918

1 INTRODUCTION

Type Theory is often claimed to be ideal for Computer Science, combining programming and proof in one unifying system, so a happy programmer can verify while they program, and program while they verify. From a broader Computer Science view, however, Type Theory lacks the ability to talk about the very thing that makes Computer Science interesting – the fact that computation is everywhere bounded by the resources in time and space that we can afford to give it.

Typically, Type Theory only speaks of the public face that programs present to the world – if you input things like this, you get things that look like that – but cannot bring itself to mention the true cost of programs' execution. One can encode costs by embedding another programming language in Type Theory, for example [Guéneau et al. 2018], or one can synthesise costs by treating resource counting as a computational effect, for example [Danielsson 2008; Niu et al. 2022], but neither of these capture the intrinsic costs of the programs we write in Type Theory. These techniques deliver only *conspicuous consumption*, not speaking of the real resources consumed.

In this paper, we propose a method for extending dependent Type Theory with a means for constraining the intrinsic computational complexity of programs written in the theory. We concentrate on linear type systems that soundly and completely capture polynomial time computation, the commonly used standard for feasible resource usage, and extend these systems to dependent types. The additional expressivity of dependent types allows us use these characterisations of polytime to

Author's address: Robert Atkey, robert.atkey@strath.ac.uk, University of Strathclyde, 26 Richmond Street, Glasgow, UK, G1 1XH.

ACM 2475-1421/2024/1-ART76
https://doi.org/10.1145/3632918

further functionally characterise the classes of non-deterministic and bounded-error probabilistic polynomial time.

We use techniques from *Implicit Computational Complexity* theory, which provides intrinsic characterisations of complexity classes in terms of logical systems or programming languages. We review the techniques that we use in Section 2. To adapt these systems to dependent types, we use *Quantitative Type Theory* (QTT) [Atkey 2018; McBride 2016], a combination of linear and dependent types. We review QTT in Section 3.

Our long-term goal is to combine the extensional reasoning of Type Theory with intensional reasoning about the resources intrinsically consumed by programs. This paper is a first step along this path, which we hope will lead both to practical systems for reasoning about programs' resource usage as well as their extensional behaviour, and to theoretical use as a form of *synthetic computational complexity theory*. We discuss these possibilities further in Section 7.

Contributions and Content. This paper makes the following contributions to the theory and use of linear dependent type theory and implicit computational complexity:

(1) We formulate two systems that combine linear type theory for polytime computation with full dependent types, using Quantitative Type Theory. The systems are presented in Section 3. The linear typing discipline required for enforcing polytime is provided by QTT, but we also need to carefully add constructs for non-iterable datatypes (Section 3.2) and the two kinds of natural number iterator that we consider (Section 3.3 and Section 3.4). Porting the natural number iterators from the simply typed to the dependently typed setting requires careful annotation of the rules to ensure that the correct information is available for type checking, while also not allowing too much information to be made available at runtime that would violate the polytime soundness property. A further contribution of this paper is the addition of reflection types to QTT, Section 3.5, which allow statements about polytime realisability to be reflected into types.

(2) We demonstrate the utility of the combination of polytime and dependent types in Section 4. Just as in the simply typed world, we have an expressive language for writing polytime programs. With the additional power of dependent types, we can also prove properties of these programs. A simple example is proving that a polytime sorting program actually sorts. Using QTT reflection, we can go further and represent the class of polytime problems, with polytime reductions between them, as dependent pairs (Section 4.2). Our final examples use dependent types to give monadic presentations of the complexity classes of Non-deterministic Polynomial (NP) time and Bounded-error Probabilistic Polynomial (BPP) time. Since these classes rely on specific semantic correctness criteria, it is not possible to capture them in a simply typed system for polytime.

(3) We prove the polytime soundness of our systems via a realisability argument in Section 5 and Section 6. Our construction is an extension of the amortised complexity realisability constructions of Dal Lago and Hofmann [2011]. We extend their work to our dependently typed setting, and also give a realisability interpretation of datatypes directly, instead of via second-order impredicative encodings. The technical content of these sections has been formalised in the Agda proof assistant [Norell 2008], and is included in the associated artefact [Atkey 2023a].

Before we get to the contributions above, we present, in Section 2, two linear simply typed systems for polytime, adaptations of systems already present in the literature. Our paper concludes with a discussion of further related work and the outlook for future work in Section 7.

2 AFFINE LINEAR TYPING AND POLYTIME

Not long after Girard introduced Linear Logic [Girard 1987], it was observed that its resource sensitivity could be turned to describing computational complexity classes by purely logical means. Typically, a logical system is described for which the process of reducing a proof to a normal form (often by cut elimination) is guaranteed to always be accomplished within a certain complexity bound. Moreover, the system is usually proven to be complete for the relevant complexity class by constructing a simulation of some known representation. Such systems that characterise polytime include Bounded Linear Logic (BLL), which uses explicit polynomials in the formulas [Girard et al. 1992] and Soft Affine Logic (SAL) [Lafont 2004], which does not explicitly represent time information in formulas, but uses a restricted form of Linear Logic's ! modality instead. Light Linear Logic (LLL) [Girard 1998] is another "counting-free" system for polytime.

Viewing logical systems though the Curry-Howard correspondence, the idea arises that one could define functional programming languages that characterise complexity classes such as polytime. SAL has been transformed into a programming language by Baillot and Mogbil [2004], and likewise for LLL by Baillot et al. [2010]. Hofmann [1999] proposed a new programming language, Linear Functional Programming Language (LFPL), that uses a novel "payment" system to track iteration.

There are at least two ways that a functional programming language can be seen as representing polynomial time, differing in how the size of the problem to be computed is measured. One approach is to consider closed expressions, combining the program with its input, and computation of the result is polynomial time in the combined size. A second approach is that the input is "externally" provided, where we consider open terms with a free variable representing the input. So a judgement $x : \mathrm{Nat} \vdash M : A$ declares a program that computes results of type A in time polynomial in the size of the natural number x. We take this latter approach in this paper.

With a view to extending to dependent types in Section 3, we take an approach slightly different to much of the polytime linear logic literature. We use explicit datatypes and eliminators, rather than using impredicative encodings via universal types. We are closer to Hofmann's original LFPL (though not a later presentation of it by Dal Lago and Hofmann [2011]) than BLL, SAL or LLL.

In this section, we review the use of linear types to capture polytime by presenting two systems, one based on ideas from SAL and the second more explicitly based on LFPL.

2.1 Affine Linear λ-Calculus

For this section, the affine linear λ-calculus we will use will have linear functions and $\otimes$-products. Contexts are treated up to permutation of entries, so uses of exchange are implicit.

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \multimap B} \qquad \frac{\Gamma_1 \vdash M : A \multimap B \qquad \Gamma_2 \vdash N : A}{\Gamma_1, \Gamma_2 \vdash M\,N : B}$$

$$\frac{\Gamma_1 \vdash M : A \qquad \Gamma_2 \vdash N : B}{\Gamma_1, \Gamma_2 \vdash (M, N) : A \otimes B} \qquad \frac{\Gamma_1 \vdash M : A \otimes B \qquad \Gamma_2, x : A, y : B \vdash N : C}{\Gamma_1, \Gamma_2 \vdash \mathrm{let}\,(x, y) = M\,\mathrm{in}\,N : C}$$

These rules are standard, so we do not describe them further except to note how affine linear typing uses presence or absences in a context to control resource usage. If a variable is in the context it must be used at most once (variables that are not used are absorbed by the additional context in the variable rule). The fact that this discipline interferes with dependent types is one of the reasons we turn to QTT when we wish to add dependent types in Section 3.

2.2 No Recursion, Only Case Analysis

It is not too difficult to see that reduction of linear λ-terms always takes a number of steps linearly proportional to the size of the term. This is because every β-redex substitutes each term into at most one variable, reducing the size of the term by one each time.

We can increase the expressivity, but not the computational complexity, of the system by adding datatypes that do not allow iteration. These can be used for representation but not for driving computation. We include the rules here to show how linearity must be preserved in these rules and to foreshadow their dependently typed counterparts in Section 3.2. The first type is the booleans, which are non-recursive and so would not allow iteration anyway:

$$\frac{}{\vdash \mathsf{true}, \mathsf{false} : \mathsf{Bool}} \qquad \frac{\Gamma_1 \vdash M : \mathsf{Bool} \qquad \Gamma_2 \vdash N_1 : A \qquad \Gamma_2 \vdash N_2 : A}{\Gamma_1, \Gamma_2 \vdash \mathsf{if}\, M\, \mathsf{then}\, N_1\, \mathsf{else}\, N_2 : A}$$

The if-then-else rule is careful to ensure that the resources used by the eliminated Bool and the resources used by the chosen branch are accounted for separately. The two branches must have the same resource usage.

Construction and case analysis of lists are given by the following rules:

$$\frac{}{\vdash \mathsf{nil} : \mathsf{List}(A)} \qquad \frac{\Gamma_1 \vdash M : A \qquad \Gamma_2 \vdash N : \mathsf{List}(A)}{\Gamma_1, \Gamma_2 \vdash \mathsf{cons}(M, N) : \mathsf{List}(A)}$$

$$\frac{\Gamma_1 \vdash M : \mathsf{List}(A) \qquad \Gamma_2 \vdash N_1 : B \qquad \Gamma_2, h : A, t : \mathsf{List}(A) \vdash N_2 : B}{\Gamma_1, \Gamma_2 \vdash \mathsf{match}\, M\, \{\mathsf{nil} \mapsto N_1; \mathsf{cons}(h, t) \mapsto N_2\} : B}$$

We can construct lists arbitrarily but only do case analysis on them. If we wish to explore a list to a arbitrary depth it must be driven by a type we can iterate over.

With booleans and lists, we can construct several other useful types. For example, to simulate Turing machines, one can construct a Tape type as a Zipper (Huet [1997]) $\mathsf{List}(\mathsf{Bool}) \otimes \mathsf{Bool} \otimes \mathsf{List}(\mathsf{Bool})$, representing a position on the tape with the items before, under, and after the head.

2.3 The Cons-Free System

Polynomial time is usually seen as a proxy for "feasible" computation. On the face of it, there does not seem to be any particular reason why polynomials have anything to do with feasibility. However, one can arrive at the definition of polynomial time in three steps, by assuming that (i) iterating over the whole input is feasible; (ii) if two computations are feasible, then so is their composition; and (iii) performing a feasible computation for every element of the input is also feasible. It is the last point that allows complexities of arbitrary polynomial degree to be constructed (we will see this in action in the completeness construction below and soundness proofs in Section 6).

Following these ideas, let us assume that the input is a natural number, so we assume that there is some type of natural numbers Nat. For point (i), we must be able to iterate over these natural numbers, so we use a linear iterator defined by this typing rule:

$$\frac{\vdash M_z : A \qquad x : A \vdash M_s : A \qquad \Gamma \vdash N : \mathsf{Nat}}{\Gamma \vdash \mathsf{rec}\, N\, \{\mathsf{zero} \mapsto M_z; \mathsf{succ}(x) \mapsto M_s\} : A}$$

Note that in the zero, M_z, and sucessor, M_s, cases, the context is empty to ensure that these cases may be invoked as many times as required. Point (ii) above is automatically satisfied by being in a typed λ-calculus, where it is difficult to stop functions from being composable. For point (iii), the iterator as given does not allow us to nest iterations. Once the natural number input n has been used for an iteration, the linear typing discipine prevents us from using it again (note the two separate contexts Γ_1, Γ_2 in the rule for application). In order to allow nested iterations, we add an operator to duplicate numbers:

$$\frac{\Gamma \vdash M : \mathrm{Nat}}{\Gamma \vdash \mathrm{dupNat}\, M : \mathrm{Nat} \otimes \mathrm{Nat}}$$

Somewhat surprisingly, this system is now sound and complete for polynomial time. Crucially, this depends on the two things we have *not* allowed. First, we have disallowed the construction of new natural numbers via the zero and succ constructors[1]. If we were to permit this, then we could use iteration over the input to construct addition, multiplication (by repeated addition), and then exponentials (by repeated multiplication). We therefore refer to this system as the *Cons-free* system. Because we cannot construct values of type Nat within the system, complete programs in this system are open terms as we explained at the start of this section.

The second prohibited feature is the ability to duplicate values of function type, even though we have allowed duplication of iterable naturals. If we were to allow this, then we would be able to sneak in a form of constructors for natural numbers by encoding them as eliminators that duplicate a function for every succ step.

We will see in Section 6.2 that this system is sound for polytime by a realisability argument. Completeness can be seen more directly by constructing a function that iterates a function for a statically known polynomial number of times in the size of the input. Assume that we have a known polynomial $p(n) = c_d n^d + \cdots + c_0$ of degree d with natural number coefficients and some single step function $f : St \multimap St$ over a state type St that runs to completion for input of size n in $p(n)$ steps. Then, using the iterator above we can iterate f over a Nat representing the size of the input:

$$\begin{aligned}&I_1 : \mathrm{Nat} \multimap St \multimap St\\ &I_1 = \lambda n.\lambda s.\mathrm{rec}\, n\, \{\mathrm{zero} \mapsto s; \mathrm{succ}(s) \mapsto f\, s\}\end{aligned}$$

To achieve higher degrees, we can use dupNat to nest iterations:

$$\begin{aligned}&I_{k+1} : \mathrm{Nat} \multimap St \multimap St\\ &I_{k+1} = \lambda n.\lambda s.\mathrm{let}\, (n, n') = \mathrm{dupNat}\, n\, \mathrm{in}\\ &\qquad\qquad\qquad \mathrm{rec}\, n\, \{\mathrm{zero} \mapsto s; \mathrm{succ}(s) \mapsto I_k\, n'\, s\}\end{aligned}$$

By further use of dupNat and composition to handle addition of polynomials, the function f can now be iterated $p(n)$ many times, where n is the input Nat. Thus, the *Cons-free* system can represent all polytime computations.

2.4 Diamond Trading with LFPL

The *Cons-free* system is sound and complete for polytime, but is quite awkward from the point of view of functional programming. It allows us to iterate over natural numbers that come from the input but does not allow us to build further values to do iteration on. For example, if our input is a list, then we cannot transform it into a binary search tree and then flatten it, we must always refer back to the original natural number input. Even dividing the input into two halves to be treated separately is difficult.

