Descriptive Complexity in Lean: Completeness by First-Order Reductions

Pierre Senellart
DI ENS, ENS, PSL University, CNRS, Inria
Paris, France
pierre@senellart.com

Anton Gnatenko
DI ENS, ENS, PSL University, CNRS, Inria
Paris, France
anton.gnatenko@ens.psl.eu

Abstract

We show that descriptive complexity can serve as a foundation for formalizing computational complexity results in a proof assistant, by constructing a Lean library centered around the following concepts: decision problems are isomorphism-invariant predicates on finite structures; complexity classes are defined by their logical characterization; membership is shown by definability witnesses; hardness is shown by first-order reductions from a known hard problem. We also establish bridges to traditional machine models such as (non)deterministic Turing machines. The library proves 73 completeness results, on 68 problems or problem families, over 14 different classes; relations between the classes established inside the logic and not by machine simulation, among them NL = coNL and the Abiteboul–Vianu theorem; and unconditional lower bounds, among them $\mathrm{FO}(\leq) \subsetneq \mathrm{FO}(\leq, \mathrm{TC})$ and the failure of order-free FO(IFP) to capture PTIME.

Keywords: descriptive complexity, computational complexity, first-order reductions, NP-completeness, complexity classes, finite model theory, Lean, Mathlib

1 Introduction

Of the 4 795 papers submitted to arXiv in 2025 whose primary category is one of the seven theoretical computer science categories[1], 475 (or 9.9%) announce, in their title or abstract, a new computational complexity hardness or completeness result (see Appendix A). Such results are almost always established by reduction from a known hard problem, and are essentially never formalized using a proof assistant.

Indeed, proof assistant libraries provide very limited support for computational complexity. Lean's Mathlib [53] defines no computational complexity class, and no reduction carrying a resource bound (§7). The few examples of computational complexity formalization each stop at a handful of completeness theorems [8, 25, 47], or at reductions whose hardness root is left admitted [44] (§7).

We claim the main reason is that standard computational complexity proofs are anchored to a machine model under resource constraints, and that such models resist mechanization. This argument was made by the researchers who attempted to prove otherwise. Forster et al. [23] built a framework for programming and verifying multi-tape Turing machines in Rocq (formerly Coq) [54], and used it to verify a universal machine and a multi-tape to single-tape compiler. Their conclusion, after some nineteen thousand lines, was nonetheless negative, and blamed not their tooling but the model: "Turing machines as model of computation are inherently infeasible for the formalisation of any computability or complexity theoretic result". Gäher and Kunze [25] obtained the first complexity-theoretic result mechanized against a concrete computational model, the Cook–Levin theorem, by giving up Turing machines for the call-by-value λ-calculus; even then machines survive as an intermediate problem in the reduction chain, and the resource analysis remains "a major overhead over just verifying the functional correctness of reductions". Balbach [8] pays that cost in full, formalizing Cook–Levin over multi-tape machines in Isabelle/HOL [42].

Once the NP-hardness of SAT was proved [8, 25], further completeness results might have followed easily. They did not: the four developments we compare in Table 2 hold seven completeness theorems between them, and the three of Gäher and Kunze [25] all sit in the module that proves Cook–Levin itself. Each further proof again depends on a computational model, needing a proof that its reduction runs under bounded resources. A solution to this was named by the authors of [25] themselves, who close by asking how "characterisations of PTIME and NP that are independent from a computational model, like via Fagin's theorem", relate to their own. This paper takes that route and demonstrates that descriptive complexity is fruitful to formalize complexity results and proofs in a proof assistant, namely Lean [16].

Descriptive complexity [31] is the subfield of finite model theory [40] concerned with logical characterizations of complexity classes. Its best known result is Fagin's theorem [22]: the class NP *is* the set of decision problems definable in the existential fragment of second-order logic (∃SO). Such characterizations make it possible to prove complexity results while sidestepping the machine model and its resource constraints entirely. A complexity class is defined by its logical characterization; a problem is shown to belong to it by exhibiting a sentence of the corresponding logic that defines it; and hardness is established through *first-order* (FO) reductions, building an instance of one problem from an instance

[1] cs.CC, cs.DM, cs.FL, cs.DS, cs.GT, cs.CG, and cs.LO.

of another through a family of first-order formulas. FO reductions are strictly weaker than Karp reductions, so hardness proved through them is the stronger statement. And they need only first-order logic, which Mathlib already provides.

Our contributions are as follows:

(i) A framework for descriptive complexity in Lean (§3): decision problems as isomorphism-invariant predicates on finite structures, reductions as first-order interpretations, and an encoding mechanism for converting user-defined instances into this form.
(ii) Complexity classes defined by their logical characterizations (§4), from LOGSPACE to RE and along the polynomial hierarchy, closed under FO reductions by construction, together with properties proved inside the logic: NL = coNL and PSPACE = coPSPACE, the fixed-point characterizations of PTIME and PSPACE, the Abiteboul–Vianu theorem [3, 4], and, on Ehrenfeucht–Fraïssé games built for the purpose, the unconditional lower bounds $\mathrm{FO}(\leq) \subsetneq \mathrm{FO}(\leq, \mathrm{TC})$ and the failure of order-free FO(IFP) to capture PTIME.
(iii) Machine bridges (§5) that validate the approach outside the logic: e.g., we show that NP, defined as ∃SO, is exactly the class of problems reducible to the acceptance problem for nondeterministic machines. The machine, its tape contents and its computations are encoded in a finite structure, with the count of the universe elements bounding the running time. We prove Cook–Levin in both its machine-free and its textbook form.
(iv) 73 completeness theorems across 14 classes on concrete problems (§6), by first-order reductions.

We then compare to existing mechanized complexity theory (§7) and discuss scope and limitations (§8). The library is 212k lines of Lean across 726 modules, using Mathlib v4.33.0 on Lean v4.33.0, available at https://github.com/PierreSenellart/descriptive-complexity.

2 Background

In his PhD thesis, Ronald Fagin [21, 22] proved that existential second-order logic *captures* NP: a property of finite structures is decidable in nondeterministic polynomial time if and only if it is expressible by a sentence in this logic. This finding kicked off the study of computational complexity through logic and finite model theory. We recall the relevant notions and results below and refer the reader to Immerman [31], Ebbinghaus and Flum [19] and Libkin [40] for more details.

2.1 Finite structures and logics

Fix a *relational vocabulary* L, or *vocabulary* for short, that is, a finite set of relation symbols, each with a prescribed arity. An L-*structure* A consists of a set Δ_{A}, called its *domain* or *universe*, and, for each k-ary relation symbol $R \in \mathsf{L}$, an interpretation $R^{\mathsf{A}} \subseteq \Delta_{\mathsf{A}}^k$. For example, a directed graph G can be seen as an $\{E\}$-structure over the universe of its nodes, with $(a, b) \in E^G$ whenever there is an edge from a to b.

First-order logic (FO) formulas are built from atoms of the forms $R(x_1, \ldots, x_k)$ and $x_i = x_j$ using Boolean connectives and quantification over the elements of the universe. A *sentence* is a formula without free variables. For example, $\exists x\, \exists y\, (x \neq y \wedge E(x, y))$ expresses that a graph has at least one edge. We write $\mathsf{A} \models \varphi$ to say that φ is *satisfied* in A.

Complexity is measured over *finite* structures, i.e., those with finite universes, and every statement below quantifies over those only. Satisfaction, on the other hand, is defined at any cardinality: finiteness is a hypothesis of the statements, not part of the semantics (§3.1).

We use several extensions of FO by *closure and fixpoint operators*. The transitive-closure operator TC forms the transitive closure of a binary relation defined by a formula. For example, $\mathrm{TC}[E](x, y)$ says "there is a path from x to y". The deterministic operator DTC is restricted to formulas that define functional binary relations.

Now, suppose that a formula $\varphi(\overline{x})$ uses a fresh relation symbol $R \notin \mathsf{L}$ of arity $|\overline{x}|$, and that R occurs in φ only positively. It induces a monotone operator F_φ on $|\overline{x}|$-ary relations over Δ_{A}, defined by

$$F_\varphi(X) = \{\, \overline{a} \in \Delta_{\mathsf{A}}^{|\overline{x}|} \mid (\mathsf{A} \text{ with } R^{\mathsf{A}} = X) \models \varphi(\overline{a}) \,\}.$$

Then $\mathrm{LFP}[\varphi](\overline{x})$ holds if and only if $\overline{x}$ belongs to the least fixed point of F_φ. For example, graph reachability is equivalently defined by the formula $\mathrm{LFP}[x = y \vee \exists z\, (R(x, z) \wedge E(z, y))](x, y)$. Two variants appear below. The *inflationary* fixed point, IFP, is that of the inflationary operator $F'_\varphi(X) = X \cup F_\varphi(X)$, so it converges without asking R to occur positively; the *partial* fixed point PFP iterates F_φ from the empty relation under no restriction at all, and returns the limit when the iteration stabilizes and the empty relation when it does not.

We also use second-order logic, which admits *relational variables* of various arities. Over a graph, for example, a unary relational variable represents a set of vertices, while a ternary relation variable stands for a set of triples. An ∃SO sentence has only existential second-order quantifiers. It can, for example, say that the graph is bipartite:

$$\exists X\, \forall x\, \forall y\, \big(E(x, y) \to (X(x) \leftrightarrow \neg X(y))\big).$$

Its universal counterpart, ∀SO, permits only universal second-order quantifiers. More generally, the second-order alternation hierarchy consists of the fragments Σ^1_k and Π^1_k, for $k \geq 1$. A sentence in Σ^1_k has a prefix with at most k alternating blocks of second-order quantifiers, beginning with an existential block; Π^1_k is defined dually, beginning with a universal block. Thus, Σ^1_1 is ∃SO and Π^1_1 is ∀SO.

The Horn and Krom fragments require the first-order matrix of an ∃SO sentence to be a universally quantified conjunction of clauses, disjunctions of literals, restricted in

their second-order literals only, with first-order atoms occurring freely as a *guard* γ on the clause. A *Horn* clause has at most one positive second-order literal, so it reads as a rule $\gamma \wedge X_1(\overline{x}_1) \wedge \cdots \wedge X_n(\overline{x}_n) \to X(\overline{y})$ over the quantified relation variables, or as the same with $\bot$ on the right; a *Krom* clause has at most two second-order literals. The respective fragments are denoted SO-Horn and SO-Krom, and over ordered structures, where the guards may use $\leq$, SO-Horn($\leq$) and SO-Krom($\leq$).

Finally, a logic can be extended with *value invention*, in the style of the object-creating query languages of Abiteboul et al. [1, ch. 18]. An $\exists\mathrm{SO}_{\mathrm{new}}$ sentence is an $\exists$SO sentence whose relation variables range over the universe of A extended with finitely many fresh elements, related to nothing, with a unary predicate old $\notin \mathsf{L}$ marking the original ones. How many may be invented is itself a resource: holding them to the number of d-tuples of the instance gives $\exists\mathrm{SO}_{\mathrm{new}}[d]$, holding them to exponentially many gives $\exists\mathrm{SO}_{\mathrm{new}}[\exp]$, and leaving them unbounded gives $\exists\mathrm{SO}_{\mathrm{new}}$.

2.2 Descriptive complexity

From the descriptive point of view, a *decision problem* P over a vocabulary L is an isomorphism-closed set of L-structures. It is *definable* in a logic $\mathcal{L}$ if some $\mathcal{L}$-sentence φ over L has $\mathsf{A} \models \varphi$ iff $\mathsf{A} \in P$, for every finite L-structure A. It is *definable over ordered structures*, or definable in $\mathcal{L}(\leq)$, if some $\mathcal{L}$-sentence φ over $\mathsf{L} \cup \{\leq\}$ has $(\mathsf{A}, \leq) \models \varphi$ iff $\mathsf{A} \in P$, for every finite L-structure A and *every* linear order $\leq$ on its universe. The order is a device of the definition and not part of the input: the problem is a set of L-structures either way, and a sentence using $\leq$ counts only if its answer is the same whichever order it is given.

Let C be a class of decision problems. A logic $\mathcal{L}$ *captures* C if, for every vocabulary L, the L-problems in C are precisely the L-problems definable in $\mathcal{L}$, and it captures C *over ordered structures* if they are precisely those definable in $\mathcal{L}$ over ordered structures. The characterizations this paper relies on are collected in Table 1, which lists them beside the classes as the library defines them.

We use a construction on classes called *exponential expansion*. Similarly to a first-order interpretation (§2.3), it defines a new structure on top of an old one, but here the new elements are given by *relations* (or tuples of relations) over the original universe, cut down by a definable condition. Second-order sentences over the old structure thus become first-order sentences over the new. As a universe of size n carries 2^{n^a} relations of arity a, the expanded universe is exponentially larger and a resource bound read there is one exponential higher. For a class C, we write $C^{\exp}$ for the problems that some exponential expansion turns into a problem of C.

Table 1. The complexity classes of the library: each is *defined* by the (first) logic in the middle column, any further one being proved equivalent to it, and a $\leq$ marking a fragment taken over ordered structures. The right-hand column gives a single problem proved complete for the class, not all. PH has no complete problem unless the hierarchy collapses, and GI is a degree (§4.2), not captured by a logic.

Class	Definition	Complete problem
LOGSPACE	FO($\leq$, DTC)	DET-REACH
NL	SO-Krom($\leq$), FO($\leq$, TC)	REACH
PTIME	SO-Horn($\leq$), FO($\leq$, LFP)	CVP
NP	$\exists$SO, $\exists\mathrm{SO}_{\mathrm{new}}[d]$	SAT
coNP	$\forall$SO	TAUT
DP	$\Sigma^1_1 \wedge \Pi^1_1$	SAT-UNSAT
Σ^{p}_k	Σ^1_k	QBF_k
Π^{p}_k	Π^1_k	$\mathrm{QBF}^{\forall}_k$
PH	SO	–
PSPACE	SO(TC), FO($\leq$, PFP), $\mathrm{NL}^{\exp}$	QSAT
EXPTIME	SO(LFP), $\mathrm{PTIME}^{\exp}$	ATM-SPACE
NEXPTIME	$\mathrm{NP}^{\exp}$, $\exists\mathrm{SO}_{\mathrm{new}}[\exp]$	WIDE-TILING
EXPSPACE	SO(PFP), $\mathrm{PSPACE}^{\exp}$	WIDE-CORRIDOR
RE	$\exists\mathrm{SO}_{\mathrm{new}}$	FINSAT
GI	degree of GRAPH-ISO	GRAPH-ISO

2.3 First-order reductions

Reductions are defined through first-order interpretations, in the manner of Dahlhaus [14]. Fix two vocabularies L and L'. A d-dimensional first-order interpretation I from L to L' consists of a finite nonempty set T of *tags*, of a first-order L-formula $\delta_t(\overline{x})$ (*domain formula*) for each tag $t \in T$, where $|\overline{x}| = d$, and, for each k-ary relation symbol $R \in \mathsf{L}'$ and each tuple $\overline{t} = (t_1, \ldots, t_k) \in T^k$ of tags, of a first-order L-formula $\rho_{R,\overline{t}}(\overline{x}_1, \ldots, \overline{x}_k)$, where $|\overline{x}_i| = d$. Applied to an L-structure A, the interpretation produces the L'-structure $I(\mathsf{A})$ whose universe holds one disjoint copy of the admitted d-tuples per tag, $\{(t, \overline{a}) \mid t \in T,\ \overline{a} \in \Delta^d_{\mathsf{A}},\ \mathsf{A} \models \delta_t(\overline{a})\}$, and whose relations are defined by $R^{I(\mathsf{A})} = \{((t_1, \overline{a}_1), \ldots, (t_k, \overline{a}_k)) \mid \mathsf{A} \models \rho_{R,\overline{t}}(\overline{a}_1, \ldots, \overline{a}_k)\}$. Textbook presentations have no tags and obtain the same copies from an order on the input, raising the dimension and letting the extra coordinates take distinguished values that the order names. We keep the tags explicit, as this does not require the L-structure to be ordered.

