Parameterized Compilability

MSc Thesis *(Afstudeerscriptie)*

written by

Noel Arteche Echeverría
(born December 22, 1998 in Donostia / San Sebastián, Spain)

under the supervision of **Dr. Ronald de Haan** and **Dr. Hubie Chen**, and submitted to the Examinations Board in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam*.

Date of the public defense:	**Members of the Thesis Committee:**
July 11, 2022	Dr. Malvin Gattinger
	Dr. Ronald de Haan
	Dr. Hubie Chen
	Dr. Balder ten Cate
	Dr. Florian Speelman

Institute for Logic, Language and Computation

Abstract

Compilability (also known as knowledge compilation) concerns the computational complexity of preprocessing intractable problems. For some hard computational problems, under the assumption that some part of the inputs will stay fixed over time, we allow this fixed part of the instances to be preprocessed in advance through some expensive precomputation whose output must be succinct (at most polynomially bigger than the input), hoping that this precomputation will help solve the complete instances efficiently (in polynomial time). The systematic study of compilability from a complexity-theoretic perspective was initiated by Cadoli, Donini, Liberatore and Schaerf (2002) and continued in an alternative style with Chen (2015), who modelled efficient compilation as a particular case of fixed-parameter tractability in his parameter compilation framework. Under such a complexity-theoretic framework, one can prove conditional lower bounds on the hardness of compilation.

In an attempt to go beyond polynomial-size compilation, this thesis studies parameterized compilability, where the compilation is allowed to be of fixed-parameter (fpt) size in the sense of parameterized complexity. This work introduces an extension of Chen's parameter compilation framework to account for doubly parameterized problems. These are problems having two parameters, where the first one indicates what part of the input is available for precomputation and the second one imposes a fixed-parameter bound on the size of that precomputation. The framework introduces new complexity classes to model fpt-size compilation and gives a new notion of reduction to study the relations between problems regarding their parameterized compilability.

Furthermore, we apply the new framework to computational problems around the parameterized complexity classes $\mathbf{W}[1]$ and $\mathbf{W}[2]$, showing hardness results in their compilability. Amongst other problems, we study the parameterized compilability of completing a satisfying assignment for a Constraint Satisfaction Problem (CSP) as well as the complexity of different inference problems for CSP.

Acknowledgements

I want to thank Ronald and Hubie for supervising this project. Ronald, thanks for introducing me to the topic and accepting to work on it at such an important moment in your life; congratulations again. Hubie, thanks for the time and effort put into this; it was a pleasure to work with you again. Thank you both for helping me learn a bit more about complexity theory.

I am also grateful to everyone who, in some form or another, was by my side during this last couple of years:

Many thanks go to my parents and sister for their unconditional support, despite my stubbornness and even despite them not being entirely sure about what this whole logic thing in Amsterdam was precisely.

Thank you, Raf, for the time spent together. You shaped me in many ways, and even if I never got to tell you what the following pages are about, there is a lot of us left in here. From you I learnt the aesthetics. Take care.

Many thanks to Bobby, for friendship, passion and eagerness to learn about the world. From you I learnt the ethics: we are what we do.

To Wittgenstein, for philosophical and moral guidance: ethics and aesthetics should be one.

To Pablo, for being there all this time. You always showed me there could be life beyond what I knew.

Thanks, Valentino, for your friendship and patience and for letting me share a bit of the warmth that complexity theory brings me. Take it easy.

To Francesco and Gian Marco: thanks for letting me intrude into the Italian safe space for those dinners.

Thanks to everyone at ASSV Esprit, for the fencing, the drinks and the people. I never thought being stabbed could be so much fun.

And, above all, to my grandma. Amatxi, we did not get to see each other again, but from you I learnt the most essential: “Il faut suivre la mode ou quitter le pays”. Thank you.

Muxu bat eta ondoloin. Bihar arte.

Contents

Chapter 1

Introduction

Life is hard, and so is computation.

For the last sixty years, theoretical computer scientists have dealt with the ubiquitous phenomenon of computational intractability. The tools of complexity theory have helped draw maps of how we believe computational hardness to extend across different domains and computational models. These are maps full of gaps and blurry regions, where exploration cannot be carried out but accompanied by strong complexity-theoretic assumptions and conjectures. And, as enlightening as it can be, the task of drawing, redrawing and exploring these maps is arguably daunting. Not only we must be equipped with heavy backpacks full of assumptions about computation if we want to intuit the landscape around us and see beyond the steepest peaks; ultimately, it is daunting because it requires a somewhat negative mindset: it is a journey toward proving impossibility results.

And while life is hard, it is often not impossible. For the most part, we do just fine. While proving the unconditional hardness of fundamental problems is a fascinating journey, some try to circumvent worst-case hardness by making some concessions to the all-mighty deities of complexity in exchange for algorithms that do just fine in practice. Naturally, the unstoppable success of SAT solving algorithms comes to mind, performing at ever-increasing speeds in most relevant instances. But even in the realm of theory, attempts at better understanding efficient computation have led to success stories. With its fixed-parameter tractable algorithms, parameterized complexity shows that many problems are easy if we can isolate the combinatorial blowups to some concealed part of the input. Another promising approach, perhaps less known, is the subject of this thesis: *compilability*.

Compilation, sometimes referred to as *knowledge compilation* or *preprocessing*, suggests that, even if a problem is intractable in the worst case, some preprocessing might make it easier! Consider the simple example of reachability in undirected graphs. Let G be a graph, and let s and t be two distinguished nodes in G. If we are asked whether s and t are connected, the reader will jump quickly to the answer: "Easy! Use breadth-first search!" Indeed, this problem is solvable in polynomial time and looks like a boring example. But imagine for a moment that the graph G is big. Really big. And imagine that it is always the same, such as, perhaps, the simplified representation of the streets of a city on Google Maps. In such a case, where only the pair of nodes changes from instance to instance, traversing the entire graph every time is like killing flies with a cannonball.

Try the following instead. Since G is fixed in advance and does not change, preprocess it as follows: compute the connected components of G and store them in a table, such that for a given node v it is easy to identify in which connected component v belongs. This table can be computed in polynomial time and is not much bigger than the graph itself. Crucially, it only has to be computed once! The advantage is that once we receive a pair (s, t) of vertices on which to check reachability, it now suffices to check if they are in the same connected component by looking up our table! We say that the graph G has been "compiled" into a more readable representation so that queries can be easily solved "online".

Fairly enough, the previous algorithm is not that impressive: the problem was already solvable in polynomial time, so this technique might look like nothing more than a smart optimization. Can compilation help us with problems that are believed to be genuinely intractable? Are there **NP**-hard problems where preprocessing part of the input can result in efficient algorithms?

The answer is positive —modulo some concessions to the deities of intractability. Naturally, the preprocessing cannot be performed in polynomial time followed by other polynomial-time computations only, unless **P** = **NP**. The trade-off is to let the compilation be expensive. Let it run in exponential, double exponential, triple exponential time even! It does not matter how long it takes since we will only do it once. Then, store the result and use that to solve the complete instances in polynomial time. Of course, to actually store the result of the compilation in memory (and to not make it too easy for ourselves), we shall ask for the compilation to be succinct: at most polynomially bigger than the data structure being preprocessed.

