Expressive Power and Abstraction in Essence

David G. Mitchell and Eugenia Ternovska
Computational Logic Laboratory
Simon Fraser University
{mitchell,ter}@cs.sfu.ca

Abstract

Development of languages for specifying or modelling problems is an important direction in constraint modelling. To provide greater abstraction and modelling convenience, these languages are becoming more syntactically rich, leading to a variety of questions about their expressive power. In this paper, we consider the expressiveness of Essence, a specification language with a rich variety of syntactic features. We identify natural fragments of Essence that capture the complexity classes P, NP, all levels Σ_i^p of the polynomial-time hierarchy, and all levels k-NEXP of the nondeterministic exponential-time hierarchy. The union of these classes is the very large complexity class ELEMENTARY. One goal is to begin to understand which features play a role in the high expressive power of the language and which are purely features of convenience. We also discuss the formalization of arithmetic in Essence and related languages, a notion of capturing NP-search which is slightly different than that of capturing NP decision problems, and a conjectured limit to the expressive power of Essence. Our study is an application of descriptive complexity theory, and illustrates the value of taking a logic-based view of modelling and specification languages.

1 Introduction

An important direction of work in constraint-based methods is the development of declarative languages for specifying or modelling combinatorial search problems. These languages provide users with a notation in which to give a high-level specification of a problem. Examples include EaCl [29], ESRA [8], Essence [11], NP-Spec [1], OPL [21] and Zinc [28], and the input languages for the solvers ASPPS [4], DLV [24], MidL [27] and MXG [30], and for the Answer Set Programming (ASP) grounders lparse [33] and GrinGo [16].

Languages of this sort, if well-designed and supported by high quality tools for practitioners, have the potential to greatly expand the range of successful applications of constraint solving technology. By reducing the need for specialized constraint programming knowledge they make the technology accessible to a wider variety of users. They also offer many other potential benefits. One

example is automated or semi-automated specification-level reasoning, such as recognition and handling of symmetries or safe-delay constraints (see, for example [13, 26, 2]), and other kinds of re-formulation. Another is facilitation of applying multiple solver technologies to a class of instances, through the use of translation or "cross-compilation" tools that allow one specification language to be used as input to different classes of solvers.

A fundamental question one may ask about a formal language is: What things can it say, or not say? For languages which describe computational problems, a natural form of this question is: *What is the computational complexity of the problems the language can describe?* This question is about the *expressive power* (or *expressiveness*) of the language. In this paper, we study the expressiveness of Essence [11].

There are many reasons, beyond curiosity, for wanting to answer questions about the expressiveness of specification and modelling languages. These include:

- At a minimum, a user needs to know that the language they use is *sufficiently expressive* to model the problem at hand. For example, a user expecting to solve their favourite NP-complete problem would be very upset to discover they were using a modelling language that could not describe it. Designers of languages or systems should be able to assure a user they *can* specify their problem — without having to actually produce a specification. A language or system designer may have some target class of applications in mind, and would like to be certain their design can express every application in the target class.

- We may wish to avoid using a language with more expressive power than required. Since a more expressive language can describe problems of greater complexity than we need, a solver for this language must implement algorithms which can solve these more complex problems. It is unlikely that a single solver or solver technology is the most effective both for these highly complex problems and the target problem class. For a system designer, choosing a language with limited expressive power may provide some assurance of "practical implementability", or that some particular solver technology is a good choice.

- We may wish to understand the contribution particular language features make to expressiveness, asking questions like: Would adding (removing) feature X to (from) language Y change what problems can be specified by the language, or merely change how easily or naturally some problems could be specified? Being able to answer such questions may help us choose the best language for an application. It can also help with selection and implementation of solver technology. Typically, a feature in-essential to the expressive power of a language is definable in terms of other features. As a simple example, we may treat the formula $(\alpha \supset \beta)$ as an abbreviation of $(\neg\alpha \vee \beta)$, thereby reducing the number of connectives to be handled. By treating many features of a rich language this way, we may obtain a much

simpler "kernel language" which has the same expressive power, but for which we may more easily produce an effective and maintainable solver.

Remark 1 *In this paper, we distinguish* expressive power*, or* expressiveness*, in the formal sense of what problems can or cannot be specified with a language, from* syntactic richness*, by which we mean the variety of features provided for a user's convenience. We are focussed on the former; our interest in the latter relates to the relationship between the two.*

Our Methodology

One area that establishes a connection between languages and complexity classes is descriptive complexity theory [22], which seeks to characterize complexity classes in terms of logics. The field began with Fagin's result [7] showing that the classes of finite structures axiomatizable in existential second order logic ($\exists$SO) are exactly those in the complexity class NP. This may be viewed as a statement about the *expressive power* of $\exists$SO. Work by others has provided analogous results for other logics and complexity classes, establishing logics corresponding to the complexity classes NL, P, all levels of polynomial hierarchy, and others (see [22, 18], for example). These results show that logics can be regarded, at least theoretically, as "declarative programming languages" or "constraint modelling languages" for corresponding complexity classes.

We are interested in the expressive power of languages such as those listed in the first paragraph of this paper. These languages vary widely in design philosophy, in syntax, and in the manner in which their semantics is specified. For example, ESSENCE semantics is a partial truth function recursively defined on the structure of expressions, while the lparse semantics is described by a translation to propositional logic programs. Fortunately, it is not hard to see that most of these languages, while formalized differently by their authors, have a common underlying logical task which makes application of standard results and techniques in descriptive complexity theory natural in many cases. The task in question is that of "model expansion" (MX), which requires finding an expansion of a given (finite) structure to a larger vocabulary, so as to satisfy a given formula. (See Section 2 for a precise definition.) Every computational search problem can be formalized as model expansion for a sufficiently expressive logic.

To apply results or methods of descriptive complexity to an existing specification language, we view it as a logic, and take specifications in the language to be axiomatizations of problems formalized as model expansion. Formally, to view a language as a logic, we construct an alternate model-theoretic semantics for the language. This "model expansion semantics" must be equivalent to the original semantics for the language, in the sense that each specification must define the same problem under either semantics.

As we will see, some natural fragments of ESSENCE are essentially minor syntactic variants of classical logic, and are easy to deal with. Further features can be handled by the notion of logical definability. Some features are more

nicely handled by other methods, such as extending the formalization to allow an infinite "background" structure, as we do in Section 5 to deal with arithmetic.

Problems and Instances

Before proceeding, we point out a potential source of confusion regarding terminology. In standard computer science terminology, a "problem" is an infinite set of instances, together with a question (or task) associated with those instances. This is the terminology we follow in this paper. For example, we may consider the set of all graphs and the question "does graph $\mathcal{G}$ have a proper 3-colouring". By a "specification", we mean a formal representation of such a problem. We take the input to a solver to be a specification together with an instance, but we think of the specification as being fixed (for a problem) while the instance varies.

In contrast, some constraint programming literature uses the term "model" where we use specification[1] and "data" for a (description of a) problem instance. When the input for a solver consists of the model and data together, this is sometimes referred to as "an instance" or even "a problem".

Remark 2 *Sometimes people use a modelling language to describe a problem which consists of a single instance, for example to answer a question like "is there a Sudoku board with 16 givens and a unique solution"? Our study does not address this use.*

Why Essence?

Our main goal in this paper is to study the expressiveness of the Essence specification language, and to learn something about the role its various features play in this expressiveness. Essence is interesting to us for several reasons. One is that it is fairly easy to get started: Essence specifications are easily seen to be specifications of model expansion tasks, and Essence syntax contains a near-copy of the syntax of classical first order logic. Another is that Essence gives the user a wide variety of features with which to specify a problem. The designers of Essence state that one of the goals of designing the language was to enable problem specifications that are similar to rigorous natural-language specifications, such as those catalogued by Garey and Johnson [15]. A related goal was to allow problems to be specified at a higher level of abstraction than is normally possible when producing models in existing "constraint modelling", "constraint logic programming", or "mathematical programming" languages, or when designing reductions to SAT or other CSP languages. Essence has many features specifically intended to support writing abstract specifications. The consequent syntactic richness of Essence presents us with many questions about its expressive power, and that of fragments which exclude various features.

[1] We do not distinguish specification from modelling here. The distinction may be important, but is not simple or of particular significance to the work presented in this paper.

1.1 In This Paper

We use the methodology described above to study the expressive power of several interesting and fairly natural fragments of ESSENCE. In particular, we apply classical results from descriptive complexity theory to show that certain fragments of ESSENCE capture the complexity classes P, NP, Σ_i^p for $i \in \mathbb{N}$ — that is, all the "existential" levels of the polynomial-time hierarchy (PH), and thus all of PH — and all the levels of the nondeterministic exponential-time hierarchy (NEXP, 2-NEXP, ...). The union of these classes is the (very large) complexity class ELEMENTARY. Results about capturing show that a language can express all and only the problems in a certain complexity class, and are important in that they exactly characterize the problems expressible in a language. This makes it possible to determine whether a problem is expressible in a language based only on a rough estimate of the complexity of the problem. It also provides valuable information regarding appropriate techniques for implementing solvers and other tools for using a language. In light of these results, we consider a number of questions about various features of ESSENCE, and the expressive power of fragments of ESSENCE that contain them.

Fragments and Features

For simplicity of exposition, the fragments we identify to capture various complexity classes are kept small and simple. However, the fragments are also "robust", in the sense that there are many ESSENCE features that can be added to each of them without affecting the expressive power. It would be a large undertaking to give a full account of all the features, let alone an extensive analysis of the extent to which each feature contributes to the expressive power. We do point out, for several fragments we define, a number of ESSENCE features that could be added to them without increasing expressiveness.

The fragment we study that is of greatest practical interest is one that we denote E_{FO}, because it corresponds closely to classical first order logic (FO). The decision problems E_{FO} can express are exactly the problems in NP. We specifically consider several issues regarding the expressiveness of fragments related to E_{FO} and NP.

Arithmetic

One of the most useful extensions one might add to a small fragment like E_{FO} is arithmetic. We consider an extension of E_{FO} with arithmetic, denoted $\mathrm{E}_{FO}[\mathcal{N}]$, where $\mathcal{N}$ stands for the natural numbers $\mathbb{N}$ with a selection of "built-in" arithmetic functions and relations, and show how to modify our logic-based formalism for this extension. Doing this brings to light a subtle, and in some ways surprising, issue. While the resulting language still captures NP, it cannot express all NP problems which involve numbers in numerical terms, as one would like. That is, there are problems in NP involving numbers which the language cannot express by direct use of numbers. To express these problems, the use of encodings of numbers (say in binary), is necessary. We show that $\mathrm{E}_{FO}[\mathcal{N}]$ can

express exactly the problems in NP that do not contain extremely large (non-encoded) numbers. The issue involved here is not something specific to E_{FO}, or to logic-based languages, but is relevant to all languages for NP that contain arithmetic.

Capturing NP-Search: Finding (only) what you want

A user with a search problem generally has a particular sort of object in mind as their desired solution. For the Knapsack problem it is a subset of the given objects; for graph colouring it is a function from vertices to colours, etc. A problem specification has vocabulary symbols for objects comprising an instance, and symbols for those comprising a solution. Sometimes, however, when writing a specification, it is useful to introduce additional "auxiliary" symbols, which denote concepts that help in describing properties of a solution. The extension of an auxiliary symbol is *not* part of the desired solution, and the solver need not report it to the user — but needs to construct it nonetheless. This raises the question: *can we always construct an* Essence *specification for a search problem in which the only objects that must be found are those the users asks for?* This amounts to a definition of *capturing search*, that is a bit different than the usual notion of capturing a class of decision problems. We will show that, in the case of E_{FO}, the answer is no: it does not capture NP-search in the desired way. We also describe an extension of E_{FO} for which the answer is yes.

Particular Abstraction Features of Essence

We also consider some features which the designers of Essence have highlighted as being particularly relevant to Essence supporting the writing of specifications at a higher level of abstraction than many other current languages [14, 11]. These are: a wide range of basic types; arbitrarily nested types; quantification over decision variables; and "unnamed" types. We defer discussing particulars of these to Section 10, but we will see that none of these features *per se* increases the expressive power of E_{FO} (or any more expressive fragment). Many of the type constructors in Essence would increase the expressiveness of E_{FO}, but this is only because they allow construction of very large domains. We may extend E_{FO} with a rich variety of type constructors without increasing expressiveness, provided we restrict the *size* of the domains so constructed.

How Expressive is Unrestricted Essence?

Because of the rich variety of features, it is not obvious what the expressive power of Essence is[2]. Even after the work reported here, we can at best make a conjecture. Our conjecture is that Essence can express exactly those combinatorial problems in the complexity class ELEMENTARY. We discuss this, and present an example of a purely combinatorial problem that, if the conjecture holds, is not expressible in Essence.

[2]This is also observed by the Essence designers [10].

1.2 Structure of the Paper

Section 2 presents some essential background in logic and computational complexity, the formal definition of model expansion, and some related explanatory material. In Section 3, we formalize a small but interesting fragment of ESSENCE, denoted E_{FO}, which corresponds to FO model expansion, and in Section 4 we show that E_{FO} captures NP. Section 5 shows how to extend E_{FO} and our logic-based formalization with a limited form of arithmetic, while still capturing NP. Section 6 examines the question of "capturing NP search". In Section 7 we identify natural fragments of ESSENCE that capture the polynomial-time hierarchy (PH), and specifically each complexity class Σ_i^p of PH. In Section 8 we show how extending E_{FO} instance descriptions with succinct representations, or E_{FO} specifications with large defined domains, results in fragments of ESSENCE which can express problems of extremely high complexity — all problems in the exponential-time hierarchy, and thus the class ELEMENTARY. In Section 9 we conjecture an upper limit on the expressive power of unrestricted ESSENCE, and provide an example of a (reasonably natural) purely combinatorial problem that we conjecture cannot be specified with ESSENCE. Finally, we end with a discussion and concluding remarks in Section 10.

2 Background

In this section we review the concepts and terminology from logic and complexity theory that are needed for this paper. We assume only that the reader is familiar with the basics of first order logic. For further background on logic, and in particular finite model theory and descriptive complexity theory, we refer the reader to [6, 25, 22].

2.1 Logical Structures

A vocabulary is a set σ of relation (a.k.a. predicate) and function symbols. Each symbol has an associated arity, a natural number. A structure $\mathcal{A}$ for vocabulary σ is a tuple containing a universe A (also called the domain of $\mathcal{A}$), and a relation (function), defined over A, for each relation (function) symbol of σ. If R is a relation symbol of vocabulary σ, the relation corresponding to R in a σ-structure $\mathcal{A}$ is denoted $R^{\mathcal{A}}$ and may be called the *extent* of R in $\mathcal{A}$ or the *interpretation* of R in $\mathcal{A}$. For example, we write

$$\mathcal{A} = (A;\ R_1^{\mathcal{A}}, \ldots R_n^{\mathcal{A}},\ f_1^{\mathcal{A}}, \ldots f_m^{\mathcal{A}},\ c_1^{\mathcal{A}}, \ldots c_k^{\mathcal{A}}),$$

where the R_i are relation symbols, the f_i are function symbols, and constant symbols c_i are 0-ary function symbols. We always assume the presence of the equality symbol $=$, which is always interpreted as the identity relation on domain elements. A structure is finite if its universe is finite.