A more flexible system was proposed by Hofmann [1999]. Instead of completely prohibiting construction of data, the *Linear Functional Programming Language* (LFPL) allows construction if it

[1] Actually, zero would be acceptable, as well as any constant natural number. It is only unrestricted use of succ that is dangerous.

is paid for by values of type ◇ (“diamonds”):

$$\frac{\Gamma \vdash M : \diamond}{\Gamma \vdash \mathsf{zero}(M) : \mathsf{Nat}} \qquad \frac{\Gamma_1 \vdash M : \diamond \qquad \Gamma_2 \vdash N : \mathsf{Nat}}{\Gamma_1, \Gamma_2 \vdash \mathsf{succ}(M, N) : \mathsf{Nat}}$$

To construct a zero, we must have a ◇ to pay for it, and likewise, to construct a succ we must pay a ◇. We can think of ◇s as an unit of iterable data. Iterability is “saved up” in data during construction, and released during iteration. Diamonds cannot be created by a program itself, for the same reason that constructors were prohibited in the *Cons-free* system, but they are released from iterable data during iteration. The LFPL natural number iterator has the following typing rule:

$$\frac{d : \diamond \vdash M_z : A \qquad d : \diamond, x : A \vdash M_s : A \qquad \Gamma \vdash N : \mathsf{Nat}}{\Gamma \vdash \mathsf{rec}\, N\, \{\mathsf{zero}(d) \mapsto M_z; \mathsf{succ}(d, x) \mapsto M_s\} : A}$$

The difference with the *Cons-free* iterator above is that the zero and succ cases now both have an additional binding of type ◇. This allows some form nesting of iterations: during an iteration over the input, the program can accumulate ◇s to use for iteration over substructures that are smaller than the current point in the iteration. A construction, due to Aehlig and Schwichtenberg [2002], illustrates how this leads to all polytime computations. As above, we assume a polynomial $p(n)$ and a step function $f : St \multimap St$ that needs to be iterated $p(n)$ times. We construct a linear iterator:

$$\begin{array}{lll} I_1 : (\mathsf{Nat} \otimes St) \multimap (\mathsf{Nat} \otimes St) & & \\ I_1 = \lambda(n, s).\, \mathsf{rec}\, n\, \{\mathsf{zero}(d) & \mapsto & (\mathsf{zero}(d), s); \\ \qquad\qquad\qquad\qquad \mathsf{succ}(d, (n, s)) & \mapsto & (\mathsf{succ}(d, n), f\, s)\} \end{array}$$

Note that this iterator returns the natural number input as well as the new state. LFPL does not allow duplication of iterable inputs, so we must always reconstruct it if we want to do further iteration. Addition of polynomials is accomplished by composition of iterators. To raise the degree, we again use a nesting iterator:

$$\begin{array}{lll} I_{k+1} : (\mathsf{Nat} \otimes St) \multimap (\mathsf{Nat} \otimes St) & & \\ I_{k+1} = \lambda(n, s).\, \mathsf{rec}\, n\, \{\mathsf{zero}(d) & \mapsto & (\mathsf{zero}(d), s); \\ \qquad\qquad\qquad\qquad \mathsf{succ}(d, (n, s)) & \mapsto & \mathsf{let}\, (n, s) = I_k\, (n, s)\, \mathsf{in}\, (\mathsf{succ}(d, n), s)\} \end{array}$$

Unlike in the *Cons-free* system, this iterator does not raise the degree of the nested iterator directly. Rather, the iterator I_k on the input n performs $\binom{n}{k}$ iterations. As observed by Aehlig and Schwictenberg, this is sufficient because the binomials form a basis for the vector space of all polynomials.

Despite this slightly more involved completeness construction, the advantage of LFPL is that it is now easy to have arbitrary iterable datatypes and to transform between them. We need only take the introduction and elimination rules for any inductive datatype and add ◇ premises to the introduction rules and ◇ bindings to the eliminators.

3 POLYTIME QUANTITATIVE TYPE THEORY

We have now seen the *Cons-free* and LFPL systems for capturing polytime by means of linear typing and restricted iteration. We now look to extend these systems to include dependent types by building upon *Quantitative Type Theory* (QTT) [Atkey 2018; McBride 2016]. This section reviews QTT and describes how we have adapted it to the polytime systems we saw in the previous section.

3.1 Quantitative Type Theory

Integrating linear and dependent types is not straightforward due to the conflict between the linear typing discipline regarding presence of a variable as only bestowing the right to use it once, and the dependent typing regime that uses variables both in types (for specification purposes) and in terms (for computational purposes), syntactically yielding multiple uses of the same variable.

QTT is a system that resolves this conflict by recording usage of variables with annotations from a semiring. It sits in the general area of systems that use semiring annotations to measure resource usage [Brunel et al. 2014; Ghica and Smith 2014; Orchard et al. 2019]. The key feature of QTT, an insight owing to McBride [2016], is that usage of variables in types counts for 0-usage in terms of the semiring used. This allows us to use normal type theory as a specification language, while also enjoying the benefits of linear typing for programs. The term typing judgement of QTT has the following form:

$$x_1 \overset{\rho_1}{:} S_1, \dots, x_n \overset{\rho_n}{:} S_n \vdash M \overset{\sigma}{:} T$$

where the annotations ρ_i are all from the semiring being used. The annotation σ is either 0 or 1, indicating whether we are in the erased ($\sigma = 0$) fragment, where all the normal rules of type theory apply, or the in the non-erased ("present", "realisable", $\sigma = 1$) fragment, where a restricted typing discipline applies. As we shall see below in the cases of Σ-types, iterable types, and LFPL's $\Diamond$ type, the separation of QTT into these two fragments allows an expressive combination of reasoning using full type theory with the benefits of linear typing.

In the remainder of this sub-section, we describe the core of QTT. As well as the term typing judgement given above, QTT also has judgements for well-formed contexts (Γ ctxt) and types ($\Gamma \vdash T$ type), and definitional equality of types and terms ($\Gamma \vdash S \equiv T$ type and $\Gamma \vdash M \equiv N \overset{\sigma}{:} S$). It is an invariant of the system that types are always well-formed in a context with all annotations 0, i.e., $\Gamma \vdash S$ type implies $0\Gamma = \Gamma$. An important admissible rule of the system, along with weakening and substitution, is that of 0-ing:

$$\frac{\Gamma \vdash M \overset{1}{:} S}{0\Gamma \vdash M \overset{0}{:} S}$$

This rule allows us to take any term M in the $\sigma = 1$ fragment and treat it as if it were in the $\sigma = 0$ fragment, and hence use it for specification purposes in types. As we add novel rules to QTT in the following sections, we will be careful to maintain the admissibility of this rule.

In this section, we give an overview of the rules of QTT. The full rules, including all equality rules, are presented in the extended version of this paper [Atkey 2023b].

3.1.1 Natural-Number Usages. We use an instantiation of QTT with the natural number semiring, with the usual semiring structure of addition and multiplication. In a mild generalisation of the original presentation of QTT, we also allow sub-usaging via the *reverse* ordering on the naturals. That is, if a variable is marked as usage n and $m \geq n$, then we can also regard it as usage m. This makes the system more like affine linear logic, since $m \sqsubseteq 0$[2] for all m, matching the system in Section 2. We do *not* have an unrestricted usage ω, since this would allow the possibility of unrestricted duplication, and hence violate our polytime soundness properties.

3.1.2 Contexts, Variables, and Conversion. As we saw above, contexts in QTT are comprised of variable $\overset{\rho}{:}$ type triples, where ρ is a natural number indicating how many times the variable x is available for use in a $\sigma = 1$ term. There are two operations on raw contexts: scaling $\pi\Gamma$, which multiplies each ρ in Γ by π, and addition $\Gamma_1 + \Gamma_2$, which adds two contexts' usage annotations assuming that the lengths and types are equal. Contexts are ordered pointwise $\Gamma' \sqsubseteq \Gamma$ on the usage annotations (which is the *reverse* ordering on naturals) The basic usage-annotation discipline of

[2] Reverse ordering!

QTT is demonstrated by the context formation and variable rules:

$$\frac{}{\epsilon\ \mathrm{ctxt}} \qquad \frac{\Gamma\ \mathrm{ctxt} \qquad 0\Gamma \vdash S\ \mathrm{type}}{\Gamma, x \overset{\rho}{:} S\ \mathrm{ctxt}} \qquad \frac{0\Gamma, x \overset{\sigma}{:} S, 0\Gamma'\ \mathrm{ctxt}}{0\Gamma, x \overset{\sigma}{:} S, 0\Gamma' \vdash x \overset{\sigma}{:} S} \qquad \frac{\Gamma \vdash M \overset{\sigma}{:} S \qquad \Gamma' \sqsubseteq \Gamma}{\Gamma' \vdash M \overset{\sigma}{:} S}$$

As with most dependent type theories, contexts are built inductively from the empty context ϵ and extension of a context by a variable with a type that is well-formed in the preceding context. Usage annotations ρ on variables are arbitrary, and types are always judged in a 0-annotated context. The variable rule marks unused variables as usage 0 and the selected variable with usage σ.

As usual, definitional equality of types impacts typing of terms via the conversion rule:

$$\frac{\Gamma \vdash M : S \qquad 0\Gamma \vdash S \equiv T\ \mathrm{type}}{\Gamma \vdash M : T}$$

Like type formation, definitional equality of types always takes place in 0-d contexts. We will describe the definitional equality rules for terms of each type as we introduce them. In QTT, it is possible for the definitional equality of terms to differ between the $\sigma = 0$ and $\sigma = 1$ fragments, as we will see below.

3.1.3 Π- and Σ-types. QTT's Π-types have the form $(x \overset{\rho}{:} S) \to T$, indicating functions that, in the $\sigma = 1$ fragment, use their arguments ρ-many times. The formation, introduction and elimination rules are similar to the standard ones, except for the addition of usage annotations:

$$\frac{0\Gamma \vdash S\ \mathrm{type} \qquad 0\Gamma, x \overset{0}{:} S \vdash T\ \mathrm{type}}{0\Gamma \vdash (x \overset{\rho}{:} S) \to T\ \mathrm{type}} \qquad \frac{\Gamma, x \overset{\sigma\rho}{:} S \vdash M \overset{\sigma}{:} T}{\Gamma \vdash \lambda x.M \overset{\sigma}{:} (x \overset{\rho}{:} S) \to T}$$

$$\frac{\Gamma_1 \vdash M \overset{\sigma}{:} (x \overset{\rho}{:} S) \to T \qquad \Gamma_2 \vdash N \overset{\sigma'}{:} S \qquad 0\Gamma_1 = 0\Gamma_2 \qquad \sigma' = 0 \Leftrightarrow (\rho = 0 \lor \sigma = 0)}{\Gamma_1 + \rho\Gamma_2 \vdash M\,N \overset{\sigma}{:} T[N/x]}$$

The side conditions on the elimination rule state that (i) both Γ_1 and Γ_2 erase to the same context, so their sum is defined; and (ii) the argument N is erased (i.e., $\sigma' = 0$) iff either the function does not use its argument, or we are in the $\sigma = 0$ fragment and everything is being erased. In the following, when we write $S \to T$ for a non-dependent function type, we mean that the argument is to be used linearly: $(x \overset{1}{:} S) \to T$, where x does not appear in T. Π-types support the usual $\beta\eta$ definitional equalities in both the $\sigma = 0$ and $\sigma = 1$ fragments.

Σ-types are a little more involved, and demonstrate the flexibility in QTT in allowing additional power in the $\sigma = 0$ fragment where we do not need to care about polytime realisability. Formation and introduction are given by the rules:

$$\frac{0\Gamma \vdash S\ \mathrm{type} \qquad 0\Gamma, x \overset{0}{:} S \vdash T\ \mathrm{type}}{0\Gamma \vdash (x \overset{\pi}{:} S) \otimes T\ \mathrm{type}} \qquad \frac{\begin{array}{c}\Gamma_1 \vdash M \overset{\sigma'}{:} S \qquad \Gamma_2 \vdash N \overset{\sigma}{:} T[M/x] \\ 0\Gamma_1 = 0\Gamma_2 \qquad \sigma' = 0 \Leftrightarrow (\pi = 0 \lor \sigma = 0)\end{array}}{\pi\Gamma_1 + \Gamma_2 \vdash (M, N) \overset{\sigma}{:} (x \overset{\pi}{:} S) \otimes T}$$

As with Π-types, the first component of a Σ-type is annotated with a usage for how many times it can be used, and this is respected by the introduction rule. Elimination of Σ-types depends on whether we are in the $\sigma = 0$ fragment or not. In the $\sigma = 0$ fragment, we are free to disregard usage restrictions, and use projections as normal:

$$\frac{\Gamma \vdash M \overset{0}{:} (x \overset{\pi}{:} S) \otimes T}{\Gamma \vdash \mathsf{fst}(M) \overset{0}{:} S} \qquad\qquad \frac{\Gamma \vdash M \overset{0}{:} (x \overset{\pi}{:} S) \otimes T}{\Gamma \vdash \mathsf{snd}(M) \overset{0}{:} T[\mathsf{fst}(M)/x]}$$

Σ-types are unrestricted in the $\sigma = 0$ fragment, and we can use them as normal for type-theoretic constructions. In the $\sigma = 1$ fragment, we must take into account the resource content of objects and use a pattern matching construct; the dependently typed analogue of the ⊗-eliminator in Section 2:

$$\frac{\Gamma_1 \vdash M \overset{\sigma}{:} (x \overset{\pi}{:} A) \otimes B \qquad \begin{array}{c} 0\Gamma, z \overset{0}{:} (x \overset{\pi}{:} A) \otimes B \vdash C \\ \Gamma_2, x \overset{\sigma\pi}{:} A, y \overset{\sigma}{:} B \vdash N \overset{\sigma}{:} C[(x, y)/z] \end{array} \qquad 0\Gamma_1 = 0\Gamma_2}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\ (x, y) = M\ \mathsf{in}\ N \overset{\sigma}{:} C[M/z]}$$

QTT also supports a unit type I with constructor $*$ and pattern-matching [Atkey 2018]. Σ- and I types support the usual $\beta\eta$ definitional equalities in the $\sigma = 0$ fragment (e.g., $\mathsf{fst}(M, N) \equiv M$), but only β equalities (i.e., $\mathsf{let}\ (x, y) = (M_1, M_2)\ \mathsf{in}\ N \equiv N[M_1/x, M_2/y]$) in the $\sigma = 1$ fragment. It would also be sound to support commuting conversions [Barber 1996] for let in the $\sigma = 1$ fragment, but this would likely bring complications for implementation.