The good news is that some **NP**-hard problems admit such a form of efficient preprocessing. In some other cases, the preprocessing can reduce complexity (say, from **PSPACE** to **NP**, in such a way that expensive preprocessing of part of the input reduces the instances to something easily manageable by a SAT solver [Cad+02]). However, as in most cases in complexity theory (and in life), not everything is perfect. Compilation, as powerful as it may seem, cannot overcome all forms of intractability, and many hard problems are bound to remain hard, even after the most expensive preprocessing.

Two questions arise regarding the limits of compilability. Firstly, assuming we have identified a part of the input that we consider reasonable to compile, how can we prove that the problem is (likely) not efficiently compilable? And secondly, why do structurally similar problems differ in compilability? Why is compilability not always preserved under polynomial-time reductions, leading to some **NP**-complete problems being compilable and others (likely) not?

Since the 1990s, some effort has been made to clarify issues related to the power of compilation. Initial work showed positive compilation procedures, while some initial uncompilability results also appeared around the same time —see [CDS96; Cad+97; Cad+99; Gog+95]. The latter tended to rely on ad hoc proofs for the uncompilability of specific problems, though some common features underly them all.

In 2002, Cadoli, Donini, Liberatore and Schaerf [Cad+02] observed the structural gaps created by compilability and started classifying problems attending to their respective compilability hardness under a unified framework. Their setting, which we refer to as the *CDLS framework*, presents complexity classes capturing the phenomenon of compilability. These classes are equipped with a new notion of reduction that, unlike the usual polynomial-time reduction, does preserve compilability dependencies. These new classes are themselves tied to classical classes such as **P**, **NP** or **PH**. In this way, if we prove hardness for a problem with respect to a compilability class, we can conjecture it to be uncompilable unless some of the classical classes collapse.

Soon after, Chen [Che05] challenged the assumption that the size of compilations must be polynomial and suggested looking at *parameterized compilability*. In this extended setting, compilation problems are parameterized, just like in parameterized complexity, and the size of the compilation is allowed to be of fixed-parameter size. That is, expensive in terms of the parameter in question. Chen extended the CDLS framework with new parameterized compilability classes, which were used briefly in some unpublished work by Bova et al. [Bov+16] regarding inference problems for propositional logic, and later in De Haan's doctoral dissertation [Haa19].

Ten years later, Chen [Che15] observed that compilability can be framed as a particular case of parameterized complexity, the idea being that the parameter of an instance can be seen as the compilable part of the instance. He presented new parameterized complexity classes that model efficient compilation and a unified framework for proving conditional lower bounds via hardness proofs for the new classes. This is the *parameter compilation framework*.

What all this line of work has in common is a constant push against the limits of intractability. It is

a balance between showing that alternative notions of efficient computation exist beyond the classical idea of polynomial time while acknowledging that intractability can never be totally defeated. This thesis continues this line of work, further clarifying the limits of what can and cannot be efficiently compiled.

1.1 Contributions

Let us clarify that, though the parameter compilation framework is embedded inside parameterized complexity, it is an analogue of the original CDLS framework. That is, the parameter compilation framework works under the assumption that the compilation must be polynomial in size, hence lacking methods to deal with parameterized compilation.

The main contribution of this thesis is to present an extension of the parameter compilation framework that can account for compilation of fixed-parameter size. This is a framework in which we deal with *doubly parameterized problems*. The first parameter tells us what part of the input is available for compilation, while the second one is there to give an fpt bound on the size of the compilation.

The new framework consists of a central complexity class, **fpt-comp-FPT**, capturing the notion of efficient fpt compilation, analogous to Chen's **poly-comp-P** class, which we review shortly. The new classes we introduce are equipped with a new notion of reduction, known as the *fpt-comp reductions*, which lets us relate the compilability of different doubly parameterized problems.

Though some of the ideas developed here were present in Chen's original attempt at parameterized compilability (in his 2005 paper), the chief difference is that while that work extended the CDLS framework, our setup extends the parameter compilation framework. Though the hardness results one can prove are quite similar, the parameter compilation framework unifies parameterized complexity and compilation and makes combining concepts from both fields easier. In this way, while the CDLS framework operates as an ad hoc extension of classical complexity, the parameter compilation framework can model compilation as a particular case of the well-studied phenomenon of fixed-parameter tractability, letting us import techniques and results from parameterized complexity. Furthermore, it is notationally and technically simpler.

Apart from setting up this general scaffolding for parameterized compilation, this thesis applies the new framework to various computational problems with new parameterizations. With the exception of De Haan's previous work [Haa19, Chapter 15], our study is the first to consistently approach the parameterized compilability of problems around the classes $\mathbf{W}[1]$ and $\mathbf{W}[2]$.

Furthermore, our study points to new structural gaps in the map of compilablity. In particular, we study the differences between efficient parameterizations of SAT and CSP, showing that while the problem of completing a satisfying assignment for SAT can be efficiently compiled under a simple (almost trivial) parameterization, the same parameterization cannot help for Constraint Satisfaction Problems (CSP). Moreover, we extend this study to a canonical example in compilability: inference problems. We define three new inference problems for CSP and compare their parameterized compilability to that of the canonical inference problems of propositional logic when parameterized by various treewidth measures.

Under the new setting for parameterized compilability, most of our contributions follow the approach of Figure 1.1, where we often end up answering all of the questions negatively.

In short, our contributions are mainly two:

- developing a new framework for parameterized compilability, extending the existing parameter compilation setup;
- researching various case studies where we apply our new framework to different computational problems, showing hardness results and conditional uncompilability.

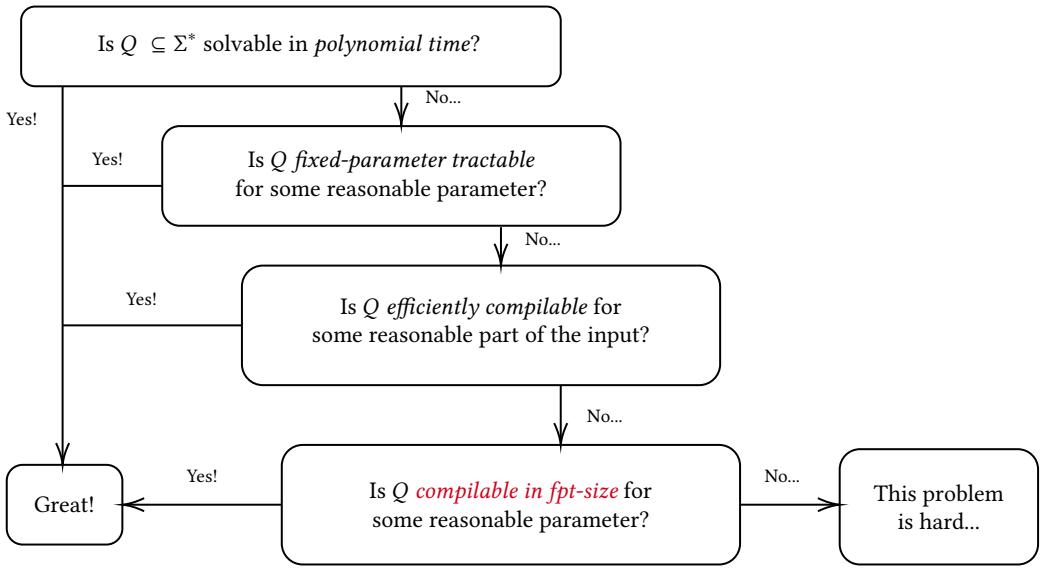


Figure 1.1: Parameterized compilability against intractability.