An example of a finite structure for vocabulary $\{E\}$, where E is a binary relation symbol, is a graph $\mathcal{G} = (V; E^{\mathcal{G}})$. Here, V is a set of vertices, $E^{\mathcal{G}}$ is a set

of pairs of vertices which represent edges. An example of an infinite structure for the vocabulary $\{s, f, c\}$, is the structure $\mathcal{B} = (\mathbb{N}; s^{\mathcal{B}}, f^{\mathcal{B}}, c^{\mathcal{B}})$ as follows. In this structure, $\mathbb{N}$ is the set of natural numbers; $s^{\mathcal{B}}$ is the (unary) successor function; $f^{\mathcal{B}}$ is a (binary) function on domain elements which maps a pair of numbers to their sum; and $c^{\mathcal{B}}$ is the smallest natural number 0. The standard choice of vocabulary for this structure would, of course, be $\{succ, +, 0\}$. We give this example to illustrate that particular symbols do not mean anything until such a meaning is provided by a structure.

2.2 Definability in a Structure

Given a structure $\mathcal{A}$, a FO formula $\phi(x_1, \ldots, x_n)$ with free variables $x_1, \ldots x_n$ defines an n-ary relation

$$\{(a_1, \ldots a_n) \mid \mathcal{A} \models \phi[a_1, \ldots a_n]\}.$$

For example, the relation $\leq$ is defined by the formula $\exists z\ x + z = y$. Thus, $\leq$ can be seen as an *abbreviation* of this longer formula, and we say that this relation is *definable* with respect to the structure $\mathcal{C} = (\mathbb{N}; succ, +, 0)$.

2.3 The Task of Model Expansion

An expansion of a σ-structure $\mathcal{A} = (A; \sigma^{\mathcal{A}})$ to vocabulary $\sigma \cup \varepsilon$ is a structure $\mathcal{B} = (A; \sigma^{\mathcal{A}}, \varepsilon^{\mathcal{B}})$, which has the same domain as $\mathcal{A}$, all the functions and relations of $\mathcal{A}$, and in addition has functions and relations interpreting ε. Model expansion (MX) as a task is defined for an arbitrary logic $\mathcal{L}$ as follows.

Definition 1 ***Model Expansion for*** $\mathcal{L}$ *(abbreviated* $\mathcal{L}$ *MX).*

Given: *1. An $\mathcal{L}$-formula ϕ, and*
2. A structure $\mathcal{A}$ for a part σ of vocab(ϕ).
Find: *An expansion $\mathcal{B}$ of $\mathcal{A}$ to vocab(ϕ) such that $\mathcal{B}$ satisfies ϕ.*

In the definition, *vocab*(ϕ) denotes the set of vocabulary symbols used in ϕ. Each expansion of $\mathcal{A}$ that satisfies ϕ gives interpretations to the vocabulary symbols of ϕ which are not interpreted by $\mathcal{A}$.

As used in this paper, the formula ϕ plays the role of a problem specification, the given structure $\mathcal{A}$ is an instance of the problem, and each expansion satisfying the definition constitutes a solution for the instance $\mathcal{A}$. We call σ, the vocabulary of $\mathcal{A}$, the *instance* or *input* vocabulary, and $\varepsilon := vocab(\phi) \setminus \sigma$ the *expansion* vocabulary.

Example 1 [Graph 3-colouring as model expansion] Let the instance vocabulary be $\{E\}$, where E is a binary relation symbol. Then the input structure is a graph $\mathcal{A} = \mathcal{G} = (V; E^{\mathcal{A}})$. Let the vocabulary of ϕ be $\{E, R, B, G\}$, where R, B, G are unary relation symbols. Any expansion of $\mathcal{A}$ to this vocabulary interprets the expansion symbols $\{R, B, G\}$ as sets of vertices, which we may

think of as those vertices being coloured Red, Blue and Green. To ensure that the colouring is total and proper, let ϕ be:

$$\begin{gathered}\forall x\ [(R(x) \vee B(x) \vee G(x)) \\ \wedge\neg(R(x) \wedge B(x)) \wedge \neg(R(x) \wedge G(x)) \wedge \neg(B(x) \wedge G(x))] \\ \wedge\ \forall x\ \forall y\ [E(x,y) \supset (\neg(R(x) \wedge R(y)) \\ \wedge\neg(B(x) \wedge B(y)) \wedge \neg(G(x) \wedge G(y)))].\end{gathered}$$

A solution is an interpretation for the expansion vocabulary $\varepsilon := \{R, B, G\}$ such that

$$\underbrace{(\overbrace{V; E^{\mathcal{A}}}^{\mathcal{A}},\ R^{\mathcal{B}}, B^{\mathcal{B}}, G^{\mathcal{B}})}_{\mathcal{B}} \models \phi.$$

The expansions of $\mathcal{A}$ that satisfy ϕ, if there are any, correspond exactly to the proper 3-colourings of $\mathcal{G}$.

The problem can also be axiomatized using a function *colour*() mapping vertices to colours. This is a more concise model expansion representation of the problem, and we invite the reader to complete the details. $\Diamond$

In Example 1, formula ϕ is essentially a specification of the 3-colouring problem, as it describes exactly the relationship between a graph and its proper 3-colourings. In addition to ϕ, we need the partition of the vocabulary into instance and expansion vocabularies.

2.4 Multi-Sorted Logic

In a multi-sorted logic there are different sorts of variables, with each sort having its own domain. It is natural to use multi-sorted logics in our study of ESSENCE. Each ESSENCE variable ranges over some finite set, and for the simple fragments of ESSENCE we study in this paper, we may let these sets correspond to sorts in logic.

Formally, we have a nonempty set I, whose members are called sorts. For each sort $i \in I$, we have variable symbols, quantifiers $\forall_i$ and $\exists_i$, and an equality symbol $=_i$. With each predicate and function symbol of a vocabulary, we specify the sorts of each argument. That is, each k-ary predicate symbol is of some sort $\langle i_1, i_2, \ldots, i_k\rangle$, where each $i_j \in I$, and each k-ary function symbol is of some sort $\langle i_1, i_2, \ldots, i_k, i_{k+1}\rangle$. Each term has a unique sort, and the sorts of predicate and function symbols agree with those of the terms they are applied to. A multi-sorted structure is a structure that has a domain for each sort, and assigns to each relation and function symbol a relation or function of the correct sort. For example, consider a vocabulary $\sigma = \{\vec{R}\}$ with sorts $I = \{1, 2, 3\}$. We may write a structure for σ as $\mathcal{A} = (A_1, A_2, A_3; \vec{R}^{\mathcal{A}})$.

Multi-sorted logics are sometimes convenient, but do not let us do anything essential that cannot be done without them. To see this, observe that we can carry out the following translation from multi-sorted to un-sorted first order logic. Suppose we have a sentence over a multi-sorted vocabulary σ. Consider

an un-sorted vocabulary σ' which has all the symbols of σ, and in addition has a predicate symbol D_i for each sort i of σ. We translate a multi-sorted formula ϕ of vocabulary σ to an un-sorted formula ϕ' of vocabulary σ' by re-writing each sub-formula of the form

$$\forall_i x\ \phi(x) \quad \text{or} \quad \exists_i x\ \phi(x)$$

where x is of sort i, as (respectively)

$$\forall x\ (D_i(x) \supset \phi(x)) \quad \text{or} \quad \exists x\ (D_i(x) \wedge \phi(x)).$$

We translate a multi-sorted σ-structure $\mathcal{A}$ to a σ'-structure $\mathcal{A}'$ by letting the domain of $\mathcal{A}'$ be the union of the domains $\mathcal{A}$, and the interpretation of each predicate symbol $D_i^{\mathcal{A}'}$ be the domain in $\mathcal{A}$ of sort i. Is it easy to show that

$$\mathcal{A} \models \phi \ \Leftrightarrow\ \mathcal{A}' \models \phi'.$$

To continue our example where $\sigma = \{\vec{R}\}$, then $\sigma' = \{D_1, D_2, D_3, \vec{R}\}$. We write a structure for σ' as $\mathcal{A}' = (A_1 \cup A_2 \cup A_3; D_1^{\mathcal{A}'}, D_2^{\mathcal{A}'}, D_3^{\mathcal{A}'}, \vec{R}^{\mathcal{A}'})$, where each $D_i^{\mathcal{A}'} = A_i$. Notice that $\mathcal{A}$ and $\mathcal{A}'$ are really just two ways of looking at the same object, except that in $\mathcal{A}'$ the three sorts have been made explicit, and have associated vocabulary symbols. In fact, it is common to make the natural and harmless assumption in a multi-sorted logic that we always have vocabulary symbols associated with each sort, so even this distinction disappears. We will make this assumption throughout the paper.

The only real difference, then, between the multi-sorted structures for σ and the un-sorted structures for σ' is that not all un-sorted structures for σ' correspond to multi-sorted structures for σ. For example, an un-sorted structure $\mathcal{A}$ for σ' may have that $R^{\mathcal{A}}$ contains both elements from $D_1^{\mathcal{A}}$ and $D_2^{\mathcal{A}}$. In this case, if $D_1^{\mathcal{A}}$ and $D_1^{\mathcal{A}}$ are disjoint, $\mathcal{A}$ does not correspond to any structure for σ. However, if we restrict our attention in the un-sorted case to suitably chosen structures, we can define a translation # from the un-sorted to the multi-sorted case, with the property that $\mathcal{A}'^{\#} = \mathcal{A}$, and such that

$$\mathcal{A} \models \phi \quad \Leftrightarrow \quad \mathcal{A}^{\#} \models \phi^{\#}.$$

In this paper, it is convenient to use multi-sorted logic because sorts correspond roughly to types in Essence. On the other hand, for most purposes it is simpler to use un-sorted logic. Thus, we will use the sorted version when this improves clarity, but un-sorted logic otherwise. All results hold in either case, but may not be equally simple to prove or explain.

2.5 Second Order Logic

Second order logic (SO), in addition to first order quantification (over individual objects), allows quantification over sets (of tuples) and functions. This is expressed through quantification over relation and function symbols. An example of a SO formula is

$$\exists E\ \forall f\ \forall R\ \forall x\ \forall y\ (R(x,y) \wedge f(x) = y\ \Rightarrow\ E(x)).$$

Here, E and R are second order variables ranging over sets (of tuples), f is a second order function variable, and x and y are first order (object) variables (those ranging over domain elements). The existential fragment of SO, denoted $\exists$SO, is the fragment where all second order quantifiers are existential.

2.6 Review of Complexity Classes

We need some background in computational complexity theory, which we give based on the exposition in [25]. Let L be a language accepted by a halting Turing machine M. Assume that for some function $f : \mathbb{N} \to \mathbb{N}$, the number of steps M makes before accepting or rejecting a string s is at most $f(|s|)$, where $|s|$ is the length of s. If M is deterministic, then we write $L \in \mathrm{DTIME}(f)$; if M is nondeterministic, then we write $L \in \mathrm{NTIME}(f)$. In this paper we are primarily interested in the complexity classes P, NP, Σ_i^p, NEXP and ELEMENTARY, which are as follows.

The class P of polynomial-time computable problems is defined as

$$\mathrm{P} = \bigcup_{k \in \mathbb{N}} \mathrm{DTIME}(n^k),$$

and the class NP of problems computable by a nondeterministic polynomial-time Turing machine as

$$\mathrm{NP} = \bigcup_{k \in \mathbb{N}} \mathrm{NTIME}(n^k).$$

The polynomial hierarchy (PH) is defined as follows. Let $\Sigma_0^p = \Pi_0^p = \mathrm{P}$. Define inductively $\Sigma_i^p = \mathrm{NP}^{\Sigma_{i-1}^p}$, for $i \geq 1$. That is, languages in Σ_i^p are those accepted by a nondeterministic Turing machine that runs in polynomial time, but that can also make "calls" to another machine that computes a language in Σ_{i-1}^p. Such a call is assumed to have unit cost. We define the class Π_i^p as the class of languages whose complements are in Σ_i^p. Notice that $\Sigma_1^p = \mathrm{NP}$ and $\Pi_1^p = $ co-NP. The union of these is

$$\mathrm{PH} = \bigcup_{i \in \mathbb{N}} \Sigma_i^p = \bigcup_{i \in \mathbb{N}} \Pi_i^p.$$

PSPACE is the class of languages accepted by a Turing machine that uses at most polynomial space. NL is the class of languages recognized by some nondeterministic logspace TM. To define NL, we need to talk about space complexity for sub-linear functions f. To do that, we use a model of Turing machines with separate input and work tapes. The input tape cannot be modified. We define NL to be the class of languages accepted by such nondeterministic machines where at most $O(\log|s|)$ cells of the work tape are used.

The relationship between these complexity classes is as follows:

$$\mathrm{NL} \subseteq \mathrm{P} \subseteq \left\{ \begin{array}{c} \mathrm{NP} \\ \text{co-NP} \end{array} \right\} \subseteq \mathrm{PH} \subseteq \mathrm{PSPACE}.$$

None of the containments of any two consecutive classes in this sequence is known to be proper, but the space classes NL and PSPACE, with which we have bracketed the time classes defined so far, are distinct: NL $\subsetneqq$ PSPACE.

We also consider several classes that involve exponential running time. We have

$$\text{EXP} = \bigcup_{c\in\mathbb{N}} \text{DTIME}(2^{n^c}) \quad \text{and} \quad \text{NEXP} = \bigcup_{c\in\mathbb{N}} \text{NTIME}(2^{n^c}).$$

EXP and NEXP both contain PSPACE, and properly contain NP.

Let $2^{[k]}(n)$ denote

$$2^{2^{\cdot^{\cdot^{2^n}}}}$$

where the stack of 2's is k high. Then we define, for each $k \in \mathbb{N}$,

$$k\text{-EXP} = \bigcup_{c\in\mathbb{N}} \text{DTIME}(2^{[k]}(n^c)) \quad \text{and} \quad k\text{-NEXP} = \bigcup_{c\in\mathbb{N}} \text{NTIME}(2^{[k]}(n^c))$$

For example, $\text{NTIME}(2^{n^c}) = \text{NEXP} = 1\text{-NEXP}$; $\text{NTIME}(2^{[2]}(n^c)) = 2\text{-NEXP}$ (doubly-exponential non-deterministic time), etc. These classes define the exponential hierarchy, where we know that all the inclusions

$$\cdots \quad k\text{-NEXP} \subsetneqq (k+1)\text{-EXP} \subsetneqq (k+1)\text{-NEXP} \quad \cdots$$

are proper. The union of all these gives us the class ELEMENTARY:

$$\bigcup_{k\in\mathbb{N}} k\text{-EXP} = \bigcup_{k\in\mathbb{N}} k\text{-NEXP} = \text{ ELEMENTARY}.$$

Beyond elementary we have

$$\text{TIME}(2^{[n]}(n)) = \text{ITERATED EXPONENTIAL TIME}.$$

Notice that we do not distinguish deterministic and nondeterministic time $2^{[n]}(n)$. At this complexity, nondeterminism (and exponents on the argument n) don't make any difference. Relating these to some more familiar classes, we have the following, where all the inclusions are proper.

$$\begin{aligned}
\bigcup_{k\in\mathbb{N}} \text{NTIME}(2^{[k]}(n^c)) &= \bigcup_{k\in\mathbb{N}} k\text{-NEXP} \\
&= \text{ELEMENTARY} \\
&\subsetneqq \text{ITERATED EXPONENTIAL TIME} \\
&\subsetneqq \text{PRIMITIVE RECURSIVE} \\
&\subsetneqq \text{RECURSIVE}.
\end{aligned}$$

2.7 Descriptive Complexity Theory

Descriptive Complexity Theory [22] deals with machine-independent characterization of complexity classes. In this section, we provide a brief review of the definitions and results we need.