A *first-order reduction* from an L-problem P to an L'-problem Q is a first-order interpretation I such that $\mathsf{A} \in P$ iff $I(\mathsf{A}) \in Q$ for every finite nonempty L-structure A (§3.3 says why nonempty); we write $P \preccurlyeq_{\mathrm{fo}} Q$, which the Lean sources write `P ≤ᶠᵒ Q`. Here the domain formulas are trivial, $\delta_t \equiv \top$, so that the output universe is all of $T \times \Delta^d_{\mathsf{A}}$; only the relativized variant below uses them. Our notions of hardness and completeness are the usual ones for this type of reduction. From the classical standpoint, such a reduction is computable by a uniform family of constant-depth, polynomial-size Boolean

```
variable (L : Language.{0, 0})
structure DecisionProblem [L.IsRelational] where
  Holds : ∀ (A : Type) [L.Structure A], Prop
  iso_invariant : ∀ {A B : Type} [L.Structure A] [L.Structure B],
    (A ≃[L] B) → (Holds A ↔ Holds B)

variable {L L' : Language.{0, 0}}
structure FOReduction [L.IsRelational] [L'.IsRelational] (P : DecisionProblem L)
    (Q : DecisionProblem L') where
  Tag : Type
  [tagFinite : Finite Tag]
  [tagNonempty : Nonempty Tag]
  dim : ℕ
  toInterpretation : FOInterpretation L L' Tag dim
  correct : ∀ (A : Type) [L.Structure A] [Finite A] [Nonempty A],
    P A ↔ Q (toInterpretation.Map A)
```

Figure 1. Decision problems, an isomorphism-closed property of relational structures, and first-order reductions between them, whose `correct` field ranges over finite nonempty structures.

circuits: FO($\leq$) sits inside AC^0, which is FO($\leq$, BIT). The containment is strict, and unconditionally so, a lower bound the library proves (§4.4). A first-order reduction is therefore computable in polynomial time, and is in particular a Karp reduction (proved in the library as `FOReduction.toLFP`, using the FO($\leq$, LFP) characterization of PTIME).

Two relaxations of $\preccurlyeq_{\mathrm{fo}}$ are needed. An *order-invariant* reduction $P \preccurlyeq_{\mathrm{fo}}^{\leq} Q$ may use the symbol $\leq$ in its formulas and, order-invariantly as above, must be correct for every linear order on the input. Any classical reduction that indexes its gadgets by position is of this kind (§4.2). A *relativized* reduction $P \preccurlyeq_{\mathrm{rfo}}^{\leq} Q$ is an order-invariant one with nontrivial domain formulas: its output universe is the definable set $\{(t, \overline{a}) \mid \mathtt{A} \models \delta_t(\overline{a})\}$. Karp's cover-testing gadget for Hamilton Circuit is of this kind. The three relations are ordered by strength, $\preccurlyeq_{\mathrm{fo}}$ being the strongest, and hardness travels forward along each of them, so a completeness proof may use whichever is convenient. Which one a reduction needs is a property of its gadget (see §6.1).

2.4 First-order logic in Mathlib

Mathlib's `ModelTheory` library formalizes classical first-order model theory: syntax and semantics, substructures and elementary maps, ultraproducts and compactness, Fraïssé limits, types, and Skolem functions. Its focus is not finite model theory: it has neither Ehrenfeucht–Fraïssé games nor logical characterizations of complexity classes, and those are developed here. What the library builds on is its first-order infrastructure: a vocabulary is a `FirstOrder.Language`, an `L`-structure on a type `A` an instance of `L.Structure A`, a formula a `BoundedFormula` or a `Sentence`, with Mathlib's realization semantics, its notion of isomorphism, written `A ≃[L] B`, and its interface for reading a relation of a structure as its order $\leq$.

Mathlib also has `Set.Definable`, but it addresses a different question: whether a subset of one fixed structure is defined by some formula, possibly with parameters. Here a decision problem is an isomorphism-invariant property of finite structures, and its defining formula must be retained. This is the object required by the capture theorems and by the logical fragments developed in the library.

3 The Reduction Framework

We begin the tour of the library with the formalization of decision problems and reductions defined in §2.2 and §2.3. Furthermore, we provide a reliable way for the user to encode decision problems given in conventional, algorithm-based terms into the language of descriptive complexity and to decode the resulting statements back.

3.1 Decision problems as predicates on structures

A vocabulary is a `FirstOrder.Language` `L`, which in Mathlib may carry function and constant symbols as well, together with an `IsRelational` instance restoring the convention of §2.1. Given such a vocabulary `L`, a `DecisionProblem L` is an isomorphism-invariant predicate on `L`-structures. It is represented by a structure with two fields: a predicate `Holds` and a proof `iso_invariant` that isomorphic structures agree on it (Figure 1).

At this point the predicate `Holds` is not restricted to finite structures. Finiteness is imposed later by the notions that use it. Thus, a user can formalize a decision problem as its natural semantic predicate, without carrying finiteness conditions through every definition. What a problem does on infinite structures is then invisible to the theory: membership and hardness are fields of a complexity class that come with `mem_congr_finite` and `hard_congr_finite`, saying that two problems agreeing on all finite structures agree on both.

```
variable (L L' : Language.{0, 0})
structure FOInterpretation (Tag : Type) (dim : ℕ) where
  relFormula : ∀ {n : ℕ}, L'.Relations n → (Fin n → Tag) → L.Formula (Fin n × Fin dim)

variable {L L' : Language.{0, 0}} {Tag : Type} {dim : ℕ}
protected def Map (_I : FOInterpretation L L' Tag dim) (A : Type) : Type :=
  Tag × (Fin dim → A)

variable (I : FOInterpretation L L' Tag dim) (A : Type) [L.Structure A]
instance mapStructure [L'.IsRelational] : L'.Structure (I.Map A) where
  funMap f := isEmptyElim f
  RelMap R xs := (I.relFormula R fun i => (xs i).1).Realize fun p => (xs p.1).2 p.2
```

Figure 2. First-order interpretation semantics: `relFormula` determines `Map`, whose interpreted universe is `Tag`×(`Fin dim` → `A`).

Isomorphism-invariance says simply that renaming the elements of a structure does not change whether it is a yes-instance. It is needed when reductions are composed: the two-step output and the output of the composite have different Lean carrier types, but are isomorphic. The invariant proof transfers the answer between them.

3.2 Interpretations

As in theory, a first-order reduction of an `L`-problem to an `L′`-problem is a first-order interpretation from `L` to `L′` that respects the `Holds` predicate. An interpretation is parameterized by a finite type `Tag` and by its dimension d. Its output elements are the tagged d-tuples, $\mathtt{Tag} \times \Delta_{\mathtt{A}}^{d}$.

These parameters fixed, an `FOInterpretation` defines a single field, `relFormula` (Figure 2). Fix a relation symbol R of `L′`, of arity n, and an assignment τ of tags to its n arguments. At that pair, `relFormula` supplies an `L`-formula $\rho_{R,\tau}$ whose free variables are the pairs (i, j) with $1 \le i \le n$ and $1 \le j \le d$: the figure writes them $\mathtt{Fin}\, n \times \mathtt{Fin}\, \mathtt{dim}$, counting from 0. Writing the arguments as in §2.3, the variable (i, j) is the j-th coordinate of $\overline{x}_i$. Applied to an `L`-structure `A`, these formulas define the relations of the interpreted `L′`-structure `I.Map A`: tagged tuples $\overline{x}_1, \ldots, \overline{x}_n$ are related by R exactly when $\rho_{R,\tau}$ holds of their coordinates, τ being the tags they carry. The library proves that interpretations map finite structures to finite structures, nonempty structures to nonempty structures when their tag type is nonempty, and isomorphic structures to isomorphic structures.

For a complete example, let $\mathtt{L} = \mathtt{L}' = \{E\}$ with E binary, and take $\mathtt{Tag} = \{\ell, r\}$ and $d = 1$. An output element is then a pair (t, u) of a tag and a vertex of `A`, so `I.Map A` carries a left and a right copy of every vertex of `A`. Since E is binary, an assignment τ is a pair (t_1, t_2) of tags, one per argument, and `relFormula` must be supplied at each of the four. As $d = 1$, each formula has one variable per argument, $(1, 1)$ and $(2, 1)$, which we write x and y; the symbol $R = E$ being fixed, we write $\rho_{t_1 t_2}$ for the formula chosen at (t_1, t_2), so that $\rho_{\ell r} = E(x, y)$ and $\rho_{\ell\ell} = \rho_{r\ell} = \rho_{rr} = \bot$. The interpreted graph has an edge from the left copy of u to the right copy of v exactly when $E^{\mathtt{A}}(u, v)$, and no other edges. Appendix F shows an interpretation of this shape written out in Lean.

The definition of §2.3 differs in one more technical respect. There, the domain formulas keep only the tagged tuples $(t, \overline{a})$ with $\mathtt{A} \models \delta_t(\overline{a})$. An `FOInterpretation` has no such field: its output universe is the whole tagged product $\mathtt{Tag} \times \Delta_{\mathtt{A}}^{d}$. Tagged tuples outside the intended universe are harmless: the relation formulas can leave them isolated. This avoids building a subtype of definable tuples into every interpreted structure. The δ_t come back as a field `domFormula` in `RelFOInterpretation`, which is what the relativized reductions $\preccurlyeq_{\mathrm{rfo}}^{\le}$ are built from.

3.3 Reductions

An `FOReduction` from `P` to `Q`, written $P \preccurlyeq_{\mathrm{fo}} Q$, packages an interpretation with a proof $\mathtt{P}(\mathtt{A}) \leftrightarrow \mathtt{Q}(I.\mathrm{Map}(\mathtt{A}))$ for every finite nonempty input structure `A` (Figure 1). In other words, it preserves the answer on exactly the finite nonempty instances to which the library's complexity classes attach membership and hardness statements. We highlight that correctness is only required on nonempty inputs. A fixed-dimensional interpretation maps an empty universe to an empty universe, and therefore cannot produce a nonempty target instance from an empty source instance.

To explain further constructions we need one more technical definition. An `L′`-formula φ is *pulled back* along an interpretation I from `L` to `L′` into an `L`-formula φ^I: every relation atom becomes the formula that I supplies for that relation, every quantifier over output elements becomes a block of d quantifiers over coordinates, one copy per tag, the tags being finite in number. The pullback preserves meaning: φ^I is true in `A` exactly when φ is true in `I.Map A` (`FOInterpretation.realize_pull`).

Reductions can be composed, as usual. Given $P \preccurlyeq_{\mathrm{fo}} Q$ and $Q \preccurlyeq_{\mathrm{fo}} R$, the library constructs $P \preccurlyeq_{\mathrm{fo}} R$ by pulling the formulas of the second interpretation back along the first. The only Lean-specific issue is that the carrier of the composite is not definitionally the same type as the carrier obtained by applying the two interpretations in sequence: one is a flattened tuple of coordinates, the other a tuple of tuples.

```
structure Encoding (L : Language.{0, 0}) (ι : Type*) where
  size : ι → ℕ
  Univ : ι → Type
  ...
  card_le : ∃ c d : ℕ, ∀ i, Nat.card (Univ i) ≤ c * (size i + 1) ^ d
  le_card : ∃ c d : ℕ, ∀ i, size i ≤ c * (Nat.card (Univ i) + 1) ^ d
```

Figure 3. An `Encoding` enforces no padding or compression

The library constructs an isomorphism between them, then invokes the target problem's `iso_invariant` proof. Hence `FOReduction.trans` establishes transitivity. Together with the one-dimensional, one-tag identity interpretation, this makes $\preccurlyeq_{\text{fo}}$ a preorder.

Definability travels backward along the same pullback: if φ defines Q and I witnesses $P \preccurlyeq_{\text{fo}} Q$, then φ^I defines P. Its contrapositive (`not_le_of_not_foDefinableFree`) is the lower-bound form: one inexpressibility result rules out every reduction to an FO-definable problem.

The two relaxations of §2.3 are defined analogously. An `OrderedFOReduction`, written `P` $\leq^{\text{fo}}$`[≤] Q`, interprets over the vocabulary expanded with ≤ and must be correct for every linear order on the input; a `RelOrderedFOReduction`, `P` $\leq^{\text{rfo}}$`[≤] Q`, is built on a `RelFOInterpretation` and so carries the domain formulas as well. Both compose, and definability travels backward along them too.

3.4 Encodings and decodings

Decision problems in the library concern finite structures, whereas users usually start from concrete data: lists, formulas, weights, and the like. An *encoding* translates such data into structures, allowing the library's definability and complexity results to apply; a *decoding* reads results back.

`Encoding` records the input type and its conventional size measure, the encoded universe and relations as computations, and two polynomial bounds relating the cardinality of that universe to that size. Both directions are needed: representing binary integers over a unary universe makes the structure exponential in the input length, and compression the other way can invalidate a hardness result. Both are fields (Figure 3). Preservation of meaning is a separate predicate, `Encoding.Faithful`, saying that a concrete instance and its encoded structure are decided alike; keeping it apart lets one encoding serve several problems over the same vocabulary.

A faithful encoding transfers membership: if the structural problem `P` lies in a class, so does the user's. Hardness does not transfer in the same way, as `P` may be hard because of malformed structures that no instance encodes. The library therefore restricts `P` to a well-formedness condition `W`, obtaining `W` ∩ `P`, and asks for a `Decoding`: a computation taking a finite-structure presentation to the concrete instance it stands for, or to `none` when it stands for none, agreeing with `P` where it succeeds and defined on every well-formed nonempty presentation. Hardness of the restricted problem then concerns only structures that some instance presents.

4 Complexity Classes, Logically Defined

A class here is a set of decision problems closed under first-order reductions and usually obtained from a logic.