1.2 Structure of the thesis

The remaining of this chapter is devoted to introducing the reader to the main concepts of compilability. Section 1.3 briefly reviews some of the notation we use together with the central technical concepts from complexity theory that appear in this work. Section 1.4 is a self-contained introduction to compilability and the technicalities of the parameter compilation framework. The reader familiar with Chen's work might still want to read this before getting to Chapter 2, since the narrative and examples presented there are the guiding thread of the rest of the thesis. In any case, the reader should definitely look at the questions posed for CSP Completion at the very end of Section 1.4, since they are the primary motivation articulating our research.

Chapter 2 is devoted to the technical work of extending the parameter compilation framework. It introduces doubly parameterized problems and the classes **fpt-comp-C**, modelling fixed-parameter size compilation. Though our study focuses on doubly parameterized problems, we also shed light on some issues left uncovered in Chen's original discussion of his framework. Crucially, we introduce the methodology theorems, which we shall use through the rest of the work to relate reductions to conditional lower bounds in the power of compilability. Moreover, we give an alternative characterization of these methodology theorems in terms of hardness for a restricted class of **fpt-comp-C**, the so-called **chopped-C**.

Chapter 3 applies the new setup to doubly parameterized problems imported from **W**[1]. We consider the problems Weighted q-SAT, CSP and Clique, and introduce *completion variants*: variants of the problems where there are side constraints that cannot be compiled. We show hardness results for these completion variants under some reasonable parameterizations, showcasing the power of our fpt-comp reductions and methodology theorems.

Chapter 4 continues with the type of research of Chapter 3, this time around problems coming from **W**[2]. We define some constrained variants of the classical Hitting Set and Dominating Set and study their compilability under additional parameterizations.

Finally, Chapter 5 goes back to CSP and compares the traditional inference problems of propositional logic (Term Inference, Clause Inference and Formula Inference) to three newly defined

inference problems for CSP. While Clause Inference and Formula Inference are uncompilable in classical terms, there are some suitable parameterizations under which they become tractable. We review these results, coming from the unpublished work of Bova et al. [Bov+16], and show that, unfortunately, the same treewidth measures they use for propositional logic cannot possibly help in the case of CSP.

Chapter 6 concludes the thesis with some closing remarks and points at future lines of work. Furthermore, it summarizes all of the complexity classifications derived throughout the thesis.

1.3 Preliminaries

We assume the reader to be familiar with the central concepts of computational complexity theory and parameterized complexity, but we review some key definitions and notation below. For further discussion on computational complexity, we refer the reader to the classic textbooks of Arora and Barak [AB09] or Papadimitriou [Pap94]. For a comprehensive introduction to parameterized complexity, see the textbooks by Downey and Fellows [DF13] or Flum and Grohe [FG06]. Finally, we refer to the exposition in [Cyg+15, Chapter 7] for issues related to tree decompositions and treewidth.

Propositional logic

Throughout this work, a *propositional* or *Boolean formula* φ is a formula made from *propositional atoms* or *variables* and the connectives $\vee$, $\wedge$ and $\neg$, under the usual semantics for propositional logic. We often also use $\varphi \rightarrow \psi$ as a shorthand for $\neg\varphi \vee \psi$.

A *literal* is a variable or its negation. A Boolean formula φ is said to be in *conjunctive normal form* (CNF) if it is of the form $\varphi = C_1 \wedge \cdots \wedge C_m$, where each C_i is a *clause*, a disjuction of literals. Furthermore, we say that φ is in k-CNF if it is in CNF and every clause contains at most k literals. A Boolean formula φ in CNF is *monotone* if no literal occurs negated, while it is *antimonotone* if all literals occur negated.

For a formula φ, we denote by $\mathrm{Vars}(\varphi)$ the set of variables on which it is defined. A *(partial) assignment* or *valuation* α for a Boolean formula φ is a (possibly partial) function $\alpha : X \rightarrow \{0,1\}$, where $\mathrm{Vars}(\varphi) \subseteq X$. We often see an assignment α as a *term* (a conjunction of literals), such that α is written as $\bigwedge_{\alpha(x)=1} x \wedge \bigwedge_{\alpha(x)=0} \neg x$.

When evaluating φ under α using the usual semantics for propositional logic, we write $\alpha \models \varphi$ if φ is *satisfied under* α, and $\alpha \not\models \varphi$ otherwise.

Furthermore, the symbol $\models$ also represents *semantic entailment*. Given two propositional formulas φ and ψ, we say that φ *entails* ψ, written $\varphi \models \psi$, if and only if every valuation $\alpha : \mathrm{Vars}(\varphi) \cup \mathrm{Vars}(\psi) \rightarrow \{0,1\}$ satisfying φ also satisfies ψ.

We assume the reader to be familiar with the usual **NP**-complete and **coNP**-complete problems SAT and UNSAT, consisting, respectively, of all satisfiable and unsatisfiable formulas of propositional logic. The problem q-SAT, for $q \in \mathbb{N}$, denotes the satisfiability problem over formulas in q-CNF.

Complexity theory

Throughout this thesis we work with some arbitrary but fixed finite set Σ as an *alphabet* with $\{0,1\} \subseteq \Sigma$. A *problem* or *language* Q is a set of strings over Σ, $Q \subseteq \Sigma^*$. A *language of pairs* is a set $Q \subseteq \Sigma^* \times \Sigma^*$. We assume that under a reasonable pairing function every language of pairs can be encoded as a usual language made of single strings.

We assume the reader to be familiar with the notion of polynomial-time many-one reductions (hereafter *polynomial-time reductions*). Whenever $A \subseteq \Sigma^*$ reduces into $B \subseteq \Sigma^*$ via a polynomial-time reduction, we write $A \leq_{\mathrm{p}} B$.

We also assume the reader to be familiar with the basic complexity classes **P**, **NP**, **coNP** and **PH**, as well as with the specific levels Σ_i^{p} and Π_i^{p} of the polynomial hierarchy, for $i \in \mathbb{N}$. Though the work in this thesis generalizes to classes higher than the ones mentioned above, those are the only ones with which we deal explicitly.

For the purposes of this work, a *classical complexity class* **C** is a set of languages that contains **P** and is closed under polynomial-time reductions. That is, $\mathbf{P} \subseteq \mathbf{C}$ and for every $A, B \subseteq \Sigma^*$, if $A \leq_{\mathrm{p}} B$ and $B \in \mathbf{C}$, then it holds that $A \in \mathbf{C}$.

For a given function $f : \Sigma^* \to \Sigma^*$, we say that f is of *polynomial size* if there exists a polynomial $p : \mathbb{N} \to \mathbb{N}$ such that for every $x \in \Sigma^*$, $|f(x)| \leq p(|x|)$.