Model Expansion and Model Checking

We will talk about two computational tasks, model checking and model expansion (as a decision problem), associated with each logic $\mathcal{L}$. We reproduce the definition of model expansion for completeness of the exposition.

1. *Model Checking* (MC): given $(\mathcal{A}, \phi)$, where ϕ is a sentence in $\mathcal{L}$ and $\mathcal{A}$ is a finite structure for $vocab(\phi)$, does $\mathcal{A}$ satisfy ϕ?
2. *Model Expansion* (MX): given $(\mathcal{A}, \phi)$, where ϕ is a sentence in $\mathcal{L}$ and $\mathcal{A}$ is a finite σ-structure with $\sigma \subseteq vocab(\phi)$, is there an expansion of $\mathcal{A}$ to $vocab(\phi)$ which satisfies ϕ?

In model checking, the entire structure is given; in model expansion *part* of the structure, which includes the domain, is given. MX for first order logic is equivalent to MC for $\exists$SO. That is, there exists a suitable expansion of a structure $\mathcal{A}$ if there exists suitable interpretations for the expansion predicates, or, equivalently, if $\mathcal{A}$ satisfies the $\exists$SO formula obtained from the FO formula by preceding it with existential quantifiers for all expansion predicates.

We now generalize this connection between MC and MX. Let $\mathcal{L}$ be any logic. Let $\exists\mathrm{SO}(\mathcal{L})$ be the set of formulas of the form $\exists P_1^{r_1} \ldots \exists P_k^{r_k} \phi$ for some k, where $\phi \in \mathcal{L}$ and each $P_i^{r_i}$ is a second order relation or function variable of arity r_i. From the definition of $\exists\mathrm{SO}(\mathcal{L})$, we have that MX for $\mathcal{L}$ is equivalent to MC for $\exists\mathrm{SO}(\mathcal{L})$. The logic $\exists\mathrm{SO}(\mathcal{L})$ may or may not be a fragment of $\exists$SO, depending on $\mathcal{L}$. When $\mathcal{L}$ is FO, $\exists\mathrm{SO}(\mathcal{L})$ is precisely $\exists$SO.

Capturing Complexity Classes

Complexity theory characterizes complexity classes in terms of resources needed by Turing machines (TMs). In descriptive complexity, we characterize them in terms of expressive power of logics. To associate structures with TM computations, we need to represent or encode structures as strings. Let $enc()$ denote such an encoding. Different encodings are possible, and as with most complexity theory any "reasonable" encoding will do. Here is an example. We assume that the elements of the universe are associated with an initial segment of the natural numbers. This allows us to have encodings of the form

$$\underbrace{000\ldots0}_{n} 1 enc(R_1) enc(R_2) \ldots enc(R_k).$$

The first n zeros represent the universe, 1 acts as a separator, and each $enc(R_i)$ represents an encoding of a relation (or function). Such an encoding is a 0-1 vector, where 1 in the i position indicates that the i-th tuple (according to lexicographic ordering) is in the relation, and 0 indicates that it is not.

The following notion was introduced in [34].

Definition 2 *Let Comp be a complexity class and $\mathcal{L}$ a logic. The* data complexity *of $\mathcal{L}$ is Comp if for every sentence ϕ of $\mathcal{L}$ the language $\{enc(\mathcal{A}) \mid \mathcal{A} \models \phi\}$ belongs to Comp.*

Definition 3 *A property P of structures is* definable *in logic $\mathcal{L}$ if there is a sentence $\phi_P \in \mathcal{L}$ such that for every structure $\mathcal{A}$, we have that $\mathcal{A} \models \phi_P$ iff $\mathcal{A}$ has property P.*

Definition 4 *Let $\mathcal{D}$ be a class of finite structures, Comp be a complexity class, and $\mathcal{L}$ be a logic. Then $\mathcal{L}$* captures *Comp on $\mathcal{D}$ ($\mathcal{L} \equiv_{\mathcal{D}} Comp$) if*

(a) for every fixed sentence $\phi \in \mathcal{L}$, the data complexity of evaluating ϕ on structures from $\mathcal{D}$ is a problem in the complexity class Comp, and

(b) every property of structures in $\mathcal{D}$ that can be decided with complexity Comp is definable in the logic $\mathcal{L}$.

The following theorem initiated the area of descriptive complexity.

Theorem 1 (Fagin [7]) *$\exists SO$ captures NP.*

Fagin's result generalizes to higher levels of polynomial hierarchy, with levels corresponding to number of alternations of second order quantifiers, and full second order logic capturing PH (see Section 7). Following Fagin's theorem, Immerman [23] and Vardi [34] independently proved that, on ordered structures, the complexity class P is captured by a fixpoint extension of FO logic. The restriction here to ordered structures means we consider only structures which have an explicit ordering relation on the universe.

2.8 Complexity and Descriptions *vs* Reductions

As is clear from Fagin's theorem and the definition of capturing a complexity class, the property of capturing NP is very different from NP-completeness. Proving NP-completeness of a problem shows us that exactly the problems in NP can be *polynomially reduced* to it. Proving a language captures NP shows that the language is universal for *defining* the problems in NP.

For example, NP-completeness of SAT tells us that for every problem in NP there is a polynomial reduction to SAT. When we reduce Graph 3-Colourability to SAT, we obtain a different propositional formula (SAT instance) for each graph. Fagin's theorem tell us that for every problem in NP there is an $\exists$SO formula that axiomatizes the problem. When we axiomatize Graph 3-Colourability as $\exists$SO (or as FO MX), we have *one* formula which describes the class of all 3-colourable graphs.

3 E_{FO}: A Small Fragment of Essence

In this section, we begin our technical development by introducing a small fragment of Essence, which we call E_{FO} because it corresponds closely to first order logic. This fragment will be useful to make our logic-based view of Essence precise, and is a natural fragment with which to begin a study of expressiveness. The Essence fragments and examples in this paper are based on Essence Version 1.2.0, as described in [11].

```
given       Vertex new type enum
given       Colour new type enum
given       Edge: rel of (Vertex × Vertex)
find        Coloured: rel of (Vertex × Colour)
such that   ∀ u : Vertex. ∃ c : Colour. Coloured(u, c)
such that   ∀ u : Vertex. ∀ c1 : Colour. ∀ c2 : Colour.
                ((Coloured(u, c1) ∧ Coloured(u, c2)) → c1 = c2)
such that   ∀ u : Vertex. ∀ v : Vertex. ∀ c : Colour.
                (Edge(u, v) → ¬(Coloured(u, c) ∧ Coloured(v, c)))
```

Figure 1: An ESSENCE specification of Graph Colouring.

A simple ESSENCE specification consist of three parts, identified by the mnemonic keywords `given`, `find`, and `such that`. The "given" part specifies the types of objects which comprise problem instances; the "find" part specifies the types of objects that comprise solutions; and the "such that" part specifies the relationship between instances and their solutions. To illustrate, we give a simple example.

Example 2 Figure 1 gives an ESSENCE specification of the Graph Colouring problem. The computational task described by this specification is as follows. We will be given a finite set *Vertex*, a finite set *Colour*, and a binary relation $Edge \subseteq Vertex \times Vertex$. Our task is to find a relation $Coloured \subseteq Vertex \times Colour$, which, in accordance with the constraints — the formulas of `such that` statements — maps each $v \in Vertex$ to a unique $c \in Colour$ so that no edge $(u, v) \in Edge$ is monochromatic. $\Diamond$

An ESSENCE instance may be described by a collection of `letting` statements, one corresponding to each `given` statement in the specification. These identify a concrete set with each symbol declared by a `given` statement.

Example 3 An example of an instance description for the specification of Figure 1 is:

```
letting   Vertex be new type enum {v1, v2, v3, v4}
letting   Colour be new type enum {Red, Blue, Green}
letting   Edge be rel {⟨v1, v2⟩, ⟨v1, v3⟩, ⟨v3, v4⟩, ⟨v4, v1⟩}
```

$\Diamond$

Definition 5 *Let* E_{FO} *denote the fragment of* ESSENCE *defined by the following rules.*

1. *The "given" part of an* E_{FO} *specification consists of a sequence of statements, each of one of the two forms:*

   (a) `given` D `new type enum`,

(b) `given` R`: rel of (`$D_1 \times \ldots \times D_n$`)`,

where $n \in \mathbb{N}$ *and each* D_i *is a type declared by a previous* `given` *statement of form 1a.*

2. *The "find" part of an* E_{FO} *specification consists of a sequence of statements, each of the form*

   `find` R`: rel of (`$D_1 \times \ldots \times D_n$`)`

   where $n \in \mathbb{N}$ *and each* D_i *is a type declared by a* `given` *statement of form 1a.*

3. *The "such that" part consists of a sequence of statements, each of the form*

   `such that` ϕ

   Here, ϕ *is an expression which is syntactically a formula of function-free first order logic, with one exception: All quantifiers of* E_{FO} *are of the form* $\exists\, x : D.$ *or* $\forall\, x : D.$*, where* x *is a variable and* D *is a type declared by a* `given` *statement of form 1a. The vocabulary of* ϕ *consists of relation symbols specified in the "given" and "find" parts, and may also contain the equality symbol* $=$*.*

   We will call formulas occurring in `such that` *statements* constraints*.*

4. *An* E_{FO} *instance description consists of* `letting` *statements which bind concrete sets or relations to the symbols of the "given" vocabulary. These are of the forms*

   (a) `letting` D `be new type enum` $\{new\_id_1, ..., new\_id_n\}$,

   (b) `letting` R `be rel` $\{\langle id_1^1, \ldots, id_n^1\rangle, \langle id_1^2, \ldots, id_n^2\rangle, \ldots\}$,

   respectively, for `given` *statements of forms (a) and (b).*

Since we make a distinction between instances and specifications, it might be more appropriate to remove the last item from this definition, and treat the specification language as independent of the instance representation. However, in order to analyze complexity or expressiveness, we need to make some assumptions about how instances are represented, and that is the role of item 4 in Definition 5. The primary point is that all elements in the instance are explicitly enumerated, so that instance descriptions correspond appropriately with standard encodings of finite structures.

This small fragment of Essence essentially corresponds to function-free multi-sorted first order logic, and indeed has the same expressive power, so we denote it by E_{FO}. Within this fragment, we can express the properties expressed by many other Essence language constructs, such as that a relation is a bijection or that all elements of a set are different. Thus, E_{FO} may be seen as a kind of "kernel" of Essence (perhaps of primarily theoretical interest).

Remark 3 *The terminology related to sorts can get confusing. In logic, we have sorts, which are associated with a vocabulary, and domains, which are concrete sets assigned to each sort by a structure.* Essence *has types and domains. Types are named sets of objects (not necessarily finite). Variables range over domains, and each domain is a subset of a type. In the context of this paper, the reader may think of types as corresponding to sorts.*

We use the term domain in the logical sense, and also informally to refer to a set of values over which a variable may range. This use does not correspond with the technical meaning of domain for Essence*, but this is probably harmless for the reader (until they want to write correct* Essence *specifications).*

3.1 FO Model Expansion and E_{FO}

We now provide the alternate, "model expansion" semantics for E_{FO}. To do this, we associate with each E_{FO} specification a class of structures, which correspond to the problem being specified. We then construct a mapping from E_{FO} specifications to FO formulas, with the property that the class of structures associated with a specification is the same as the class of structures defined by the formula it is mapped to.

Let Γ be an E_{FO} specification. We associate with Γ the following class $\mathcal{K}_\Gamma$ of multi-sorted structures. We have a sort for each type declared in Γ by a `given` statement of form 1a of Definition 5. The vocabulary of $\mathcal{K}_\Gamma$ is $\sigma \cup \varepsilon$, where σ has one relation symbol for each symbol declared by a `given` statement of Γ, and ε has one relation symbol for each symbol declared by a `find` statement of Γ. The sorts of these symbols correspond to the types declared in Γ in the obvious way. (Of course, σ and ε are our instance and expansion vocabularies.)

Instances for Γ are exactly the (multi-sorted) finite σ-structures. If we take a σ-structure, and expand it by adding interpretations for ε, we have a structure of vocabulary $\sigma \cup \varepsilon$, consisting of an instance together with a "candidate solution". $\mathcal{K}_\Gamma$ contains exactly those structures consisting of an instance for Γ expanded with a (real, not candidate) solution. The role of the constraints of Γ is to distinguish real solutions from candidate solutions.

Example 4 The specification of Graph Colouring in Figure 1 is an E_{FO} specification. The instance vocabulary is $\sigma = \{\mathit{Vertex}, \mathit{Colour}, \mathit{Edge}\}$, where *Vertex* and *Colour* are unary relation symbols denoting sorts, and *Edge* is a binary relation symbol. An instance structure $\mathcal{A}$ gives interpretations to the symbols of σ, that is, the sets $\mathit{Vertex}^{\mathcal{A}}$ and $\mathit{Colour}^{\mathcal{A}}$ and the binary relation $\mathit{Edge}^{\mathcal{A}} \subseteq \mathit{Vertex}^{\mathcal{A}} \times \mathit{Vertex}^{\mathcal{A}}$:

$$\mathcal{A} = (\mathit{Vertex}^{\mathcal{A}} \cup \mathit{Colour}^{\mathcal{A}}; \mathit{Vertex}^{\mathcal{A}}, \mathit{Colour}^{\mathcal{A}}, \mathit{Edge}^{\mathcal{A}}).$$

An example of a σ-structure is given by:

$$\begin{aligned}
&A = \{1,2,3,4\} \cup \{R,B,G\},\\
&\mathit{Vertex}^{\mathcal{A}} = \{1,2,3,4\},\\
&\mathit{Colour}^{\mathcal{A}} = \{R,B,G\},\\
&\mathit{Edge}^{\mathcal{A}} = \{(1,2),(1,3),(3,4),(4,1)\}.
\end{aligned}$$

This corresponds to the E_{FO} instance of Example 3. $\Diamond$

We will now show that, corresponding to Γ, there is a formula ϕ_Γ of (un-sorted) first order logic whose finite models are exactly the (un-sorted versions of the) structures in $\mathcal{K}_\Gamma$. The formula ϕ_Γ is over vocabulary $\sigma \cup \varepsilon$, and will be constructed from two sets of formulas, $\phi_{\Gamma,\mathrm{C}}$ and $\phi_{\Gamma,\mathrm{T}}$, reflecting the constraint and the type information of Γ.