3.1.4 The Identity Type. QTT also supports an extensional equality type with equality reflection:

$$\frac{0\Gamma \vdash S\ \mathsf{type} \qquad 0\Gamma \vdash M \overset{0}{:} S \qquad 0\Gamma \vdash N \overset{0}{:} S}{0\Gamma \vdash M =_S N\ \mathsf{type}} \qquad \frac{\Gamma \vdash M \overset{\sigma}{:} S}{\Gamma \vdash \mathsf{refl}(M) \overset{\sigma}{:} M =_S M} \qquad \frac{\Gamma \vdash N \overset{0}{:} M_1 =_S M_2}{\Gamma \vdash M_1 \equiv M_2 \overset{0}{:} S}$$

The equality type also has an η rule demonstrating $\mathsf{refl}(M)$ as the only proof of equality [Hofmann 1997]. Note that equality reflection only targets the $\sigma = 0$ fragment, we cannot use equality reflection to convey realisability information.

3.1.5 Universe. QTT has universe types U, as in standard type theory [Atkey 2018]. For our examples below, we do not explicitly mark the use of terms of type U as types – i.e., we use a Russell-style presentation. Universes are where the definitional equality on terms affects the definitional equality on types.

3.1.6 Data Types. QTT, as we have presented it so far, has no interesting base types to perform computation on. Following our presentation of the simply typed linear systems in Section 2, we add two kinds of datatype to QTT. First, in Section 3.2, we add non-iterable datatypes that allow construction and case analysis, but no recursion. Then, in Section 3.3 we describe how to extend QTT to be a dependently typed adaptation of the *Cons-free* system of Section 2.3 by adding a type of iterable naturals. In Section 3.4 we apply the same treatment to the LFPL-style system.

3.2 Non-Iterable Data Types

3.2.1 Booleans. The boolean type for QTT was described in [Atkey 2018]. Booleans offer no possibility for iteration, but it is useful to see how the QTT rules extend the simply typed rules from Section 2.2 before moving to more complex types.

$$\frac{\Gamma\ \mathrm{ctxt}}{\Gamma \vdash \mathrm{Bool}\ \mathrm{type}} \qquad\qquad \frac{\Gamma\ \mathrm{ctxt}}{0\Gamma \vdash \mathsf{true}, \mathsf{false} \overset{\sigma}{:} \mathrm{Bool}}$$

$$\frac{\begin{array}{c} 0\Gamma_1, x \overset{0}{:} \mathrm{Bool} \vdash P\ \mathrm{type} \\ \Gamma_1 \vdash M \overset{\sigma}{:} \mathrm{Bool} \qquad \Gamma_2 \vdash N_t \overset{\sigma}{:} P[\mathsf{true}/x] \qquad \Gamma_2 \vdash N_f \overset{\sigma}{:} P[\mathsf{false}/x] \qquad 0\Gamma_1 = 0\Gamma_2 \end{array}}{\Gamma_1 + \Gamma_2 \vdash \mathrm{if}_{x.P}\ M\ \mathrm{then}\ N_t\ \mathrm{else}\ N_f \overset{\sigma}{:} P[M/x]}$$

The introduction rules for booleans both use a 0-d context, indicating that construction of boolean values is free. Elimination of booleans via a dependently typed if-then-else is more subtle with its resource usage. The boolean to be eliminated must be constructed in a context Γ_1, while the two branches are constructed in context Γ_2. Since only one of the branches will be used, sharing resources between the branches is expected. Booleans and their eliminator obey the usual β laws for definitional equality: $\mathrm{if}_{x.P}\ \mathsf{true}\ \mathrm{then}\ N_t\ \mathrm{else}\ N_f \equiv N_t$, and similarly for false.

One might wonder how, since constructing booleans is 0-cost by their introduction rules, the Γ_1 context will ever be non-0. This is resolved by observing that booleans may be the output of processes that consume time (e.g., the iteration constructs defined below), and so Γ_1 will represent a requirement that the necessary resource is provided.

3.2.2 Lists. Lists are a little more complex than booleans, because the cons constructor takes two arguments, so their resource usage must be combined. The type formation and introduction rules are as follows:

$$\frac{0\Gamma \vdash T\ \mathrm{type}}{0\Gamma \vdash \mathrm{List}(T)\ \mathrm{type}} \qquad \frac{\Gamma \vdash T\ \mathrm{type}}{0\Gamma \vdash \mathsf{nil} \overset{\sigma}{:} \mathrm{List}(T)} \qquad \frac{\Gamma_1 \vdash M \overset{\sigma}{:} T \qquad \Gamma_2 \vdash N \overset{\sigma}{:} \mathrm{List}(T) \qquad 0\Gamma_1 = 0\Gamma_2}{\Gamma_1 + \Gamma_2 \vdash \mathsf{cons}(M, N) \overset{\sigma}{:} \mathrm{List}(T)}$$

Lists do have the potential for iteration by their recursive nature, but in order to ensure the polytime complexity guarantees we only permit matching without recursion in the $\sigma = 1$ fragment. Here is the rule for dependently typed case analysis on lists, which also obeys the usual β-equalities for case analysis, analogous to the ones for booleans:

$$\frac{\begin{array}{c} 0\Gamma_1, x \overset{0}{:} \mathrm{List}(T) \vdash P\ \mathrm{type} \qquad \Gamma_1 \vdash M \overset{\sigma}{:} \mathrm{List}(T) \\ \Gamma_2 \vdash N_1 \overset{\sigma}{:} P[\mathsf{nil}/x] \qquad \Gamma_2, h \overset{\sigma}{:} T, t \overset{\sigma}{:} \mathrm{List}(T) \vdash N_2 \overset{\sigma}{:} P[\mathsf{cons}(h,t)/x] \qquad 0\Gamma_1 = 0\Gamma_2 \end{array}}{\Gamma_1 + \Gamma_2 \vdash \mathrm{match}_{x.P}\ M\ \{\ \mathsf{nil} \mapsto N_1; \mathsf{cons}(h,t) \mapsto N_2\ \} \overset{\sigma}{:} P[M/x]}$$

In the $\sigma = 0$ fragment, however, we are free to iterate on lists because computations in this fragment are only meant for type-level computation, not for the program itself. Put another way, the type checker may perform arbitary recursion on lists to type check the program, but the program itself may not do so without correctly accounting its costs as described in the following sections. The $\sigma = 0$ fragment recursor for lists has the following typing rule, which is the standard dependent eliminator for lists except that everything annotated as 0 usage.

$$\frac{\begin{array}{c} 0\Gamma, x \overset{0}{:} \mathrm{List}(T) \vdash P\ \mathrm{type} \qquad 0\Gamma \vdash M \overset{0}{:} \mathrm{List}(T) \\ 0\Gamma \vdash N_1 \overset{0}{:} P[\mathsf{nil}/x] \qquad 0\Gamma, h \overset{0}{:} T, t \overset{0}{:} \mathrm{List}(T), p \overset{0}{:} P[t/x] \vdash N_2 \overset{0}{:} P[\mathsf{cons}(h,t)/x] \end{array}}{0\Gamma \vdash \mathrm{recList}_{x.P}\ M\ \{\ \mathsf{nil} \mapsto N_1; \mathsf{cons}(h,t;p) \mapsto N_2\ \} \overset{0}{:} P[M/x]}$$

This eliminator also obeys the usual β-equality laws for a list eliminator, using the resource freedom of the $\sigma = 0$ fragment to duplicate the N_2 term in the cons case.

3.3 Cons-Free Natural Numbers and their Recursor

The datatypes of the previous section still only allow us to write programs in the $\sigma = 1$ fragment that are constant time in the size of their input. As with the simply typed linear system, if we are handed a list of an unknown length, we can only explore it to a fixed depth, determined statically by the program. To write programs that do work proportional to the size of their input, we need some form of iterable datatype. In both our *Cons-free* and LFPL-style QTT systems, we use a natural number datatype.

The *Cons-free* system cannot allow the programmer to construct natural numbers in the $\sigma = 1$ fragment, as this would violate the complexity guarantees. However, we can use the flexibility of QTT to allow free construction of naturals in the $\sigma = 0$ fragment, which allows us to use natural numbers freely in types. Therefore, we have the following introduction rules, only usable in the $\sigma = 0$ fragment:

$$\frac{\Gamma\ \mathsf{ctxt}}{\Gamma \vdash \mathsf{zero} \overset{0}{:} \mathsf{Nat}} \qquad\qquad \frac{\Gamma \vdash M \overset{0}{:} \mathsf{Nat}}{\Gamma \vdash \mathsf{succ}(M) \overset{0}{:} \mathsf{Nat}}$$

The cons-free system allows free duplication of complete natural numbers. This is accomplished by a special construct copying the simply linear typed rule we gave above:

$$\frac{\Gamma \vdash M \overset{\sigma}{:} \mathsf{Nat}}{\Gamma \vdash \mathsf{dupNat}(M) \overset{\sigma}{:} \mathsf{Nat} \otimes \mathsf{Nat}}$$

Anyone who has reasoned about the metatheory of, or implemented a type checker for, dependent types will view this rule with unease as it appears to grant the ability to construct non-canonical values of pair type, and consequently generate non-canonical naturals. We fix this by adding an equational rule to the $\sigma = 0$ fragment, ensuring definitionally that dupNat acts as its name implies:

$$\frac{\Gamma \vdash M \overset{0}{:} \mathsf{Nat}}{\Gamma \vdash \mathsf{dupNat}(M) \equiv (M, M) \overset{0}{:} \mathsf{Nat} \otimes \mathsf{Nat}}$$

Note that this rule is well-typed by the 0-needs-0 property of QTT, and the fact that $0 + 0 = 0$.

The eliminator for these natural numbers takes the following form. Disregarding the usage annotations, it has the same type structure as the normal dependently typed recursor for naturals:

$$\frac{\begin{array}{l} 0\Gamma, x \overset{0}{:} \mathsf{Nat} \vdash P\ \mathsf{type} \\ \Gamma \vdash M \overset{\sigma}{:} \mathsf{Nat} \\ 0\Gamma \vdash N_z \overset{\sigma}{:} P[\mathsf{zero}/x] \\ 0\Gamma, n \overset{0}{:} \mathsf{Nat}, p \overset{\sigma}{:} P[n/x] \vdash N_s \overset{\sigma}{:} P[\mathsf{succ}(n)/x] \end{array}}{\Gamma \vdash \mathsf{rec}_{x.P}\, M\, \{\mathsf{zero} \mapsto N_z; \mathsf{succ}(n; p) \mapsto N_s\} \overset{\sigma}{:} P[M/x]}$$

In the successor case, N_s, there are two bound variables: n for the natural number and p for its induction hypothesis. Note that n is required to be usage 0 no matter what σ is. We need the variable n to be present in order to correctly type the induction hypothesis and the conclusion, but it must be marked as usage 0 to ensure that the resources captured by the number are not duplicated.

This eliminator cannot have a β-equality in the $\sigma = 1$ fragment because there is no way to construct any natural numbers to iterate on in this fragment. In the $\sigma = 0$ fragment, this eliminator obeys the usual β-equality laws for a natural number recursor. This allows us to use it to compute and reason about operations on naturals in this fragment.

The reader is invited to compare this dependently typed rule with the simply typed linear version in Section 2.3. Removing the 0-annotated parts of the rule, and the type dependency, yield the exact same rule. Conversely, when $\sigma = 0$, this rule is identical (up to 0-annotations) to the usual dependently typed recursor for natural numbers, and so we can use it in the types to prove properties of programs just as we do in normal type theory. We will see in Section 6.2 that this rule is realisable by polynomial-time computation, and so is sound for polynomial time.

3.4 LFPL-Style Diamonds, Natural Numbers, and a Recursor that Gives Back

As explained in Section 2.4, the LFPL system differs from the *Cons-free* system in that it is possible to construct natural numbers (and other iterable datatypes), provided one has the necessary diamonds to pay for the construction. As with the natural number type in the *Cons-free* system, it ought not be possible to construct diamonds in the $\sigma = 1$ fragment, as this would amount to the free distribution of diamonds to all which would lead to a collapse in the complexity guarantees of the system. It is possible construct diamonds in the $\sigma = 0$, though:

$$\frac{\Gamma\ \mathsf{ctxt}}{0\Gamma \vdash \Diamond\ \mathsf{type}} \qquad\qquad \frac{\Gamma\ \mathsf{ctxt}}{0\Gamma \vdash * \overset{0}{:} \Diamond} \qquad\qquad \frac{\Gamma \vdash M \overset{0}{:} \Diamond}{\Gamma \vdash M \equiv * \overset{0}{:} \Diamond}$$

The $\Diamond$ type also supports an η-rule in the $\sigma = 0$ fragment, indicating that, in this fragment, it acts the same as a unit type. This allows us to freely use diamonds in types, and to not have to care about the identity of particular diamonds, since by this rule all diamonds are definitionally equal[3].

Construction of natural numbers now requires a $\Diamond$ for zero and a $\Diamond$ and a predecessor for succ:

$$\frac{\Gamma \vdash M \overset{\sigma}{:} \Diamond}{\Gamma \vdash \mathsf{zero}(M) \overset{\sigma}{:} \mathsf{Nat}} \qquad\qquad \frac{\Gamma_1 \vdash M \overset{\sigma}{:} \Diamond \qquad \Gamma_2 \vdash N \overset{\sigma}{:} \mathsf{Nat} \qquad 0\Gamma_1 = 0\Gamma_2}{\Gamma_1 + \Gamma_2 \vdash \mathsf{succ}(M, N) \overset{\sigma}{:} \mathsf{Nat}}$$

In the $\sigma = 0$ fragment, we can construct $\Diamond$s for free, and so construct natural numbers freely as well just as we did for the *Cons-free* system above.