For a every classical complexity class **C**, we also consider the class **C**/poly of problems solvable within the resources of **C** with the aid of *polynomial-size advice*. That is, **C**/poly is the class of languages $Q \subseteq \Sigma^*$ for which there exists a (possibly non-computable) polynomial-size function $\alpha : \Sigma^* \to \Sigma^*$ said to provide *advice* and a language $Q_{+\alpha} \in \mathbf{C}$ such that for every $x \in \Sigma^*$,

$$x \in Q \text{ if and only if } (x, \alpha(1^{|x|})) \in Q_{+\alpha}.$$

Throughout this thesis we often invoke the classical *Karp-Lipton theorem* [KL80], a result relating advice classes to the different levels of the polynomial hierarchy. The theorem states that if $\mathbf{NP} \subseteq \mathbf{P}/\mathrm{poly}$, then it holds that $\mathbf{PH} = \Sigma_2^{\mathrm{p}}$. We shall also invoke the more general version, due to Yap [Yap83], stating that for every $i \in \mathbb{N}$, if $\Sigma_{i+1}^{\mathrm{p}} \subseteq \Pi_i^{\mathrm{p}}/\mathrm{poly}$, then $\mathbf{PH} = \Sigma_{i+2}^{\mathrm{p}}$.

Finally, the *Exponential Time Hypothesis* (ETH) of Impagliazzo, Paturi and Zane [IP01; IPZ01] is the conjecture that the infimum over all $\delta \in \mathbb{R}$ such that 3SAT can be decided in time $O(2^{\delta n})$, where n denotes the number of variables, satisfies $\delta > 0$. Intuitively, one often thinks of ETH as stating that there is no subexponential-time algorithm for 3SAT (that is, no algorithm deciding 3SAT in time $2^{o(n)}$). If ETH holds, then in particular $\mathbf{P} \neq \mathbf{NP}$.

Parameterized complexity

Let $Q \subseteq \Sigma^*$ be a language. A function $\kappa : \Sigma^* \to \Sigma^*$ is said to be a *parameterization* if for every $x \in \Sigma^*$, $\kappa(x)$ can be computed in time $f(\kappa(x)) \cdot p(|x|)$, for some computable $f : \Sigma^* \to \mathbb{N}$ and some polynomial $p : \mathbb{N} \to \mathbb{N}$. The pair (Q, κ) is then called a *parameterized language*. For every $x \in \Sigma^*$, we say that $\kappa(x)$ is the *parameter* of instance x. Often we abuse notation and denote the parameterization by the object in the input of the problem. For example, in the problem Clique getting as input a graph G and a natural number k for the size of the potential clique, we denote by $(\textsc{Clique}, k)$ the parameterized problem where the parameter is the function giving access to the value of k. More on this notation is discussed on Remark 1.3. Throughout this work we frequently use three basic parameterizations with specific notation:

- the *i-th projection function* π_i is the function $\pi_i : \Sigma^* \to \Sigma^*$ that returns the i-th component of a tuple given as input;
- the *empty parameterization* is the function $\epsilon : \Sigma^* \to \Sigma^*$ mapping every string to the empty string;
- the function $\mathrm{len} : \Sigma^* \to \Sigma^*$ is the function giving a string with the length of the input; that is, $\mathrm{len} : x \mapsto 1^{|x|}$.

We say that a parameterized language (Q, κ) is *fixed-parameter tractable* (fpt) if there exists a deterministic Turing machine, a computable function $f : \Sigma^* \to \mathbb{N}$ and a polynomial $p : \mathbb{N} \to \mathbb{N}$ such that on input $x \in \Sigma^*$, the machine decides whether $x \in Q$ in time $f(\kappa(x)) \cdot p(|x|)$. We then say that such an algorithm runs in *κ-fpt-time*. In particular, a parameterization κ is *fpt-computable with respect to itself*.

We denote by **FPT** the class of all parameterized languages that are fixed-parameter tractable. Furthermore, every classical complexity class **C** induces a *parameterized complexity class* **para-C**, containing all parameterized languages (Q, κ) for which there exists a computable function $c : \Sigma^* \to \Sigma^*$ and a language of pairs $Q_{+c} \in \mathbf{C}$ such that for every $x \in \Sigma^*$, $x \in Q$ if and only if $(x, c(\kappa(x))) \in Q_{+c}$. We often say that c is *compiling* $\kappa(x)$. In particular, it holds that **FPT** = **para-P** [FG06, Theorem 1.37].

Given two parameterized languages (A, κ) and (B, λ), we say that (A, κ) *fpt-reduces into* (B, λ), written $(A, \kappa) \leq_{\text{fpt}} (B, \lambda)$, if there exists an κ-fpt-time computable function $f : \Sigma^* \to \Sigma^*$ and a computable function $s : \Sigma^* \to \mathcal{P}_{\text{fin}}(\Sigma^*)$ such that for every $x \in \Sigma^*$,

(i) $x \in A$ if and only if $f(x) \in B$;

(ii) $\lambda(f(x)) \in s(\kappa(x))$.

and where $\mathcal{P}_{\text{fin}}(\Sigma^*)$ denotes the finite power set of Σ^*, that is, the set of all finite subsets of Σ^*.

The first condition in the definition of fpt reductions is known as *correctness*, while the second one ensures that *parameter dependencies are preserved*. That is, it requires that the parameter of the reduced instance is bounded by a function depending solely on the old parameter. Note that ours is somewhat unusual notation, since most texts in parameterized complexity often consider parameters to be natural numbers. The reason we use strings is that it lets us more naturally parameterize over different parts of the inputs, without having to first encode objects as natural numbers.

For a given complexity class **C** (parameterized or otherwise) and a notion of reduction $\preccurlyeq$ under which **C** is closed, we say that a problem is *hard* for **C** if every other problem in **C** reduces to it via $\preccurlyeq$. We say that the problem is **C**-*complete* if it is hard and it is in **C**. For a given problem Q (parameterized or otherwise) and a suitable notion of reduction $\preccurlyeq$, we denote by $[Q]^{\preccurlyeq}$ the set of all problems that reduce into Q via $\preccurlyeq$. Like for classical complexity, when quantifying over an arbitrary parameterized complexity class **C**, we assume **FPT** $\subseteq$ **C** and **C** to be closed under fpt reductions.

Throughout this thesis, we work extensively with the *Weft hierarchy* (or **W** *hierarchy*). In particular, we focus on the first two levels of the hierarchy, $\mathbf{W}[1]$ and $\mathbf{W}[2]$. We refer the reader to the expositions by Downey and Fellows [DF13, Chapters 21-25] or Niedermeier [Nie06, Chapter 13] for the rather technical formal definition of these complexity classes. Here we just note some of the canonical problems in these classes. For $\mathbf{W}[1]$, the problem (Weighted q-SAT, k) for every $q \geq 2$ is $\mathbf{W}[1]$-complete, and so is the problem restricted to antimonotone formulas, (Weighted Antimonotone q-SAT, k). The well-known graph-theoretic problems (Clique, k) and (Independent Set, k) are also $\mathbf{W}[1]$-complete. For $\mathbf{W}[2]$, the canonical complete problem is (Weighted CNF SAT, k), as well as (Hitting Set, k) and (Dominating Set, k). For readers not familiar with these problems, see, again, Chapter 13 in [Nie06].

The central conjecture of parameterized complexity is that **FPT** $\neq$ $\mathbf{W}[1]$ —often regarded as the parameterized analogue of **P** $\neq$ **NP**. Interestingly, if the Exponential Time Hypothesis holds, then **FPT** $\neq$ $\mathbf{W}[1]$ [DF13, Chapter 29.2]. Or, using the usual contrapositive phrasing of complexity theorists, "**FPT** $\neq$ $\mathbf{W}[1]$ unless ETH fails".