4.1 Defining a complexity class

A class is defined by the constructor `ComplexityClass.ofMem`, from a membership predicate together with two proofs about it: that membership is stable under first-order reductions, and that it is determined by the finite structures (Figure 4). The stability proof comes from a *pullback lemma*: if the target problem is definable in the logic, so is the problem reducing to it. Being an argument of the constructor, it must be in hand before the class exists, and there is one such lemma per logic (Appendix B), because the pullback of §3.3 is first-order work only for FO. A relation variable of arity n in the target sentence ranges over relations on the output universe $\texttt{Tag} \times \Delta_{\text{A}}^{d}$, which no relation on Δ_{A} is. It is replaced by a *block* of $|\texttt{Tag}|^n$ relation variables of arity nd, one per assignment of tags to its arguments, and every atom of the sentence is rewritten to read the variable its tags select (`SOBlock.pull`). The rewriting is atom by atom, so it takes a clause to a clause and preserves the Horn and Krom fragments (`SigmaSOHornDefinable.of_orderedReduction`, `SigmaSOKromDefinable.of_orderedReduction`): this is why PTIME and NL are closed without leaving their defining fragments. A fixed-point operator is pulled the same way, its relation variable as a block and its step formula as first-order (`LFPDefinable.of_orderedReduction`, and likewise for IFP, PFP and TC). Hardness is then determined by the membership predicate: P is hard for the class when every member of it reduces to P (`cofinalHard_iff`). Completeness for a class is a pair of certificates, for membership and for hardness. Each class of Table 1 has a problem proved complete for it (§6); PH has none unless the polynomial hierarchy collapses, hence the dash in the table.

4.2 A catalog of classes

Most definitions through logics are standard, and we cite Immerman [31] by theorem number wherever the book has the result. LOGSPACE is FO(≤, DTC) (Thm. 9.11); NL is FO(≤, TC) (Cor. 9.22) and SO-Krom(≤) (Thm. 9.32); PTIME is FO(≤, LFP) (Thm. 4.10), after Immerman [28] and Vardi

```
def ComplexityClass.ofMem
    (Mem : ∀ {L₀ : Language.{0, 0}} [L₀.IsRelational], DecisionProblem L₀ → Prop)
    (mem_of_foReduction : ∀ {L₁ L₂ : Language.{0, 0}} [L₁.IsRelational] [L₂.IsRelational]
      {P : DecisionProblem L₁} {Q : DecisionProblem L₂}, (P ≤ᶠᵒ Q) → Mem Q → Mem P)
    (mem_of_orderedReduction : ∀ {L₁ L₂ : Language.{0, 0}} [L₁.IsRelational] [L₂.IsRelational]
      {P : DecisionProblem L₁} {Q : DecisionProblem L₂}, (P ≤ᶠᵒ[≤] Q) → Mem Q → Mem P)
    (mem_congr_finite : ∀ {L₁ : Language.{0, 0}} [L₁.IsRelational] {P Q : DecisionProblem L₁},
      (∀ (A : Type) [L₁.Structure A] [Finite A], P A ↔ Q A) → (Mem P ↔ Mem Q)) :
    ComplexityClass
```

Figure 4. Constructor for `ComplexityClass`: a membership predicate, and the three obligations it must meet

[58], and SO-Horn(≤) (Thm. 9.32), the two second-order fragments being due to Grädel [26]. NP is ∃SO by Fagin's theorem [22] (Thm. 7.8); the levels Σ^{p}_k and Π^{p}_k are Σ^1_k and Π^1_k (Thm. 7.21), after Stockmeyer [49], and PH is SO (Cor. 7.22). PSPACE is FO(≤, PFP) (Thm. 10.13), due to Vardi [58], and SO(TC) (Cor. 10.29). DP is from Papadimitriou and Yannakakis [43], read as $\Sigma^1_1 \wedge \Pi^1_1$ from Fagin's theorem.

The last four rows are of a different kind, and the book has none of them. Several are reached by the exponentiation operator $C \mapsto C^{\exp}$ (§2.1), which the library proves to relate the classes one exponent apart: PSPACE = $\mathrm{NL}^{\exp}$ (`PSPACE_eq_NL_exp`), EXPTIME = $\mathrm{PTIME}^{\exp}$, EXPSPACE = $\mathrm{PSPACE}^{\exp}$. EXPTIME and EXPSPACE are nevertheless *defined* by SO(LFP) and SO(PFP), after the fixpoint logics of Abiteboul et al. [2]. NEXPTIME is third-order in the literature [38], a syntax the library lacks, so it is $\mathrm{NP}^{\exp}$ here, with $\exists\mathrm{SO}_{\mathrm{new}}[\exp]$ for its logic. RE is $\exists\mathrm{SO}_{\mathrm{new}}$, unbounded value invention, the device of the object-creating query languages of Abiteboul et al. [1, ch. 18], its complete problem FINSAT being Trakhtenbrot's [55].

Not every class need come from a logic. The *degree* `ComplexityClass.below` of a problem P, the problems that reduce to P, is itself a class, closed under reductions by construction, so completeness becomes statable with no logic anywhere. Any class with a complete problem is the degree of that problem, and where both apply the two agree, as in `NP_eq_below_sat` and `PTIME_eq_below_hornSat`: SAT-hardness *is* NP-hardness as a theorem.

4.3 Relations between the classes

The library proves several structural results about the classes of Table 1. Many inclusions follow directly from the logical definitions. For example, NL contains LOGSPACE because a deterministic transitive closure is a transitive closure (`LOGSPACE_subset_NL`); NP and coNP sit inside DP because an ∃SO or ∀SO condition is a conjunction of both with a tautology (`NP_subset_DP`, `coNP_subset_DP`); DP sits inside each level of the polynomial hierarchy by the same argument (`DP_subset_sigmaP_two`, `DP_subset_piP_two`). For NL ⊆ PTIME we also use reductions and hardness: 2-SAT, a complete problem for NL, is defined by a Horn second-order sentence and thus belongs to PTIME, implying the inclusion of the whole class as the degree of 2-SAT. The polynomial hierarchy is handled uniformly in k: four step lemmas $\Sigma^{\mathrm{p}}_k \subseteq \Sigma^{\mathrm{p}}_{k+1}$, $\Sigma^{\mathrm{p}}_k \subseteq \Pi^{\mathrm{p}}_{k+1}$, $\Pi^{\mathrm{p}}_k \subseteq \Sigma^{\mathrm{p}}_{k+1}$, $\Pi^{\mathrm{p}}_k \subseteq \Pi^{\mathrm{p}}_{k+1}$, each level included into PH. Likewise `QBF_complete` gives completeness for every level in one statement, after Wrathall [59].

Capture theorems. We prove Grädel's equivalence [26]: the Horn fragment of ∃SO coincides with FO(LFP) on ordered structures, because the Horn rules can be evaluated by iterating a least fixed point, and conversely any LFP definition can be written as a Horn program by making the iteration explicit in the rules (`lfpDefinable_iff_sigmaSOHornDefinable`). Read against the definition of PTIME as SO-Horn(≤), this is the Immerman–Vardi theorem [28, 58], that FO(IFP) over ordered structures captures PTIME (`lfpDefinable_iff_mem_PTIME`). The latter uses the equivalence FO(IFP) = FO(LFP) for ordered structures, where the least and inflationary fixed points coincide: the order lets us detect convergence (`ifpDefinable_iff_lfpDefinable`). The partial fixed point captures PSPACE: FO(PFP) over ordered structures equals PSPACE, proved by showing FO(PFP) = SO(TC) (`pfpDefinable_iff_sotcDefinable`). These theorems connect the fragment-based definitions (SO-Horn for PTIME, SO(TC) for PSPACE) with the operator-based ones (FO(IFP) for PTIME, FO(PFP) for PSPACE): `ifpDefinable_iff_mem_PTIME`, `pfpDefinable_iff_mem_PSPACE`.

Closure under complementation. The class NL is closed under complementation by the Immerman–Szelepcsényi theorem [30, 51]. In the library, it is proved for FO(TC) via inductive counting (`TCDefinable.compl`), and the Krom fragment inherits this through the translation to FO(TC) (`sigmaSOKromDefinable_compl_iff`); at the level of classes this is `NL_eq_coNL`. Then, PTIME is closed because FO(LFP) is closed under negation (`LFPDefinable.compl`); and PSPACE is closed because every SO(TC) problem reduces to QSAT [50], flipping the answer of which gives the complement within SO(TC) (`PSPACE_eq_coPSPACE`). Further, EXPTIME and EXPSPACE inherit closure from PTIME and PSPACE via the exponential operator $C \mapsto C^{\exp}$ (`ComplexityClass.exp`), which commutes with complement (`ComplexityClass.exp_compl`).

Abiteboul–Vianu theorem. This result is of a different nature, an equivalence with an open question on either side: FO(IFP) and FO(PFP) coincide over *unordered* finite structures exactly when PTIME = PSPACE [3, 4], `ifpDefinableFree_eq_pfpDefinableFree_iff_ptime_eq_pspace`. We follow the purely logical route of Dawar et al. [15], whose invariant structure and its definable order replace the relational machine of the original proof.

4.4 Unconditional lower bounds

Ehrenfeucht–Fraïssé games [20] are the tool the machine-first developments have no analogue of: they argue about what a *logic* cannot say. The witness throughout is EVEN, whether a finite set has an even number of elements. It is not first-order definable, even order-invariantly (`even_not_foDefinable`).

It is one walk along an order, though, so $\mathrm{FO}(\leq) \subsetneq \mathrm{FO}(\leq, \mathrm{TC})$ outright (`exists_tcDefinable_not_foDefinable`); and arithmetic defines it too, which separates $\mathrm{FO}(\leq)$ from AC^0, read here as the logic $\mathrm{FO}(\leq, +, \times)$ and not as a circuit class (`exists_ac0Definable_not_foDefinable`).

The k-pebble game carries EVEN to the fixed-point logics [19, ch. 7], implying that order-free FO(IFP) does not capture PTIME (`exists_mem_PTIME_not_ifpDefinableFree`). The $\leq$ in every capture theorem is doing work.

5 Machine Bridges

The *machine bridges* link the logically defined classes to the standard definitions by Turing machine. For each class we define, in the library's vocabulary, the canonical *acceptance problem*; for NP an instance describes a nondeterministic machine together with its input, and the question is whether it accepts within as many steps as the instance has elements. We then prove that NP, defined by ∃SO-definability, coincides with the degree of that problem (`mem_NP_iff_le_ntmAccept`). No machine model is declared as a Lean type: the machine is *data in the instance*.

5.1 Acceptance as a decision problem

The vocabulary `FirstOrder.Language.turing` encodes a computation as a finite structure. Two unary symbols sort the universe, the *positions* – tape cells and time steps at once – and the *transitions*; the remaining symbols mark a start state, the accepting states, the blank and the rightward transitions, order the positions into a tape by `le`, give their initial contents, and tie each transition to the states it runs between and the symbols it reads and writes. States and tape symbols get no sort of their own: no part of the definition quantifies over them.

`NTMAccept` then asks whether the machine an instance describes accepts within as many steps as there are positions. It holds when the structure is well-formed – the tape is ordered linearly, there is at least one position, the input is functional, the blank unique – and some run reaches an accepting configuration in time. `DTMAccept` adds a determinism guard; the space-bounded `NTMAcceptSpace` and `DTMAcceptSpace` drop the step bound, acceptance becoming reachability in the configuration graph. Since positions serve as time steps, the budget is a count of universe elements and the vocabulary needs no arithmetic: a d-dimensional reduction (§3.3) builds n^d of them, which is where the polynomial bound comes from.

5.2 Recovering the machine classes

Hardness is the easy half: SAT is complete for NP, so it suffices to reduce it to `NTMAccept`, building a machine that guesses an assignment and checks clauses, and PTIME enters the degree of `DTMAccept` the same way from HORN-SAT. For membership, `NTMAccept` is in NP by writing the run as an ∃SO sentence: one existential block guesses the state, head position and tape contents at each step, and a first-order kernel checks that they form a valid accepting run. Determinism makes the run of `DTMAccept` unique, so it is derivable by a Horn program and the problem lands in PTIME. Each direction closes into a biconditional: `mem_NP_iff_le_ntmAccept`, `mem_PTIME_iff_le_dtmAccept`. This is how Fagin's theorem is proved. Since NP is *defined* as ∃SO, it is not an equality of classes; it is instead the fact that ∃SO is exactly the degree of nondeterministic polynomial-time acceptance.

Below PTIME the bridge is a capture rather than a complete problem. A logarithmic-space machine has polynomially many configurations, each a tuple over the instance, so its configuration graph is definable on the instance and acceptance *is* reachability, leaving nothing to complete. LOGSPACE and NL are exactly what two-way multi-head automata recognize (`mem_LOGSPACE_iff_automaton`, `mem_NL_iff_automaton`), and AC^0 what constant-alternation logarithmic time does (`ac0Definable_iff_ltDecidable`).

Above PTIME the run outgrows the instance. The bridge there keeps the complete problem of NP and PTIME. The space-bounded `NTMAcceptSpace` and `DTMAcceptSpace` are complete for PSPACE (`spaceMachines_PSPACE_complete`), and EXPTIME is reached by the space-bounded alternating machines of Chandra et al. [11] (`atmAcceptSpace_EXPTIME_complete`). NEXPTIME and EXPSPACE need *wide machines*, whose universe is the power set of the instance together with the instance itself, so that tape and clock run to 2^n (`wideRegAccept_NEXPTIME_complete`, `wideAcceptSpace_EXPSPACE_complete`). Appendix C details each level.

For RE no machine appears on either side. An $\exists\mathrm{SO}_{\mathrm{new}}$ problem is semi-decidable because its invented values can be found by unbounded search (`RE_subset_rePred`); conversely a semi-decidable problem reduces to the RE-complete halting problem. Together they give `mem_RE_iff_rePred`, in Mathlib's own `REPred` sense.