The constraints are, except for a small syntactic change, formulas of multi-sorted first order logic. For example, a constraint of the form $\forall\, x : D.\ \phi(x)$ can be mapped to the formula $\forall_D x\, \phi(x)$ of our multi-sorted logic, which we can then re-write in un-sorted logic as described in Section 2.4. Thus, our formula will be obtained in part by applying the following rules to each constraint, translating all "Essence quantifiers" into standard un-sorted FO quantifiers:

- Rewrite $\forall\, x : D.\ \phi(x)$ as $\forall\, x\ (D(x) \Rightarrow \phi(x))$,
- Rewrite $\exists\, x : D.\ \phi(x)$ as $\exists\, x\ (D(x) \wedge \phi(x))$.

We call such quantification *guarded*, and we call D a *guard* for the corresponding quantified variable. For E_{FO} specification Γ, denote by $\phi_{\Gamma,\mathrm{C}}$ the set of first order formulas produced by applying this translation to all of the constraints of Γ.

The guards in $\phi_{\Gamma,\mathrm{C}}$ ensure that the formulas playing the role of constraints only "talk about" things of the right sorts. In our un-sorted logic, expansion predicates may contain elements of any sort, and it remains to restrict them to the intended domains. In Γ, this is the role of the type declarations in the `find` statements. In particular, an expression of the form

$$R\colon \texttt{rel of}\ (D_1 \times \ldots \times D_n)$$

in a `find` statement indicates that the interpretation of `R` is a subset of the cross-product of the indicated D_i. We can enforce this requirement with the following first order formula.

$$\forall\, x_1, \ldots\ \forall\, x_n\ (R(x_1, \ldots, x_n)\ \rightarrow\ D_1(x_1) \wedge \ldots \wedge D_n(x_n)) \tag{1}$$

Let us denote by $\phi_{\Gamma,\mathrm{T}}$ the set containing one formula of the form (1) for each `find` statement in Γ. We call the formulas of $\phi_{\Gamma,\mathrm{T}}$ *upper guard axioms*.

Now, let ϕ_Γ denote the conjunction of the formulas in $\phi_{\Gamma,\mathrm{C}} \cup \phi_{\Gamma,\mathrm{T}}$ and we have the following.

Proposition 1 *Let Γ be an E_{FO} specification and $\mathcal{K}_\Gamma$ the associated class of $\sigma \cup \varepsilon$-structures. Then $\mathcal{B} \in \mathcal{K}_\Gamma$ if and only if $\mathcal{B} \models \phi_\Gamma$.*

A rigorous proof of Proposition 1 requires establishing the relationship between the truth conditions of the property $\mathcal{B} \models \phi_\Gamma$, and those of the property $\mathcal{B} \in \mathcal{K}_\Gamma$. This entails a precise definition of the class $\mathcal{K}_\Gamma$, which in turn relies on the formal semantics of Essence. A formal semantics for a fragment of Essence is given in [10]. While this fragment does not fully contain E_{FO}, the

intended semantics of those parts which are not specified, but which are part of E_{FO}, are obvious.

The ESSENCE semantics is a partial function $[\![P]\!]^{\Omega} \in \{true, false, \bot\}$ (where $\bot$ denotes undefined), specified recursively. Here, P is an ESSENCE expression and Ω is a mapping. We are not concerned here with the details of Ω in the general case. For the case we need, P is a complete E_{FO} specification Γ, and Ω is an assignment to the set of decision variables (i.e., the "find" or expansion symbols).

The semantics is defined for "instantiated specifications", i.e., where the `given` statements have been replaced by `letting` statements specifying exactly an instance. Let Γ be an ESSENCE specification, $\mathcal{A}$ an instance structure for Γ, and E a candidate assignment for the decision (expansion) variables of Γ. Denote by `letting`($\mathcal{A}$) a representation of instance structure $\mathcal{A}$ by `letting` statements, as described in the definition of E_{FO} (Definition 5), and let Γ`(letting(`$\mathcal{A}$`))` denote the instantiation of specification Γ with instance `letting`($\mathcal{A}$). Then the truth of Proposition 1 is immediate from the following fact.

$$[\![\, \Gamma(\texttt{letting}(\mathcal{A}))\,]\!]^{E} = true \;\text{ iff }\; (\mathcal{A}; E) \models \phi_{\Gamma}$$

where $(\mathcal{A}; E)$ is the structure $\mathcal{A}$ expanded by the relations of E.

4 E_{FO} Captures NP

In this section, we consider the question: How expressive is our fragment E_{FO}? The answer is given by the following theorem.

Theorem 2 *Let $\mathcal{K}$ be a class of finite σ-structures. The following are equivalent*

1. *$\mathcal{K} \in NP$;*
2. *There is a first order formula ϕ with vocabulary $vocab(\phi) \supset \sigma$ such that $\mathcal{A} \in \mathcal{K}$ if and only if there exists a structure $\mathcal{B}$ which is an expansion of $\mathcal{A}$ to $vocab(\phi)$ and satisfies ϕ;*
3. *There is an E_{FO} specification Γ with instance vocabulary σ and expansion vocabulary ε such that $\mathcal{A} \in \mathcal{K}$ if and only if there is structure $\mathcal{B}$ which is an expansion of $\mathcal{A}$ to $\sigma \cup \varepsilon$ and satisfies the formula ϕ_{Γ}.*

Proof: The equivalence of 1 and 2 are by Fagin's Theorem [7]. The implication from 3 to 2 is immediate from the Proposition 1 above. Notice that in 2 we are viewing the structures of $\mathcal{K}$ as un-sorted, and in 3 we are viewing them (the very same structures) as being multi-sorted. To go from 2 to 3, consider an arbitrary FO formula ϕ as in 2. Construct an E_{FO} specification as follows.

1. The "given" part consists of a sequence of statements as follows:
   (a) `given` D `new type enum`
   where D is the domain of $\mathcal{A}$,

(b) `given` R`: rel of (`$D \times \ldots \times D$`)`
for each relation R (with corresponding arity) occurring in ϕ.

2. The "find" part consists of a sequence of statements each of the form

   `find` R`: rel of (`$D \times \ldots \times D$`)`

   for each R in $vocab(\phi) \backslash \sigma$.

3. The "such that" part is just

   `such that` ϕ'

   where ϕ' is identical to ϕ except each $\forall\, x$ is replaced by $\forall\, x : D.$, and each $\exists\, x$ is replaced by $\exists\, x : D.$.

To verify that this E_{FO} specification satisfies the condition in item 3, let ϕ be the original formula, and Γ be the E_{FO} specification produced as described. Now, consider the formula ϕ_Γ as defined in Section 3.1. The formulas are identical, except that all quantification in ϕ has turned into guarded quantification in ϕ_Γ. That is, any sub-formula of ϕ of the form $\forall\, x\ \psi(x)$ appears in ϕ_Γ as a sub-formula of the form $\forall\, x\, (D(x) \Rightarrow \psi(x))$. This reflects the translation of views from un-sorted to sorted and back. The models of ϕ are un-sorted. If we view them as multi-sorted models with a single sort, then we see that ϕ and ϕ_Γ have exactly the same models, which are those in $\mathcal{K}$. The guard D in ϕ_Γ is superfluous, because D is the entire domain. ∎

The theorem tells us that the search problems definable by E_{FO} are exactly those definable by FO model expansion, and that the decision problems associated with search problems specifiable in E_{FO} are exactly those in NP.

Remark 4 *There are two related claims which are tempting to make. One could be expressed as "*E_{FO} *can express all and only the NP-search problems", or "*E_{FO} *captures NP-search (a.k.a. FNP)". The other might be expressed as "We may specify decision problems in* E_{FO} *by writing specifications in which there is just one* `find` *statement, and the object found is a Boolean value denoting 'yes' or 'no'. The decision problems we can specify this way in* E_{FO} *are exactly those in NP." The results of Section 6 show why both of these claims would be false.*

4.1 Extending E_{FO}

E_{FO} is a nearly minimal fragment of Essence that captures NP. Smaller fragments are of marginal interest. For example, one could restrict the constraint formulas to be in CNF, but it is not clear why we would study this case. It is more interesting to ask what sorts of features of Essence we could *extend* E_{FO} with, without increasing its expressive power. Here are some examples:

- **Functions:** We made E_{FO} function-free to keep the fragment small and the proofs simple. Extending E_{FO} to allow use of functions in the "given" and "find" vocabularies does not increase expressiveness, since

unrestricted FO has no more expressive power than function-free FO. (In E_{FO}, as well as function-free FO, we can always represent functions by their graphs.)

- **Arithmetic:** We show, in Section 5, how to extend E_{FO} with a limited form of arithmetic, while still capturing NP.
- **User defined types and domains:** In E_{FO}, if the specification or solution requires a domain larger than any instance domain, it must be "simulated", for example with tuples of instance domain elements. It is not hard to extend E_{FO} with the ability to define larger domains which are either fixed or are a function of the instance. To do this, we extend the formalization with an infinite "background" structure that provides access to an unlimited number of "reserve" elements. This is a simplified version of the method we use to handle arithmetic in Section 5. The only source of difficulty is that, to retain capturing of NP, defined domains must be of size polynomial in the instance size. We want the method of specifying them to be as generous as possible, while ensuring this restriction.
- **Domain modifiers:** Essence contains many useful domain modifiers for restricting sets, functions, and relations. Many of these, such as those requiring a function to be a bijection, and those which bound the size of a set or relation, are definable in FO MX (and thus E_{FO}). Extending E_{FO} with syntax for these is easy and does not affect expressiveness.

5 Extending E_{FO} with Arithmetic

In E_{FO} specifications (and their associated structures) numbers must be encoded using non-numeric domain elements. It is desirable to have a language in which numbers and numeric variables can be used directly. Here we consider an extension of E_{FO} to a language we denote $\mathrm{E}_{FO}[\mathcal{N}]$, with "built-in" arithmetic over the natural numbers. It has domains which are sets of natural numbers, the arithmetic operations $+$ and $\times$, sum and product operators $\sum_{x \in D} .F(x)$ and $\prod_{x \in D} .F(x)$, and the standard order relation $<$.

We formalize $\mathrm{E}_{FO}[\mathcal{N}]$ by applying the notion of "metafinite" structures, introduced in [19], which consist of a finite structure together with an infinite "secondary" structure. The structures associated with $\mathrm{E}_{FO}[\mathcal{N}]$ are arithmetical meta-finite structures, where the secondary structure is the natural numbers with arithmetic. To obtain a capturing result for $\mathrm{E}_{FO}[\mathcal{N}]$, we restrict our attention to classes of arithmetical structures which do not have extremely large numbers. We call these "small cost structures". The restriction ensures that encodings of domain elements are never super-polynomial in the domain size. Our definitions and methods adapt and generalize some from [19, 18].

Intuitively, $\mathrm{E}_{FO}[\mathcal{N}]$ seems "more powerful" than E_{FO}, but it is easy to verify that $\mathrm{E}_{FO}[\mathcal{N}]$ cannot specify any problem beyond NP. The hard part of obtaining a capturing result is to show it is sufficiently expressive. $\mathrm{E}_{FO}[\mathcal{N}]$ can express

every problem in NP using encoded numbers, just as E_{FO} can. We need also to show that it can express every problem when using built-in numbers. The difficulty here stems from the fact that complexity is measured with respect to encoding size, and in adding built-in arithmetic to E_{FO}, we have changed the relationship between the objects the language talks about and their encodings.

The size of the encoding of a (standard) structure as a string (see Section 2.7) is polynomial in the number of domain elements. When domain elements are abstract, as in E_{FO}, numbers must be encoded, and encoding a large number requires having a sufficient number of domain elements. As a consequence, there is always a large enough domain that any NP property of (the encodings of) structures can be described by an $\exists SO$ formula. However, in $\mathrm{E}_{FO}[\mathcal{N}]$ an individual domain element may be an arbitrary number, and the polynomial relationship between domain size and encoding size is lost. For example, a structure may have a domain of size one, but this one element may be an arbitrarily large number. It may no longer be possible for an $\exists SO$ formula to describe every NP property of a class of such structures, because the variables in this formula range only over the domain of size one, but the encodings it must talk about are arbitrarily large, depending on the size of the number. The restriction to small cost structures limits our attention to those classes of arithmetical structures where the polynomial relationship between domain size and encoding size is retained.

5.1 $\mathrm{E}_{FO}[\mathcal{N}]$

Let $\mathrm{E}_{FO}[\mathcal{N}]$ denote the following extension of E_{FO}.

1. The "given" part may contain statements of the forms allowed for E_{FO}, and also may have one or more declarations of the forms

   - `given` D`: set of int {0..}`,
   - `given` C`: int {0..}`,
   - `given` F`: function {total} tuple` $\langle D_1, \ldots, D_k \rangle \longrightarrow D_{k+1}$

   These declare D to be a domain of non-negative integers; C to be a constant symbol denoting a non-negative integer; and F to be a k-ary function, respectively. Each D_i in the specification of function F must have been declared by a `given` statement. If the arguments of F include both sets of integers and enumerated types, we call F a "mixed" function. We also extend the "given" relations for E_{FO} to mixed relations.

2. There is a required pair of `letting` statements. The first is

   `letting` *Size* `be` $(|D_i| + \ldots + |D_j \cup \ldots|)$,

   which sets *Size* to be the number of distinct elements in the instance structure. (We compute this by summing the sizes of enumerated domains and adding the size of the union of the integer domains.)

   The second is

`letting` *Bound* `be` $2 ** (Size ** k)$

for some natural number k. This sets *Bound* to be the upper bound on values allowed in the instance by the small-cost criteria.

3. There is a required "where" part consisting of one statement:

   `where` $\forall x \in D_i.\ x < Bound$

   for each "given" domain D_i which is a subset of $\mathbb{N}$, and one statement:

   `where` $C < Bound$

   for each "given" integer C.

   A `where` statement is an ESSENCE statement that places a constraint on the allowed values in an instance. The `where` statements here ensure that no number in the instance is larger than *Bound*, so that instances which violate the small-cost condition will be "rejected".

4. The "find" part is as in E_{FO}, extended with:

   `find` F `: function(total) tuple` $\langle D_1, \ldots, D_k \rangle \longrightarrow D_{k+1}$

   where each D_i has been declared by a `given` statement. F may be over enumerated types, or sets of integers, or may be mixed. We also extend the "find" relations of E_{FO} to mixed relations.

5. The "such that" part is the same as in E_{FO}, but with the following extensions to the syntax of formulas:

   - we have constant symbols $\{0, 1, 2, \ldots\}$, which always denote the natural numbers;
   - we have arithmetic operations $+$, $\times$, and the sum and product operators $\sum_{x \in D} .F(x)$ and $\prod_{x \in D} .F(x)$, where F denotes an expression with free variable x that evaluates to a natural number;
   - the standard order relation $<$ may be used, applied to the natural numbers ($\leq$,$>$, $\geq$, etc., are definable.);
   - quantifiers over numeric domains are of the form $\forall x \in D.$ and $\exists x \in D.$ (to conform with ESSENCE syntax rules);
   - we allow function symbols that have been defined by `given` or `find` statements.