We can also consider parameterized complexity classes with access to *fpt-size advice*. We refer the reader to Definition 2.18 later in the main text for a formal definition of classes of the form **C**/fpt, for a parameterized class **C**. Analogously to how **FPT** $\neq$ $\mathbf{W}[1]$ is regarded as the parameterized phrasing of **P** $\neq$ **NP**, we see $\mathbf{W}[1] \subseteq$ **FPT**/fpt as the unlikely analogue of the inclusion **NP** $\subseteq$ **P**/poly. Unfortunately, there is no immediate translation of the Karp-Lipton theorem to the **W** hierarchy. That is, we do not know whether $\mathbf{W}[1] \subseteq$ **FPT**/fpt implies that the entire hierarchy collapses to $\mathbf{W}[2]$. In fact, it is an open question whether local collapses in the **W** hierarchy also imply a global collapse, as it happens for the polynomial hierarchy. De Haan [Haa19, Chapter 15.4] has studied the issue of parameterized analogues of the Karp-Lipton theorem, but the existing versions involve other more sophisticated complexity classes that do not play a role in this thesis.

Constraint Satisfaction Problems (CSP)

A *constraint satisfaction problem* (CSP) is a triple $\langle X, D, C\rangle$ consisting of a finite set X of *variables*, a finite *domain* D and a finite set $C = \{C_1, \ldots, C_m\}$ of *constraints*. A constraint $C_i \in C$ is a pair (X_i, R_i) consisting of a sequence $X_i = (x_1, \ldots, x_k)$ of variables in X together with a *relation* $R_i \subseteq D^k$. We say that k is the *arity* of the constraint C_i.

Let $X' \supseteq X$. A total function $\alpha : X' \to D$ is said to be a *satisfying assignment* for $\langle X, D, C\rangle$ if it holds that for every $C_i = (X_i, R_i) \in C$, with $X_i = (x_1, \ldots, x_k)$, $(\alpha(x_1), \ldots, \alpha(x_k)) \in R_i$. If a CSP instance has a satisfying assignment, we say that the CSP instance is *satisfiable*. If α is not a total function, we say it is a *partial assignment*.

The set CSP of all satisfiable constraint satisfaction problems is an **NP**-complete problem. Membership in **NP** follows from the fact that a satisfying assignment is a certificate verifiable in polynomial time. Hardness follows by reducing from 3SAT. Every 3-CNF formula φ can be mapped to a CSP instance having as variables the atoms of φ, as domain the set $\{0, 1\}$ and where every constraint corresponds to a clause, with the relation encoding the valuations that make the clause true. Since all clauses contain at most three literals, the relation of each constraint is of size at most the constant $2^3 = 8$, so the reduction can be carried out in polynomial time. We refer back to this translation of Boolean formulas into CSP a few times throughout this thesis.

Graphs and treewidth

We review some of the notation we use when dealing with graphs. A *graph* G is a pair (V, E), where V is a finite set of *vertices* or *nodes* and $E \subseteq V \times V$ is a set of *edges*. We mostly work with *undirected* graphs, meaning that if $(u, v) \in E$, we also have $(v, u) \in E$. We sometimes use $|G|$ to refer to the number $|V|$ of vertices in G. For a given subset $S \subseteq V$, we denote by $G[S]$ the *subgraph induced by* S, that is, the subgraph $G' = (S, E')$, where $E' = \{(u, v) \in E \mid u, v \in S\}$. Finally, for a given graph $G = (V, E)$ and a vertex $v \in V$, we denote by $N_G(v)$ the *neighborhood of* v, the set $\{u \in V \mid (v, u) \in E\} \cup \{v\}$ of vertices adjacent to v, together with v itself.

A *tree* is a connected undirected acyclic graph. A *tree decomposition* of a graph $G = (V, E)$ is a pair $(T = (V_T, E_T), \{B_t\}_{t \in V_T})$ consisting of a tree T and a set of *bags*, one subset $B_t \subseteq V$ for each vertex $t \in V_T$ in the tree, such that the following three properties hold:

(i) *Vertex coverage*: for every $v \in V$, there is a $t \in V_T$ such that $v \in B_t$;

(ii) *Edge coverage*: for every $(u, v) \in E$, there is a $t \in V_T$ such that $\{u, v\} \subseteq B_t$;

(iii) *Connectivity*: for every $t, t' \in V$, if a vertex $v \in V$ appears both in B_t and $B_{t'}$, when v also appears in every B_s, for every s on the path from t to t' in T.

The *width* $\mathrm{w}(T)$ of a tree decomposition T of G is defined as the maximum number of elements appearing in any bag of the decomposition, minus one. That is,

$$\mathrm{w}(T) := \max_{t \in V_T} |B_t| - 1.$$

Finally, the *treewidth* of a graph G, denoted $\mathrm{tw}(G)$, is the minimum w such that there is a tree decomposition of G of width w. It is useful to note that a complete graph of n nodes has treewidth $n-1$, while a tree has unit treewidth.

Deciding whether a graph G has treewidth w is **NP**-complete. However, it can be solved in fpt-time when parameterized by the treewidth itself. See Chapter 7 in [Cyg+15] for more on treewidth and fpt algorithms exploiting tree decompositions.

1.4 Compilability: a tour d'horizon

Let us consider three simple problems related to propositional logic that will motivate and guide our exposition on compilability: Term Inference, Clause Inference and Formula Inference. The first one takes as input a propositional formula φ together with a term (a conjunction of literals $\ell_1 \wedge \cdots \wedge \ell_k$) and asks whether φ semantically entails $\ell_1 \wedge \cdots \wedge \ell_k$. The second problem is identical but for clauses: does $\varphi \models C$, where C is a clause? The third problem, Formula Inference, is a bit more general. It consists of two propositional formulas φ and ψ, asking whether $\varphi \models \psi$. It is not difficult to show that all three of these problems are quite hard: they are all **coNP**-complete and hence not solvable in polynomial time unless **P** = **NP**.

Though seemingly toy puzzles in propositional logic, these inference problems are central to computer science and artificial intelligence. Imagine a propositional knowledge base encoded into the formula φ in such a way that the query $\varphi \models \psi$ is an inference some AI needs to carry out. We would like our AI to perform such an inference as rapidly as possible. Furthermore, we can probably assume that φ is some big but fixed knowledge base that does not change over time, while we query it with different formulas every time.

This is precisely the context in which compilation can be helpful! We might be willing to invest in some expensive computation preprocessing φ into a friendlier format (say, some more readable formula φ') in such a way that $\varphi \models \psi$ if and only if $\varphi' \models \psi$ and such that the latter entailment can be decided in polynomial time! The only constraint we will impose in such a preprocessing is that the formula φ' should be at most polynomially bigger than φ so that we can efficiently store it in our machine.

Luckily for us, Term Inference admits such a form of preprocessing! The idea is simple. Recall that the problem instances are of the form $(\varphi, \ell_1 \wedge \cdots \wedge \ell_k)$, where we ask whether $\varphi \models \ell_1 \wedge \cdots \wedge \ell_k$, and we allow ourselves unlimited computational power to preprocess φ. We can do the following:

1. For every variable x occurring in φ, check whether $\varphi \models x$ and whether $\varphi \models \neg x$. Note down all the literals for which entailment holds.