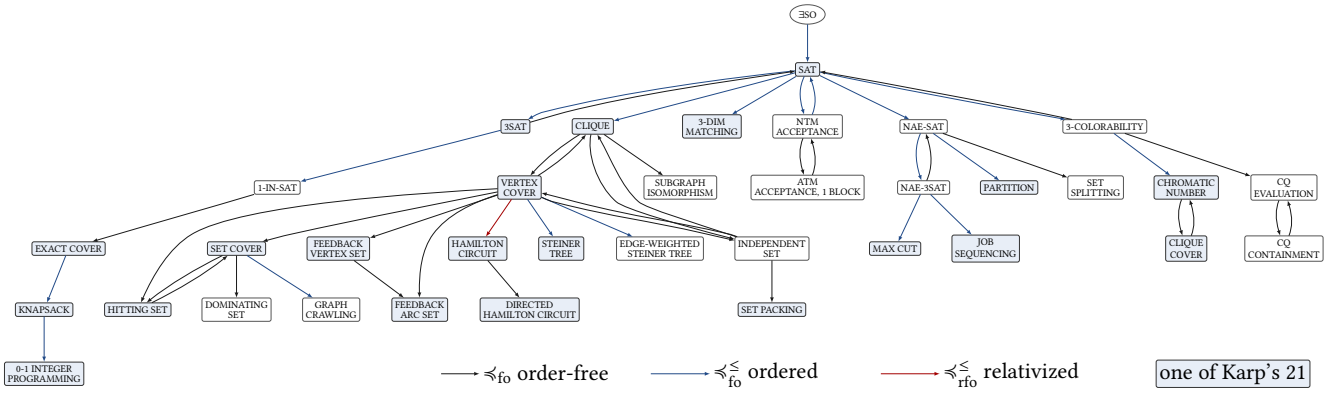

Figure 5. Problems the library proves NP-complete, and reductions between them. Karp's 21 are highlighted. An arrow between problems is a proved first-order reduction, colored by the form of the reduction. The graph is not strongly connected only because *membership* in NP is proved through ∃SO definability rather than through a reduction.

5.3 Cook–Levin in its textbook form

A reduction from `NTMAccept` to SAT is the classical Cook–Levin theorem [13, 39]: encode the run as an ∃SO sentence, then introduce a fresh variable per subformula with clauses enforcing the equivalence (Tseitin construction [57]). That is `ntmAccept_interreducible_sat`, both reductions in one statement. `SAT_complete_for_ntmAccept` is the same content with hardness quantified over the class, i.e., SAT complete for the NP the *machine* defines. This is what the developments compared in §7 prove.

6 A Catalog of Complete Problems

The library proves 73 completeness theorems across 14 classes, counted by class in Appendix D. Most are NP or coNP; the rest follow a few patterns, and one problem fits no logically defined class.

Adding a problem costs little once the framework is there. Setting aside the machine models, whose modules Appendix C costs instead, the median problem of the catalog is 869 lines of its own modules, with quartiles 521 and 1 192 (Appendix E costs them one by one). Those are source lines of what a problem adds, since we are measuring the cost of adding another problem onto a base library.

Part of that cost is notation. Mathlib's `BoundedFormula` is locally nameless: a variable bound by the k-th enclosing quantifier block is `Sum.inr i` under $k-1$ applications of `Sum.inl`, so writing a clause means counting blocks, and reading one means counting them back. The library's `fo%` syntax, which Appendix F uses throughout, names the variables and applies a symbol to them as `R(x, y)`. It is a macro, not a new type of formulas: it elaborates to exactly the term one would have written, so a definition converted to it keeps its equation lemma, and every `simp only` proof about it, verbatim. Two commands, `fo_language` and `fo_block`, generate the index types, summed vocabularies and per-symbol abbreviations that a vocabulary or a second-order block needs, all of it determined by the arities. Converting the example of Appendix F took its problem module from 314 to 275 lines, the interpretation losing half of its own.

6.1 NP- and coNP-complete problems

Figure 5 is the NP part of the catalog, read out of the elaborated library: every node is a problem proved NP-complete and every arrow between problems a proved first-order reduction. Its core is Karp's 21 problems [35], highlighted in the figure and all proved complete here, SAT [13, 39] among them.

Hardness enters the graph once, at the arrow out of ∃SO. That arrow is the hardness half of Cook–Levin, proved without a machine: `sat_hard_of_sigmaSODefinable` sends every problem definable in Σ^1_1 to SAT – by Fagin's theorem, every problem in NP – and every other problem inherits hardness by composing the reductions along a path from SAT to it, with no second-order argument of its own. Membership is proved separately, and cheaply, by a definability witness per problem; that is why the graph is not strongly connected.

Fourteen problems beside Karp's list round the catalog out. Five are required as intermediate problems; two are the machine problems of §5, NTM-ACCEPT and one-block ATM-ACCEPT, on the alternating machines of Chandra et al. [11], interreducible and complete for NP; four are common starting points for hardness proofs elsewhere (SUBGRAPH ISOMORPHISM, DOMINATING SET, SET SPLITTING, EDGE-WEIGHTED STEINER TREE); finally, the last three (CQ EVALUATION and CQ CONTAINMENT [12], and GRAPH CRAWLING) are domain-specific problems the library ships as tutorials. Appendix F walks a smaller problem, SUBGRAPH ISOMORPHISM, through every step of its completeness proof.

Most arrows are the order-free $\preccurlyeq_{\mathrm{fo}}$: tags do the work a textbook first-order reduction gives to the order (§3.2), so a gadget that only builds a fixed number of copies of its input never mentions one. The other forms of reduction are used when required. First, a problem involving numbers (e.g.,

Table 2. Mechanized complexity theory, by what the complexity classes are defined over. *Thms* counts completeness theorems on concrete problems. * `poly-reductions` proves reductions, not completeness: the NP-hardness at the root of its tree rests on an admitted lemma (§7).

Library	Prover	Defined over	Classes	Completeness for	Thms
`coq-library-complexity` [25]	Rocq	call-by-value λ-calculus	PTIME, NP	NP	3
AFP `Cook_Levin` [8]	Isabelle/HOL	multi-tape Turing machines	DTIME, PTIME, NP	NP	1
`poly-reductions` [44]	Isabelle/HOL	IMP-while-language [34]	PTIME, NP	none*	0
`Complexitylib` [47]	Lean	multi-tape Turing machines	LOGSPACE, NL, PTIME, NP, coNP, PSPACE, EXPTIME, NEXPTIME, and 21 others	NP, coNP	3
Our library	Lean	logic over finite structures	LOGSPACE, NL, PTIME, NP, coNP, DP, Σ^{p}_k, Π^{p}_k, PSPACE, EXPTIME, NEXPTIME, EXPSPACE, RE, GI degree	all	73

weights) needs its gadget to build them, and a number is a positional encoding over the universe, so the bit blocks are laid along the order. This is why KNAPSACK and PARTITION are reached by ordered reductions $\preccurlyeq^{\leq}_{\mathrm{fo}}$. Second, spanning problems, such as HAMILTON CIRCUIT, must visit every vertex, so the meaningless tuples that other reductions can leave isolated in the output universe would have to be visited too. Only VERTEX COVER to HAMILTON CIRCUIT therefore needs the relativized $\preccurlyeq^{\leq}_{\mathrm{rfo}}$, whose domain formula cuts the universe down to the real gadget.

The coNP side relies on taking the complement, which yields completeness results at little cost. Since coNP is $\forall$SO, a Π^1_1-definable problem has a Σ^1_1-definable complement, which that same theorem sends to SAT; complementing that reduction yields a reduction to TAUT, whence `taut_hard_of_piSODefinable` and `TAUT_coNP_complete`.

6.2 The rest of the catalog

Problems shown complete for the remaining classes of Table 1 usually fit one of four patterns. (i) *The defining logic's own construct, as a problem.* Hardness translates that logic into the problem's vocabulary: HORN-SAT [18] for PTIME, which is SO-Horn, and DET-REACH for LOGSPACE, which is FO($\leq$, DTC). (ii) *The complement of a complete problem*, as TAUT was in §6.1. What that costs varies. It is free where the defining logic is closed under complement, so QSAT [50], complete for PSPACE, is complete for coPSPACE as well; whereas UNREACH is NL-complete only through NL = coNL. (iii) *A succinct reading of a problem one exponent down.* Tiling the $n \times n$ square is in NP, and tiling the corridor of width n is in PSPACE. Read the instance's elements as addresses, not as positions and the same two problems, over a grid of side 2^n, are NEXPTIME-complete and EXPSPACE-complete. (iv) *A classical complete problem*, reached by reduction from the problem of (i): CVP for PTIME [37] and REACH for NL [33], each complete here under order-invariant first-order reductions, strengthening the classical logarithmic-space statements.

6.3 The GI degree

The graph isomorphism problem, GRAPH-ISO, fits none of the logically defined classes of Table 1. It is in NP, not known to be NP-complete and not known to be in PTIME, and it has no logical characterization to be complete for. A notion of completeness tied to a logic would have no place for it at all.

The degree construction of §4.2 gives it one anyway, since the downward closure of a fixed problem is itself a class. Three problems are proved complete for the degree of GRAPH-ISO: `graphIso_GI_complete`, `digraphIso_GI_complete` and `dagIso_GI_complete`. Undirected isomorphism is the anchor, that being what the literature calls GI, and the other two are interreducible with it.

Reductions between isomorphism problems are unlike the rest of the catalog, and the digraph-to-graph gadget runs to some seven hundred and fifty lines for that reason: a reduction that *translates* an instance is cheap, its correctness following the construction, whereas one that must *reflect* isomorphism has to rule out every accidental automorphism of the output, one argument for each way one could arise.

7 Related Work

Lean's general-purpose libraries stop short of complexity classes. Mathlib's `Computability` tree offers polynomial-time computability of functions and the many-one reducibility of computability theory; the official Lean computer science library [9] adds Turing machines with `TimeComputable`

Table 3. Lines strictly necessary to prove the Cook–Levin theorem (NP-completeness of SAT under each library's machine model). *Closure* counts the declarations the proof reaches; *In full* counts the same modules in full; *External libraries* is what the closure reaches in the libraries the development imports.

Library	Theorem measured	Modules	Closure	In full	External libraries
`coq-library-complexity` [25]	`CookLevin`	72	19 658	24 322	25 762 (undec. lib. + Coq stdlib)
AFP `Cook_Levin` [8]	`NP_complete_SAT`	18	43 946	50 509	16 838 (HOL only)
`Complexitylib` [47]	`NPComplete_language`	56	29 693	38 017	25 796 (Mathlib, core)
Our library	`SAT_complete_for_ntmAccept`	32	6 262	9 236	14 996 (Mathlib, core)

and `PolyTimeComputable` predicates. Neither defines a complexity class, a reduction carrying a resource bound, or a completeness result.

Four developments do define complexity classes, as recorded in Table 2. Gäher and Kunze [25] work over the call-by-value λ-calculus, their three completeness theorems all in the module proving Cook–Levin; Balbach [8] works over multi-tape Turing machines, Cook–Levin being the completeness theorem proved. `poly-reductions` [44] proves reductions rather than completeness: 26 problems, all 21 of Karp's among them, form a tree rooted at SAT, whose NP-hardness is proved only inside an Isabelle locale assuming the step from a polynomial-time verifier to a reduction to SAT, a lemma left admitted. `Complexitylib` [47], back over multi-tape machines, has the widest inventory: 29 classes at the revision we cite, with completeness for NP and coNP on three concrete problems, and, beside them, first- and second-order syntax over finite structures with one-dimensional first-order reductions, Fagin's theorem being open on its roadmap and no class defined through them. Machine models have a clock in return, so `Complexitylib` proves the deterministic time hierarchy theorem and circuit lower bounds, and Forster et al. [23] verify a universal machine and a multi-tape to single-tape compiler with polynomial overhead, results our library cannot state.

These four are the only developments we could find; others advertising machine-checked complexity results were excluded once we confirmed their headline theorems admitted or vacuous, reducing to `True` or resting on classes that are opaque axioms.

Cook–Levin is the one theorem all but `poly-reductions` prove, and Table 3 measures, for each, the lines of code *strictly necessary* to prove it: the dependency closure of the theorem, not of the files holding it. Every row measures the machine form of the theorem, SAT complete for the NP that the development's own machine model defines under a polynomial time bound. We report each closure split into the development's own modules and the external libraries beneath them, a small count resting on a large library being a different achievement, and give the same modules counted in full beside it. How each closure is computed is Appendix G.

We checked what each theorem relies on with each prover's own facility: the Rocq row depends on no axiom, the Isabelle row on no oracle, and the two Lean rows on the three standard axioms `propext`, `Classical.choice`, and `Quot.sound`; no row is admitted. Over the whole libraries rather than these closures, ours declares no axiom of its own and contains no `sorry`.

The closures differ by a factor of about seven on the developments' own side, from 6 262 lines to 43 946; the two Lean rows compare most directly, using the same prover and the same libraries beneath, at 29 693 lines against 6 262. Line counts across four provers are a weak measure at best: the scopes differ, the inputs differ – binary strings in the Isabelle and `Complexitylib` rows, encoded λ-terms in the Rocq one, finite structures here (§8.2) – and so do the libraries beneath them. Read as an order of magnitude, though, they say something: ours is the smallest of the four, and that with the whole machine bridge of §5 inside the closure (Appendix C costs it class by class). That is evidence that descriptive complexity is a useful route to complexity results.

The Rocq library of undecidable problems [24] fixes one notion of reduction, grows a catalog around it, and takes contributions from others, which is the shape of our library one level down, with first-order reductions in place of many-one ones and completeness in place of undecidability; Grange et al. [27] specify reductions as graphical recipes equivalent to quantifier-free first-order interpretations, so that candidate reductions can be validated automatically. On the model-theoretic side, Kirst and Larchey-Wendling [36] prove Trakhtenbrot's theorem in Rocq, FINSAT being the problem closest to it in our catalog; first-order logic over finite relations has been mechanized for query evaluation [45], and the least-fixpoint semantics of Datalog, the iteration behind Grädel's theorem in §4.3, more than once [10, 52]; and the logic–automata connection has verified decision procedures for WS1S [17, 56]. We know of no prior mechanization of Ehrenfeucht–Fraïssé games, of the fixpoint and transitive-closure logics, or of the capture theorems of §4.

8 Design Decisions and Limitations

Two decisions shape the library: hardness is proved by first-order reduction, and an instance is a finite structure. We take each in turn, then the questions the approach cannot reach.

8.1 First-order vs. Karp reductions

Hardness is classically defined by Karp reductions, that is, many-one reductions computable in polynomial time, or by their logarithmic-space refinement (§2.3). Ours are first-order, and every first-order reduction is both, so hardness proved under $\preccurlyeq_{\text{fo}}$ transfers verbatim to the classical notion.

The converse fails already against logarithmic space, and the library proves it (`exists_dtcReduction_not_orderedReduction`). In FO($\leq$, DTC), EVEN of §4.4 reduces to NONEMPTY-MARK, whether an element is marked: mark everything when the universe is even, nothing when it is odd, one walk along the order. No first-order reduction does so, the target being first-order definable and EVEN not. That walk never has a choice, so this is the deterministic notion, the many-one reduction of the textbooks: completeness under the weaker reduction is the stronger statement.

In practice the restriction has cost the library nothing. Every reduction in the catalog is FO($\leq$)-definable, across all 73 completeness results and 14 classes (§6); no problem had to be dropped, and none needed a weaker statement, because its textbook reduction would not fit in the logic. Where the textbook route does resist, another is available. Murray and Williams [41] observe that reductions with local structure suffice for almost all completeness results, and name one potential counterexample: the usual reduction from SUBSET-SUM to PARTITION outputs two numbers that require summing every weight in the instance, so it does not seem computable even by AC^0 circuits of size $2^{n^{o(1)}}$. The library reaches PARTITION by an ordered first-order reduction from NAE-SAT [46] instead, and a recent note [32] gives first-order projections.