The dependently typed recursor for LFPL-style natural numbers again augments the simply typed linear recursor from Section 2.4 with dependency information:

$$\frac{\begin{array}{l} 0\Gamma, x \overset{0}{:} \mathsf{Nat} \vdash P\ \mathsf{type} \\ \Gamma \vdash M \overset{\sigma}{:} \mathsf{Nat} \\ 0\Gamma, d \overset{\sigma}{:} \Diamond \vdash N_z \overset{\sigma}{:} P[\mathsf{zero}(*)/x] \\ 0\Gamma, d \overset{\sigma}{:} \Diamond, n \overset{0}{:} \mathsf{Nat}, p \overset{\sigma}{:} P[n/x] \vdash N_s \overset{\sigma}{:} P[\mathsf{succ}(*, n)/x] \end{array}}{\Gamma \vdash \mathsf{rec}\, M\, \{\mathsf{zero}(d) \mapsto N_z; \mathsf{succ}(d, n; p) \mapsto N_s\} \overset{\sigma}{:} P[M/x]}$$

We have used $* : \Diamond$ as the value in the types for the zero and successor cases. By the η-rule for diamonds, we could have equally well used the d variable that is in scope in each case.

Unlike the natural number iterator in the *Cons-free* system, this iterator has β-equalities in both the $\sigma = 0$ and $\sigma = 1$ fragments. In the succ case, for example, we have:

$$\begin{array}{rl} & \mathsf{rec}\, (\mathsf{succ}(M_d, M_n))\, \{\mathsf{zero}(d) \mapsto N_z; \mathsf{succ}(d, n; p) \mapsto N_s\} \\ \equiv & N_s[M_d/d, M_n/n, \mathsf{rec}\, M_n\, \{\mathsf{zero}(d) \mapsto N_z; \mathsf{succ}(d, n; p) \mapsto N_s\}/p] \end{array}$$

Note that the fact that the variable n in the N_s term is annotated 0, which allows us to use M_n twice even when we are in the $\sigma = 1$ fragment.

Just as for the *Cons-free* system iterator above, in the $\sigma = 0$ fragment this rule is identical to the usual dependently typed recursor for the natural numbers, so it can be used in the types to reason

[3] *Fungible*, if one wishes to use a monetary metaphor.

about programs. Moreover, we will see in Section 6.3 that this rule is also sound for polynomial time in a system with ◇s.

3.5 Reflection of Realisability

Our final addition to QTT is *reflection of realisability*. In QTT thus far, it has been possible to reason about the non-resourced behaviour of programs. This is because the 0-ing process moving from the $\sigma = 1$ fragment to the $\sigma = 0$ fragment erases all resource information. This is sufficient for reasoning about the extensional behaviour of programs via types, but it is useful to be able to make statements like "this function is realisable in polynomial time" in the types of QTT, something that is not currently possible with the system we have seen so far.

We remedy this by adding a *realisable* type to QTT with the following type formation and introduction and elimination rules:

$$\frac{0\Gamma \vdash A \text{ type}}{0\Gamma \vdash \mathbf{R}(A) \text{ type}} \qquad \frac{0\Gamma \vdash M \overset{1}{:} A}{0\Gamma \vdash \mathbf{R}(M) \overset{\sigma}{:} \mathbf{R}(A)} \qquad \frac{\Gamma \vdash M \overset{\sigma}{:} \mathbf{R}(A)}{\Gamma \vdash \mathbf{R}^{-1}(M) \overset{\sigma'}{:} A}$$

Intuitively, the type $\mathbf{R}(A)$ is inhabited whenever the type A is realisable in the $\sigma = 1$ fragment of the system. In particular, the type $\mathbf{R}(\mathrm{Nat} \to \mathrm{Nat})$ is the type of all realisable functions from natural numbers to natural numbers. In the polynomial time systems we are concerned with here, this is exactly the type of polynomial functions. Note that in the introduction rule, the premise is required to be in the $\sigma = 1$ fragment, to ensure that the type is realisable, while in the elimination rule, the conclusion is in an arbitrary fragment σ'. This flexibility is require to maintain the admissibility of the 0-ing rule.

The equality rules for $\mathbf{R}$ state that the two operations are mutually inverse: $\mathbf{R}(\mathbf{R}^{-1}(M)) \equiv M$, in both fragments, and $\mathbf{R}^{-1}(\mathbf{R}(M)) \equiv M$ in the $\sigma = 0$ fragment. By congruence, the $\sigma = 1$ fragment's definitional equality affects the definitional equality of the $\sigma = 0$ fragment via the $\mathbf{R}(-)$ constructor.

With just the rules given here, the type $\mathbf{R}(A)$ is no more than a statement that a given type is realisable with a polytime implementation. This is enough to do the constructions that we present in the next section, e.g., that polytime functions are closed under composition, but one could imagine stronger reflection principles that allow deeper logical consequences of polytime realisability to be proved internally. We discuss this further in Section 7.2.

Readers familiar with Benton [1994]'s Linear/Non-linear system will note that the $\mathbf{R}(A)$ constructor is the QTT analogue of the right adjont G type constructor in that system. The Σ-types play the role of the left adjoint F types, in a similar way to the dependent linear type system of Krishnaswami et al. [2015].

4 PROGRAMMING AND PROVING WITH POLYTIME

We now explore the possibilities afforded by the combination of polytime guarantees with the specification expressivity of dependent types.

4.1 Building Data Types

We have only defined an iterable natural number datatype for both of our systems above. We could extend both systems to include further iterable inductive types, although in the *Cons-free* system this is not particularly useful due to the prohibition of construction. However, sticking with just the natural numbers, we can use the power of dependent types with a universe to create further datatypes whose size is measured by some iterable natural number. Iteration on the size yields iteration over the full datastructure. For example, in the LFPL system, we can define a type of iterable lists by pairing a size with a type of elements defined by recursion on the size:

$$\mathrm{IList}\,A = (n \overset{1}{:} \mathrm{Nat}) \otimes (\mathrm{rec}_{x.\mathrm{U}}\, n\, \{\mathrm{zero}(d) \mapsto I; \mathrm{succ}(d, n; p) \mapsto A \otimes p\})$$

The nil and cons constructors can now be defined in terms of zero and succ, provided the caller supplies sufficient $\diamond$s. These definitions live in the $\sigma = 1$ fragment, so we annotate them appropriately:

$$\begin{array}{ll} \mathrm{nil} \overset{1}{:} \diamond \to \mathrm{IList}\,A & \qquad \mathrm{cons} \overset{1}{:} \diamond \to A \to \mathrm{IList}\,A \to \mathrm{IList}\,A \\ \mathrm{nil}\, d = (\mathrm{zero}(d), *) & \qquad \mathrm{cons}\, d\, x\, xs = \mathrm{let}\, (n, \mathit{elems}) = xs\, \mathrm{in}\, (\mathrm{succ}(d, n), (x, \mathit{elems})) \end{array}$$

Using the LFPL iterator it is also possible to construct a dependently typed iterator for $\mathrm{IList}\,A$ values. Unfortunately, the current types of the LFPL system are not sufficient to type this as a function, as we have no way of stating that the successor case must be arbitrarily duplicable. Lifting this restriction by means of some modality is future work. The typing rule for the derived list iterator is:

$$\frac{\begin{array}{l} 0\Gamma \vdash A\ \mathrm{type} \qquad 0\Gamma, x \overset{0}{:} \mathrm{IList}\,A \vdash P\ \mathrm{type} \\ \Gamma \vdash M \overset{\sigma}{:} \mathrm{IList}\,A \\ 0\Gamma, d \overset{1}{:} \diamond \vdash N_1 \overset{\sigma}{:} P[\mathrm{nil}(*)/x] \\ 0\Gamma, d \overset{1}{:} \diamond, x \overset{\sigma}{:} A, \mathrm{xs} \overset{0}{:} \mathrm{IList}\,A, p \overset{\sigma}{:} P[xs/x] \vdash N_2 \overset{\sigma}{:} P[\mathrm{cons}(*, x, xs)/x] \end{array}}{\Gamma \vdash \mathrm{rec}_{x.P}\, M\, \{\mathrm{nil}(d) \mapsto N_1; \mathrm{cons}(d, x, xs; p) \mapsto N_2\} \overset{\sigma}{:} P[M/x]}$$

Note that, in the cons case, we have access to the result of iterating over the tail of the list (p), but not to the actual tail of the list (xs).

With our list iterator, it is now possible to write interesting polytime programs. For example, the example used by Hofmann [2003] to demonstrate the expressivity of LFPL is insertion sort. First we define insertion of a natural into a sorted list:

$$\mathrm{insert} \overset{1}{:} \diamond \to \mathrm{Nat} \to \mathrm{IList}\,\mathrm{Nat} \to \mathrm{IList}\,\mathrm{Nat}$$

which requires some ingenuity to write to handle the case where we find the place to insert the item and need access to the remainder of the list. Note that, also, the function consumes a $\diamond$ to construct the new element of the output list, and also that the items in the list are themselves iterable natural numbers. This is needed to account for the comparisons between elements.

Insertion sort is repeated insertion of elements from an original list into a new list. The new list is constructed from the $\diamond$s yielded by the original list:

$$\mathrm{insertionSort} \overset{1}{:} \mathrm{IList}\,A \to \mathrm{IList}\,A$$

The immediate benefit of dependent types in this situation is that it is now possible to state and prove the correctness property of this sorting procedure. Using the fact that the $\sigma = 0$ fragment of QTT is exactly normal type theory, we can use normal dependently typed programmming techniques to establish:

$$\mathrm{insertionSortCorrect} \overset{0}{:} (xs \overset{1}{:} \mathrm{IList}\,A) \to \mathrm{Sorted}(xs, \mathrm{insertionSort}\,\mathrm{xs})$$

where $\mathrm{Sorted}(x, y)$ is some predicate stating that y is a sorted permutation of x. Note that, despite the 1 annotation on the Π-type here, we are free to duplicate xs because types are constructed in the $\sigma = 0$ fragment.

4.2 Polytime Problems

Define a decision problem to be a pair (A, P), where A is a type in the universe U, and $P : A \to \mathsf{U}$ is a predicate on A. For what follows, we are only interested in whether or not $P\,a$ is inhabited for each a. Therefore, we use $P \Leftrightarrow Q$ to stand for equi-inhabitation of two $P, Q : \mathsf{U}$, i.e., $P \Leftrightarrow Q \equiv (P \to Q) \times (Q \to P)$.

We can use the reflection type former defined in Section 3.5 to define a predicate on decision problems that establishes whether or not they are polytime decision problems. Specifically, we can state that there is a polytime realisable boolean-value predicate that reports true exactly when the given element of a is in the predicate:

$$\mathrm{PTIME}(A, P) = (f \overset{1}{:} \mathbf{R}(A \to \mathsf{Bool})) \otimes \Big((a \overset{1}{:} A) \to (\mathbf{R}^{-1}(f)\,a = \mathsf{true}) \Leftrightarrow P\,a\Big)$$

Thus, $\mathrm{PTIME}(A, P)$ is a logical proposition stating that the decision problem (A, P) is decidable in polytime. We make three notes about this definition: (i) *proofs* of $\mathrm{PTIME}(A, P)$, are carried out in the $\sigma = 0$ fragment, where we have the full power of Type Theory to aid us; (ii) this definition is intrinsic, in the sense that, whichever of the polytime systems is chosen, proving that a decision problem is solvable in polytime is a matter of programming, without having to reason directly about machine models and step counting; and (iii) we have defined problems to have arbitrary types A as domains, rather than bitstrings, and so the notion of size attached to an input is intrinsic to the type A chosen.

We can also declare a type of polytime *reductions* between problems. A problem (A, P) can be polytime reduced to a problem (B, Q) if there is an inhabitant of the following type:

$$(A, P) \overset{\textsc{Poly}}{\Rightarrow} (B, Q) = (f \overset{1}{:} \mathbf{R}(A \to B)) \otimes \Big((a \overset{1}{:} A) \to Q(\mathbf{R}^{-1}(f)\,a) \Leftrightarrow P\,a\Big)$$

In words, there must be a polytime function f that preserves and reflects decisions. With this definition, it is possible to prove in our systems that polytime computations are closed under polytime reductions. We note that this definition is, up to the reflection modality, the same as the definition of cartesian container morphism, well known in dependent type theory [Abbott et al. 2005], and speaks to a general conception of containers as "problem/solution" pairings and container morphisms as problem reductions.

4.3 Polytime-Based Complexity Classes

The fact that we can characterise polytime decision problems is perhaps to be expected from a system designed to capture polynomial time realisable programs. However, we can go further to capture the complexity classes of Non-deterministic Polynomial time (NP) and Probabilistic Polynomial time (PP), both of which are based on polytime. We do this by augmenting our polytime functions with additional power in the form of computational effects.

4.3.1 Non-deterministic Polynomial Time. To capture the complexity class NP, we use polynomial time programs augmented with non-determinism, as one might expect. We will not need to reason about equality of these non-deterministic programs, so we can represent non-deterministic choices as binary trees. We suppose a non-iterable datatype defined like so:

$$\begin{array}{l} \mathbf{data}\,\mathrm{ND}\,(A : \mathsf{U}) : \mathsf{U}\,\mathbf{where} \\ \quad \mathsf{return} : A \to \mathrm{ND}\,A \\ \quad \mathsf{choice} : (\mathsf{Bool} \to \mathrm{ND}\,A) \to \mathrm{ND}\,A \end{array}$$

The crucial point here is that the subtrees are represented as a function $\mathsf{Bool} \to \mathrm{ND}\,A$. By the typing rules of QTT, this means that the two branches of this function can share resources (see the encoding of the additive product types by Atkey [2018]). Thus, each branch of this tree can be explored in polynomial time, but not the whole tree itself.

The type ND supports a monad interface via the usual free monad construction, as well as an effect $\mathrm{flip} \overset{1}{:} \mathrm{ND}\,\mathrm{Bool}$ providing access to a bit of non-deterministic information. Thus a program of type $A \to \mathrm{ND}\,B$ in the $\sigma = 1$ fragment will be a polytime program with access to an oracle. In the $\sigma = 0$ fragment, we can write a function that resolves non-determinism using a list of booleans. This function returns nothing if the list of booleans is insufficient to resolve all the choices:

$$\mathrm{runWithOracle} \overset{0}{:} \mathrm{ND}\,A \to \mathrm{List}(\mathrm{Bool}) \to \mathrm{Maybe}\,A$$

With these definitions, we can define Non-deterministic Polynomial time as a predicate on problems:

$$\begin{aligned}\mathrm{NP}(A, P) = {} & (f \overset{1}{:} \mathbf{R}(A \to \mathrm{ND}(\mathrm{Bool}))) \otimes \\ & \Big((a \overset{1}{:} A) \to \Big((bs \overset{1}{:} \mathrm{List}(\mathrm{Bool})) \otimes (\mathrm{runWithOracle}\,(\mathbf{R}^{-1}(f)\,a)\,bs = \mathrm{just\,true})\Big) \Leftrightarrow P\,a\Big)\end{aligned}$$

Thus, a problem is in NP if there is a non-deterministic boolean-valued polynomial time function that has a path to returning true exactly when the input satisfies the predicate. Moreover, it is a quick matter of programming to see that problems in NP are closed under the type of polytime reductions given above.