6. We also are allowed the following `letting` statements defining new domains:

   `letting` *Dom* `be` $\{0 .. Size - 1\}$,

   and

   `letting` *BigDom* `be` $\{0 .. (Size ** k) - 1\}$,

`given`	U `new type enum`	$ The set of objects.
`given`	w`: function` $\fatsemi$ `total` $\fatsemi$ `tuple` $\langle U \rangle \longrightarrow$ `int` $\fatsemi$ `0..` $\fatsemi$	$ weights.
`given`	v`: function` $\fatsemi$ `total` $\fatsemi$ `tuple` $\langle U \rangle \longrightarrow$ `int` $\fatsemi$ `0..` $\fatsemi$	$ values.
`given`	B`: int` $\fatsemi$ `0..` $\fatsemi$	$ The weight bound.
`given`	K`: int` $\fatsemi$ `0..` $\fatsemi$	$ The value target.
`letting`	*Size* `be` $(\lvert U \rvert)$	
`letting`	*Bound* `be` $2 ** (\mathit{Size} * * k)$	$ for some $k \in \mathbb{N}$.
`where`	$\forall\, x : U.\ w(x) < \mathit{Bound}$	
`where`	$\forall\, x : U.\ v(x) < \mathit{Bound}$	
`where`	$B < \mathit{Bound}$	
`where`	$K < \mathit{Bound}$	
`find`	U' `: rel of` (U)	$ The objects in the knapsack.
`such that`	$\sum_{u \in U'}.\ w(u) \leq B$	
`such that`	$\sum_{u \in U'}.\ v(u) \geq K$	

Figure 2: Specification of Knapsack in $\mathrm{E}_{FO}[\mathcal{N}]$.

where k is the same value used in computing *Bound*. These letting statements are not necessary in all $\mathrm{E}_{FO}[\mathcal{N}]$ specifications, but they are used in our proof of capturing NP.

Example 5 Figure 2 gives an $\mathrm{E}_{FO}[\mathcal{N}]$ specification of the KNAPSACK problem. The specification is almost the same as the ESSENCE specification for KNAPSACK given in [11], except for the addition of the `letting` and `where` statements enforcing the small-cost condition on instances. $\Diamond$

$\mathrm{E}_{FO}[\mathcal{N}]$ is not a fragment of ESSENCE, primarily because ESSENCE does not (so far) have a product operator. We may obtain a true fragment of ESSENCE from $\mathrm{E}_{FO}[\mathcal{N}]$ by deleting the product operator and addressing some minor syntactic issues. We include the product operator in $\mathrm{E}_{FO}[\mathcal{N}]$ because our proof of capturing NP, in Section 5.3, uses this operator.

We now need to formally extend the underlying task of model expansion to account for a logical treatment of arithmetic. We do it along the lines of the metafinite model theory of Grädel and Gurevich [19].

5.2 Metafinite Model Expansion with Arithmetic

A metafinite structure, as defined by Grädel and Gurevich [19, 18], is a two-sorted structure consisting of a finite primary structure, a (typically infinite) secondary structure (such as the structure of the natural numbers), and functions from primary to secondary part. That definition is too restrictive for our purposes since it does not allow mixed predicates where elements of both primary and secondary structures may occur as arguments. For this reason, we introduce the following definition.

Metafinite Structures with Mixed Relations

Definition 6 *A metafinite structure with mixed relations (mixed structure, for short), is a tuple $\mathcal{D} = (\mathcal{A}, \mathcal{R})$, where*

(i) $\mathcal{A} = (A; M)$ *is a finite two-sorted structure, called the* primary *part of* $\mathcal{D}$.

(ii) $\mathcal{R}$ *is a finite or infinite structure, called the* secondary *part of* $\mathcal{D}$. $\mathcal{R}$ *always contains two distinguished elements, 0 and 1, and may contain multi-set operations.*

(iii) The domain of $\mathcal{R}$ *is called the secondary domain. The two sorts of* $\mathcal{A}$ *are called the primary domain and the active domain (denoted* $adom(\mathcal{A})$*). The active domain is a finite subset of the secondary domain.* $|\mathcal{A}|$ *denotes the size of the combined domain of* $\mathcal{A}$*, that is, the total number of elements in both primary and active domains.*

(iv) M *is a set of finite relations and functions such that each relation* $R_i \in M$ *satisfies* $R_i \subseteq U_1 \times U_2 \times \ldots \times U_k$*, and each function* $f_i \in M$ *satisfies* $f_i : U_1 \times U_2 \times \ldots \times U_k \longrightarrow U_{k+1}$*, where each* U_j *is either* $\mathcal{A} \backslash adom(\mathcal{A})$ *(the primary domain) or R (the secondary domain). If a function or relation has arguments of both kinds, we call it "mixed".*

Typically, we are interested in studying the class of metafinite structures obtained by taking $\mathcal{R}$ to be a fixed infinite structure, such as the natural numbers or integers with standard arithmetic operations. We may view the secondary structure as describing "built-in" functions and relations.

The term "active domain" comes from constraint databases, and denotes the set of elements from a potentially infinite set (such as the natural numbers) which actually occur in relations of the database. Here, the active domain consists of the elements of the secondary domain, for example numbers, which appear in an instance.

First Order Logic for Metafinite Structures

Logics for metafinite structures are an essential component of metafinite theory. These are two-sorted logics, interpreted over combined two-sorted structures. In our case, the logic is essentially standard two-sorted first order logic, which we interpret over metafinite structures with mixed relations as defined above, but with all quantification guarded.

We extend first order logic to include the characteristic function $\chi[\phi](\bar{x})$, which maps a formula ϕ and an assignment of domain elements to its free variables $\bar{x}$ to 0 or 1 (the distinguished elements of the secondary structure). That is, in our formulas we may use $\chi[\phi](\bar{x})$ as a term, where ϕ is any formula. The semantics is that, for any structure $\mathcal{A}$, $\chi[\phi]$ is always interpreted by the characteristic function of the relation defined by ϕ in $\mathcal{A}$. We also have *active domain quantifiers*, $\exists(\forall)\, x \in adom$.

Our main goal here is to ensure that in the presence of infinite secondary structures, the property of capturing NP is preserved. While in the corresponding theorem for E_{FO} we could translate an arbitrary FO formula it into an ESSENCE specification, here we need to impose restrictions on these formulas to avoid infinite interpretations for expansion relations. This is because expansion relations may be mixed, and since the secondary domain is infinite they may be given infinite interpretations.

Fortunately, we don't need anything new to address this issue. In ESSENCE, all decision variables are restricted to be from finite sets, and all quantification is guarded by finite sets. We do the same with our logic, and require that all quantification is guarded, and that all expansion predicates have upper guards, as in the axioms $\phi_{\Gamma,\mathrm{C}}$ and $\phi_{\Gamma,\mathrm{T}}$ of Subsection 3.1. Thus, the restriction to the formulas of our logic is:

- All quantification is guarded;
- For every expansion predicate or function there is an upper guard axiom;
- All guards are instance relations.

Predicates from the instance vocabulary don't need to be guarded from above since their interpretations are already given.

Strictly speaking, the logic is parameterized by the vocabulary of the secondary structure and the instance vocabulary. For the remainder of this section, we will denote the logic $\mathcal{L}$, with a subscript to denote the secondary structure used. The instance vocabulary will be clear from the context.

Arithmetical Structures with Mixed Relations

Arithmetical structures with mixed relations are mixed structures $\mathcal{D} = (\mathcal{A}, \mathcal{N})$, where $\mathcal{N}$ is a structure containing at least $(\mathbb{N}; 0, 1, <, +, \cdot, \sum, \prod)$, with the natural numbers $\mathbb{N}$ as its domain, and where $\sum$,$\prod$ are multi-set operations. Other functions, predicates, and multi-set operations may be included, provided every function, relation and operation of $\mathcal{N}$ is polytime computable. The first order logic for this metafinite structure will be denoted $\mathcal{L}_{\mathcal{N}}$.

Small Cost Arithmetic Structures

For a version of capturing NP with arithmetical structures, we need to restrict the range of numbers that can appear in a structure, as follows. Define $|\mathcal{D}|$, the size of a metafinite structure $\mathcal{D} = (\mathcal{A}, \mathcal{R})$, to be the *total* number of elements of both primary and secondary sort occurring in the finite structure $\mathcal{A}$. That is $|\mathcal{D}| = |\mathcal{A}|$.

Definition 7 *For a mixed arithmetical structure $\mathcal{D} = (\mathcal{A}, \mathcal{N})$, define cost$(D)$, the cost of $\mathcal{D}$, to be $\lceil log(l) \rceil$, where l is the largest number in adom$(\mathcal{A})$.*

That is, the cost of a mixed structure is the size of the binary encoding of its largest number. If $adom(\mathcal{A})$ is empty, the cost is zero.

Definition 8 *A class $\mathcal{K}$ of mixed arithmetical structures has* small cost *if there is some $k \in \mathbb{N}$ such that $cost(\mathcal{D}) < |\mathcal{D}|^k$, for every $\mathcal{D} \in \mathcal{K}$.*

For structures in a small-cost class, each number is encoded by a number of bits which is polynomial in the size of the domain of the structure, and so has a value no larger than $2^{poly(|\mathcal{D}|)}$.

5.3 Capturing NP with $\mathrm{E}_{FO}[\mathcal{N}]$

Theorem 3 *Let $\mathcal{K}$ be a class of small-cost arithmetical mixed metafinite structures over vocabulary σ. Then the following are equivalent:*

1. *$\mathcal{K} \in NP$;*
2. *There is a first order formula ϕ of logic $\mathcal{L}_{\mathcal{N}}$ with vocabulary $\sigma' \supsetneq \sigma$, such that $\mathcal{D} \in \mathcal{K}$ if and only if there is an expansion $\mathcal{D}'$ of $\mathcal{D}$ to σ' with $\mathcal{D}' \models \phi$;*
3. *There is an $\mathrm{E}_{FO}[\mathcal{N}]$ specification Γ with instance vocabulary σ and vocabulary $\sigma' \supsetneq \sigma$, such that $\mathcal{D} \in \mathcal{K}$ iff there is an expansion $\mathcal{D}'$ of $\mathcal{D}$ to σ' with $\mathcal{D}' \models \phi_\Gamma$.*

Proof: First we show the equivalence of 1 and 2. Suppose 2 holds. Observe that upper guards ensure interpretations of expansion predicates are of size polynomial in $|\mathcal{A}|$, and the small cost condition ensures that encodings of structures are of size polynomial in $|\mathcal{A}|$. Checking the truth of the formula in the expanded structure is polytime, since lower guards ensure that quantification is always over a number of elements that is polynomial in $|\mathcal{A}|$, and every operation of $\mathcal{N}$ is polynomial-time computable. Thus, we can guess the extensions of our expansion predicates and check the truth of our formula in that expanded structure in polynomial time. Membership in NP follows.
For the other direction, from 1 to 2, assume $\mathcal{K}$ is in NP. We will show that we can construct the formula ϕ described in item 2 of the theorem. We use a generalization of a method used by Grädel and Gurevich in [19].
We first outline the proof idea. Let $\mathcal{K}$ be a class of mixed metafinite structures $\mathcal{D} = (A; M, \mathcal{N})$, where the primary structure is $\mathcal{A} = (A; M)$, for a set M of mixed functions and relations. We will expand each structure $\mathcal{D} \in \mathcal{K}$ with several new relations. In particular, for each relation (or function) $R \in M$, we will add new relations $\{I_R, P_R, \{S_R^l\}_{1 \leq l \leq r}\}$. We will construct these new relations to encode exactly the same information as is represented in M, but using domain elements only as abstract objects, never as numbers. We call the resulting class of structures $\mathcal{K}'$. We then apply Fagin's theorem, in the following way. We write a first order formula $\bigwedge_R \phi_R$ that defines the correspondence between the relations and functions of M and the new relations encoding them. We then assume $\mathcal{K} \in$ NP and perform the following reasoning. $\mathcal{K} \in$ NP iff $\mathcal{K}' \in$ NP (by construction of $\mathcal{K}'$) iff there is a formula ψ constructed as in Fagin's theorem (by observing that structures in $\mathcal{K}'$ can be viewed as standard, not metafinite structures) iff the formula $\phi = \psi \wedge \bigwedge_R \phi_R$ is the required formula demonstrating statement (2) of the theorem.

Now we give the details. For simplicity, we give the proof for the relational case only. We may think of functions as being represented by their graphs. Consider a mixed relation $R(\overline{x}, \overline{s})$, where $\overline{x}$ denotes the tuple of arguments of primary sort, and $\overline{s} = s_1 s_2 \ldots s_r$ denotes the tuple of arguments of secondary sort. For each such relation, we will introduce expansion relations I_R, P_R, and $S_R^1, \ldots S_R^r$ where:

- $I_R(\bar{i})$ denotes that there is an $\bar{i}^{th}$ tuple in R.
- $P_R(\bar{i}, \overline{x})$ denotes that, if there is an $\bar{i}^{th}$ tuple in R, the tuple of primary elements in that tuple is $\overline{x}$.
- $S_R^l(\bar{i}, \bar{j})$ denotes that, if there is an $\bar{i}^{th}$ tuple in R, the $\bar{j}^{th}$ bit of the binary representation of the secondary structure element s_l is 1.

For each relation R with arguments of secondary sort, we have one relation S_R^l for each argument to R of secondary sort, where l is the index of the argument. The role of S_R^l is to encode the binary representation of each number occurring in position l of some tuple of R. Argument $\bar{i}$ indexes the possible tuples of R while $\bar{j}$ indexes the possible bits in a binary representation of a number. The S_R^l s do not distinguish absence of the $\bar{i}^{th}$ tuple from R from the bits in the $\bar{i}^{th}$ tuple being all zero, so we use I_R to indicate which of the n^k possible tuples for R are actually present.
Now, we must write a first order formula, over the vocabulary of $\mathcal{K}$ plus the expansion to the I_R, P_R and $S_R^1, \ldots, S_R^r$, that says that these encode R correctly. Our formula is

$$\begin{array}{l}\phi_R \;=\; \forall \overline{x}\, \forall\, s_1 \ldots \forall\, s_r\, [R(\overline{x}, \overline{s}) \leftrightarrow \exists\, \bar{i}\, (I_R(\bar{i}) \wedge P_R(\bar{i}, \overline{x}) \\ \wedge\; \bigwedge_l (\, \Sigma_{\bar{j}} (\, \chi[S_R^l(\bar{i}, \bar{j})] \times \Pi_y(2 : y < value(\bar{j}))) = s_l\,))],\end{array}$$

where $value(\bar{j})$ is $j_0 n^0 + \cdots + j_m n^m$, where n is the size of the domain and m is the length of the tuple $\bar{j}$, i.e., $\bar{j}$ is an encoding of that number in n-ary.
For each particular l and i, the term

$$\Sigma_{\bar{j}}(\chi[S_R^l(\bar{i}, \bar{j})] \times \Pi_y(2 : y < value(\bar{j})))$$

computes the value of a secondary element s_l from its representation in $S_R^l(\bar{i}, \bar{j})$. Recall that the number s_l is represented in $S_R^l(\bar{i}, \bar{j})$, which is true (for a fixed i) on those tuples $\bar{j}$ which encode numbers of positions which are 1 in the binary representation of s_l. We have such a formula ϕ_R for each R.
Let $\mathcal{K}$ be a class of small-cost arithmetical mixed metafinite structures over vocabulary σ. Assume $\mathcal{K} \in$ NP. Now, we construct our formula ϕ which is required in item 2 of the theorem. Let $\mathcal{K}'$ be the class of structures $\mathcal{B}$ consisting of $(\mathcal{A}, \{I_R, P_R, \{S_R^l\}_{1 \le l \le r}\}_{R \in M}, \mathcal{N})$ for each $(\mathcal{A}, \mathcal{N}) \in \mathcal{K}$. That is, for each $R \in M$, we add the new relations $\{I_R, P_R, \{S_R^l\}_{1 \le l \le r}\}_{R \in M}$ to each structure in $\mathcal{K}$. Now, $\mathcal{K}$ is in NP if and only if $\mathcal{K}'$ is, if and only if there is a first order formula ψ in the vocabulary of $\mathcal{B}$ such that:

> Every structure $\mathcal{B}$ is in $\mathcal{K}'$ if and only if there is an expansion $\mathcal{B}'$ of $\mathcal{B}$ to the vocabulary of ψ so that $\mathcal{B}' \models \psi$.