Such reductions are usually first-order *projections* [29], refining those of Skyum and Valiant [48]: each bit of the output is a constant, or one bit of the input, or its negation, independently of the input. Natural complete problems seem to stay complete under them [6], although an artificial set complete for NP under $\mathsf{AC}^0[\text{mod } 2]$ reductions but not under AC^0 ones exists [5]; what our catalog adds is machine-checked evidence for the problems one actually meets.

8.2 Finite structures vs. strings

A machine model is classically defined on strings, with its resource bounds given as explicit functions of the input length. Here an instance is a finite structure, a machine is data inside it, and the budget is a count of universe elements, the polynomial coming from the dimension of the reduction that built the instance (§5.1); no arithmetic appears in the vocabulary.

The two conventions differ most in how order enters. Strings come ordered and structures do not, so order-invariance is an explicit variant here, $\preccurlyeq_{\text{fo}}^{\leq}$, and theorems whose whole content is the *absence* of an order become statable: Abiteboul–Vianu (§4.3), and the failure of order-free FO(IFP) to capture PTIME (§4.4), which is why the $\leq$ carried by the fragments of Table 1 is necessary. Neither convention dominates, though: the string one makes cost functions and hierarchy theorems natural, and those are exactly what we give up (§8.3).

The two meet at the top of the catalog. Finite structures over a finite vocabulary carry a numbering, so a problem denotes a set of codes, and `mem_RE_iff_rePred` proves $\exists\text{SO}_{\text{new}}$-definability to coincide with Mathlib's own `REPred` on that set. The witness is CODEHALT, where a partial recursive code *is* an instance, its syntax tree flattened, so nothing is simulated. Below RE the agreement with the usual presentations over string encodings is classical [22, 31] and is not formalized here.

The encodings that carry a result back to concrete data (§3.4) raise the same question. They require polynomial size bounds in both directions and a faithfulness proof, but the encoding itself may be any computation, so a completeness statement transported through one inherits a cost that is never measured. A machine model faces the same step: its inputs are strings, so a graph or a formula has to be written out as one before the machine runs, and that cost goes unmeasured too.

8.3 What the descriptive approach cannot do

There is no cost model here, hence no $O(\cdot)$, no fine-grained complexity and no Δ_k^p. It is therefore impossible to prove a hierarchy theorem: separating $\mathsf{DTIME}(f)$ from $\mathsf{DTIME}(2^f)$ is a statement about a clock, and `Complexitylib` proves the deterministic time hierarchy theorem because it carries one (§7), whereas here there is nothing to diagonalize against.

9 Conclusion

Taking a decision problem to be an isomorphism-invariant property of finite structures, a complexity class to be a definability predicate, and hardness to be a first-order reduction, we obtain a library in which membership and completeness are logical statements, the relations between classes are proved inside the logic, and the machine models are theorems. New results come cheaply on that footing: membership is a sentence, hardness a reduction, completeness the pair of them, and a problem joins the catalog for a median of 869 lines of its own modules – which is how 73 completeness theorems across 14 classes were reached.

Three extensions look worthwhile. Locality, in Hanf's and Gaifman's forms [40], would give the inexpressibility results that now need a strategy built by hand. Counting fits the same idiom: the quantitative second-order logic of Arenas et al. [7] layers arithmetic over an unchanged Boolean one, so that #P arrives as $\Sigma\text{QSO}(\text{FO})$ with #SAT complete for it. And tracking quantifier-freeness through composition would carry the catalog to the projections of §8.1.

The library is available at https://github.com/PierreSenellart/descriptive-complexity under the Apache 2.0 license; the numbers reported here were measured at release 1.2.2.

Acknowledgments

This work was funded in part by the French government under management of Agence Nationale de la Recherche (ANR) as part of the "France 2030" program, reference ANR-23-IACL-0008 (PR[AI]RIE-PSAI). It is also part of the program DesCartes and is supported by the National Research Foundation, Prime Minister's Office, Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) program. We are grateful to Dan Suciu who suggested FO reductions as the right tool to formalize reductions in Lean.

The Lean code of the library, including most proofs and docstrings, was written with the assistance of generative models from Anthropic (Claude); the formalization choices, the library architecture, the main definitions and theorems, and their alignment with the standard ones were designed and written or reviewed by the authors, who take responsibility for the whole.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley.

[2] Serge Abiteboul, Moshe Y. Vardi, and Victor Vianu. 1997. Fixpoint logics, relational machines, and computational complexity. *Journal of the ACM* 44, 1 (1997), 30–56. doi:10.1145/256292.256295

[3] Serge Abiteboul and Victor Vianu. 1991. Generic Computation and Its Complexity. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing (STOC)*. ACM, 209–219. doi:10.1145/103418.103444

[4] Serge Abiteboul and Victor Vianu. 1995. Computing with First-Order Logic. *J. Comput. Syst. Sci.* 50, 2 (1995), 309–335. doi:10.1006/jcss.1995.1025

[5] Manindra Agrawal, Eric Allender, Russell Impagliazzo, Toniann Pitassi, and Steven Rudich. 1997. Reducing the Complexity of Reductions. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC)*. ACM, 730–738. doi:10.1145/258533.258671

[6] Eric Allender, José L. Balcázar, and Neil Immerman. 1997. A First-Order Isomorphism Theorem. *SIAM J. Comput.* 26, 2 (1997), 557–567. doi:10.1137/S0097539794270236

[7] Marcelo Arenas, Martín Muñoz, and Cristian Riveros. 2020. Descriptive Complexity for Counting Complexity Classes. *Log. Methods Comput. Sci.* 16, 1 (2020). doi:10.23638/LMCS-16(1:9)2020

[8] Frank J. Balbach. 2023. The Cook-Levin theorem. *Archive of Formal Proofs* (January 2023). https://isa-afp.org/entries/Cook_Levin.html Formal proof development.

[9] Clark Barrett, Swarat Chaudhuri, Fabrizio Montesi, Jim Grundy, Pushmeet Kohli, Leonardo de Moura, Alexandre Rademaker, and Sorrachai Yingchareonthawornchai. 2026. CSLib: The Lean Computer Science Library. https://github.com/leanprover/cslib. arXiv:2602.04846 [cs.LO] Release v4.33.0, 2026-08-10.

[10] Véronique Benzaken, Évelyne Contejean, and Stefania Dumbrava. 2017. Certifying Standard and Stratified Datalog Inference Engines in SSReflect. In *8th International Conference on Interactive Theorem Proving, ITP 2017 (Lecture Notes in Computer Science, Vol. 10499)*. Springer, 171–188. doi:10.1007/978-3-319-66107-0_12

[11] Ashok K. Chandra, Dexter Kozen, and Larry J. Stockmeyer. 1981. Alternation. *J. ACM* 28, 1 (1981), 114–133. doi:10.1145/322234.322243

[12] Ashok K. Chandra and Philip M. Merlin. 1977. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*, John E. Hopcroft, Emily P. Friedman, and Michael A. Harrison (Eds.). ACM, 77–90. doi:10.1145/800105.803397

[13] Stephen A. Cook. 1971. The Complexity of Theorem-Proving Procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman (Eds.). ACM, 151–158. doi:10.1145/800157.805047

[14] Elias Dahlhaus. 1983. Reduction to NP-complete problems by interpretations. In *Logic and Machines: Decision Problems and Complexity (Lecture Notes in Computer Science, Vol. 171)*, Egon Börger, Gisbert Hasenjaeger, and Dieter Rödding (Eds.). Springer, 357–365. doi:10.1007/3-540-13331-3_51

[15] Anuj Dawar, Steven Lindell, and Scott Weinstein. 1995. Infinitary Logic and Inductive Definability over Finite Structures. *Inf. Comput.* 119, 2 (1995), 160–175. doi:10.1006/INCO.1995.1084

[16] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12699)*, André Platzer and Geoff Sutcliffe (Eds.). Springer, 625–635. doi:10.1007/978-3-030-79876-5_37

[17] Christian Doczkal and Gert Smolka. 2018. Regular Language Representations in the Constructive Type Theory of Coq. *Journal of Automated Reasoning* 61, 1–4 (2018), 521–553. doi:10.1007/s10817-018-9460-x

[18] William F. Dowling and Jean H. Gallier. 1984. Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae. *J. Log. Program.* 1, 3 (1984), 267–284. doi:10.1016/0743-1066(84)90014-1

[19] Heinz-Dieter Ebbinghaus and Jörg Flum. 1995. *Finite Model Theory*. Springer.

[20] Andrzej Ehrenfeucht. 1961. An application of games to the completeness problem for formalized theories. *Fundamenta Mathematicae* 49 (1961), 129–141. doi:10.4064/fm-49-2-129-141

[21] Ronald Fagin. 1973. *Contributions to the Model Theory of Finite Structures*. Ph. D. Dissertation. University of California, Berkeley.

[22] Ronald Fagin. 1974. Generalized first-order spectra and polynomial-time recognizable sets. In *Complexity of Computation*, Richard M. Karp (Ed.). SIAM-AMS Proceedings, Vol. 7. American Mathematical Society, 43–73.

[23] Yannick Forster, Fabian Kunze, and Maximilian Wuttke. 2020. Verified Programming of Turing Machines in Coq. In *9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20–21, 2020*. ACM, 114–128. doi:10.1145/3372885.3373816

[24] Yannick Forster, Dominique Larchey-Wendling, Andrej Dudenhefner, Edith Heiter, Dominik Kirst, Fabian Kunze, Gert Smolka, Simon Spies, Dominik Wehr, and Maximilian Wuttke. 2020. A Coq Library of Undecidable Problems. In *The Sixth International Workshop on Coq for Programming Languages (CoqPL 2020)*. https://github.com/uds-psl/coq-library-undecidability

[25] Lennard Gäher and Fabian Kunze. 2021. Mechanising Complexity Theory: The Cook-Levin Theorem in Coq. In *12th International Conference on Interactive Theorem Proving, ITP 2021, Rome, Italy (Virtual Conference), June 29 - July 1, 2021 (LIPIcs, Vol. 193)*, Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 20:1–20:18. doi:10.4230/LIPIcs.ITP.2021.20

[26] Erich Grädel. 1992. Capturing Complexity Classes by Fragments of Second-Order Logic. *Theor. Comput. Sci.* 101, 1 (1992), 35–57. doi:10.1016/0304-3975(92)90149-A

[27] Julien Grange, Fabian Vehlken, Nils Vortmeier, and Thomas Zeume. 2024. Specification and Automatic Verification of Computational Reductions. In *49th International Symposium on Mathematical Foundations of Computer Science, MFCS 2024, Bratislava, Slovakia, August 26-30, 2024 (LIPIcs, Vol. 306)*, Rastislav Královic and Antonín Kucera (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 56:1–56:14. doi:10.4230/LIPIcs.MFCS.2024.56

[28] Neil Immerman. 1986. Relational Queries Computable in Polynomial Time. *Inf. Control.* 68, 1-3 (1986), 86–104. doi:10.1016/S0019-9958(86)80029-8

[29] Neil Immerman. 1987. Languages that Capture Complexity Classes. *SIAM J. Comput.* 16, 4 (1987), 760–778. doi:10.1137/0216051

[30] Neil Immerman. 1988. Nondeterministic Space is Closed Under Complementation. *SIAM J. Comput.* 17, 5 (1988), 935–938. doi:10.1137/0217058

[31] Neil Immerman. 1999. *Descriptive Complexity*. Springer. doi:10.1007/978-1-4612-0539-5

[32] Paúl Risco Iturralde. 2025. A Note on the NP-Hardness of PARTITION Via First-Order Projections. arXiv:2512.21448 [cs.CC] Preprint.

[33] Neil D. Jones. 1975. Space-Bounded Reducibility among Combinatorial Problems. *J. Comput. Syst. Sci.* 11, 1 (1975), 68–85. doi:10.1016/S0022-0000(75)80050-X

[34] Kevin Kappelmann, Fabian Huch, Lukas Stevens, and Mohammad Abdulaziz. 2025. Proof-Producing Translation of Functional Programs into a Time & Space Reasonable Model. arXiv:2503.02975 [cs.LO]

[35] Richard M. Karp. 1972. Reducibility Among Combinatorial Problems. In *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, Raymond E. Miller and James W. Thatcher (Eds.). Plenum Press, New York, 85–103. doi:10.1007/978-1-4684-2001-2_9

[36] Dominik Kirst and Dominique Larchey-Wendling. 2022. Trakhtenbrot's Theorem in Coq: Finite Model Theory through the Constructive Lens. *Logical Methods in Computer Science* 18, 2 (2022). doi:10.46298/lmcs-18(2:17)2022

[37] Richard E. Ladner. 1975. The Circuit Value Problem is Log Space Complete for P. *SIGACT News* 7, 1 (1975), 18–20. doi:10.1145/990518.990519

[38] Daniel Leivant. 1989. Descriptive Characterizations of Computational Complexity. *J. Comput. Syst. Sci.* 39, 1 (1989), 51–83. doi:10.1016/0022-0000(89)90019-6

[39] Leonid A. Levin. 1973. Universal sequential search problems. *Problems of Information Transmission* 9, 3 (1973), 265–266. Russian original: Problemy Peredachi Informatsii 9(3):115–116.

[40] Leonid Libkin. 2004. *Elements of Finite Model Theory*. Springer. doi:10.1007/978-3-662-07003-1

[41] Cody D. Murray and R. Ryan Williams. 2017. On the (Non) NP-Hardness of Computing Circuit Complexity. *Theory Comput.* 13, 1 (2017), 1–22. doi:10.4086/toc.2017.v013a004

[42] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science, Vol. 2283. Springer. doi:10.1007/3-540-45949-9

[43] Christos H. Papadimitriou and Mihalis Yannakakis. 1984. The Complexity of Facets (and Some Facets of Complexity). *J. Comput. Syst. Sci.* 28, 2 (1984), 244–259. doi:10.1016/0022-0000(84)90068-0

[44] Poly-Reductions Contributors. 2026. Automatic Refinements and Polynomial-Time Reductions in Isabelle/HOL. https://github.com/rosskopfs/poly-reductions. Commit d741d95, 2026-07-01. Successor to https://github.com/wimmers/poly-reductions.

[45] Martin Raszyk. 2022. First-Order Query Evaluation. *Archive of Formal Proofs* (February 2022). https://isa-afp.org/entries/Eval_FO.html Formal proof development.