4.3.2 Bounded-Error Probabilistic Polynomial Time. By changing the computation effects supplied to a program, we can change the complexity class. To capture the class BPP of Bounded-error Probabilistic Polynomial time [Arora and Barak 2009], we use a (non-iterable) data structure representing trees of probabilistic choices, where $\mathbb{Q}[0, 1]$ is some type of (non-iterable) rationals in the closed interval $[0, 1]$:

$$\begin{array}{l}\mathbf{data}\,\mathrm{Dist}\,(A : \mathrm{U}) : \mathrm{U}\,\mathbf{where} \\ \quad \mathrm{return} : A \to \mathrm{Dist}\,A \\ \quad \mathrm{choice} : \mathbb{Q}[0,1] \to (\mathrm{Bool} \to \mathrm{Dist}\,A) \to \mathrm{Dist}\,A\end{array}$$

As in the non-deterministic case, a function $A \to \mathrm{Dist}\,B$ in the $\sigma = 1$ fragment is a polytime probabilistic computation. Again, the use of a function type here ensures that each branch of the tree is constructable in polynomial time, not the whole tree. In the $\sigma = 0$ fragment we can write a function that computes the probability of a Dist Bool computation being true:

$$\mathrm{probTrue} \overset{0}{:} \mathrm{Dist}\,\mathrm{Bool} \to \mathbb{Q}[0, 1]$$

We can now define the class of probabilistic polynomial time decision problems, where the decider is allowed to make probabilistic choices as long as it is correct with probability at least $\frac{2}{3}$:

$$\mathrm{BPP}(A, P) = \quad (f \overset{1}{:} \mathbf{R}(A \to \mathrm{Dist}(\mathrm{Bool}))) \otimes \Big((a \overset{1}{:} A) \to (\mathrm{probTrue}\,(\mathbf{R}^{-1}(f)\,a) \geq \tfrac{2}{3}) \Leftrightarrow P\,a\Big)$$

Again, problems in BPP are easily seen to be closed under polytime reductions.

Probabilistic Polynomial time has previously been considered in the setting of implicit computational complexity by Dal Lago et al. [2021] and Dal Lago and Toldin [2015]. In both cases, they must build probabilistic choice into the language, and have difficulty in directly capturing the class BPP due to its semantic nature, where the correctness of implementation is probabilistic. With a dependently-typed host language, adding probabilistic choice as an effect and capturing the semantic constraint of BPP is straightforward.

5 POLYTIME SOUNDNESS VIA REALISABILITY

In this section and the next, we establish the polytime soundness of our extensions of QTT by adapting a realisability method due to Dal Lago and Hofmann [2011]. This approach is based on a three way coupling between abstract mathematical elements (the *what*), values from a machine model (the *how*), and resource potentials (the *fuel*). Each type in the system is defined as a three way relation between these elements. The set of abstract elements depends on the type being interpreted (e.g., types of natural numbers will be defined in terms of the set $\mathbb{N}$). The machine model is fixed across all types. We describe the particular machine model we use for this paper in Section 5.1. Potentials are arranged into *resource monoids* that we define in Section 5.2. Unlike Dal Lago and Hofmann [2011], we explicitly construct realisers for inductive datatypes (both iterable and non-iterable) instead of relying on second-order polymorphic encodings and special !-style modalities. These explicit constructions are essential for constructing models of our systems.

Agda Formalisation. The key soundness results in this section have been formalised in the Agda proof assistant [Norell 2008]. The Agda formalisation can be found in the associated artefact [Atkey 2023a]. After each definition and result we provide a pointer to the Agda modules where the corresponding formalisation can be found, and note interesting features of the mechanisation.

5.1 Machine Model and Operational Semantics

We demonstrate that every program that can be written in extensions of QTT has the complexity bounds that we claim by translating QTT terms into an untyped CBV λ-calculus with a costed operational semantics. The syntax and rules of our target language are given in Figure 1.

Variables are represented as de Bruijn indicies i, j. Expressions $E \in \mathcal{E}$ can be (anonymous) λ-abstractions, unit, pairing and boolean values, variables, sequencing, application, pair elimination, and conditionals. Note that, with the exception of λ-abstraction and sequencing, expressions never contain nested expressions; instead referring to variables already defined. Values $V \in \mathcal{V}$ can be closures $\mathrm{clo}\langle E, \eta \rangle$, where η is an environment for the closure, unit values, pairs and booleans.

Costed evaluation of expressions in environments is defined by a big-step operational semantics $E, \eta \Downarrow_k V$, where k is the number of steps. For simplicity, all operations cost 1 unit, though this could be generalised. We use $\eta[i]$ to access the ith variable in the environment, counting from the right. The evaluation rules are mostly as one would expect, except that the application rule includes a self reference to the closure being invoked in order to allow recursive definitions.

Agda Formalisation. The machine model is defined in the Agda module `MachineModel`. We use an intrinsically well-scoped syntax, which ensures that all variable accesses are well defined.

5.2 Resource Monoids

As we mentioned above, resource potentials are attached to values to represent the amount of intrinsic potential they have to fuel computation. Resource potentials are organised into resource monoids. To be able to account for the combined potential attached to composite data and programs (e.g., pairs, or functions applied to arguments) we will require monoid structure on potentials. The action of turning potential difference into fuel for computation will be modelled by a difference function. Finally, we require that our resource monoid contains sufficient elements to fuel constant time operations. We gather these requirements into a formal definition as follows, which is a slight reformulation of the resource monoids of Dal Lago and Hofmann [2011]:

Definition 5.1. A *resource monoid* M consists of:

(1) A carrier set $|M|$, whose elements represent amounts of potential. We use Greek letters α, β, γ to denote elements of a resource monoid.

Syntax

$$\begin{array}{lcl}
i, j & \in & \mathbb{N} \\
E \in \mathcal{E} & ::= & \lambda E \mid * \mid (i, j) \mid \mathsf{true} \mid \mathsf{false} \mid i \mid \mathsf{let}\, E_1 \,\mathsf{in}\, E_2 \mid i \cdot j \mid \mathsf{letpair}\, i \,\mathsf{in}\, E \mid \mathsf{if}\, i\, E_1\, E_2 \\
V \in \mathcal{V} & ::= & \mathsf{clo}\langle E, \eta \rangle \mid * \mid (V_1, V_2) \mid \mathsf{true} \mid \mathsf{false} \\
\eta & ::= & [] \mid \eta :: V
\end{array}$$

Evaluation: Construction

$$\frac{}{\lambda E, \eta \Downarrow_1 \mathsf{clo}\langle E, \eta \rangle}\ \textsc{MkClo} \qquad \frac{}{*, \eta \Downarrow_1 *}\ \textsc{MkUnit} \qquad \frac{\eta[i] = V_1 \qquad \eta[j] = V_2}{(i, j), \eta \Downarrow_1 (V_1, V_2)}\ \textsc{MkPair}$$

$$\frac{}{\mathsf{true}, \eta \Downarrow_1 \mathsf{true}}\ \textsc{MkTrue} \qquad \frac{}{\mathsf{false}, \eta \Downarrow_1 \mathsf{false}}\ \textsc{MkFalse}$$

Evaluation: Variable access and Sequencing

$$\frac{\eta[i] = v}{i, \eta \Downarrow_1 v}\ \textsc{Access} \qquad \frac{E_1, \eta \Downarrow_{k_1} V \qquad E_2, (\eta :: V) \Downarrow_{k_2} V'}{\mathsf{let}\, E_1 \,\mathsf{in}\, E_2, \eta \Downarrow_{k_1+1+k_2} V'}\ \textsc{Seq}$$

Evaluation: Elimination

$$\frac{\eta[i] = \mathsf{clo}\langle E, \eta' \rangle \qquad \eta[j] = V \qquad E, (\eta' :: \mathsf{clo}\langle E, \eta' \rangle :: V) \Downarrow_k V'}{(i \cdot j), \eta \Downarrow_{1+k} V'}\ \textsc{App}$$

$$\frac{\eta[i] = (V_1, V_2) \qquad E, (\eta :: V_1 :: V_2) \Downarrow_k V}{\mathsf{letpair}\, i \,\mathsf{in}\, E, \eta \Downarrow_{1+k} V}\ \textsc{LetPair} \qquad \frac{\eta[i] = \mathsf{true} \qquad E_1, \eta \Downarrow_k V}{\mathsf{if}\, i\, E_1\, E_2, \eta \Downarrow_{1+k} V}\ \textsc{IfTrue}$$

$$\frac{\eta[i] = \mathsf{false} \qquad E_2, \eta \Downarrow_k V}{\mathsf{if}\, i\, E_1\, E_2, \eta \Downarrow_{1+k} V}\ \textsc{IfFalse}$$

Fig. 1. Language with CBV Big-step Costed Evaluation Semantics

(2) Commutative monoid structure $(\oplus, \emptyset)$ on $|M|$, so we can add potentials.

(3) a *difference function* $M : |M| \times |M| \to \mathbb{N}_{-\infty}$, where $\mathbb{N}_{-\infty}$ is the natural numbers extended with a negative infinity $-\infty$ and $-\infty + k = -\infty$. A difference $M(\alpha, \beta) = k \in \mathbb{N}$ means that starting with potential α and ending with potential β yields k units of fuel. A difference of $-\infty$ means that α contains insufficient potential to reach β. Differencing must satisfy:

  (a) for all α, $M(\alpha, \alpha) = 0$; and

  (b) for all α, β, γ, $M(\alpha, \beta) + M(\beta, \gamma) \leq M(\alpha, \gamma)$.

  The latter is a "reverse triangle inequality": the fuel recoverable by moving between potential levels α and γ via β may be less than the fuel recoverable moving from α to γ directly.

(4) Differencing and the commutative monoid structure must satisfy:

  (a) $M(\alpha, \beta) \leq M(\alpha \oplus \gamma, \beta \oplus \gamma)$; and

  (b) $M(\alpha, \emptyset) = 0$.

(5) An *accounting function* $acct : \mathbb{N} \to |M|$ such that for all k, $k \leq M(acct(k), \emptyset)$.

For any resource monoid M, we can define an action of $\mathbb{N}$ on M as $n \cdot \alpha = \alpha \oplus \cdots \oplus \alpha$, where the right-hand side has n summands.

Alternative definition. Every resource monoid induces a pre-ordering on its carrier set by $\alpha \leq \beta$ iff $0 \leq M(\alpha, \beta)$. Taking this idea further, we can reformulate a resource monoid as a symmetric monoidal category enriched in the symmetric monoidal category $\mathbb{N}_{-\infty}$, where the monoid structure is addition. The conditions in the definition above amount to the usual identity and composition laws for enriched categories. With this reading, we can see the value $M(\alpha, \beta)$ when it is ≥ 0 as the possibility of moving from α to β levels of potential resource with some amount of residual resource emitted for computation; when it is $-\infty$, moving from α to β is not possible.

Agda Formalisation. Resource monoids are defined in the module `Algebra.ResourceMonoid`. We use a formulation closer to the enriched category theory definition for the actual formalisation, because it avoids having to treat equality in the monoid structure separately from the induced preorder on elements. Thinking of proofs involving the resource monoid as a process of finding a composable sequence of morphisms in a category was a helpful intuition when constructing the realisability model below.

5.2.1 *Specific Resource Monoids.*

The simplest example of a resource monoid is given by the natural numbers $\mathbb{N}$, where each number stands directly an amount of stored fuel.

Definition 5.2 (Natural Number Resource Monoid). Monoid structure is given by normal addition. Differencing is defined as

$$\mathbb{N}(m,n) = \begin{cases} m - n & m \geq n \\ -\infty & \text{otherwise} \end{cases}$$

and $acct(k) = k$. Note that this is the simplest possible resource monoid due to the requirement that the *acct* function must exist.

The differencing operator of the natural number resource monoid can only supply as much fuel as is contained in the potential. For the two polynomial time systems, we need more sophisticated structures, both originally presented by Dal Lago and Hofmann. The fundamental idea with both is to represent potentials as pairs (m, p), where m is a natural number and p is a polynomial. The m tracks the "size" of data as it pertains to the number of times an operation will be repeated by iterating over it — for example, an iterable natural number will have size equal to itself, but a non-iterable natural number may be assigned zero size. The polynomial p tracks the complexity of a program as a function of the size of the input. This leads to a differencing operator that evaluates the polynomial with the size of the data:

Definition 5.3 (Polynomial Resource Monoids). The *Max-Polynomial* resource monoid MaxPoly has carrier set consisting of pairs (m, p) where m is natural number and p is a polynomial with natural number coefficients. Addition of elements is defined as $(m, p) \oplus (n, q) = (m \sqcup n, p + q)$, where $\sqcup$ is the max operator, with $\emptyset = (0, 0)$. Difference is defined as:

$$\text{MaxPoly}((m,p),(n,q)) = \begin{cases} p(m) - q(m) & m \geq n \text{ and } \forall k \geq m. p(k) \geq q(k) \\ -\infty & \text{otherwise} \end{cases}$$

MaxPoly accounts for constant time with constant polynomials: $acct(k) = (0, \lambda x.k)$.

The *Plus-Polynomial* resource monoid PlusPoly is defined the same way as MaxPoly except that the monoid addition adds the natural number components instead of taking their maximum: $(m, p) \oplus (n, q) = (m + n, p + q)$.

It is perhaps easier to see how the differencing operator works in the special case of the difference $\text{MaxPoly}((m, p), (0, 0)) = p(m)$. I.e., if we have code that contains data of size m and a program with complexity p, then running the combination with no expectation of remaining potential yields $p(m)$ available steps. The MaxPoly and PlusPoly resource monoids will be used for the *Cons-free* and LFPL-style systems respectively, as we explain in Section 6 and show how these resource monoids yield the required polytime bounds on programs.

Agda Formalisation. The $\mathbb{N}$ resource monoid is defined in `Algebra.ResourceMonoid.Nat` and the polynomial monoids are both defined in `Algebra.ResourceMonoid.Polynomial`. The definition is parameterised by the "size monoid" operation (either $\sqcup$ or $+$) used to compose sizes.