We take the conjunction of ψ with all the ϕ_R, and we have a first order formula ϕ such that:

> Every structure $\mathcal{D}$ is in $\mathcal{K}$ if and only if there is an expansion $\mathcal{D}'$ of $\mathcal{D}$ to σ' with $\mathcal{D}' \models \phi$.

The formula ϕ is not a formula of $\mathcal{L}_{\mathcal{N}}$, because it is not guarded. We can rewrite it in guarded form, to produce a formula of $\mathcal{L}_{\mathcal{N}}$, as follows. Considering the ϕ_R, we first rewrite the biconditional into a conjunction of two material implications. In the $\Rightarrow$ direction, the upper guard for R suffices as the guard. In the $\Leftarrow$ direction, we use the active domain quantifiers, which suffice as guards. In ψ, all quantification is over the primary domain, so is trivially guarded.
Finally, we need to show the equivalence of items 1 and 3 in the theorem. To do this, we execute a proof just like the one showing equivalence of items 1 and 2, but we must construct an $\mathrm{E}_{FO}[\mathcal{N}]$ specification rather than a formula of logic $\mathcal{L}_{\mathcal{N}}$. To show we can do this, we sketch how to modify the formulas ϕ_R to produce $\mathrm{E}_{FO}[\mathcal{N}]$ constraints, as follows.

1. Rewrite the quantifiers over the variables $\bar{x}$ and s_i as ESSENCE quantifiers. (The domain for each of these quantifiers is either $adom(\mathcal{A})$ or the primary domain, and we have vocabulary symbols for each.)

2. The variable tuple $\bar{i}$ indexes possible bits in a binary encoding of a number in $adom(\mathcal{A})$, so ranges over $Dom^k = \{0, \ldots, \texttt{Size} - 1\}^k$. So we rewrite $\exists \bar{i}$ as $\exists i_1 \in Dom.\ \exists i_2 \in Dom.\ \ldots\ \exists i_k \in Dom.$

3. $\Sigma_{\bar{j}}$ is an abbreviation for $\Sigma_{j_1} \Sigma_{j_2} \ldots \Sigma_{j_k}$ where the j_i range over Dom. The ESSENCE function `toInt()` maps Booleans to $\{0, 1\}$. So, we rewrite

$$\Sigma_{\bar{j}}(\chi[S^l_R(\bar{i}, \bar{j})] \times \ldots)$$

   as

$$\Sigma_{j_1 \in Dom}.(\Sigma_{j_2 \in Dom}.(\ldots \Sigma_{j_k \in Dom}.(\texttt{toInt}(S^l_R(\bar{i}, \bar{j})) \times \ldots).$$

4. We rewrite $\Pi_y(2 : y < value(\bar{j}))$ as $\Pi_{y \in BigDom}(2 \times \texttt{toInt}(y < value(\bar{j})))$, where $value(\bar{j})$ is rewritten as $(j_0 \times Size{*}{*}0 + \ldots + j_m \times Size{*}{*}m)$.

Completing the $\mathrm{E}_{FO}[\mathcal{N}]$ specification is straightforward. ∎

This formalization of arithmetic is rather limited, in the following sense. The numbers that may be explicitly quantified over, including those that may appear in any solution, consist exactly of those that occur in the instance. This is because upper guards on expansion predicates must be instance predicates. Even with this restriction we can naturally specify many problems (see, e.g., Example 5). More importantly, it is possible to extend the formalization to allow a more general class of upper guards. We do not do this here to keep the presentation simple.

Small Cost Structures: Discussion

Every $\mathrm{E}_{FO}[\mathcal{N}]$ specification requires `letting` and `where` statements enforcing the small-cost condition. A solver could easily detect instances violating the condition, and inform the user, who then would have the option of revising the specification.

If we removed the requirement in $\mathrm{E}_{FO}[\mathcal{N}]$ for statements enforcing the small-cost condition, some axiomatizations would be correct on many instances but incorrect when instances contained sufficiently large numbers. The same probably holds for all existing languages which are intended to capture NP and which have built-in arithmetic.

The difference between our specification of Figure 2 and the specification in [11] deserves remark. There are instances of KNAPSACK with large numbers on which they will disagree: there is a solution, which will be found according to the specification of [11], but not according to the specification of Figure 2, because one of the `where` statements is violated. Both specifications are correct, but they specify slightly different problems. One specifies KNAPSACK restricted to small-cost structures; the other specifies the KNAPSACK problem. In the case of KNAPSACK, the specification without enforcing cost bounds is correct. Intuitively, this is because all computation with numbers is done by the summation operators. But there are other problems where a specification (using only the first-order features of ESSENCE) without cost bound enforcement would be incorrect.

Extending $\mathrm{E}_{FO}[\mathcal{N}]$

We could add the following to $\mathrm{E}_{FO}[\mathcal{N}]$ without increasing expressiveness:

- Any polytime function or multi-set operation on natural numbers.
- Any feature that can be added to E_{FO} without increasing its expressiveness.
- User-defined domains consisting of natural numbers not in the instance structure, provided the structure that results from expanding the instance with these new domains is still a small-cost structure. These domains can also be used as guards.

6 Capturing NP-Search

Complexity theory primarily studies decision problems. A decision problem is formalized as the set of "yes" instances. The use of the keyword `find` in ESSENCE and similar languages reflects the fact that, for many applications, we are interested in search problems. That is, given an instance we want to *find* an object of a certain sort — called a solution — not just answer a yes-no question.

A search problem is formalized as a binary relation R, such that the solutions for instance x are the elements of $\{y : R(x, y)\}$. An NP-search problem is a

polytime binary relation $R(x, y)$ such that there is some polynomial p such that $R(x, y) \Rightarrow |y| \leq p(|x|)$. The set $\{x : \exists\ y\ R(x, y)\}$ is in NP, and R is sometimes called an "NP checking relation". The class of NP-search problems is also called Functional NP (FNP).

Now, suppose you are faced with an NP-search problem $R(x, y)$. You have a specific notion of what constitutes an instance and a solution, which determines your instance vocabulary σ, and a solution vocabulary ε. Theorem 2 tells us that there must be some E_{FO} specification Γ, with "given" vocabulary σ, which specifies the set $\{x : \exists\ y\ R(x, y)\}$. It *does not* tell us that there is such an Γ which has `find` (expansion) vocabulary ε. Indeed, there are cases for which there is no such Γ. For these problems, an E_{FO} specification will have "given" vocabulary σ and `find` vocabulary $\varepsilon \cup \alpha$, where α is a set of "auxiliary" vocabulary symbols. These are symbols for relations or functions that are used in writing the specification, but are not part of the desired solution. Intuitively, we need them because one cannot write the "such that" conditions in a FO formula using only the "given" and "find" vocabulary — it requires that additional concepts be represented explicitly, for which the auxiliary vocabulary symbols are needed.

We say a *search problem R is representable* in a language $\mathcal{L}$ iff there is an $\mathcal{L}$ specification Γ such that for all x and y, x is a Γ-instance with Γ-solution y iff $R(x, y)$. Here, x is over vocabulary σ, and y is over vocabulary ε. We say that $\mathcal{L}$ *captures NP-search* if it can represent exactly the NP-search problems.

From the point of view of logic, we can understand a search problem $R(x, y)$ as a class of $\sigma \cup \varepsilon$-structures $\mathcal{K}_R$. Each $\sigma \cup \varepsilon$-structure $\mathcal{B}$ in this class expands a σ-structure $\mathcal{A}$. Then we can talk about axiomatizing the class of structures $\mathcal{K}_R$ by $\mathcal{L}$-sentences. Logic $\mathcal{L}$ captures NP-search iff it can axiomatize $\mathcal{K}_R$ for exactly those R in FNP.

To capture NP-search, we need to use a logic for which model checking captures exactly the complexity class P. Model expansion for this logic will then capture NP-search. Examples of logics that capture P are the least fixed point logic FO(LFP) and FO(IFP), first order with iterated fixed points [23, 34]. These logics augment FO with a construction to represent induction. The details of the logics are not important, and can be found in, for example, [25]. The important point is that there are properties which are expressible in the logics with fixpoint constructions, but not expressible in FO. An example is graph connectivity. This fact gives us the following theorem.

Theorem 4 *E_{FO}, even if extended with the features listed in subsection 4.1, does not capture NP-search.*

In our opinion, constructs to represent induction are an important feature missing from most constraint modelling languages.

Theorem 5 *E_{FO}, when extended by allowing FO(LFP) formulas in* `such that` *statements, captures NP search over ordered structures.*

Proof: Follows from the fact that FO(LFP) captures P on ordered structures [23, 34]. ∎

Remark 5 *The requirement for ordered structures in the theorem indicates that every domain must have a linear order. This is essential to the proof, and thus one would be restricted in the use of the so-called "unnamed types" of* Essence.

7 E_{SO} and The Polynomial-Time Hierarchy

So far, we have considered Essence specifications when the constraints are essentially first order formulas. We called the corresponding fragment E_{FO}. Essence has many features which are not first order. Here we consider an extension of E_{FO} with second order quantification. This gives constraints the power of second order logic (over finite domains). Together with this comes the power to express any problem in the polynomial-time hierarchy (PH).

7.1 E_{SO}: Second Order Essence

Define E_{SO} to be just as E_{FO}, but with constraints (formulas of the "such that" part) extended to be those obtained as follows:

- every E_{FO} constraint is an E_{SO} constraint;
- if ϕ is an E_{SO} constraint, P a relation symbol, and $D_1, \ldots D_n$, for $n \in \mathbb{N}$, are types specified by `given` statements in accordance with Definition 5, then:

  $$\forall\, P : \texttt{rel of } (D_1 \times \ \ldots \ \times D_n).\ \phi$$

  and

  $$\exists\, P : \texttt{rel of } (D_1 \times \ \ldots \ \times D_n).\ \phi$$

  are E_{SO} constraints.

We call quantifiers of the form $\mathcal{Q}\ P$: `rel of` $(D_1 \times \ \ldots \ \times D_n)$., where $\mathcal{Q}$ is $\forall$ or $\exists$, *second order quantifiers*. Here, P must be a relation symbol. That is, P can occur in ϕ in the form $P(\vec{x})$, but not in the form $R(P)$, where R is a relation symbol. An example of an E_{SO} constraint is in the following `such that` statement.

$$\begin{array}{ll} \texttt{such that} & \exists\, R : \texttt{rel of } (V).\ \exists\, B : \texttt{rel of } (V).\ (\forall\, x : V.\, (R(x) \vee B(x)) \wedge \\ & (\forall\, x : V.\ \forall\, y : V.\, (E(x,y) \rightarrow \neg((R(x) \wedge B(y)) \vee (B(x) \wedge R(y)))))) \end{array}$$

It states that there are two sets R and B which constitute a proper two-colouring of edge relation E. Here, R and B are bound variables, not decision variables (expansion symbols): they are not declared by `find` statements, and a solver would not report their values.

7.2 Obtaining the Polynomial Hierarchy

With E_{FO}, we can express exactly the problems in NP. We are at Σ_1^p with no second order quantifiers because the expansion (`find`) relations are implicitly

existentially quantified. If we allow second order quantification, each alternation of SO quantifiers gives us a jump in the polynomial hierarchy. If the specification contains only universal ($\forall$) second order quantifiers, then we can express problems in Σ_2^p, or $\mathrm{NP}^{\mathrm{NP}}$. If the quantification pattern is $\forall\exists$, then we have Σ_3^p, or $\mathrm{NP}^{\mathrm{NP}^{\mathrm{NP}}}$, etc. This way, all Σ levels of the polynomial hierarchy are precisely captured by ESSENCE specifications with second order quantification.

Definition 9 *Let* $\mathrm{E}_{SO}[\Pi_i]$*, where* $i \geq 1$ *denote the fragment of* E_{SO} *where second order quantification in any* `such that` *statement has at most* i *alternations of second order quantifiers, counting from the first block of* $\forall$*s. In counting, we disregard existential second order quantifiers before the first block of* $\forall$*s because they don't give us any additional expressive power — we already have the implicit existential second order quantifiers given by the "find" part.*

Theorem 6 E_{SO} *specifications in the fragment* $\mathrm{E}_{SO}[\Pi_i]$ *capture the complexity class* Σ_{i+1}^p*.*

That is, every $\mathrm{E}_{SO}[\Pi_i]$-definable class of finite structures is in the complexity class Σ_{i+1}^p, and every class of finite structures in Σ_{i+1}^p is $\mathrm{E}_{SO}[\Pi_i]$-definable.
Proof: *(Sketch)* Since a similar theorem holds for model expansion for the corresponding fragments of second order logic, it is sufficient to establish a correspondence between $\mathrm{E}_{SO}[\Pi_i]$ specifications and SO. We already saw that E_{SO} specifications amount to model expansion for SO where an arbitrary depth of quantifier alternations is possible.
We also need to show that every SO model expansion axiomatization with i alternations of quantifiers (counting from the left-most universal), can be represented by an $\mathrm{E}_{SO}[\Pi_i]$ specification. The idea is that we include all the predicates from the expansion vocabulary ε in the "find" part of our ESSENCE specification. Then we replace $\mathcal{Q}P$ with $\mathcal{Q}P$: `rel of` $(D \times \cdots \times D)$ where $\mathcal{Q}$ is either $\forall$ or $\exists$, and where D is precisely the universe of the structure in our model expansion task. The resulting formula becomes a constraint in a `such that` statement. Capturing follows immediately because it holds for model expansion for the corresponding fragment of SO (see [30] for details). ∎

Corollary 1 E_{SO} *captures the polynomial hierarchy.*

Proof: It is enough to recall that $PH = \bigcup_{i\in\mathbb{N}} \Sigma_i^p$. ∎

Theorem 6 identifies an ESSENCE fragment capturing every Σ level of the polynomial hierarchy except for $\mathrm{P} = \Sigma_0^p = \Pi_0^p$.