[46] Thomas J. Schaefer. 1978. The Complexity of Satisfiability Problems. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing, May 1-3, 1978, San Diego, California, USA*, Richard J. Lipton, Walter A. Burkhard, Walter J. Savitch, Emily P. Friedman, and Alfred V. Aho (Eds.). ACM, 216–226. doi:10.1145/800133.804350

[47] Samuel Schlesinger. 2026. Complexitylib: A Lean 4 Formalization of Computational Complexity Theory. https://github.com/SamuelSchlesinger/complexitylib. Commit b673821, 2026-08-19.

[48] Sven Skyum and Leslie G. Valiant. 1985. A Complexity Theory Based on Boolean Algebra. *J. ACM* 32, 2 (1985), 484–502. doi:10.1145/3149.3158

[49] Larry J. Stockmeyer. 1976. The Polynomial-Time Hierarchy. *Theor. Comput. Sci.* 3, 1 (1976), 1–22. doi:10.1016/0304-3975(76)90061-X

[50] Larry J. Stockmeyer and Albert R. Meyer. 1973. Word Problems Requiring Exponential Time: Preliminary Report. In *Proceedings of the 5th Annual ACM Symposium on Theory of Computing (STOC)*. 1–9. doi:10.1145/800125.804029

[51] Róbert Szelepcsényi. 1988. The Method of Forced Enumeration for Nondeterministic Automata. *Acta Informatica* 26, 3 (1988), 279–284. doi:10.1007/BF00299636

[52] Johannes Tantow, Lukas Gerlach, Stephan Mennicke, and Markus Krötzsch. 2025. Verifying Datalog Reasoning with Lean. In *16th International Conference on Interactive Theorem Proving, ITP 2025 (LIPIcs, Vol. 352)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 36:1–36:19. doi:10.4230/LIPIcs.ITP.2025.36

[53] The mathlib Community. 2020. The Lean Mathematical Library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, Jasmin Blanchette and Catalin Hritcu (Eds.). ACM, 367–381. doi:10.1145/3372885.3373824

[54] The Rocq Development Team. 2025. *The Rocq Prover*. https://rocq-prover.org/

[55] Boris A. Trakhtenbrot. 1950. The impossibility of an algorithm for the decidability problem on finite classes. *Proceedings of the USSR Academy of Sciences* 70, 4 (1950), 569–572. In Russian.

[56] Dmitriy Traytel. 2015. A Coalgebraic Decision Procedure for WS1S. In *24th EACSL Annual Conference on Computer Science Logic, CSL 2015 (LIPIcs, Vol. 41)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 487–503. doi:10.4230/LIPIcs.CSL.2015.487

[57] G. S. Tseitin. 1968. On the complexity of derivation in propositional calculus. In *Studies in Constructive Mathematics and Mathematical Logic, Part II*, A. O. Slisenko (Ed.). Steklov Mathematical Institute, 115–125. Translated from the Russian.

[58] Moshe Y. Vardi. 1982. The Complexity of Relational Query Languages (Extended Abstract). In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*, Harry R. Lewis, Barbara B. Simons, Walter A. Burkhard, and Lawrence H. Landweber (Eds.). ACM, 137–146. doi:10.1145/800070.802186

[59] Celia Wrathall. 1976. Complete Sets and the Polynomial-Time Hierarchy. *Theor. Comput. Sci.* 3, 1 (1976), 23–33. doi:10.1016/0304-3975(76)90062-1

A How Often the Literature Proves a Hardness Result

We explain how the number §1 opens with was measured. We took every arXiv submission of 2025 whose *primary* category is one of the 7 theoretical computer science categories, 4 795 records. For each we checked whether its title or abstract announces a hardness, completeness or undecidability result, and whether it announces one as the authors' own.

We measure what a paper announces in title or abstract only, so 475 is a lower bound. Mentioning is not claiming either: 745 papers (15.5%) mention such a result, most of them to justify a heuristic, while 475 (9.9%) claim one of their own. We use simple English patterns: to count as claiming, a paper must put a claim verb and a hardness term in one sentence, the verb first.

We need to pay attention to three potential pitfalls. Abstracts carry markup, so a pattern written against NP-complete misses the same words set as mathematics, and we strip the markup first. The OAI-PMH interface, recommended for bulk harvesting, keys its windows on a datestamp that moves to the latest revision, so a December paper revised in January is assigned the wrong year; we use the query interface, which filters on the submission date. And "complete for" is overloaded, as in "sound and complete for bounded postconditions", so we require a complexity class to follow it.

We hand-checked the classifier against samples. Of 60 sampled positives, 58 are correct. The two failures are a paper writing "given the *known* NP-hardness" of its problem, and a claimed (obviously bogus) PTIME versus NP separation in which the NP-completeness of SAT is background, not the result. Precision is therefore near 96%. Of 40 sampled papers that mention a hardness result without being credited with proving one, three were misses, all gaps in the list of claim verbs; adding those verbs caught six papers across the corpus and moved the count from 469 to 475. Of 42 further sampled papers with a weaker cue or none, none were misses. The two kinds of error are of comparable size and opposite in sign, so 475 is accurate to within a few percent.

We checked the choice of arXiv categories for theoretical computer science. Leaving out a theoretical one would understate the count, and taking in an applied one would dilute it. The seven differ widely among themselves, from 6.0% (cs.LO) to 24.8% (cs.CC), and every plausible neighbor sits an order of magnitude below: cs.DB at 1.2%, cs.SC at 1.1%, cs.PL at 0.7%, and cs.DC, cs.IT, cs.CR and cs.MS below those. Over all of cs.* the proportion would be 0.5%, due to the archive current dominance by machine learning papers.

Two caveats remain. First, arXiv is not the literature: it mixes preprints with published work, and authors choose their own primary category. Second, §1 says such results are almost always established by reduction. We checked that only on a sample: of 40 counted papers read in full, 35 use explicit reduction language.

B How Each Logic Is Represented in Lean

Mathlib supplies first-order syntax and semantics; it has no second-order quantifier and no fixed-point or closure operator, and the library adds none as syntax. Each logic is a *definability predicate* on `DecisionProblem`: a type of data, a sentence, a program or a specification; a Lean definition giving each datum a meaning on each structure; and the predicate itself, that some datum agrees with the problem on every finite nonempty structure. Two conventions run through all of them. A logic taken over ordered structures reads the vocabulary `L.sum Language.order` under a `[LinearOrder A]` instance, and the agreement is required for *every* linear order, which is what makes the notion order-invariant and the predicate a property of the problem alone; the order-free variants (`FODefinableFree`, `IFPDefinableFree`, `PFPDefinableFree`) drop the expansion and the instance. And each predicate comes with a lemma `_congr`, saying it depends only on the finite structures, and a lemma `.of_orderedReduction`, pulling it back along an ordered reduction: the obligations of `ComplexityClass.ofMem` (Figure 4), the plain reduction being an ordered one that ignores the order.

First-order logic. `FODefinableFree` asks for a Mathlib `Sentence` over the vocabulary itself, `FODefinable` for one over its ordered expansion, and `AC0Definable` for one over the arithmetic expansion, where $\leq$, $+$ and $\times$ are relation symbols computed from the order.

Second-order logic. A quantifier block is a finite index type of relation variables with their arities; its vocabulary `B.lang` has one relation symbol per variable, and an assignment gives each variable a relation on the universe. A list of blocks is folded into the base vocabulary by `soLang`, and `SORealize` quantifies the blocks alternately, existentially first or universally first, over a first-order kernel in the summed vocabulary. So a Σ^1_k sentence is a list of k blocks and a Mathlib sentence, and `SigmaSODefinable` k and `PiSODefinable` k are the two polarities. Pulling a block back along an interpretation (§4.1) is a construction on blocks alone, `SOBlock.pull`: an index becomes a pair of an index and a tuple of tags, an arity is multiplied by the dimension.

```
structure SOBlock : Type 1 where
  ι : Type
  [ιFinite : Finite ι]
```

```
  arity : ι → ℕ

variable {L : Language.{0, 0}}
def SigmaSODefinable [L.IsRelational] (k : ℕ) (P : DecisionProblem L) : Prop :=
  ∃ Bs : List SOBlock, Bs.length = k ∧
    ∃ φ : (soLang L Bs).Sentence,
      ∀ (A : Type) [L.Structure A] [Finite A] [Nonempty A], P A ↔ SORealize L A Bs φ true

variable (Tag : Type) [Finite Tag] (d : ℕ)
def SOBlock.pull (B : SOBlock) : SOBlock where
  ι := Σ i : B.ι, Fin (B.arity i) → Tag
  arity p := B.arity p.1 * d
```

The Horn and Krom fragments are not recognized on sentences but built as data. A `HornClause` is a first-order guard over the input vocabulary, a list of positive second-order body atoms, and an optional head; a program is a list of clauses, universally quantified over k shared first-order variables. `SigmaSOHornDefinable` asks for a block and a program, over the ordered vocabulary, such that the problem holds exactly when some assignment satisfies every clause; `SigmaSOKromDefinable` is the same with clauses of at most two signed second-order literals. Value invention (`SigmaSONewDefinable`) extends the vocabulary by the unary mark `old`, and reads the block over `A ⊕ Fin m`, the invented elements related to nothing, for some `m : ℕ` existentially quantified in Lean and not in the logic, which is what leaves the number of invented elements unbounded.

Fixed points. An `LFPDef` is a block, a list of Horn rules defining its variables, and an *unrestricted* output sentence over the ordered vocabulary expanded by the block; it holds when the output is true at the least fixed point of the rules, which is the inductive predicate `Derives` of derivable atoms. Negating the output negates the value, so closure under complement (`LFPDefinable.compl`) is one line, and Grädel's theorem (§4.3) is the translation between an `LFPDef` and a Horn program, in both directions.

```
structure HornClause (L : Language.{0, 0}) (B : SOBlock) (k : ℕ) where
  guard : L.Formula (Fin k)
  body : List (SOAtom B k)
  head : Option (SOAtom B k)

structure LFPDef (L : Language.{0, 0}) : Type 1 where
  B : SOBlock
  k : ℕ
  rules : List (HornClause (L.sum Language.order) B k)
  out : ((L.sum Language.order).sum B.lang).Sentence
```

The inflationary and partial fixed points share their data: a `StepDef` is a block, one first-order step formula per variable, over the vocabulary expanded by the block and with the variable's arguments as free variables, and an output sentence. The step formulas are unrestricted, and the two logics differ only in the iteration they apply: `IFPHolds` accumulates each stage into the previous one and reads the output at the limit; `PFPHolds` iterates the bare step from the empty assignment and holds when some stage is a fixed point at which the output is true, so a diverging iteration makes the definition false whatever its output. The textbook reading of §2.1, which returns the empty relations instead, differs only when the output holds at the empty assignment; `StepDef.realize_pfpValue_iff` compares the two, and a definition is carried from one reading to the other by guarding its output with "the stage is a fixed point", a first-order condition. The convention is chosen because it makes the translation to SO(TC) direct: acceptance of the walk *is* "some stable stage satisfying the output is reachable", with no divergence detection, which unordered structures could not supply.

```
structure StepDef (L : Language.{0, 0}) : Type 1 where
  B : SOBlock
  step : ∀ i : B.ι, (L.sum B.lang).Formula (Fin (B.arity i))
  out : (L.sum B.lang).Sentence

variable {L : Language.{0, 0}}
def IFPDefinable [L.IsRelational] (P : DecisionProblem L) : Prop :=
  ∃ d : StepDef (L.sum Language.order),
    ∀ (A : Type) [L.Structure A] [LinearOrder A] [Finite A] [Nonempty A],
      P A ↔ d.IFPHolds A
```

Transitive closure. A `TCSpec` is a walk on k-tuples carrying a finite *mode*: a transition formula per pair of modes, with the current tuple and the next as free variables, and start and accepting formulas per mode, all over the ordered vocabulary. A node is a mode with a tuple, the transition formulas define the edges, and the specification accepts when some accepting node is reachable from some starting node; `TCDefinable` asks the problem to coincide with acceptance. The mode is to a walk what a tag is to an interpretation: a one-element universe has one tuple, so finite state cannot live in the tuple. `DTCDefinable` replaces the transition by its determinization `TCSpec.det`, "this step, and no other step out of the current node", so that determinism is a formula and not a side condition on the specification. SO(TC) is the same walk on assignments of a block (`SOTCSpec`), its transition a sentence over two copies of the block.

```
structure TCSpec (L : Language.{0, 0}) where
  Mode : Type
  [modeFinite : Finite Mode]
  k : ℕ
  step : Mode → Mode → (L.sum Language.order).Formula (Fin k ⊕ Fin k)
  src : Mode → (L.sum Language.order).Formula (Fin k)
  tgt : Mode → (L.sum Language.order).Formula (Fin k)
```

Exponential expansions. An `ExpExpansion` is the exponential expansion of §2.2 as a Lean `structure`: a finite type of tags, a block whose assignments are the points, a domain sentence δ_t per tag, and, for each relation symbol of the expanded vocabulary and each tuple of tags, a defining sentence over as many copies of the block as the symbol has arguments. Its `Map` sends `A` to the structure whose universe is $\{(t,\rho) \mid (A,\rho) \models \delta_t\}$, and `ExpDefinable` C holds of P when some expansion turns it into a member of C. `ComplexityClass.exp` is that predicate handed to `ComplexityClass.ofMem`; `SOLFPDefinable` and `SOPFPDefinable`, the logics of EXPTIME and EXPSPACE in Table 1, are `LFPDefinable` and `PFPDefinable` read through an expansion.

```
structure ExpExpansion (L : Language.{0, 0}) : Type 1 where
  Tag : Type
  B : SOBlock
  E : Language.{0, 0}
  dom : Tag → ((L.sum Language.order).sum B.lang).Sentence
  relSentence : ∀ {n : ℕ}, E.Relations n → (Fin n → Tag) →
    ((L.sum Language.order).sum (B.replicate n).lang).Sentence
  ...

variable {L : Language.{0, 0}} [L.IsRelational]
def ExpDefinable (C : ComplexityClass) (P : DecisionProblem L) : Prop :=
  ∃ (X : ExpExpansion L) (Q : DecisionProblem X.E), C.Mem Q ∧
    ∀ (A : Type) [L.Structure A] [LinearOrder A] [Finite A] [Nonempty A], P A ↔ Q (X.Map A)
```

Games. The Ehrenfeucht–Fraïssé game is the predicate `efStage`, by recursion on the number of rounds: a position is a pair of tuples, one in each structure, and survives zero rounds when it is a partial isomorphism, $n+1$ rounds when it is one and every element chosen on either side can be answered on the other so that the extended position survives n. `EFEquiv` plays from the empty position, and the one lemma the lower bounds rest on, `realize_efStage`, says that a formula of quantifier rank at most n cannot separate the two sides of a position surviving n rounds. The strategies are then combinatorics: on bare sets of at least n elements (`efEquiv_bare`) and on linear orders of at least 2^n elements (`efEquiv_linearOrder`), whence `even_not_foDefinableFree` and `even_not_foDefinable`.

```
def efStage (L : Language.{0, 0}) {M N : Type} [L.Structure M] [L.Structure N] :
    ℕ → ∀ {j : ℕ}, (Fin j → M) → (Fin j → N) → Prop
  | 0, _, a, b => PartialIso L a b
  | n + 1, _, a, b =>
      PartialIso L a b ∧
        (∀ c : M, ∃ d : N, efStage L n (Fin.snoc a c) (Fin.snoc b d)) ∧
        (∀ d : N, ∃ c : M, efStage L n (Fin.snoc a c) (Fin.snoc b d))

variable {L : Language.{0, 0}} {M N : Type} [L.Structure M] [L.Structure N]
variable [L.IsRelational]
theorem realize_efStage : ∀ {j : ℕ} (φ : L.BoundedFormula Empty j) {n : ℕ}, qdepth φ ≤ n →
    ∀ {a : Fin j → M} {b : Fin j → N}, efStage L n a b →
      (φ.Realize default a ↔ φ.Realize default b)
```

The k-pebble game is kept apart from the logic. `EquivK` is the greatest fixed point of one round of pebble exchange over an abstract initial relation, instantiated at agreement on the atomic type over the finitely many symbols a definition mentions; `realize_equivK` is the k-variable invariance lemma, its budget hypothesis standing in for a syntactic k-variable fragment, which is never defined. Each stage of an induction expands the structure by an invariant relation and leaves `EquivK` unchanged (`equivK_inf_eq`), so an inflationary definition whose formulas fit in k variables cannot separate two $\equiv^k$-equivalent structures (`StepDef.ifpHolds_equivK`$_2$), and bare sets of $2k + 2$ and $2k + 3$ elements are such a pair, whence `exists_mem_PTIME_not_ifpDefinableFree`.