5.2.2 Resource Sub-Monoids. The separation between sizes of data and complexity of code in the polynomial resource monoids motivates the use of resource sub-monoids to ensure that programs themselves (as opposed to higher order code which may contain closed over data) do not contain data that can be iterated. We do this by requiring that programs' potential must come from a specified resource sub-monoid:

Definition 5.4 (Resource Sub-Monoids). A *resource sub-monoid* $M_0 \subseteq M$ of a resource monoid M consists of a subset $|M_0| \subseteq |M|$ that is closed under the monoid operations and *acct*.

For both MaxPoly and PlusPoly, the elements with zero size component, i.e., of the form $(0, p)$, form a resource sub-monoid that we will use for interpreting programs. We will call these sub-monoids MaxPoly_0 and PlusPoly_0.

5.3 Models of Quantitative Type Theory from Indexed Preorders

Atkey [2018] described a general class of QTT models termed *Quantitative Categories with Families* (QCwFs). Atkey [2018] constructs QCwFs from certain Linear Combinatory Algebras (LCAs), where terms in the $\sigma = 1$ fragment are realised by elements of the LCA. However, there is a mistake in that paper where the interpretation of contexts is stated to be the category of *assemblies* over the LCA, where it ought to be the category of sets paired with realisability relations, with no guarantee that all elements be realisable.

Here, we fix the mistake of Atkey [2018] and provide a more general construction of QCwFs in terms of indexed linear preorders. We construct indexed linear preorders specific to our polytime setting below. They could also be constructed from LCAs.

Definition 5.5. A $\mathbb{N}$-*linear preorder*[4] is a preordered set $(L, \leq)$:

(1) a commutative monoid $(I, - \otimes -)$ that is monotone w.r.t. the order;
(2) is closed: there is an operation $\multimap: L \times L \to L$ such that $x \otimes y \leq z$ iff $x \leq y \multimap z$; and
(3) has a function $! : \mathbb{N} \to L \to L$, to interpret resource requirement adjustments, satisfying:
   (a) $!_0 X \simeq I$, for discarding;
   (b) $!_{m+n} X \leq (!_m X) \otimes (!_n X)$, for duplication;
   (c) $!_m !_n X \leq !_{mn} X$ for nesting;
   (d) $!_1 X \leq X$ for extraction / dereliction;
   (e) $(!_n X) \otimes (!_n Y) \leq !_n (X \otimes Y)$, for distribution; and
   (f) $n \leq m$ implies $!_n X \leq !_m X$, for usage weakening.

The collection of all linear preorders and functions that preserve the order and the operations forms a category LinPreorder.

An *indexed linear preorder* $L : \text{Set}^{\text{op}} \to \text{LinPreorder}$ is a contravariant function, where we write $f^* : L(B) \to L(A)$ for the action of L on functions $f : A \to B$, such that such that reindexing along projections has a right adjoint $L_{\Sigma_{a \in A}.B}(\pi_1^* X, Y) \cong L_A(X, \forall_B Y)$ that commutes with reindexing.

Given an indexed linear preorder $L : \text{Set}^{\text{op}} \to \text{LinPreorder}$, we construct a QCwF model of QTT with the basic type formers from Section 3.1:

[4] We specialise to the semiring $\mathbb{N}$ here, but the same definition works for any suitable semiring $\mathcal{R}$.

(1) Define a category $\mathcal{L}$ of interpretations of contexts with objects that are pairs $(A \in \mathrm{Set}, X \in L(A))$ and morphisms $f : (A, X) \to (B, Y)$ that are functions $f : A \to B$ such that $X \le f^* Y$ (this is the Grothendieck category of L). There is a faithful functor $U : \mathcal{L} \to \mathrm{Set}$. The category $\mathcal{L}$ will be used for interpreting contexts in the $\sigma = 1$ fragment of QTT.
(2) Define scaling of objects of $\mathcal{L}$ by $\pi(A, X) = (A, !_\pi X)$, and addition of (A, X) and (A, Y) as $(A, X \otimes Y)$.
(3) For each set A, define the collection of semantic types $\mathrm{Ty}(A)$ as the collection of $B : A \to \mathrm{Set}$ and $X \in L(\Sigma_{a \in A}.B(a))$. Thus a QTT type consists of an extensional meaning B and its realisability specification X.
(4) For each A and $(B, X) \in \mathrm{Ty}(A)$, the $\sigma = 0$ fragment terms $\mathrm{Tm}(A, (B, X))$ are functions $\Pi_{a \in A}.\ B(a)$. For each context interpretation (A, X) in $\mathcal{L}$ and type interpretation $(B, Y) \in \mathrm{Ty}(A)$, the $\sigma = 1$ fragment terms $\mathrm{RTm}((A, X), (B, Y))$ are functions $f : \Pi_{a \in A}.\ B(a)$ such that $X \le \overline{f}^* Y$, where $\overline{f} : A \to \Sigma_{a \in A}.\ B(a)$ is the section associated with f.
(5) The empty context is interpreted as $(\{*\}, I)$ and context extension $(A, X).n(B, Y)$ (i.e., comprehension) by $(\Sigma_{a \in A}.B(a), \pi_1^* X \otimes !_n Y)$.
(6) Given $(A, X) \in \mathrm{Ty}(C)$ and $(B, Y) \in \mathrm{Ty}(\Sigma_{c \in C}.A(c))$, Σ-types are interpreted similarly to context extension and Π-types are interpreted as $(\lambda c.\ (\Pi_{a \in A(c)}.\ B(c, a)), \forall_A (X \multimap (ev\, f)^* Y))$, where $ev\, f : (\Sigma_{c \in C}.A(c)) \to (\Sigma_{c \in C}.\Sigma_{a \in A(c)}.B(c, a))$ is defined using application of f.
(7) Universe and Equality types are interpreted as normal in Set with the realisability component set to I in both cases. Note that the universe of small types includes resource-relevant realisability information for each type.
(8) Realisability reflection for a type $(B, X) \in \mathrm{Ty}(A)$ is interpreted as the type $(\lambda a.\{b \in B(a) \mid I \le (\lambda a.(a, b))^* X\}, I)$. Thus the set-component of the type is restricted to the elements that are realisable, while the actual realisability component is the "empty" I realisability specification.

Agda Formalisation. The indexed linear preorders are defined in the Agda module `IndexedLinear`. We have not yet completed a formalisation of the construction of a full model of QTT from an indexed linear preorder so this part is currently unmechanised.

5.4 Amortised Complexity Realisability Model

Equipped with our underlying costed model of computation (Section 5.1) and a compositional notion of resource potential (Section 5.2), we can construct models of QTT that witness the resource and type soundness of our complexity constrained systems. We fix a resource monoid M with sub-monoid M_0 and proceed to build an indexed linear preorder of resource accounted realisers.

5.4.1 Indexed Linear Preorder. We now define an indexed linear poset L of realisers over Set that ties together our "mathematical" model of types in Set with our machine model and resource monoid. This construction is a reformulation of Dal Lago and Hofmann [2011]'s realisability models to make it suitable for dependent types. For a set A, the carrier of $L(A)$ is the set of ternary relations $X \subseteq A \times M \times \mathcal{V}$ and we define the ordering $X \le Y$ to hold iff there exists a realising expression $E \in \mathcal{E}$ and potential $\gamma \in M_0$ such that for all $a \in A$, $\alpha \in M$ and $v \in \mathcal{V}$ with $(a, \alpha, v) \in X$, we have that there exists a result $v' \in \mathcal{V}$, step count $k \in \mathbb{N}$ and result potential $\beta \in M$ with:

(1) $E, v \Downarrow_k v'$ (evaluation successfully completes in k steps);
(2) $(a, \beta, v') \in Y$ (the result is well-resourced and satisfies Y); and
(3) $k \le M(\alpha \oplus \gamma, \beta)$ (the step count is within the difference between the initial potential and the result potential).

Note that the definition of realisablity is uniform in the element a – the realising expression E and the potential γ must work for all a – thus the implementation and complexity measure of the transition being modelled cannot depend on what the input is. Put in implementation terms, the input a is not present at runtime. Moreover note that the potential γ attached to the expression E must come from the sub-monoid M_0, indicating that is intended to be data-free, while the potential α for the input is from the full monoid M, so it can contain data and functions.

For $X, Y \in L(A)$, the required elements for symmetric monoidal closed structure are defined as follows. For the tensor product $X \otimes Y \in L(A)$, the realising value must be a pair (v_1, v_2) and the potential of the pair must split into suitable potentials α_1, α_2 for the components. For the residual $X \multimap Y$, the realising value must be a closure with potential to, when added to the potential of an input, compute the output with enough remaining. Note that the potential attached to a closure (α, here) need not be from the sub-monoid M_0. Unlike top-level term interpretations, closures may contain data.

$$\begin{aligned}
X \otimes Y \; &= \{(a, \alpha, (v_1, v_2)) \mid \exists \alpha_1, \alpha_2.\, 0 \le M(\alpha, \alpha_1 \oplus \alpha_2) \wedge X(a, \alpha_1, v_1) \wedge Y(a, \alpha_2, v_2)\} \\
X \multimap Y &= \{(a, \alpha, \mathsf{clo}\langle E, \eta\rangle) \mid \forall \alpha' \in M, v, w \in \mathcal{V}.\, X(a, \alpha', v) \Rightarrow \\
&\qquad\qquad \exists v', k, \beta.\, E, (\eta :: w :: v) \Downarrow_k v' \wedge Y(a, \beta, v') \wedge k \le M(\alpha \oplus \alpha', \beta)\}
\end{aligned}$$

The seemingly useless $w \in \mathcal{V}$ in the formula for $X \multimap Y$ is a dummy argument standing for the self-referential reference to the closure used for defining recursive programs.

Each $L(A)$ has a terminal (i.e. top) element, which is also the unit for $\otimes$, defined as $I_A = \{(a, \alpha, *) \mid a \in A, \alpha \in M\}$. The potential α here is unrestricted, so I_A can consume an arbitrary resource.

$\mathbb{N}$-Graded exponentials in each $L(A)$ are defined using the action of $(\mathbb{N}, \le)$ on M defined above. When $n > 0$, the modality $!_n$ has no effect on realising values. It only serves to alter the resource potentials. In the $n = 0$, case the realising value must be $*$, in order to satisfy the $!_0 X \cong I$ condition in Definition 5.5 3(a):

$$\begin{aligned}
!_0 X &= \{(a, \alpha, *) \mid a \in A, \alpha \in M\} \\
!_n X &= \{(a, \alpha, v) \mid \exists \alpha'.\, M(n \cdot \alpha', \alpha) = 0 \wedge (a, \alpha', v) \in X\}
\end{aligned}$$

L also has arbitrary Set-indexed products, realised “lazily” as functions that take dummy arguments. For $A \in \mathrm{Set}$ and $B \in A \longrightarrow \mathrm{Set}$ and $X \in L(\Sigma A.\, B)$, we define $\forall_B X \in L(A)$ similarly to $\multimap$ above, but with different resource and indexing requirements:

$$\forall_B X = \{(a, \alpha, \mathsf{clo}\langle E, \eta\rangle) \mid \forall b, v.\, \exists v', \beta, k.E, (\eta :: v :: *) \Downarrow_k v' \wedge X((a, b), \beta, v') \wedge k \le M(\alpha, \beta)\}$$

Note, as with the definition of $X \le Y$ above, the realiser closure $\mathsf{clo}\langle E, \eta\rangle$ must be chosen uniformly for all b. This definition also appears to allow arbitrary computation (paid for by α) to happen when the realising closure is applied, but the potential α will only ever be greater than β by enough to handle the administrative costs of applying the function.

To complete the construction of L as an indexed linear preorder, we need to give realisers for each of the required inequalities in Definition 5.5. In each case, this is a matter of programming in the language of Section 5.1. For example, transitivity of the order is realised by sequencing of expressions. The potentials are calculated by counting the steps in the ensuing programs.

Proposition 5.6. *L, with I, $\otimes$, $\multimap$, $!_n$, and $\forall_B$ defined above, is an indexed linear preorder.*

Agda Formalisation. The construction of this indexed linear preorder and the proof of Proposition 5.6 are formalised in the Adga module `AmortisedRealisabilityModel`.

5.4.2 Non-Iterable Data Types. The model of QTT constructed in Proposition 5.6 does not yet include any useful base types. Iterable types, which are the ones that induce non-constant time complexities, require specific properties of resource monoids that we introduce in Section 6.

Before that, we show how to define realisers for the representative examples of non-iterable types from Section 2.2 and Section 3.2. Booleans are the simplest case, with only two cases and no chance of iteration. Lists are more complex: we can have non-iterable lists containing iterable data.

Booleans. Fix $\mathbb{B} = \{\mathit{tt}, \mathit{ff}\}$ as our set of boolean elements. We define an element of $L(\mathbb{B})$ to represent boolean values:

$$\mathrm{Bool} = \{(\mathit{tt}, \alpha, \mathrm{true}) \mid \alpha \in M\} \cup \{(\mathit{ff}, \alpha, \mathrm{false}) \mid \alpha \in M\}$$

Thus, the boolean tt is represented by the value true and ff is represented by false. In both cases, we allow arbitrary potential α to be attached.

Realisability of the construction and elimination of booleans amounts to the existence of the following inequalities. In any preorder $L(A)$, we have $I_A \leq \mathit{tt}^*\mathrm{Bool}$ and $I_A \leq \mathit{ff}^*\mathrm{Bool}$ (treating tt and ff as constant functions $A \longrightarrow \mathbb{B}$). These inequalities are realised by the corresponding true/false expression. For conditionals, the types involved are a little more complex to ensure agreement between boolean manipulations at the Set-level and the realising computations. To get a realiser for a conditional, we require a set A, an element $X \in L(A)$ (standing for the context) and an element $Y \in L(A \times \mathbb{B})$ (standing for the target type) and the existence in $L(A)$ of inequalities $X \leq (\lambda a.(a, \mathit{tt}))^*Y$, for the true case, and $X \leq (\lambda a.(a, \mathit{ff}))^*Y$, for the false case. When we have all these, we get in $L(A \times \mathbb{B})$ an inequality $\pi_1^*X \otimes \pi_2^*\mathrm{Bool} \leq Y$. This construction suffices to realise the rules for QTT booleans in Section 3.2.