2. To decide whether $\varphi \models \ell_1 \wedge \cdots \wedge \ell_k$, check whether all of the literals $\ell_1, \ldots, \ell_k$ appear in the table compiled in the previous step. If so, accept; else, reject.

Naturally, the first step will be expensive: on a formula φ over n variables, the brute-force approach to check entailment is to verify each of the 2^n valuations, and we must repeat such a check for each of the $2n$ possible literals. However, the table we compile is of size $O(n)$ and the second step can be computed in polynomial time, so this is an efficient form of preprocessing. We say that Term Inference is *efficiently compilable*.

After this initial success, one may try to extend this approach to the more general Formula Inference, where the query can be any propositional formula and not just a conjunction of literals. In this more general scenario, there does not seem to be any structure we can exploit... So much so that we do not think there is an efficient compilation procedure for this problem. Formula Inference is very likely to be uncompilable.

Theorem (Cadoli et al. [Cad+02, Theorem 2.4])**.** Formula Inference *with compilation access to the first formula is not efficiently compilable unless* **P** = **NP**.

This result is not difficult to prove. What seems interesting from a complexity-theoretic point of view is a form of structural gap that appears inside the class **coNP**. Recall that both Term Inference and Formula Inference are **coNP**-complete; hence, classical complexity theory sees these two problems as almost identical in structural terms. Yet, as soon as we relax our notion of efficient computation to one including compilation power, one problem stays intractable, while the other becomes easy!

Let us have a closer look at what is going on. One naïve idea to compile instances of Formula Inference would be to simply reduce instances of Formula Inference to instances of

Term Inference and then apply the compilation procedure described above. This should provide efficient compilation for Formula Inference... but it does not. The issue is with what the polynomial-time reductions between the two problems are actually doing. Essentially, on input (φ, ψ) to Formula Inference, it holds that

$$\varphi \models \psi \text{ if and only if } \neg(\varphi \to \psi) \models \bot$$

and $(\neg(\varphi \to \psi), \bot)$ can be seen as an input to Term Inference. This is technically a polynomial-time reduction, but the trick is that we moved ψ to the left-hand side of the entailment, so our compilation procedure is now preprocessing both φ and ψ! This is not a reasonable compilation since allowing unlimited computation for the entire instance obviously solves every possible problem. We say that the reduction *does not preserve compilability dependencies*.

In 2002, Cadoli, Donini, Liberatore and Schaerf [Cad+02] observed this structural gap and presented a comprehensive complexity-theoretic framework to study these differences more closely, the already mentioned CDLS framework. Though our contributions do not use the CDLS toolbox formally, the underlying ideas are essentially theirs.

Cadoli et al. model compilation as a form of reduction. For them, problems amenable to compilation must be *languages of pairs*, that is, problems where there are clearly two parts in the input: the *offline part* and the *online part*, where we are only allowed to compile the former. The compilation is carried out by some function whose output cannot be much longer than the input. The online part, together with the compiled offline instance, must be solvable in polynomial time. Compilation is seen as a form of reduction because one reduces to the language consisting of the instances padded with the compilation. The class of problems amenable to such a form of efficient compilation is called $\rightsquigarrow$**P**: the class of problems "compilable to **P**".

1.4.1 The parameter compilation framework

In their original paper, Cadoli et al. already observed that compilation seems conceptually related to another form of efficient computation: fixed-parameter tractability. The link with parameterized complexity would be that, like in compilability, parameterized algorithms are allowed to be "expensive" in some restricted part of the input (the parameter). Furthermore, the structural gap in compilation is reminiscent of a well-known gap in the hardness theory of parameterized complexity. Namely, the fact that the problems Vertex Cover and Clique are both **NP**-complete and hence interreducible under polynomial-time reductions, yet (Vertex Cover, k) $\in$ **FPT**, while (Clique, k) $\notin$ **FPT** unless the Exponential Time Hypothesis fails [DF13, Chapter 29.2]. This is a similar type of separation to the one between Term Inference and Formula Inference.

Interestingly, one can give an account of fixed-parameter tractability in terms of compilation! Every classical complexity class induces a parameterized class via compilation.

Definition 1.1 (**para-C** [FG03])**.** Let **C** be a classical complexity class. We denote by **para-C** the class of parameterized languages (Q, κ) for which there is a computable function $c : \Sigma^* \to \Sigma^*$ and a language $Q_{+c} \in$ **C** such that for all $x \in \Sigma^*$,

$$x \in Q \text{ if and only if } (x, c(\kappa(x))) \in Q_{+c}.$$

It is not difficult to show that **FPT** = **para-P** [FG06, Theorem 1.37], and hence fixed-parameter tractability can be characterized in terms of compilation. However, the existing characterization of **FPT** as **para-P** only requires the compilation function to be computable, not polynomially bounded in size. Hence, we cannot guarantee that the compilation is space-efficient.

In 2015, Chen devised a framework that makes such an assumption explicit: the *parameter compilation framework* [Che15]. He considers the class of parameterized problems having compilation in

the **para-C** sense, but where the compilation function is forced to be polynomial in size. The resulting framework is an alternative to CDLS, where efficient compilation can be effectively modelled as a particular case of fixed-parameter tractability.

The restrictions imposed to get space-efficient compilation are two. First, we will work with parameterized problems (Q, κ) where $\kappa : \Sigma^* \to \Sigma^*$ is a polynomial-time function. Intuitively, κ extracts from the input the part of the instance that can be compiled. It should not do much more computation than some simple "reading of the input", hence the restriction that it works in polynomial time. Secondly, we shall consider a subset of **para-P**, where compilations are of polynomial size.

Definition 1.2 (**poly-comp-P** [Che15])**.** We denote by **poly-comp-P** the class of parameterized problems (Q, κ) where κ is computable in polynomial time and for which there exists a polynomial-size computable function $c : \Sigma^* \to \Sigma^*$ and a language of pairs $Q_{+c} \in \mathbf{P}$ such that for all $x \in \Sigma^*$,

$$x \in Q \text{ if and only if } (x, c(\kappa(x))) \in Q_{+c}.$$

Of course, nothing in the previous definition is specific about **P**. For every complexity class **C** closed under polynomial-time reductions, one can define the class **poly-comp-C** of problems that can be solved within the resources of **C** after some possibly expensive precomputation. Observe that this is indeed a restriction of the **para-C** classes. That is, for every **C**, **poly-comp-C** $\subseteq$ **para-C**.

Under this setting, the class **poly-comp-P** accounts for efficient compilation and is the analogue of $\rightsquigarrow$**P** in the CDLS framework. As an example, we would say that (Term Inference, π_1) $\in$ **poly-comp-P**. However, for the sake of readability, we shall use the following notation, (Term Inference, φ), to refer to that problem.

Remark 1.3 (Notation for parameterizations). For a given computational problem, we often abuse notation and denote a parameterization directly by the symbols referring to the object in the problem's input. For example, consider the problem Term Inference. The input to Term Inference is a pair (φ, T) consisting of a propositional formula φ together with a term T. Then, the problem (Term Inference, π_1) would give us compilation access to φ but not to T. For the sake of readability, we will be writing (Term Inference, φ) instead.