C What the Machine Characterization Costs at Each Level

Table 4 applies the metric of Table 3 along the hierarchy instead of across provers: each number is the declaration closure of a theorem, counted in the library's own modules, with the rows and the order of Table 1. The two rows below PTIME carry no membership and hardness split because the bridge there is a capture rather than a complete problem (§5.2).

The *marginal* column is a difference of declaration *sets*, not of totals, and it is taken over the entry in the *over* column, what the library had already paid for when the row was added. For most rows that is the class's problem in Table 1, the machine being the second complete problem for its class: CVP for PTIME, SAT for NP. At the three exponential classes the machine comes first instead, the tilings being drawn out of it afterwards, so baselining on a tiling would measure backwards; the baseline is the machine of the class each one is the exponential of – `DTMAccept` for EXPTIME – and the marginal measures what reading the same machine over an exponential expansion costs. A † row is taken over the first machine of its class. The marginal can be far larger than the gap between the totals, because the closures overlap only in part. The closure of `ATMAcceptSpace` exceeds that of `DTMAccept` by 4 746 lines, and yet 8 471 of its lines are ones the deterministic machine's proof never touches.

The cost of machine characterization is often comparable to that at the NP level. In particular, every row from PTIME to EXPTIME, and RE above them, closes below 19 658 lines, the smallest Cook–Levin closure of any other library in Table 3.

The † rows are cheap: a second machine for a class is reached from the first and not from the logic. 124 lines buy the nondeterministic space machine over the deterministic one, the deterministic side being proved hard and transferred, so that no Savitch argument [Sav70] is needed on the machine side; 55 lines buy the deterministic wide machine over the nondeterministic one, and 442 the one-block alternating machine over `NTMAccept`.

The two *wide* rows are the exception, at 40 832 and 47 007 lines against 12 641 for EXPTIME, which is exponential too. The construction sets them apart: a reduction into a wide machine quantifies over an exponential expansion of the instance (§5.2), so its interpretation has to write the machine's program down inside the logic instead of naming it, and the hardness half alone runs to 46 161 lines.

D Complete Problems per Class

Table 5 counts, class by class, the 73 completeness theorems of §6 across 14 classes. We count theorems, not problems, and the 73 theorems are about 68 distinct problems, a family such as QBF$_k$ counting once: a class whose complete problem comes in several forms – TAUT, 3-DNF-TAUT and 3-UNSAT for coNP – contributes one per form, and a problem complete for two classes proved equal – QSAT for PSPACE and coPSPACE – contributes a theorem to each. A theorem that merely restates another – an instantiation of a statement already counted for every k, say – is not counted twice. The NP row still exceeds the 35 problems of Figure 5 by three, each a further theorem attached to a problem already drawn there: 3-COLORABILITY is also proved complete as `KCol` for every $k \geq 3$, and CQ EVALUATION and GRAPH CRAWLING are also proved complete on their well-formed instances alone (§3.4).

E What a Complete Problem Costs

We cost the catalog of §6 problem by problem, measuring what a further problem adds to a library that already has the framework. That differs from the declaration closures of Tables 3 and 4, which count the framework beneath a theorem, paid once. So a problem is a module `Problems/X.lean` with its directory `Problems/X/` – the vocabulary, the encoding, the definability witness, the reduction gadgets and the completeness statement – and we count the non-blank source lines of those modules. Of the 42 problems here, 35 come in under two thousand lines. The theorem counts here sum to those of Table 5, and a problem whose completeness theorem is proved elsewhere is still costed where its work is: `REACH_NL_complete` is stated in `TransitiveClosureReach`, but its subject is defined under `Problems/Reachability/` and is counted there.

The machine models are the one exception. Their cost is the cost of building a machine model, the subject of Appendix C. Nothing else is left out. The two wide tilings share a directory with the wide machines, and are costed here on the modules that are theirs alone, the machines keeping the rest. So the two tables account for every completeness theorem between them: 61 of

Table 4. Cost of the machine characterization, class by class, with the rows of Table 1. Each number is a *declaration closure* in the sense of Table 3: the declarations the theorem's proof term transitively reaches, counted as the source lines they occupy, so a module contributes only the part of itself the proof uses. Only the library's own modules are counted, not the Mathlib declarations beneath them. *Membership* and *Hardness* are the closures of the two halves and overlap both each other and *Complete*, so the three do not add up. *Marginal* is closure(*Complete*) \ closure(*over*) on declaration sets, not a difference of totals; *over* is what was already paid for, a prototypical complete problem of the class where there is one and a comparable machine result otherwise. A dash means the theorem has no separately named halves. * A capture biconditional, not a complete problem, so it has no membership/hardness split. † A second machine for a class already rowed above.

Class	Machine statement	Mod.	Membership	Hardness	Complete	Marginal	over
LOGSPACE	`HeadAutomaton` (det.)*	22	–	–	5 035	4 024	DET-REACH
NL	`HeadAutomaton`*	31	–	–	6 987	2 279	REACH
PTIME	`DTMAccept`	37	4 707	4 731	7 895	3 897	CVP
NP	`NTMAccept`	32	2 485	5 640	6 268	2 949	SAT
NP	`ATMAccept` (1 block)†	35	2 848	5 978	6 706	442	`NTMAccept`
coNP	`ATMAccept`	57	–	–	10 908	7 776	TAUT
Σ_k^{P}	`ATMAccept`	57	4 411	8 586	10 871	6 522	QBF_k
Π_k^{P}	`ATMAccept`	57	4 444	8 620	10 905	6 522	$\mathrm{QBF}_k^{\forall}$
PSPACE	`DTMAcceptSpace`	51	1 859	9 302	9 807	3 435	QSAT
PSPACE	`NTMAcceptSpace`†	52	1 750	9 462	9 888	124	`DTMAcceptSpace`
EXPTIME	`ATMAcceptSpace`	87	5 151	11 639	12 641	8 471	`DTMAccept`
NEXPTIME	`WideRegAccept`	209	3 464	39 584	40 832	38 162	`NTMAccept`
EXPSPACE	`WideAcceptSpace`	207	11 759	46 161	47 007	37 243	`NTMAcceptSpace`
EXPSPACE	`DWideAcceptSpace`†	206	11 791	46 047	46 924	55	`WideAcceptSpace`
RE	`HALT`	59	3 170	14 253	15 614	8 890	FINSAT

Table 5. Completeness theorems proved, by class, with the rows of Table 1. PH is the one class of that table with no entry, and its empty cell is a theorem, not a gap: a complete problem for PH would collapse the hierarchy. GI is set below the rule because it is a degree (§4.2).

Class	Completeness theorems
LOGSPACE	2
NL	3
PTIME	4
NP	38
coNP	3
DP	1
Σ_k^{P}	2
Π_k^{P}	2
PH	–
PSPACE	5
EXPTIME	1
NEXPTIME	2
EXPSPACE	3
RE	4
GI	3

the 73 are costed here, one problem often carrying several, and the remaining 12 belong to the machine models, which Table 4 lists, second machines included. Its coNP row is the one-block instance of the Π_k^{P} theorem above it, and so is not a further one.

F A Worked Example: Subgraph Isomorphism

The figures of §3 show the definitions of the framework. Here we show how a user proves a completeness result against them. We take one problem of the catalog and walk it through the six steps every problem follows – vocabulary, semantics, invariance, membership, hardness, completeness – and a seventh, the encoding of concrete data and its decoding, showing

Table 6. What each problem of the catalog costs: the non-blank source lines of its own modules, with the classes it is proved complete for and the number of completeness theorems it carries. Not a declaration closure, unlike Tables 3 and 4. The machine models are excluded and costed in Table 4 instead. Read down the left panel, then the right.

Module	Complete for	Thms	Lines	Module	Complete for	Thms	Lines
Taut	coNP	1	147	ConjunctiveQueries	NP	3	878
ThreeDnfTaut	coNP	2	159	Cvp	PTIME	1	881
NaeThreeSat	NP	1	256	TwoSat	NL	1	922
NaeSat	NP	1	270	MaxCut	NP	1	924
DigraphIso	GI	1	295	HornSat	PTIME	1	944
Reachability	NL	2	356	GraphCrawling	NP	2	964
WideCorridor	EXPSPACE	1	401	Qbf	Π_k^{p}, Σ_k^{p}	2	968
SatUnsat	DP	1	423	Knapsack	NP	1	973
DominatingSet	NP	1	434	ThreeDimMatching	NP	1	1 025
SubgraphIso	NP	1	474	GraphIso	GI	1	1 036
ZeroOneIP	NP	1	521	Steiner	NP	2	1 192
Coloring	NP	3	558	SuccinctReach	PSPACE	1	1 363
ThreeColorability	NP	1	568	Partition	NP	1	1 482
Game	PTIME	1	607	Sat	NP	1	1 730
ReachabilityDet	LOGSPACE	2	617	Hamilton	NP	2	2 032
ThreeSat	NP	1	622	JobSequencing	NP	1	2 164
Feedback	NP	2	677	WideTiling	NEXPTIME	1	2 516
DagIso	GI	1	711	CodeHalt	RE	1	3 051
CliqueFamily	NP	3	750	Qsat	PSPACE	2	3 609
SetFamily	NP	5	836	Pcp	RE	1	4 170
OneInSat	NP	1	861	FinSat	RE	1	5 038

every declaration and eliding every proof. The problem is SUBGRAPH ISOMORPHISM: does the host graph contain a subgraph isomorphic to the pattern graph? It is among the cheapest rows of Table 6, and a fairly self-contained example. It is also one of the few catalog problems that come with an encoding and a decoding to concrete Mathlib classes. Of its 474 lines, the problem itself is `Problems/SubgraphIso.lean`, 275 lines, roughly half of them the reduction and its correctness, a quarter the definability witness, and the rest definitions and invariance; the remaining 199, `Problems/SubgraphIso/Encoding.lean`, are Step 7.

Step 1: the vocabulary

An instance carries two graphs, so the vocabulary has two unary marks, telling the vertices of the pattern from those of the host, and two binary relations, one adjacency for each. The `fo_language` command declares it, a symbol and its arity per line, under a prefix for the abbreviations it is to generate, here `tg` for `twoGraphs`. It generates what one would otherwise write out: an inductive type of relation symbols indexed by arity, packaged as a `Language` with no function symbols, whence the `IsRelational` instance that `DecisionProblem` requires (§3.1), and the four abbreviations `tgPatV`, `tgHostV`, `tgPatE` and `tgHostE` naming the symbols at their arity, for use in formulas.

```
fo_language twoGraphs with tg where
  patV : 1
  hostV : 1
  patE : 2
  hostE : 2
```

Nothing forces an element of the universe to be a vertex of either graph. Elements outside both marks are “junk elements” that no condition below mentions. That is deliberate: it lets an interpretation build such a structure inside a tagged power of its input universe (§3.2), as Step 5 does.

Step 2: the semantics

The property is stated first generically, for arbitrary predicates on a type, then instantiated to the relations of a structure. `SubgraphIsoOn` asks for a map that sends pattern vertices to host vertices, injectively on the pattern, carrying pattern edges to host edges: an injective homomorphism of the pattern into the host, which is the usual reading (not an induced subgraph, so that non-edges of the pattern are unconstrained and a clique is a special case). The map is total on the universe and unconstrained

off the pattern, junk included. `fo_predicates` generates the four shorthands `TGPatV` to `TGHostE`, which read the relations of a `twoGraphs`-structure as predicates, and `HasSubgraphIso` is the property on structures. Its finiteness conjunct is there for uniformity with the threshold problems of the catalog, whose cardinality comparisons need it; by `mem_congr_finite` (§3.1) it changes no complexity-theoretic statement.

```
fo_predicates Language.twoGraphs tg

variable {A : Type}
def SubgraphIsoOn (PV HV : A → Prop) (PE HE : A → A → Prop) : Prop :=
  ∃ f : A → A, (∀ x, PV x → HV (f x)) ∧
    (∀ x y, PV x → PV y → f x = f y → x = y) ∧
    ∀ x y, PV x → PV y → PE x y → HE (f x) (f y)

variable (A : Type) [Language.twoGraphs.Structure A]
def HasSubgraphIso : Prop :=
  Finite A ∧ SubgraphIsoOn (TGPatV (A := A)) TGHostV TGPatE TGHostE
```

Step 3: invariance and the bundled problem

Isomorphism-invariance is the proof obligation of Figure 1. The generic property transports along any equivalence of types that commutes with the four predicates – the map is conjugated by the equivalence, a twenty-line lemma (`SubgraphIsoOn.of_equiv`, made a biconditional by `SubgraphIsoOn.equiv_iff`) – and an isomorphism of structures commutes with every relation by the library's transport lemmas $\texttt{relMap\_equiv}_1$ and $\texttt{relMap\_equiv}_2$, so the invariance theorem `hasSubgraphIso_iso` is a six-line application of the two. Here is its statement, and the bundled problem:

```
variable {A B : Type} [Language.twoGraphs.Structure A] [Language.twoGraphs.Structure B]
theorem hasSubgraphIso_iso (e : A ≃[Language.twoGraphs] B) :
    HasSubgraphIso A ↔ HasSubgraphIso B

def SubgraphIso : DecisionProblem Language.twoGraphs where
  Holds := fun A inst => @HasSubgraphIso A inst
  iso_invariant := fun e => hasSubgraphIso_iso e
```

Step 4: membership, by a definability witness

Membership in NP is ∃SO-definability, since that is how the library defines NP (Table 1), so the witness is a sentence: guess the map as a binary relation variable, then check it first-order. The `fo_block` command declares all of it at once. A second-order block is a finite index type of relation variables with their arities, here the single variable `map` of arity two. The kernel is a sentence not over `Language.twoGraphs` but over its sum with the block's own symbols, which the command names `subgraphSOLang`; and under the second prefix it generates an abbreviation for every symbol of that sum, the four of Step 1 injected on the left, `sgPatVSym` for `tgPatV` and so on, and the guessed `sgMapSym` on the right. The prefix says over which vocabulary a symbol is read: `tg` over `twoGraphs`, as in Step 1, and `sg` over `subgraphSOLang`.

```
fo_block isoGuessBlock over Language.twoGraphs tg into subgraphSOLang with sg where
  map : 2
```

The kernel is the conjunction of three clauses, each a first-order sentence over that sum, and each written in the `fo%` syntax of §6: variables are named and bound by `∀` and `∃`, a relation is applied to terms as `R(x, y)`, the connectives are the usual ones, and `≐` is equality of terms, kept apart from Lean's own `=`. We write V_p and V_h for the two vertex marks, E_p and E_h for the two adjacencies, and M for the guessed relation.