Lists. Lists are a little more involved, due to the need to explicitly manage a context that applies to all elements of the list. Let $\mathrm{List}(B)$ be the set of lists with elements from a set B. If we have $A : \mathrm{Set}$ and $B : A \longrightarrow \mathrm{Set}$ and $X \in L(\Sigma a : A.\, Ba)$, then the resourced lists predicate $\mathrm{RList}(X) \in L(\Sigma a : A.\, \mathrm{List}(Ba))$ must satisfy the equation:

$$\begin{aligned}\mathrm{RList}(X) = \;& \{((a, []), \alpha, (\mathrm{false}, *)) \mid \alpha \in M\} \\ & \cup \\ & \{((a, b :: bs), \alpha, (\mathrm{true}, (v_1, v_2))) \mid \\ & \quad \exists \alpha_1, \alpha_2 . 0 \leq M(\alpha, \alpha_1 \oplus \alpha_2) \wedge ((a, b), \alpha_1, v_1) \in X \wedge ((a, bs), \alpha_2, v_2) \in \mathrm{RList}(X)\}\end{aligned}$$

This equation has a least solution, by induction on the length of the list being realised. This definition is somewhat involved, but in essence states that a list is represented by tagged pairs, where false represents nil and true represents cons, and that the potential is distributed amongst the elements of the list as needed.

Agda Formalisation. The construction of realisers for booleans and lists are carried out in the Agda modules `AmortisedModel.Bool` and `AmortisedModel.List`.

6 REALISING ITERATION FOR IMPLICIT POLYNOMIAL TIME

The models constructed in the previous section only allow for constant-time programs to be realised. To interpret the iterators of the *Cons-free* and LFPL-style systems, we need to use the MaxPoly and PlusPoly resource monoids. We do this in this section, where first we establish some operations that will be useful to see how they capture the nesting of iterations inherent to polytime computation.

6.1 Iteration Resource Monoids

To interpret iteration over a resource monoid (M, M_0), we require additional structure, which we call an *Iteration Resource Monoid* to account for measurement of the sizes of iterable data structures and the effects of iteration on potentials.

6.1.1 Definition. We require:

(1) a function *size* : $\mathbb{N} \longrightarrow M$ that gives the potential of an iterable data structure of a given size;
(2) a function *raise* : $M \longrightarrow M$ that raises the (polynomial) degree of some potential; and
(3) a function *scale* : $\mathbb{N} \times M \longrightarrow M$ that scales a potential for a fixed number of iterations.

These functions must satisfy the following properties:

(1) M_0 is closed under the *raise* operation;
(2) for all α and n, $0 \leq M(\mathit{raise}(\alpha) \oplus \mathit{size}(n), \mathit{scale}(n, \alpha) \oplus \mathit{size}(n))$; and
(3) for all $\alpha \in M_0$ and n, $0 \leq M(\mathit{scale}(1 + n, \alpha), \alpha \oplus \mathit{scale}(n, \alpha))$.

The first property states that *raise* is suitable as potential for whole programs, meaning that it does not make any requirements on the existence of iterable data. Note that we do *not* require M_0 to contain $\mathit{size}(n)$ – programs themselves may not contain iterable data, all potential for iteration must be delivered externally. A useful intuition is that $\mathit{scale}(n, \alpha)$ represents the potential required for at most n iterations that require potential α, whereas $\mathit{raise}(\alpha)$ represents the potential required for a number of iterations that depends on the context. This is the motivation behind the second required property, which states that having $\mathit{raise}(\alpha)$ potential implies having $\mathit{scale}(n, \alpha)$ potential when the current size is n. The third property states that *scale* decomposes as expected on potentials that do not include any size potential.

Note that $\mathit{scale}(n, \alpha)$ is not the same as the action $n \cdot \alpha$ defined in Section 5.2. The latter operation scales both size and function potential, but the former only scales the function potential.

6.1.2 Polynomial Iteration Resource Monoids. Both of the polymonial resource monoids defined in Definition 5.3 support the structure of an Iteration Resource Monoid. We define:

(1) $\mathit{size}(n) = (n, 0)$
(2) $\mathit{raise}(n, p) = (n, xp)$
(3) $\mathit{scale}(m, (n, p)) = (n, m \cdot p)$

Note that *raise* does indeed raise the degree of the polynomial involved. Property 2 above is satisfied because for any polynomial we have $(m \cdot p)(x) \leq (xp)(x)$ whenever $m \leq x$.

6.1.3 Realising Iterable Natural Numbers. For any natural number n, we define its representation as a value $\mathrm{natValue}(n) \in \mathcal{V}$ by recursion:

$$\mathrm{natValue}(0) = (\mathrm{true}, *) \qquad\qquad \mathrm{natValue}(1 + n) = (\mathrm{false}, \mathrm{natValue}(n))$$

This representation uses a tagged pair approach similar to our representation of lists in Section 5.4.2. Using this, we can define what it means for a natural number to be realisable via $\mathrm{Nat} \in L(\mathbb{N})$:

$$\mathrm{Nat} = \{(n, \alpha, \mathrm{natValue}(n)) \mid n \in \mathbb{N}, 0 \leq M(\alpha, \mathit{size}(n + 1))\}$$

So a natural number n is realised by the value $\mathrm{natValue}(n)$ as long as we have at least $\mathit{size}(n + 1)$ potential (we add one to make the LFPL soundness proof easier). This gives us the ability to represent natural numbers as a type in QTT, but in order to iterate (and construct in the case of LFPL), we need to construct specific realisers for the *Cons-free* and LFPL systems.

6.2 The Cons-Free System

The *Cons-free* system uses the MaxPoly resource monoid, with the distinguished sub-monoid being those elements that are 0 in the size component. We enumerate the features of the *Cons-free* system and justify their realisability with the MaxPoly resource monoid:

(1) Duplication of natural numbers by $\mathrm{dupNat}(M)$ is realisable by the expression $(0, 0)$, which creates a pair by copying the input variable twice. By the cost semantics in Section 5.1, this takes 1 step of computation (we assume that it is actually implemented via some pointer copy). The resource accounting for this realiser works because the size component required for the output is the maximum of the size components of the two elements, and since $n \sqcup n = n$, we have enough resources to fulfil this.

(2) Construction of natural numbers is not realisable. In a putative succ rule, we would need to get an additional unit of size resource from nowhere.
(3) Iteration is realised by constructing a realising expression in the expression language from the given expressions for the zero and successor cases that uses the in-built recursion of the language. The proof that resources are correctly accounted for is carried out by induction on the natural being iterated over. For n, we require potential $\mathit{scale}(n, \mathit{acct}(4) \oplus \gamma_{succ}) \oplus (\mathit{acct}(2) \oplus \gamma_{zero})$, where γ_{succ} and γ_{zero} are the potentials required by the successor and zero cases respectively. By Property (2) of iteration resource monoids, above, we know that $\mathit{raise}(\mathit{acct}(4) \oplus \gamma_{succ}) \oplus (\mathit{acct}(2) \oplus \gamma_{zero})$ always dominates this requirement when paired with the potential $\mathit{size}(n)$ from the input. Therefore, this latter expression, plus some administrative set up costs, is the required potential for the whole iterator.

Together, we have a soundness result for the *Cons-free* system, that ensures that every term in the $\sigma = 1$ fragment is realisable by a *correct* program that terminates in polynomial time for all inputs:

Theorem 6.1 (Soundness for the *Cons-free* System). *If we have a term* $n \overset{1}{:} \mathrm{Nat} \vdash M \overset{1}{:} T(n)$ *then there exists a realising expression* E *and polynomial* p *such that for all* $n \in \mathbb{N}$*, there exists* $v \in \mathcal{V}$ *and* $k \in \mathbb{N}$ *such that* $E, [\mathrm{natValue}(n)] \Downarrow_k v$*,* $k \leq p(n)$ *and* v *is a realising value for* $[\![M]\!](n) \in [\![T]\!](n)$.

Agda Formalisation. The realisability of the *Cons-free* system iterator and the soundness property of the whole system are formalised in the Agda modules `ConsFree` and `ConsFree.Iterator`. The soundness theorem is a combination of this and the QTT model sketched in Section 5.3.

6.3 The LFPL System

The LFPL system uses the PlusPoly resource monoid, with the distinguished sub-monoid again being those elements that are 0 in the size component. With this resource monoid, the capabilities offered at the QTT level are altered:

(1) We can no longer duplicate natural numbers, because Nat ⊗ Nat requires twice as much size resource as Nat, due to the combining operation on size potentials being addition.
(2) We define the realisability specification for diamonds $\diamond \in L(1)$ to be $\diamond = \{(*, \alpha, *) \mid 0 \leq M(\alpha, \mathit{size}(1))\}$. Thus, a diamond represents at least one unit of size resource, matching the intuitive explanation given in Section 2.4.
(3) With this definition of realisability for ◇s, it is possible to realise the zero and succ constructors for natural numbers. By the additive combination of size resources we get 1 from the diamond and $n + 1$ from the predecessor to total $n + 2$ for a new number. Note that, even if we add a ◇ type to the *Cons-free* system, it would still not be possible to realise the constructors, because we would only have $1 \sqcup (n + 1) = n + 1$ size resource for the output.
(4) The construction of the recursor follows a very similar proof to the realisability of *Cons-free* iterator, up to some additional work to make sure that the dummy $*$ values representing the diamond components end up in the right places. This additional work is revealed in the required potential for the LFPL iterator being $\mathit{raise}(\mathit{acct}(8) \oplus \gamma_{succ}) \oplus (\mathit{acct}(2) \oplus \gamma_{zero})$, so slightly higher in the successor case.

Soundness for the LFPL system is similar to the *Cons-free* system, except for a +1 to the input to the polynomial, to account for the fact that we cost one size unit for the zero constructor.

Theorem 6.2 (Soundness for the LFPL-style System). *If we have a term* $n \overset{1}{:} \mathrm{Nat} \vdash M \overset{1}{:} T(n)$ *then there exists a realising expression* E *and polynomial* p *such that for all* $n \in \mathbb{N}$*, there exists* $v \in \mathcal{V}$ *and* $k \in \mathbb{N}$ *such that* $E, [\mathrm{natValue}(n)] \Downarrow_k v$*,* $k \leq p(n + 1)$ *and* v *is a realising value for* $[\![M]\!](n) \in [\![T]\!](n)$.

The proofs of well-accounted realisability for the LFPL iterator, and the *Cons-free* iterator, could be adapted to any other inductively defined type that is finitely branching. This is not immediately necessary, as evidenced by the construction of other datatypes in Section 4.1. Nevertheless, native tree type where the iterability is proportional to the total number of nodes would be useful.

Agda Formalisation. The realisability of the LFPL system iterator and the soundness property of the whole system are formalised in the Agda modules `LFPL` and `LFPL.Iterator`.

7 RELATED AND FUTURE WORK

We have presented two extensions of Quantitative Type Theory that soundly and completely capture polynomial time. This allows for an expressive combination of verification and complexity constrained computation, including characterisations of the classes P, NP, and BPP. We now discuss related work, and take a look at where the combination of polytime and dependency could take us.

7.1 Related Work

Implicit Computational Complexity with Linear Types. Implicit Computational Complexity [Dal Lago 2011] is a vast field, so we only survey closely related works. We have already mentioned the Bounded Linear Logic [Girard et al. 1992], Soft Affine Logic [Lafont 2004], Light Linear Logic [Girard 1998] and LFPL [Hofmann 1999] systems, which all use linear typing to implicitly capture polynomial time. Jones [2001] characterises polynomial time using first-order functional programs without constructors. Thus it shares a method with our *Cons-free* system, but we use linear typing to permit controlled use of higher-order functions. Other approaches to polynomial time use stratification or information flow tracking to ensure that the outputs of iteration may not be used unrestrictedly to drive further iteration. For example, [Bellantoni and Cook 1992] and [Hainry and Péchoux 2023]. Below polynomial time, systems have be devised to capture LOGSPACE [Dal Lago and Schöpp 2016]. Above polynomial time, systems such as Elementary Affine Logic (EAL) capture all Elementary-time functions [Coppola and Martini 2001].

We have used Dal Lago and Hofmann [2011]'s technique to prove soundness of our extension of QTT. This technique has been successfully applied to many other linear typing based systems, such as BLL [Dal Lago and Hofmann 2010a; Hofmann and Scott 2004] and LLL [Dal Lago and Hofmann 2010b] and EAL. In contrast to most of those systems, we do not use restricted !-modalities and second order encodings to express datatypes. Our explicit datatype approaches enabled our combination of dependent types and polynomial time.

Explicit Resource Accounting with Dependent Types. In contrast to the implicit systems, previous works have constructed systems that give explicit resource bounds via typing. Examples include Hoffmann et al. [2017]'s Resource Allocated ML (RAML) and Rajani et al. [2021], both of which are based on ideas of type-based amortised complexity analysis arising from Hofmann [1999]'s ideas, via the work of Hofmann and Jost [2003]. More details are to be found in the survey paper of Hoffmann and Jost [2022]. Another approach is to track costs at the value level instead of the types. Danielsson [2008] describes a system that uses a "tick" effect to count steps of computation, which can be reasoned about via dependent types. Niu et al. [2022] take this idea further by employing a modality-based phase separation to ensure that tick counting never interferes with the functional business of programs. McCarthy et al. [2016] is another tick effect based system in Coq. All of these tick-counting techniques rely on the programmer correctly annotating the program with tick effects to count the resource usage they are interested in, in contrast our intrinsic approach.

Linear and Substructural Dependent Types. We chose QTT as the particular combination of linear and dependent types for our systems. Other systems include systems such as those by Cervesato and Pfenning [2002], Krishnaswami et al. [2015], and Vákár [2014] which all use a strict separation between linear and non-linear variables. This strict separation would mean that we could not as easily move programs from the linear fragment into the types, as in Section 4. Systems that are more like QTT in that they do not have a strict separation of variables include those of Moon et al. [2021], Choudhury et al. [2021], and Abel et al. [2023]. These systems differ from QTT in that they do not include a complete copy of unrestricted type theory as QTT does in its $\sigma = 0$ fragment, because they all track usage in types as well as terms, so it is not clear how to use them for unrestricted reasoning as we do with QTT. Fu et al. [2022] present a system that is closer to QTT but does not include a universe type, which we used in Section 4.3 to be able to characterise complexity classes as predicates decidable in restricted complexity.