Definition 10 *Call an* E_{FO} *constraint universal Horn if it consists of a block of universal (first order) quantifiers, followed by a conjunction of clauses in which each clause has at most one positive occurrence of a relation of the "find" vocabulary.*

Theorem 7 E_{FO}*, restricted to universal Horn constraints, captures* P *on ordered structures.*

Proof: Follows immediately from the fact that ∃SO universal Horn captures P on ordered structures [17]. ∎

8 The Exponential Hierarchy

In this section, we show that very simple extensions to E_{FO} give it very high expressive power — far beyond that obtained by adding second order quantification. The extensions simply give us the ability to define domains with size larger than any polynomial in the instance size. This has the same effect on expressiveness whether done in the problem specification, the instance description, or both.

The special case when the instance description language allows a domain to be expressed simply as its size or — equivalently for our purposes — as a range of integers, is of particular interest. We present that case as an illustrative example in Section 8.1. Then, we devote the remainder of this section to presenting the more general case of producing very large domains by composing large types, or by expressing their size with arithmetic expressions. We do this in the context of simple extensions to the problem specification language.

8.1 NEXP-time with Succinct Domains

Recall that, in an E_{FO} instance, domains must be given by enumeration. (Equivalently, for current purposes, we could give them by their size expressed in unary.) If we extend the instance description language to allow domains to be specified by their size expressed in binary, then we may express problems of much higher complexity. The same results hold if we express sizes in decimal, or if we may specify them as a range of integers in decimal.

Intuitively, the situation is as follows. Giving a domain of size n as its size expressed in unary requires n bits. Similarly, an enumerated set of size n requires at least $n \log n$ bits to describe. On the other hand, if we allow features in the instance description language to describe a set by a range of integers (expressed in binary or decimal), or as a size expressed in binary or decimal, we require only $\log n$ bits to describe the same set. This is an exponentially more succinct representation, and increases the complexity of the problems specifiable in the problem specification language by an exponential.

Suppose we extend E_{FO} specifications with `given` statements of the form

`given` D`: set of int (..)`

and E_{FO} instance descriptions with corresponding `letting` statements of the form

`letting` D `be` $\{0..n\}$,

for any $n \in \mathbb{N}$. These allow us to specify a domain of size n by giving the range $\{0..n\}$ in the instance description. Call a domain D specified in this way a "succinct instance domain".

```
given       Index: set of int ⦇ 0.. ⦈
given       Tiles new type enum
given       t: Tiles
given       H: rel of (Tiles × Tiles)
given       V: rel of (Tiles × Tiles)
find        f: function ⦇ total ⦈ tuple ⟨Index, Index⟩ ⟶ Tiles
such that   f(⟨1, 1⟩) = t  ∧
            ∀ i: Index. ∀ j: Index. (i < n → H(f(⟨i, j⟩), f(⟨i + 1, j⟩))) ∧
            ∀ i: Index. ∀ j: Index. (j < n → V(f(⟨i, j⟩), f(⟨i, j + 1⟩)))]
```

Figure 3: An ESSENCE specification of a Tiling problem.

Theorem 8 *There is an NEXP-complete class $\mathcal{K}$ of finite structures that can be expressed by the fragment of* ESSENCE *consisting of* E_{FO} *extended with succinct instance domains.*

Proof: We demonstrate the claim by providing a specification of the required form for the following Tiling problem. We are given a set of square tile types $T = \{t_0, \ldots, t_k\}$, together with two relations $H, V \subseteq T \times T$ (the horizontal and vertical compatibility relations, respectively). We are also given an integer n (in decimal). An $n \times n$ tiling is a function $f : \{1, \ldots, n\} \times \{1, \ldots, n\} \to T$ such that

1. $f(1, 1) = t_0$, and
2. for all $i < n$ and $j \leq n$ $(f(i, j), f(i + 1, j)) \in H$, and
3. for all $i \leq n$ and $j < n$ $(f(i, j), f(i, j + 1)) \in V$.

Given T, H, V and n, deciding whether an $n \times n$ tiling exists is NEXP-complete. (This is Problem 20.2.10 (a) in [32].) We exhibit the required ESSENCE specification in Figure 3. For convenience and readability we use an extension of E_{FO} with arithmetic and function symbols (much as in Section 5). We can easily write this specification without either feature, by replacing functions with their graphs, and defining an ordered set with successor to use in place of integers. An instance description for the specification could, in part, look like this:

```
letting     Index be {0..n}
letting     Tiles be {t_0, t_1, ..., t_k}
letting     t be t_0
⋮
```

∎

NEXP is known to properly contain NP, and thus this problem is provably not in NP, and not expressible in E_{FO} without succinct domain representation.

8.2 Large Domains in ESSENCE Specifications

ESSENCE allows defining new domains as a function of what is given, and these domains can be extremely large — much larger than the single exponential that is demonstrated in Section 8.1. We show here that, when definition of large domains is allowed, even fragments of ESSENCE which are otherwise very limited can specify problems of extremely high complexity. Indeed, every problem within the exponential time hierarchy can be expressed in the fragment that consists of E_{FO} extended with a very simple form of `letting` statement.

There are at least two ways to construct very large domains in ESSENCE. Suppose we are given an instance domain D of size n. Then, using ESSENCE type constructors and `letting` statements, we may construct new domains such as `set of` D, which is of size 2^n; `rel of` $(D \times \ldots \times D)$, which is of size 2^{n^k}, where k is the arity; and `function tuple` $\langle D, \ldots, D \rangle \longrightarrow D$, etc. We can also define an ESSENCE type in terms of its size, and since ESSENCE includes an exponentiation operator, given n we can define a type of size 2^n. Recall that $2^{[k]}(n)$ denotes the value

$$2^{2^{\cdot^{\cdot^{2^n}}}}$$

where the stack of 2's is k high. Since both type constructors and exponentiation can be composed to arbitrary depth, using either feature we can write a `letting` statement that defines a domain of size $2^{[k]}(n)$, for any $k \in \mathbb{N}$.

An example using type composition could look like this:

```
given       D new type enum
letting     Big be set of set of ... set of set of D
find        f: function ( total ) Big ⟶ Big
such that   ...
```

An example using exponentiation could look like this:

```
given       n int ( 0.. )
letting     Big be new type of size 2 ** (2 ** (2 ** (... (2 ** n) ...)))
find        f: function ( total ) Big ⟶ Big
such that   ...
```

8.3 E_{FO} with Large Domains

Let E_{FO}[big] be the fragment of ESSENCE consisting of E_{FO} extended with `letting` statements of the form

```
letting  B be new type of size expr
```

where *expr* is $2 ** (2 ** (\ldots (2 ** |D|) \ldots))$, for D a domain declared in a `given` statement.

Recall that, for each $k \in \mathbb{N}$, the complexity class k-NEXP is defined to be $\bigcup_{c \in \mathbb{N}} \mathrm{NTIME}(2^{[k]}(n^c))$. For notational simplicity, it is common to leave the union over $c \in \mathbb{N}$ implicit, and write this as k-NEXP $= \mathrm{NTIME}(2^{[k]}(n^c))$.

Theorem 9 *For every $k, c \in \mathbb{N}$, and every problem* Π *in the complexity class* NTIME$(2^{[k]}(n^c))$, there is an E_{FO}[big] specification defining Π.

Proof:(Sketch) Theorem 2 states that every NP class of finite structures has an E_{FO} specification. This follows from Fagin's theorem, which states in part that every NP class of finite structures consists of the models of a sentence of $\exists$SO or, equivalently, has a representation as FO model expansion. To prove the theorem, suppose that a class of structures is in NP. Then it can can be recognized by a time n^c-bounded non-deterministic Turing machine, for some $c \in \mathbb{N}$, where n is the universe size. We must exhibit an $\exists SO$ sentence which defines our class of structures. Let the formula be of the form $\exists\, T\, \phi$, where T is of arity $2c + 1$. We write ϕ so that T encodes the execution of the Turing machine. The first c places are used to index the n^c tape cells needed; the second c places index the n^c possible time steps; the remaining place records the tape symbol. That is, $T(s_1, \ldots, s_c, t_1, \ldots, t_c, a)$ denotes that at time $\bar{t}$, tape cell $\bar{s}$ has symbol a. Additional predicates are useful (but not strictly necessary) to encode the machines state, head position, etc. Any book containing an introduction to finite model theory, including [25, 22, 5], will have details of a proof along these lines.
Suppose that, given an instance with universe of size n, we may construct a domain of size $f(n)$ and quantify over its elements. Now, the same formula as above but with variables ranging over this larger domain describes (non-deterministic) computations of length $f(n)^c$. In our fragment E_{FO}[big] we may construct domains of size $2^{[k]}(n)$, for any $k \in N$, so we can describe any nondeterministic computation with time bounded by $2^{[k]}(n^c)$, and thus may specify any problem in NTIME$(2^{[k]}(n^c))$. ∎

By fixing the depth of composition of exponentiation in E_{FO}[big] to a constant, we can obtain a language capturing k-NEXP for any k. The nondeterministic exponential time hierarchy is the collection of all the classes k-NEXP. NEXP = 1-NEXP properly contains NP, and all the inclusions k-NEXP $\subsetneq$ $(k+1)$-NEXP are proper. Thus, ESSENCE can express problems that are widely considered hopelessly intractable.

Example 6 For illustrative purposes, consider that the number of elementary particles in the known universe has about 85 digits, whereas $2^{[3]}(5)$ and $2^{[2]}(32)$ completely dwarf this number with more than 1.2 million digits. Problems requiring this many steps to solve are certainly intractable. ◊

Remark 6 *The ability to define large domains gives much more expressive power than second order quantification. This does not imply that second order quantification is pointless, as it is certainly more natural to describe some problems with second order quantifiers than with only first order quantifiers and a large defined domain.*

9 On The Expressive Limit of Essence

In Section 8, we showed that Essence can express problems of quite high complexity: every problem in the very large class ELEMENTARY has an Essence specification. This brings us to one of the questions which first motivated our study: What is the upper limit on what Essence can express? The answer could be viewed as informing two particular concerns. One relates to implementability, because a complete solver for Essence must have running time at least that required for the "hardest" problem Essence can specify. On the other side of this coin, it is our understanding that the Essence designers were aiming high in terms of expressiveness, and would be interested in understanding what problems *cannot* be expressed with an Essence specification. We believe we know the answer to the question, but here must restrict ourselves to making two related conjectures.

9.1 Conjecturing a Limit

Our conjecture is that the results of Section 8 account for all the expressive power of unrestricted Essence.

Conjecture 1 *For every* Essence*-specifiable problem P, there exists $k, c \in \mathbb{N}$ such that P is in* $\text{NTIME}(2^{[k]}(n^c))$.

If the conjecture is correct, then unrestricted Essence can express exactly the problems in ELEMENTARY. One obstacle to *proving* an upper bound on the expressiveness of Essence is that it is not easy to be sure that all the many features of Essence — and their interactions — are accounted for. A more serious one is that, as of time of writing, we know of no published complete formal semantics for the language.

9.2 Problems Beyond Essence

Suppose that our conjecture on the expressive limit of Essence is correct. Then the answer to the question "Are there problems Essence cannot specify" is clearly yes. ELEMENTARY relates to some other familiar classes as follows.

$$\begin{aligned}
\bigcup_{k\in\mathbb{N}} \text{NTIME}(2^{[k]}(n^c)) &= \bigcup_{k\in\mathbb{N}} k\text{-NEXP} \\
&= \text{ELEMENTARY} \\
&\subsetneq \text{ITERATED EXPONENTIAL TIME} \\
&\subsetneq \text{PRIMITIVE RECURSIVE} \\
&\subsetneq \text{RECURSIVE}
\end{aligned}$$

Notice that the inclusions here are all proper. Thus, there would seem to be a vast number of problems, even of decidable problems, that Essence cannot

specify. However, most known problems in this category are word or language problems, which can justifiably be said to be outside the class of problems targeted by ESSENCE. The papers on ESSENCE explicitly describe the language as targeting combinatorial problems.

It is therefore interesting to ask a more focused question[3]: Are there (natural) *purely combinatorial* problems that are not expressible in ESSENCE? If Conjecture 1 is correct, there are some. Friedman [9] has exhibited natural combinatorial problems that are complete for ITERATED EXPONENTIAL TIME, and indeed for even higher complexity classes. ITERATED EXPONENTIAL TIME is TIME($2^{[n]}(n)$). Here the height of the stack of exponentials grows with the instance size, so this is significantly beyond problems in the class we conjecture ESSENCE to capture. (At this complexity, non-determinism and exponents on n don't make any difference.)

Here is an example, from [9], of a problem that is complete for ITERATED EXPONENTIAL TIME. We will call it the Maximum Omitted Function problem.

> Instance: Integer $k \geq 1$; two finite sets K, S of finite k-ary functions.
> Find: A largest k-ary function omitting K but not omitting S.

The size of a finite function is the size of its domain. We say that a function f *omits* a finite set K of finite functions if no function $g \in K$ is isomorphic to any restriction of f. The function asked for, if it exists, is finite — but may be exceedingly large.

Conjecture 2 *There is no* ESSENCE *specification for the Maximum Omitted Function problem.*

Conjecture 1 clearly implies Conjecture 2, but Conjecture 2 may be true even if Conjecture 1 is false.

10 Discussion

In this paper, we adopted a descriptive complexity theory viewpoint to study the expressive power of the specification language ESSENCE and several fragments of it. Our study brings to light several points worthy of further discussion.

10.1 High expressiveness of simple languages

The ESSENCE designers, in our understanding, set out to produce a very general and feature-rich language for defining combinatorial problems. It is therefore not surprising that the language is highly expressive. Perhaps more surprising is the fact that, insofar as we are able to determine, very tiny fragments of the language account for all of the expressive power. In particular, we need only extend E_{FO}, our fragment of ESSENCE that captures NP, with type constructors

[3] At least one main ESSENCE designer has expressed interest in this question.

that produce large domains to give it what we conjecture to be the full expressive power of Essence.

Turning to more practical examples than E_{FO}, a claim is made in [12] that every Essence specification can be refined to an equivalent one in Essence′, a much smaller language which could be seen as a "kernel" of Essence. Essence′ *is a very simple language but, like* Essence*, can express every problem in the complexity class* ELEMENTARY. Indeed, the proof of this property for Essence (see Section 8) adapts almost trivially to Essence′.

We are not sure if Essence′ has *exactly* the same power as Essence, because we are not certain of an upper bound on the expressiveness of Essence. The question of expressive equivalence of the two languages is important because Essence′ is very close to many other existing constraint languages which have been designed with very limited features to "ensure implementability". In fact *every reasonable constraint language* which allows arrays as decision variables (i.e., in an equivalent of *find* statements), and which allows those arrays to have indices in the range $1 \dots 2^n$ (where n is the size of an instance domain), can express NEXP-complete problems. If the exponentials can be stacked, then we get ELEMENTARY, as for Essence and Essence′.

Even languages which have been designed specifically to capture NP often express much more than that. This is because they allow succinct representations of instance domains, for example by allowing an instance domain to be expressed as a range of integers $\{1 \dots n\}$. Our proof in Section 8.1 that E_{FO} extended with succinct domain representations expresses NEXP can be trivially adapted to the (actual, not extended) language NP-Spec [1], the input languages for the solvers MXG [30] and ASPPS [4], and other languages that specifically target NP-search problems.