The first clause says that every pattern vertex is mapped to some host vertex:

$$\forall x_0 \, \big(V_p(x_0) \to \exists y \, (M(x_0, y) \wedge V_h(y))\big).$$

```
private noncomputable def sgTotalClause : subgraphSOLang.Sentence :=
  fo% ∀ x₀, sgPatVSym(x₀) → ∃ y, sgMapSym(x₀, y) ∧ sgHostVSym(y)
```

The second says that the map is injective on the pattern:

$$\forall x_0 x_1 x_2 \, \big(V_p(x_0) \wedge V_p(x_1) \wedge M(x_0, x_2) \wedge M(x_1, x_2) \to x_0 = x_1\big).$$

```
private noncomputable def sgInjClause : subgraphSOLang.Sentence :=
  fo% ∀ x₀ x₁ x₂,
    ((sgPatVSym(x₀) ∧ sgPatVSym(x₁)) ∧ sgMapSym(x₀, x₂)) ∧ sgMapSym(x₁, x₂) → x₀ ≐ x₁
```

The third says that it carries pattern edges to host edges:

$$\forall x_0 x_1 x_2 x_3 \left(V_p(x_0) \wedge V_p(x_1) \wedge E_p(x_0, x_1) \wedge M(x_0, x_2) \wedge M(x_1, x_3) \rightarrow E_h(x_2, x_3)\right).$$

```
private noncomputable def sgEdgeClause : subgraphSOLang.Sentence :=
  fo% ∀ x₀ x₁ x₂ x₃,
    (((sgPatVSym(x₀) ∧ sgPatVSym(x₁)) ∧ sgPatESym(x₀, x₁)) ∧
      sgMapSym(x₀, x₂)) ∧ sgMapSym(x₁, x₃) → sgHostESym(x₂, x₃)
```

The kernel is their conjunction. The quantifier $\exists M$ is no part of it: the block carries that, so that `SigmaSODefinable 1 SubgraphIso` is the $\exists$SO sentence $\exists M$ of the three clauses.

```
noncomputable def subgraphKernel : subgraphSOLang.Sentence :=
  sgTotalClause ⊓ (sgInjClause ⊓ sgEdgeClause)

theorem subgraphIso_sigmaSODefinable : SigmaSODefinable 1 SubgraphIso
```

`SigmaSODefinable 1 P` unfolds to a list of one block and a sentence over the summed vocabulary that agrees with P on every finite nonempty structure. Its proof relies on a private lemma `realize_subgraphKernel` that reads the realization of the kernel back as a statement of Lean about the guessed relation, some twenty-five lines, most of them one `simp` call listing the realization lemmas of the connectives; the two directions are then the mathematics, in thirty lines: an injective homomorphism is the graph of a function satisfying the three clauses, and a relation satisfying them yields one by choice.

Step 5: hardness, by a reduction from CLIQUE

The reduction is from CLIQUE, on marked graphs – a graph with a marked set whose cardinality is the threshold k. The host is the input graph, and the pattern is the complete graph on the marked set, so that an injective homomorphism of the pattern is exactly a clique at least as large as the marked set. This is the interpretation of §3.2 with two tags and dimension one, `Bool` standing for $\{\ell, r\}$: the tag `true` carries the pattern copy of every vertex and `false` its host copy. The listing reads against Figure 2: for each symbol of the output vocabulary and each assignment t of tags to its arguments, a formula over the input vocabulary of marked graphs, whose mark and adjacency symbols are `mgMarked` and `mgAdj`. It is written in the surface syntax of Step 4, in its `fo%⟨u, v⟩` form, which names the arguments of the symbol: the dimension is one, so an argument is a single element of the input, `u` the first and `v` the second, where a hand-written formula would address them by argument and coordinate as $(0, 0)$ and $(1, 0)$. The `if` chooses between formulas by the tags, and $\bot^{\mathrm{f}}$ is the false formula, apart from Lean's own $\bot$. The pattern vertices are exactly the pattern copies of marked vertices, and the host vertices exactly the host copies; two pattern copies are pattern-adjacent when they are copies of distinct marked vertices, two host copies host-adjacent when they are copies of distinct adjacent ones, and every other combination is $\bot$. The pattern copy of an unmarked vertex is thus in neither graph, junk the guessed map may send anywhere, as §3.2 said it would be. Every formula is quantifier-free, and all but one is a *first-order projection* in the sense of §8.1: a constant, or a single bit of the input, under a condition on the tags and the coordinates alone. The pattern-adjacency formula misses only by conjoining the two marks.

```
def cliquePatternInterp :
    FOInterpretation Language.markedGraph Language.twoGraphs Bool 1 where
  relFormula {n} R :=
    match n, R with
    | _, .patV => fun t => fo%⟨u⟩ if t 0 then mgMarked(u) else ⊥ᶠ
    | _, .hostV => fun t => if t 0 then ⊥ else ⊤
    | _, .patE => fun t => fo%⟨u, v⟩
        if t 0 then (if t 1 then ¬ u ≐ v ∧ (mgMarked(u) ∧ mgMarked(v)) else ⊥ᶠ) else ⊥ᶠ
    | _, .hostE => fun t => fo%⟨u, v⟩
        if t 0 then ⊥ᶠ else (if t 1 then ⊥ᶠ else ¬ u ≐ v ∧ mgAdj(u, v))

theorem hasLargeClique_iff_subgraphIso_map (A : Type) [Language.markedGraph.Structure A] :
    HasLargeClique A ↔ HasSubgraphIso (cliquePatternInterp.Map A)

def clique_fo_reduction_subgraphIso : Clique ≤ᶠᵒ SubgraphIso where
  Tag := Bool
  dim := 1
  toInterpretation := cliquePatternInterp
  correct A _ _ _ := hasLargeClique_iff_subgraphIso_map A
```

Correctness is the theorem `hasLargeClique_iff_subgraphIso_map` between the interpretation and the reduction. Six characterization lemmas, the `clPat_` family, first say what each relation of the interpreted structure means on tagged copies,

each one a `simp` call through `FOInterpretation.relMap_map`; the two directions then take fifty lines, an embedding of the marked set into a clique being turned into an injective homomorphism and back. The threshold of CLIQUE is consumed by the shape of the pattern, not by a counting argument: the one cardinality fact used is that a set at least as large as the marked set receives an injection from it, `cliqueOn_iff_embedding`, which CLIQUE already provides for its own witness. The reduction record then packages the interpretation with its correctness, exactly as in Figure 1.

Step 6: completeness

Membership is the witness, since NP is ∃SO by definition; hardness composes the reduction with the hardness of CLIQUE, closure under reductions being part of what a class is (§4.1); completeness is the pair.

```
theorem subgraphIso_mem_NP : SubgraphIso ∈ NP :=
  subgraphIso_sigmaSODefinable

theorem subgraphIso_NP_hard : NP.Hard SubgraphIso :=
  NP.hard_of_foReduction clique_fo_reduction_subgraphIso clique_NP_hard

theorem subgraphIso_NP_complete : NP.Complete SubgraphIso :=
  ⟨subgraphIso_mem_NP, subgraphIso_NP_hard⟩
```

No machine appears anywhere. The hardness of CLIQUE is that of SAT, by the ordered reduction of Figure 5, and that of SAT is the arrow out of ∃SO in the same figure.

Step 7: from concrete graphs and back

The problem so far is stated on structures. A user holds two concrete graphs, a pattern on p vertices and a host on h, each a finite set of edges, and the encoding and decoding of §3.4 tie those to `SubgraphIso`. The concrete instance is a record. Its size is the textbook one, the vertices and edges of both graphs, and the two bounds below are stated against it. Its predicate is the textbook one too: an injective map from the pattern's vertices to the host's carrying every edge to an edge. The encoder puts the pattern's vertices to the left of a sum and the host's to the right. The marks then read the side, and the two adjacency relations decide membership in the two edge sets. The encoder is a plain `def`, so the compiler vouches that it computes. The bundle of Figure 3 carries the two bounds as proofs, elided below. The universe is the vertex set, so nothing is padded, and an edge set has at most quadratically many elements, so nothing is compressed. Faithfulness closes the listing. Its proof, and the decoder's, go through one lemma: against an enumeration of the pattern vertices and one of the host vertices, the generic property of Step 2 is the textbook predicate (`subgraphIsoOn_iff_concrete`); the encoding enumerates them by the two injections of the sum.

```
structure SubgraphIsoInstance where
  pat : ℕ
  host : ℕ
  patEdges : Finset (Fin pat × Fin pat)
  hostEdges : Finset (Fin host × Fin host)
  deriving DecidableEq

def sgSize (i : SubgraphIsoInstance) : ℕ :=
  i.pat + i.host + i.patEdges.card + i.hostEdges.card

def ConcreteSubgraphIsoHolds (i : SubgraphIsoInstance) : Prop :=
  ∃ f : Fin i.pat → Fin i.host, Function.Injective f ∧
    ∀ a b, (a, b) ∈ i.patEdges → (f a, f b) ∈ i.hostEdges

def sgRelBool (i : SubgraphIsoInstance) {n : ℕ} (R : Language.twoGraphs.Relations n) :
    (Fin n → (Fin i.pat ⊕ Fin i.host)) → Bool :=
  match n, R with
  | _, .patV => fun x => (x 0).isLeft
  | _, .hostV => fun x => (x 0).isRight
  | _, .patE => fun x =>
    match x 0, x 1 with
    | Sum.inl a, Sum.inl b => decide ((a, b) ∈ i.patEdges)
    | _, _ => false
  | _, .hostE => fun x =>
```

```
    match x 0, x 1 with
    | Sum.inr a, Sum.inr b => decide ((a, b) ∈ i.hostEdges)
    | _, _ => false

def subgraphIsoEncoding : Encoding Language.twoGraphs SubgraphIsoInstance where
  size := sgSize
  Univ := fun i => Fin i.pat ⊕ Fin i.host
  ...
  relBool := fun i {n} R => sgRelBool i R
  ...

theorem subgraphIsoEncoding_faithful :
    subgraphIsoEncoding.Faithful ConcreteSubgraphIsoHolds SubgraphIso
```

The decoding direction reads hardness back (§3.4), and here it needs no well-formedness condition. The semantics ignores the elements in neither mark, and never relates the pattern role and the host role of one element, so a structure marking an element as both is the same instance as one where that element is split in two. The decoder therefore reads every presented structure back, a `FinPresentation` being a finite structure whose relations come as computable tables: the pattern vertices are the pattern-marked elements and the host vertices the host-marked ones, each enumerated in order (`Finset.orderEmbOfFin`), and the edges are read off the tables. Its soundness is the lemma above with the other pair of enumerations, and its totality is `rfl`, the condition `W` of §3.4 being `⊤`. The consequence is the last statement: every finite nonempty structure is decided by `SubgraphIso` exactly as some pair of concrete graphs is by the textbook predicate, so the hardness of Step 5 is about concrete graphs and not about junk.

```
variable (S : FinPresentation Language.twoGraphs)
def sgDecode : Option SubgraphIsoInstance :=
  some ⟨(sgPatVs S).card, (sgHostVs S).card,
    Finset.univ.filter fun q => S.relBool tgPatE
      ![(sgPatVs S).orderEmbOfFin rfl q.1, (sgPatVs S).orderEmbOfFin rfl q.2],
    Finset.univ.filter fun q => S.relBool tgHostE
      ![(sgHostVs S).orderEmbOfFin rfl q.1, (sgHostVs S).orderEmbOfFin rfl q.2]⟩

def subgraphIsoDecoding : Decoding Language.twoGraphs (DecisionProblem.ofSentence ⊤)
    ConcreteSubgraphIsoHolds SubgraphIso where
  dec := sgDecode
  sound := sgDecode_sound
  total := sgDecode_total

theorem exists_concreteSubgraphIso_iff (A : Type) [Language.twoGraphs.Structure A]
    [Finite A] [Nonempty A] : ∃ i, ConcreteSubgraphIsoHolds i ↔ SubgraphIso A
```

The compiler enforces that the encoder and the decoder compute: it would demand `noncomputable` of a definition deciding an undecidable predicate. Their complexity is not bounded, the interface measuring nothing against a machine, and the reader sees from `sgRelBool` that it decides membership in two finite sets.

What the other problems pay for

The more expensive rows of Table 6 pay for three things this problem does not need. An ordered reduction, when the gadget needs an element the input does not single out – the fresh variable of the reduction to NAE-SAT, the three marked copies of the minimum in the one to CHROMATIC NUMBER. A number laid along the order, when the target carries weights (§6.1). And a well-formedness condition, when a universe has junk that no decoder can read back without deciding something: the two tutorials of the library, CQ EVALUATION and GRAPH CRAWLING, restrict their completeness theorems to the well-formed instances – a website with exactly one root, for the latter – where Step 7 restricts nothing.

G How the Closures of Table 3 Are Computed

In Lean, proofs are terms, so we take the declarations a theorem's proof term mentions, close under that relation through statements and proof terms alike, map each declaration to its source range, merge overlapping ranges per module, and count the lines of code they cover. A line counts when it carries something other than whitespace, comment, or docstring, a convention reused in every measurement in this paper. The Rocq and Isabelle rows are the same measurement made with each

prover's own machinery: a kernel-level dependency graph from `coq-dpdgraph`, and a walk of the proof body with Isabelle's `Proofterm.fold_body_thms`.

References for the Appendix

[Sav70] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *J. Comput. Syst. Sci.*, 4(2):177–192, 1970.