7.2 Future Work

Implementation. We currently lack an implementation of our extension of QTT, which hampers further investigation of programming and proving with polytime along the lines of Section 4. Idris 2 [Brady 2021] is an implementation of QTT, but cannot be used directly because its facility for defining datatypes is too liberal, not making a distinction between iterable and non-iterable datatypes. A further implementation-focused question is whether or not the term-level polytime guarantees can be turned to type-level guarantees to guarantee polytime typechecking.

Other Complexity Classes. We have been able to characterise the classes NP and BPP in terms of our underlying characterisation of P (Section 4.3). It seems straighforward to extend this to related classes like coNP, RP, etc. It also seems feasible to adapt the techniques presented here to other complexity classes such as LOGSPACE and ELEMENTARY, given the simply typed linear systems mentioned above. Complexity classes based on circuits may be more challenging, but we do now have a way to characterise circuits that are generatable in polynomial time.

Explicit Resource Tracking. Our construction already includes soundness of a system with intrinsic but explicit resource tracking where $\diamond$s are used to pay for every step of computation but never returned, via the natural number resource monoid defined in Section 5.2.1. Investigation of such a system may yield a system that tracks the intrinsic cost of programs precisely and explicitly.

Towards a Synthetic Computational Complexity Theory? The realisability type $\mathbf{R}(A)$ described in Section 3.5 allows us to internalise the realisability of certain functions into the logical ($\sigma = 0$) fragment of the calculus. However, it is not possible to derive any logical consequences from this other than turning it back into a function. This limitation becomes acute when trying to prove results from standard Computational Complexity theory. Even though we can characterise the class NP, as we did in Section 4.3.1, and it is a “matter of programming” to show that 3-SAT is in NP, we cannot prove the Cook-Levin theorem that 3-SAT is NP-complete. This is because the proof relies on obtaining the *source code* of the program solving an NP problem and then encoding that program in 3-SAT. To do this in our setting, we would need to internalise the soundness property (Theorem 6.2) as an axiom, stating that for a realisable polytime function there (merely) exists a realising expression E that completes in polynomial time, and then writing polytime encodings into 3-SAT. We hope that the addition of such an axiom to our system would lead to an expressive machine-free *Synthetic Computational Complexity Theory*, analogous to the Church-Turing axiom for Synthetic Computability Theory as described by Bauer [2005].

ACKNOWLEDGMENTS

Thanks to Anton Lorenzen and Fredrik Nordvall Forsberg for their comments on an earlier revision, and to the anonymous POPL reviewers for their detailed, perceptive, and interesting reviews.

This work was funded by the Engineering and Physical Sciences Research Council: Grant number EP/T026960/1, *AISEC: AI Secure and Explainable by Construction*.

DATA AVAILABILITY STATEMENT

The Agda source files and rendered HTML for this paper is available from Zenodo [Atkey 2023a]. The source files are also available online at GitHub: https://github.com/bobatkey/qtt-models.

REFERENCES

Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. 2005. Containers: Constructing strictly positive types. *Theor. Comput. Sci.* 342, 1 (2005), 3–27. https://doi.org/10.1016/j.tcs.2005.06.002

Andreas Abel, Nils Anders Danielsson, and Oskar Eriksson. 2023. A Graded Modal Dependent Type Theory with a Universe and Erasure, Formalized. *Proc. ACM Program. Lang.* 7, ICFP, Article 220 (aug 2023), 35 pages. https://doi.org/10.1145/3607862

Klaus Aehlig and Helmut Schwichtenberg. 2002. A syntactical analysis of non-size-increasing polynomial time computation. *ACM Trans. Comput. Log.* 3, 3 (2002), 383–401. https://doi.org/10.1145/507382.507386

Sanjeev Arora and Boaz Barak. 2009. *Computational Complexity - A Modern Approach.* Cambridge University Press. http://www.cambridge.org/catalogue/catalogue.asp?isbn=9780521424264

Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 56–65. https://doi.org/10.1145/3209108.3209189

Robert Atkey. 2023a. *Agda formalisation of Polynomial Time and Dependent Types.* https://doi.org/10.5281/zenodo.8425923

Robert Atkey. 2023b. Polynomial Time and Dependent Types - Extended Version. (2023). https://doi.org/10.48550/arXiv.2307.09145 arXiv:2307.09145.

Patrick Baillot, Marco Gaboardi, and Virgile Mogbil. 2010. A PolyTime Functional Language from Light Linear Logic. In *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6012)*, Andrew D. Gordon (Ed.). Springer, 104–124. https://doi.org/10.1007/978-3-642-11957-6_7

Patrick Baillot and Virgile Mogbil. 2004. Soft lambda-Calculus: A Language for Polynomial Time Computation. In *Foundations of Software Science and Computation Structures, 7th International Conference, FOSSACS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 2987)*, Igor Walukiewicz (Ed.). Springer, 27–41. https://doi.org/10.1007/978-3-540-24727-2_4

Andrew Barber. 1996. *Dual Intuitionistic Linear Logic.* Technical Report. University of Edinburgh.

Andrej Bauer. 2005. First Steps in Synthetic Computability Theory. In *Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics, MFPS 2005, Birmingham, UK, May 18-21, 2005 (Electronic Notes in Theoretical Computer Science, Vol. 155)*, Martín Hötzel Escardó, Achim Jung, and Michael W. Mislove (Eds.). Elsevier, 5–31. https://doi.org/10.1016/j.entcs.2005.11.049

Stephen J. Bellantoni and Stephen A. Cook. 1992. A New Recursion-Theoretic Characterization of the Polytime Functions. *Comput. Complex.* 2 (1992), 97–110. https://doi.org/10.1007/BF01201998

P. N. Benton. 1994. A Mixed Linear and Non-Linear Logic: Proofs, Terms and Models (Extended Abstract). In *Computer Science Logic, 8th International Workshop, CSL '94, Kazimierz, Poland, September 25-30, 1994, Selected Papers (Lecture Notes in Computer Science, Vol. 933)*, Leszek Pacholski and Jerzy Tiuryn (Eds.). Springer, 121–135. https://doi.org/10.1007/BFb0022251

Edwin C. Brady. 2021. Idris 2: Quantitative Type Theory in Practice. In *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference) (LIPIcs, Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:26. https://doi.org/10.4230/LIPIcs.ECOOP.2021.9

Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. 2014. A Core Quantitative Coeffect Calculus. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014*. 351–370. https://doi.org/10.1007/978-3-642-54833-8_19

Iliano Cervesato and Frank Pfenning. 2002. A Linear Logical Framework. *Inf. Comput.* 179, 1 (2002), 19–75. https://doi.org/10.1006/inco.2001.2951

Pritam Choudhury, Harley Eades III, Richard A. Eisenberg, and Stephanie Weirich. 2021. A graded dependent type system with a usage-aware semantics. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–32. https://doi.org/10.1145/3434331

Paolo Coppola and Simone Martini. 2001. Typing Lambda Terms in Elementary Logic with Linear Constraints. In *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001, Krakow, Poland, May 2-5, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2044)*, Samson Abramsky (Ed.). Springer, 76–90. https://doi.org/10.1007/3-540-45413-6_10

Ugo Dal Lago. 2011. A Short Introduction to Implicit Computational Complexity. In *Lectures on Logic and Computation - ESSLLI 2010 Copenhagen, Denmark, August 2010, ESSLLI 2011, Ljubljana, Slovenia, August 2011, Selected Lecture Notes (Lecture Notes in Computer Science, Vol. 7388)*, Nick Bezhanishvili and Valentin Goranko (Eds.). Springer, 89–109. https://doi.org/10.1007/978-3-642-31485-8_3

Ugo Dal Lago and Martin Hofmann. 2010a. Bounded Linear Logic, Revisited. *Log. Methods Comput. Sci.* 6, 4 (2010). https://doi.org/10.2168/LMCS-6(4:7)2010

Ugo Dal Lago and Martin Hofmann. 2010b. A Semantic Proof of Polytime Soundness of Light Affine Logic. *Theory Comput. Syst.* 46, 4 (2010), 673–689. https://doi.org/10.1007/s00224-009-9210-x

Ugo Dal Lago and Martin Hofmann. 2011. Realizability models and implicit complexity. *Theor. Comput. Sci.* 412, 20 (2011), 2029–2047. https://doi.org/10.1016/j.tcs.2010.12.025

Ugo Dal Lago, Reinhard Kahle, and Isabel Oitavem. 2021. A Recursion-Theoretic Characterization of the Probabilistic Class PP. In *46th International Symposium on Mathematical Foundations of Computer Science (MFCS 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 202)*, Filippo Bonchi and Simon J. Puglisi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 35:1–35:12. https://doi.org/10.4230/LIPIcs.MFCS.2021.35

Ugo Dal Lago and Ulrich Schöpp. 2016. Computation by interaction for space-bounded functional programming. *Inf. Comput.* 248 (2016), 150–194. https://doi.org/10.1016/j.ic.2015.04.006

Ugo Dal Lago and Paolo Parisen Toldin. 2015. A higher-order characterization of probabilistic polynomial time. *Inf. Comput.* 241 (2015), 114–141. https://doi.org/10.1016/J.IC.2014.10.009

Nils Anders Danielsson. 2008. Lightweight semiformal time complexity analysis for purely functional data structures. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, George C. Necula and Philip Wadler (Eds.). ACM, 133–144. https://doi.org/10.1145/1328438.1328457

Peng Fu, Kohei Kishida, and Peter Selinger. 2022. Linear Dependent Type Theory for Quantum Programming Languages. *Log. Methods Comput. Sci.* 18, 3 (2022). https://doi.org/10.46298/lmcs-18(3:28)2022

Dan R. Ghica and Alex I. Smith. 2014. Bounded Linear Types in a Resource Semiring. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014*. 331–350. https://doi.org/10.1007/978-3-642-54833-8_18

Jean-Yves Girard. 1987. Linear Logic. *Theor. Comput. Sci.* 50 (1987), 1–102. https://doi.org/10.1016/0304-3975(87)90045-4

Jean-Yves Girard. 1998. Light Linear Logic. *Inf. Comput.* 143, 2 (1998), 175–204. https://doi.org/10.1006/inco.1998.2700

Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. 1992. Bounded Linear Logic: A Modular Approach to Polynomial-Time Computability. *Theor. Comput. Sci.* 97, 1 (1992), 1–66. https://doi.org/10.1016/0304-3975(92)90386-T

Armaël Guéneau, Arthur Charguéraud, and François Pottier. 2018. A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 533–560. https://doi.org/10.1007/978-3-319-89884-1_19

Emmanuel Hainry and Romain Péchoux. 2023. A General Noninterference Policy for Polynomial Time. *Proc. ACM Program. Lang.* 7, POPL (2023), 806–832. https://doi.org/10.1145/3571221

Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards automatic resource bound analysis for OCaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 359–373. https://doi.org/10.1145/3009837.3009842

Jan Hoffmann and Steffen Jost. 2022. Two decades of automatic amortized resource analysis. *Math. Struct. Comput. Sci.* 32, 6 (2022), 729–759. https://doi.org/10.1017/S0960129521000487

Martin Hofmann. 1997. Syntax and Semantics of Dependent Types. In *Semantics and Logics of Computation*. Cambridge University Press, 79–130.

Martin Hofmann. 1999. Linear Types and Non-Size-Increasing Polynomial Time Computation. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*. IEEE Computer Society, 464–473. https://doi.org/10.1109/LICS.1999.782641

Martin Hofmann. 2003. Linear types and non-size-increasing polynomial time computation. *Inf. Comput.* 183, 1 (2003), 57–85. https://doi.org/10.1016/S0890-5401(03)00009-9

Martin Hofmann and Steffen Jost. 2003. Static prediction of heap space usage for first-order functional programs. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, Alex Aiken and Greg Morrisett (Eds.). ACM, 185–197. https://doi.org/10.1145/604131.604148

Martin Hofmann and Philip J. Scott. 2004. Realizability models for BLL-like languages. *Theor. Comput. Sci.* 318, 1-2 (2004), 121–137. https://doi.org/10.1016/j.tcs.2003.10.019

Gérard P. Huet. 1997. The Zipper. *J. Funct. Program.* 7, 5 (1997), 549–554. https://doi.org/10.1017/s0956796897002864

Neil D. Jones. 2001. The expressive power of higher-order types or, life without CONS. *J. Funct. Program.* 11, 1 (2001), 5–94. https://doi.org/10.1017/s0956796800003889

Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. 2015. Integrating Linear and Dependent Types. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 17–30. https://doi.org/10.1145/2676726.2676969

Yves Lafont. 2004. Soft linear logic and polynomial time. *Theor. Comput. Sci.* 318, 1-2 (2004), 163–180. https://doi.org/10.1016/j.tcs.2003.10.018

Conor McBride. 2016. I Got Plenty o' Nuttin'. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Lecture Notes in Computer Science, Vol. 9600)*, Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella (Eds.). Springer, 207–233. https://doi.org/10.1007/978-3-319-30936-1_12

Jay A. McCarthy, Burke Fetscher, Max S. New, Daniel Feltey, and Robert Bruce Findler. 2016. A Coq Library for Internal Verification of Running-Times. In *Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9613)*, Oleg Kiselyov and Andy King (Eds.). Springer, 144–162. https://doi.org/10.1007/978-3-319-29604-3_10

Benjamin Moon, Harley Eades III, and Dominic Orchard. 2021. Graded Modal Dependent Type Theory. In *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12648)*, Nobuko Yoshida (Ed.). Springer, 462–490. https://doi.org/10.1007/978-3-030-72019-3_17

Yue Niu, Jonathan Sterling, Harrison Grodin, and Robert Harper. 2022. A cost-aware logical framework. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–31. https://doi.org/10.1145/3498670

Ulf Norell. 2008. Dependently typed programming in Agda. In *International school on advanced functional programming*. Springer, 230–266.

Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative program reasoning with graded modal types. *Proc. ACM Program. Lang.* 3, ICFP (2019), 110:1–110:30. https://doi.org/10.1145/3341714

Vineet Rajani, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2021. A unifying type-theory for higher-order (amortized) cost analysis. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434308

Matthijs Vákár. 2014. Syntax and Semantics of Linear Dependent Types. *CoRR* abs/1405.0033 (2014). http://arxiv.org/abs/1405.0033

Received 2023-07-11; accepted 2023-11-07