Often we also use the bracket notation $\langle\cdot\rangle$ to refer to parameterizations that give us access to multiple things simultaneously. For example, consider the problem Clause Inference, getting as input pairs (φ, C) of propositional formulas and clauses. A possible parameterization for compilation may be giving access to the size of the formula together with the size of the clause. We would denote such a problem by (Clause Inference, $\langle|\varphi|, |C|\rangle$). ■

Chen's parameter compilation framework is equipped with a new notion of reducibility: poly-comp reductions. In the same way that **poly-comp-P** is a restriction of **FPT**, the so-called poly-comp reduction is a particular case of the usual fpt reduction.

Definition 1.4 (poly-comp reductions [Che15])**.** We say that (A, κ) *poly-comp-reduces* to (B, λ), written $(A, \kappa) \leq^{\text{poly}}_{\text{comp}} (B, \lambda)$, if there is

- a polynomial-size computable function $c : \Sigma^* \to \Sigma^*$,
- a polynomial-time computable function $f : \Sigma^* \times \Sigma^* \to \Sigma^*$

such that for all $x \in \Sigma^*$,

$$x \in A \Leftrightarrow f(x, c(\kappa(x))) \in B$$

and there is a polynomial-size function $s : \Sigma^* \to \mathcal{P}_{\text{fin}}(\Sigma^*)$ such that

$$\lambda(f(x, c(\kappa(x)))) \in s(\kappa(x)).$$

Recall that, as we saw, polynomial-time reductions do not necessarily preserve the dependencies between the compilable parts of the inputs. Observe instead how poly-comp reductions do preserve the dependencies between the online and the offline parts of the instances! This is precisely the goal of the requirement that $\lambda(f(x, c(\kappa(x)))) \in s(\kappa(x))$. That is, s should be able to generate all the possible needed compilations, which cannot be too many.

Clearly, the polynomial-time reduction we described earlier from Formula Inference to Term Inference does not fit in this definition! Naturally, the new **poly-comp-C** classes are closed under this new notion of reduction [Che15, Theorem 12], and poly-comp reducibility is transitive [Che15, Theorem 13]. Furthermore, it is easy to find complete problems for **poly-comp-C** under poly-comp reductions. It turns out that if Q is **C**-complete, then the problem (Q, ϵ), where we do not have compilation power because we use the empty parameterization, is complete for **poly-comp-C**.

Proposition 1.5. *If Q is* **C***-complete (under polynomial-time reductions), then (Q, ϵ) is* **poly-comp-C***-complete (under poly-comp reductions).*

Proof. Membership is clear. For hardness, let $(R, \kappa) \in$ **poly-comp-C**. By definition of **poly-comp-C**, there is a computable c and a language of pairs $R_{+c} \in$ **C** such that for every $x \in \Sigma^*$, $x \in R$ if and only if $(x, c(\kappa(x))) \in R_{+c}$. Under the perspective of compilation as reduction, we have that

$$(R, \kappa) \leq^{\text{poly}}_{\text{comp}} (R_{+c}, \epsilon)$$

and $(R_{+c}, \epsilon) \leq^{\text{poly}}_{\text{comp}} (Q, \epsilon)$, by the fact that $R_{+c} \in$ **C** and Q is **C**-complete. □

This lets us prove the theorem we stated earlier, using the tools of the parameter compilation framework.

Theorem 1.6. Formula Inference *is* **poly-comp-coNP***-complete (under poly-comp reductions) and hence not in* **poly-comp-P** *unless* **P** = **NP**.

Proof. We show that

$$(\text{UNSAT}, \epsilon) \leq^{\text{poly}}_{\text{comp}} (\textsc{Formula Inference}, \varphi) \leq^{\text{poly}}_{\text{comp}} (\textsc{Formula Inference}, \epsilon).$$

The first reduction gives hardness, since by the previous proposition (UNSAT, ϵ) is **poly-comp-coNP**-complete. See that

$$\varphi \in \text{UNSAT} \Leftrightarrow \top \models \neg\varphi \Leftrightarrow (\top, \neg\varphi) \in \textsc{Formula Inference}.$$

The second reduction is immediate and gives membership (since $(\textsc{Formula Inference}, \epsilon) \in$ **poly-comp-coNP** and the class is closed under poly-comp reductions). Observe how compilability dependencies are preserved now!

Now, suppose Formula Inference $\in$ **poly-comp-P**. Then, **poly-comp-P** = **poly-comp-coNP**, and since $(\text{UNSAT}, \epsilon) \in$ **poly-comp-P** that means there is a polynomial-time algorithm for UNSAT, since having access to ϵ does not help in compilation. Hence **P** = **coNP** = **NP**. □

The reader may feel suspicious of how easy the previous proof was. The suspicion is somewhat confirmed: while the previous technique gives a strong conditional lower bound for Formula Inference, most other problems cannot be dealt with in this way.

The chief example is Clause Inference. This time the input is a propositional formula φ together with a clause $C = \ell_1 \vee \cdots \vee \ell_k$, and the goal is to decide whether $\varphi \models C$. Although Clause Inference is also **coNP**-complete, it cannot be **poly-comp-coNP**-complete unless **P** = **NP** [Cad+02, Theorem 2.5]. Is it because Clause Inference is efficiently compilable? Not really. In [SK96], Selman and Kautz showed

that Clause Inference is not compilable unless the polynomial hierarchy collapses to its second level! The idea is essentially the following: we allow compilation access to the size of the entire input, including the online part. In this way, having polynomial-size compilation means having *polynomial-size advice*. If one can then show that SAT having compilation access to the length of the input reduces to, say, Clause Inference, and Clause Inference is in **poly-comp-P**, then SAT can be solved in polynomial time with polynomial-size advice. By the Karp-Lipton theorem, the polynomial hierarchy collapses. The only issue is that such a reductiom needs to map every 3-CNF formula φ to an equisatisfiable pair (ψ, α), in such a way that ψ can somehow be computed just by knowing the length of φ, which seems impossible at first glance. The idea of Selman and Kautz is to build a "superinstance" that contains all the components needed to represent any formula of a fixed length. Then, the partial assignment α can assign values to some of the atoms so that the superinstance is "configured" to represent exactly φ. We call this the "superinstance technique", or "carving technique"

Let us reproduce this proof inside the parameter compilation framework to showcase the superinstance technique. Observe that giving compilation access to the length of the input can be done using the parameterization $\text{len} : x \mapsto 1^{|x|}$. The reduction in point is from 3SAT to the problem SAT Completion, defined as follows (and known as Constrained SAT in the paper by Cadoli et al.).

SAT Completion	
Instance	A Boolean formula φ and a partial assignment α.
Question	Is there a satisfying assignment for φ extending α?

The idea would be that on input (φ, α), we allow compilation access to φ but not to α. That is, we study the problem (SAT Completion, φ).

Theorem 1.7 (Selman and Kautz [SK96, Theorem 2.5.3.1], rephrased)**.** $(3\text{SAT}, \text{len})$ *reduces into* (SAT Completion, φ) *via poly-comp reductions.*

Proof. The key observation is that a 3-CNF formula over n variables cannot have more than $O(n^3)$ clauses. Indeed, there are at most $\binom{2n}{3} \in O(n^3)$ clauses over three literals. Thus, we can easily build the set of all such possible clauses: $C := \{C_1, C_2 \ldots, \}$ and consider the superinstance formula

$$\psi_n := \bigwedge_{C_i \in C} (\neg c_i \vee C_i)$$

where the lower case c_i are fresh variables, one per clause $C_i \in C$.