Should we be concerned?

The tools that have been designed primarily to solve NP-search problems, in many respects, do an admirable job. There is a simple reason that, most of the time, we should not worry about the fact that they can express exponentially harder problems. Consider, for example, a problem where the input is a graph $G = \langle V, E \rangle$. If, by convention, we number the vertices of each n-vertex graph $\{1, \dots n\}$, requiring the user to enumerate the n numbers explicitly rather than give the range is an inconvenience, whereas allowing them to be stated as a range is typically of no computational significance. The reason is that the "hardness" of a graph problem is captured in the set of edges, which will be given explicitly, and has size $\Omega(n)$ in almost every application. From this point of view, nothing is amiss, but the formal properties do not exactly match "normal use".

However, there are dangers. One requirement the Essence designers set for themselves is that an Essence specification, together with an instance, can be reduced — not necessarily efficiently — to standard finite-domain CSP languages. Let us call this reduction "grounding". Almost all solvers for constraint modelling languages involve some form of grounding. If we want to ground Essence to an NP-complete language or problem, the grounding algorithm is

of very high complexity. In particular, for every $k \in \mathbb{N}$, there are Essence specifications for which the grounder takes time $2^{[k]}(n)$.

Now, a reasonable Essence grounder will ground many Essence specifications for NP-complete problems in polynomial time. But, it will not be able do this with *all* specifications for NP-complete problems. (Consider this fact, which we will not prove here: given an Essence specification, it is undecidable what the complexity of the specified problem is.) Therefore, it is not possible to rule out the case that a user writes a perfectly reasonable (at least to them) specification for an NP-complete problem, but the grounding time itself is not only super-polynomial, but super-exponential.

10.2 Abstraction and Expressiveness in Essence

Several features of Essence are highlighted in [14, 11] as setting it apart from most other languages in the level of abstraction at which it can specify problems. Since these features seem to be of particular importance in meeting the goals of Essence, it is natural to ask whether their presence has significant implications in terms of expressiveness. We may address this question by considering the effect of adding these features to E_{FO}, our fragment for capturing NP.

Wide range of types

The types and type constructors of Essence include enumerated and integer types along with sets, multi-sets, functions, relations, partitions, and others. Having such a variety of types does not, in itself, contribute significant expressive power. That is, all of these types can be added to E_{FO} without changing its expressive power, provided that the sizes of the domains that variables may range over is limited to a polynomial in the instance size. In practice, these constructors are used in ways that violate this condition, but they are also often used in ways that do not. For example, many Essence specifications involving sets include explicit constant size bounds on those sets.

Nested Types

The Essence type constructors may be composed to arbitrary depth. For example, given a domain D, one may construct a domain which contains all functions mapping sets of matrices of partitions of D to sets of sets of sets of triples of elements of D. Even decision variables may be of such types. This feature immediately leads to extremely high expressive power, but this power is associated with the resulting *size* of objects so defined. If we restrict this size to be polynomial in the instance size, then such complex types do not give expressive power beyond that of E_{FO}.

Quantification over Decision Variables

For E_{FO}, and most existing constraint modelling languages, in any constraint of the form

$$\forall\, x : D\ \phi(x)$$

(or some equivalent), the domain D over which x ranges must be given by the instance. In Essence, this is not necessary, and D may in fact be a decision variable. Because D is declared elsewhere to be a subset of some finite domain, x still ranges over a finite set. In the case of E_{FO}, this set is given by the instance, and it is easy to check that our proof that E_{FO} captures NP remains correct if we extend the syntax to allow D to be a decision variable. The expressiveness also remains unaltered if the domain D may be constructed, as in the Essence complex type constructors, provided the size of D is polynomially bounded.

Unnamed Types

In many constraint modelling languages, all variables are integers, or of some other "structured" type. That is, domains are not just sets of values but sets of values with operations and properties such as ordering. A consequence of this (see, e.g., [13]) is the introduction of symmetries. Essence "unnamed indistinguishable types" are domains consisting of abstract objects which have no properties other than identity. That is, they can be distinguished by an equality test, but in no other way. When such domains are used in the specification, symmetries that occur in a ground instance as a result of ordering these values can be automatically detected and eliminated with symmetry-breaking constraints [12]. To see that this feature does not add expressiveness, one need only observe that all E_{FO} domains are abstract sets, and thus are unnamed types. To see that eliminating them does not reduce power, one need only observe that making all domains ordered does not reduce the expressive power of E_{FO}.

10.3 Toward a Theory of Declarative Languages

We believe there is a need for development of a logic-based theory for constraint modelling languages, and in particular a model-theoretic foundation for the study of these languages and the tools that use them. Fagin's theorem suggested the possibility of viewing logics as declarative languages for problems in interesting complexity classes, but this idea was widely considered of little practical value. As our work here illustrates, researchers in constraint modelling have essentially been executing this idea for some years now, although often implicitly.

Cadoli and Mancini [26, 2] have argued that "∃SO can be considered as the formal logical basis for virtually all available languages for constraint modelling, being able to represent all search problems in the complexity class NP". This view is certainly in line with our own. Indeed, we have previously proposed that *starting* with $\exists\mathrm{SO}(\mathcal{L})$, for some suitable logic $\mathcal{L}$, is a desirable way to construct modelling languages and tools [31, 30]. However, as our work here shows, much

more power than ∃SO is required to account for the power of Essence, and in fact most existing modelling languages. Further, although arithmetic over bounded domains can be axiomatized in ∃SO (as is pointed out, for example, in [26, 2]), the question of whether specifications using this axiomatized arithmetic are equivalent to those in a language with built-in arithmetic may not be trivial. In our view, arithmetic is one of several aspects of constraint modelling languages that should be the subject of a more rigorous treatment than it generally receives. We think that a general, logic-based theory of such languages would support such work.

In Section 6 we have shown that E_{FO} does not capture NP-search. A way to extend E_{FO} to capture NP-search would be to add a least-fixpoint operator, or other form of inductive definition. In our opinion, inductive definitions (or other forms of fixpoint operators) are an important element missing from almost all constraint modelling languages, and not only because of the capturing results. Induction is an extremely useful tool in natural and effective modelling, for example in describing important properties like reachability, and can also provide computational benefits. Languages created outside of the constraint programming community often have a construct to represent induction. For example, the database query language SQL has been extended with induction, and fixpoints are fundamental in the semantics of ASP languages. In [31], we argued for using model expansion for FO(ID), an extension of FO with inductive definitions (see [3]) as the basis of a language for modelling problems in NP. We might speculate that one reason induction has, by and large, not been included in constraint modelling languages, is that deciding on the exact form and semantics is non-trivial. Much relevant work has been done in logic.

The idea of viewing constraint languages as logics specifying model expansion tasks gives us a uniform logic-based view on constraint modelling languages. As we have shown, it provides a basis for the exploration of language expressiveness through the application of tools of descriptive complexity theory. While there are many questions about such languages that descriptive complexity does not answer, having a uniform formal view of many languages should provide a basis for development and application of a variety of theoretical tools that can answer questions about families of languages and their relationships, not just individual languages. We also strongly believe that there will be many *practical* benefits of having such theory. Consider, in analogy, the fields of databases and compilers. There is much in the practice of each area that is not application of theory, but there can be little doubt that the fields and the practical tools they produce are much stronger than they would have been without development of their underlying theories.

10.4 Concluding Remarks

We consider development of Essence and related languages an extremely important direction for the constraints field. Like the Essence designers, we think a good industrial modelling language should support a rich type system, and that an industrial worker with basic discrete math background, but little train-

ing specific to constraint modelling, should be able to specify problems in such a language.[4]

We also strongly believe that *the languages used in practice to specify or model problems should have expressive power commensurate with the complexity of the problems being specified.* This is, in our view, important to producing high-quality and high-performance tools. In this paper, we have shown that a number basic features of Essence lead to extremely high expressive power. This does not constitute a criticism of the design of Essence. Expressive power is, in and of itself, neither good nor bad. As with other things, any notion of "too much" or "too little" must be related to particular goals. However, we do believe there are potential negative consequences of extremely high expressiveness in contexts where it is not necessary and where its sources may not be fully understood.

Thus, we believe it is useful to understand the expressive power of languages we design and use, and to be able to control as well as effectively exploit this power. For languages, like Essence, that are very expressive, we think it is useful to identify the largest possible fragments with limited expressive power. Modellers may choose to remain within the smallest fragment that can specify their problem, or not, depending upon their priorities. But, we also think that, with well designed languages, users should not find the need to use a more expressive language than required. Therefore, we consider it a challenge to researchers in the area to find ways to incorporate the kind of features that Essence provides in such a way that expressiveness and complexity can be constrained and controlled without sacrificing ability to write natural specifications.

Acknowledgements

We would like to thank the editors for their patience and encouragement, Alan Frisch for fruitful discussions and suggestions as well as help with Essence syntax, and the anonymous reviewers for many useful comments. The work reported here was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] M. Cadoli, G. Ianni, L. Palopoli, A. Schaerf, and D. Vasile. NP-SPEC: An executable specification language for solving all problems in NP. *Computer Languages*, 26:165–195, 2000.

[2] M. Cadoli and T. Mancini. Automated reformulation of specifications by safe delay of constraints. *Artificial Intelligence*, 170(8-9):779–801, 2006.

[4] We expect that tackling the hardest problems will always require substantial modelling experience and understanding of solver technology. We also expect the classes of problems for which the best available technologies require human assistance to grow smaller with time.

[3] Marc Denecker and Eugenia Ternovska. A logic of non-monotone inductive definitions. *ACM transactions on computational logic*, 2008. To Appear.

[4] D. East and M. Truszczynski. Predicate-calculus based logics for modeling and solving search problems. *ACM Transactions on Computational Logic*, 7(1):38–83, 2006.

[5] H.-D. Ebbinghaus and J. Flum. *Finite model theory*. Springer Verlag, 1995.

[6] H. B. Enderton. *A Mathematical Introduction to Logic*. Harcourt/Academic Press, New York, 2001.

[7] R. Fagin. Generalized first-order spectra and polynomial-time recognizable sets. In R. Karp, editor, *In: Complexity and Computation, SIAM-AMS Proc., 7*, pages 43–73, 1974.

[8] P. Flener, J. Pearson, and M. Agren. Introducing ESRA, a relational language for modelling combinatorial problems. In *In: M. Bruynooghe (ed), Logic Based Program Synthesis and Transformation: 13th International Symposium (LOPSTR '03), Revised Selected Papers, Lecture Notes in Computer Science, volume 3018.*, pages 214–232. Springer, 2004.

[9] Harvey M. Friedman. Some decision problems of enormous complexity. In *Proceedings, 14th Symposium on Logic in Computer Science (LICS '99)*, pages 2–12, 1999.

[10] Alan M Frisch, Matthew Grum, Chris Jefferson, Bernadette Martinez Hernandez, and Ian Miguel. The essence of ESSENCE: a constraint language for specifying combinatorial problems. In *Proc., Fourth International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, pages 73–88, October 2005.

[11] Alan M. Frisch, Warwick Harvey, Christopher Jefferson, Bernadette Martínez Hernández, and Ian Miguel. ESSENCE: A constraint language for specifying combinatorial problems. *Constraints*, this issue, 2008.

[12] Alan M. Frisch, Christopher Jefferson, Bernadette Martínez Hernández, and Ian Miguel. The rules of constraint modelling. In *Proc., 19th International Joint Conference on Artificial Intelligence*, pages 109–116, 2005.

[13] Alan M. Frisch, Christopher Jefferson, Bernadette Martínez Hernández, and Ian Miguel. Symmetry in the generation of constraint models. In *Proceedings of the International Symmetry Conference*, 2007.

[14] A.M. Frisch, M. Grum, C. Jefferson, B. Martinez Hernandez, and I. Miguel. The design of ESSENCE: A constraint language for specifying combinatorial problems. In *Proc., Twentieth International Joint Conference on Artificial Intelligence (IJCAI '07)*, pages 80–87, 2007.

[15] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[16] Martin Gebser, Torsten Schaub, and Sven Thiele. Gringo : A new grounder for answer set programming. In Chitta Baral, Gerhard Brewka, and John S. Schlipf, editors, *Proc., 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR '07)*, volume 4483 of *Lecture Notes in Computer Science*, pages 266–271. Springer, 2007.

[17] E. Grädel. Capturing Complexity Classes by Fragments of Second Order Logic. *Theoretical Computer Science*, 101:35–57, 1992.

[18] E. Grädel. *Finite Model Theory and Descriptive Complexity*, chapter 3, pages 125–230. In Grädel et al. [20], 2007.

[19] E. Grädel and Y. Gurevich. Metafinite model theory. *Information and Computation*, 140(1):26–81, 1998.

[20] E. Grädel, P.G. Kolaitis, L. Libkin, M. Marx, J. Spencer, M.Y. Vardi, Y. Venema, and S. Weinstein, editors. *Finite Model Theory and Its Applications*. Springer-Verlag, 2007.

[21] P. Van Hentenryck. *The OPL Optimization Programming Language*. MIT Press, 1999.

[22] N. Immerman. *Descriptive complexity*. Springer Verlag, New York, 1999.

[23] Neil Immerman. Relational queries computable in polytime. *Information and Control*, 68:86–104, 1986.

[24] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3), 2006.

[25] L. Libkin. *Elements of Finite Model Theory*. Springer, 2004.

[26] T. Mancini and M. Cadoli. Detecting and breaking symmetries by reasoning on problem specifications. In *Proc., 6th International Symposium on Abstraction, Reformulation and Approximation (SARA 2005). Lecture Notes in Computer Science Volume 3607*, pages 165–181. Springer, 2005.

[27] Maarten Mariën, Johan Wittocx, and Marc Denecker. The IDP framework for declarative problem solving. In *Giunchiglia, E. and Marek, V. and Mitchell, D. and Ternovska, E. (eds), Proc., Search and Logic: Answer Set Programming and SAT (LaSh '06)*, pages 19–34, 2006.

[28] Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J. Stuckey, Maria Garcia de la Banda, and Mark Wallace. The design of the Zinc modelling language. *Constraints*, this issue, 2008.

[29] P. Mills, E.P.K. Tsang, R. Williams, J. Ford, and J. Borrett. EaCL 1.0: an easy abstract constraint programming language. Technical Report CSM-321, University of Essex, December 1998.

[30] David Mitchell, Eugenia Ternovska, Faraz Hach, and Raheleh Mohebali. Model expansion as a framework for modelling and solving search problems. Technical Report TR 2006-24, School of Computing Science, Simon Fraser University, 2006.

[31] David G. Mitchell and Eugenia Ternovska. A framework for representing and solving NP search problems. In *Proc. of the 20th National Conf. on Artificial Intelligence (AAAI '05)*, pages 430–435, 2005.

[32] Christos Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[33] Tommi Syrjänen. Lparse 1.0 user's manual.
Available from: http://www.tcs.hut.fi/Software/smodels/.

[34] Moshe Y. Vardi. The complexity of relational query languages. In *14th ACM Symp.on Theory of Computing, Springer-Verlag*, 1982.