The formula ψ_n can be computed just by knowing the number n of variables of φ, which is in turn bounded by its length! Then, consider the partial assignment α that configures the superinstance. For each $C_i \in C$, if C_i appears in φ, then α maps $c_i \mapsto 1$; if, on the other hand, C_i is not in φ, it maps $c_i \mapsto 0$. The original n variables are then left unassigned. It is straightforward to see that φ is satisfiable if and only if $\psi_n \wedge \alpha$ is satisfiable.

Besides, one can easiliy formalize this as a poly-comp reduction. The function c does nothing, and all the work is carried out by f, which is in charge of building ψ_n and defining the partial assignment α. Then, the function s simply builds all possible superinstances up to the length of the input: $\psi_n \in s(\text{len}(\varphi)) = \{\psi_i \mid i \leq |\text{len}(\varphi)|\}$, where s is clearly computable and polynomial-size. □

The following is an immediate consequence.

Theorem 1.8. (SAT Completion, φ) $\notin$ **poly-comp-P** *unless* $\mathbf{NP} \subseteq \mathbf{P}/\text{poly}$.

By the Karp-Lipton theorem, we get that (SAT Completion, φ) $\notin$ **poly-comp-P** unless $\mathbf{PH} = \Sigma_2^{\text{p}}$. This in turn lets us prove the conditional lower bound for Clause Inference mentioned earlier.

Theorem 1.9. (Clause Inference, φ) $\notin$ **poly-comp-P** *unless* $\mathbf{PH} = \Sigma_2^{\mathrm{p}}$.

Proof. It suffices to present the reduction

$$(\overline{\text{SAT Completion}}, \varphi) \leq_{\mathrm{comp}}^{\mathrm{poly}} (\text{Clause Inference}, \varphi)$$

since (SAT Completion, φ) is now known to us to be (conditionally) uncompilable. Simply observe that a pair $(\varphi, \alpha) \notin$ SAT Completion if and only if $\varphi \models \neg\alpha$. Every assignment that makes φ true disagrees with α at some variable. Since α can be seen as a conjunction of literals, $\neg\alpha$ is a clause, and so $(\varphi, \neg\alpha)$ is a valid input to Clause Inference. Note how φ stays the same, and compilability dependencies are preserved: the compilable part of the new input depends exclusively on the compilable part of the old input. This completes the reduction.

Now, if (Clause Inference, φ) $\in$ **poly-comp-P**, then so is (SAT Completion, φ). But then so is (3SAT, len), which is to say $\mathbf{NP} \subseteq \mathbf{P}/\mathrm{poly}$. By the Karp-Lipton theorem, $\mathbf{PH} = \Sigma_2^{\mathrm{p}}$. □

The previous approach to lower bounds can be phrased more generally as follows.

Theorem 1.10 (Methodology theorem [Che15])**.** *If* $(A, \mathrm{len}) \leq_{\mathrm{comp}}^{\mathrm{poly}} (B, \lambda)$ *for some* **C***-hard problem* A *and* $(B, \lambda) \in$ **poly-comp-C**$'$*, then* $\mathbf{C} \subseteq \mathbf{C}'/\mathrm{poly}$.

Superinstance reductions and the methodology theorem are the central tools to show uncompilability results. In order to rephrase such results in terms of hardness for some complexity class, Chen defines the following subclasses of **poly-comp-C**.

Definition 1.11 (**chopped-C** classes for classical **C** [Che15])**.** Let **C** be a classical complexity class. We define **chopped-C** as the class

$$\textbf{chopped-C} := \bigcup_{Q \in \mathbf{C}} [(Q, \mathrm{len})]^{\leq_{\mathrm{comp}}^{\mathrm{poly}}}.$$

The name of the classes comes from the intuition that, if you are in **chopped-C**, you reduce to some (Q, len) for $Q \in \mathbf{C}$ and you can hence compile the length of the input; that is to say, each "chop" of the language is mapped to the same instance. Hence, "chopped"-**C**.

It is not difficult to show that if A is **C**-hard, then (A, len) is **chopped-C**-hard. As a result, the following classifications immediately follow from the previous reductions.

Corollary 1.12. (SAT Completion, φ) *is* **chopped-NP***-complete and* (Clause Inference, φ) *is* **chopped-coNP***-complete.*

1.4.2 Beyond polynomial-size compilation

Our discussion of compilation so far made an unchallenged assumption: the precomputation involved in the compilation can be as expensive as desired, but its output must be succinct (at most polynomially bigger). Interestingly, the parameter compilation framework is actually defined in greater generality to allow the possibility of, say, exponential-size compilation. In fact, for every class $\mathcal{F}$ of functions used to bound the length of strings, we can imagine the class $\mathcal{F}$-**comp-C** of problems that can be solved within the resources of **C** with compilation bounded by some function in $\mathcal{F}$.

In particular, one may imagine the class **exp-comp-P** of problems efficiently compilable with exponential-size compilations. Many problems that are hard for classical compilation immediately become easy under this new definition. For example, (SAT Completion, φ) $\in$ **exp-comp-P**: given the formula φ, write down all its possible valuations. This compilation is exponential in size, and with it, one can easily check whether the partial assignment can be extended into a complete satisfying assignment.

Fairly enough, calling this "efficient compilation" might be stretching the term a bit too much. However, consider the following scenario, also around SAT Completion. We justified expensive computation on φ assuming that the formula stays fixed and is always the same. We could then further assume that all the partial assignments also share a lot in common. Perhaps they are all almost complete assignments, in the sense that the number of variables left unassigned is always very small. In such a scenario, one may wonder whether we could take the number of variables left unassigned, u, as a *parameter*, in the sense of parameterized complexity, and let the compilation be of fpt-size: of length $f(u) \cdot \text{poly}(|\varphi|)$, for some computable function $f : \mathbb{N} \to \mathbb{N}$.

It is easy to see that SAT Completion with this additional parameter u is "fpt-compilable": the compilation contains all the possible 2^u valuations to the variables left unassigned, and it is a matter of trying them all! Since we assume u to be small, this might be a more reasonable notion of efficient compilation beyond the traditional polynomial-size requirement.

The reader may be worried, however, that allowing this sort of parameterization trivializes the theory. But this does not seem to be the case in general. Consider, instead of SAT, the more general CSP, where variables are interpreted in a domain that could be bigger than binary. Analogously to SAT Completion, define CSP Completion as the problem consisting of pairs (I, α) where $I = \langle X, D, C \rangle$ is a CSP instance and $\alpha : X \to D$ is a partial assignment to I, the goal being to provide a complete satisfying assignment extending α.

Given that (SAT Completion, φ) is **chopped**-**NP**-complete and that SAT is just a particular case of CSP, it is evident that (CSP Completion, I) is likely not efficiently compilable. In fact, it is also **chopped**-**NP**-complete. What happens if we allow this extra power, adding the number u of variables left unassigned by α as a parameter? The trick used for SAT Completion no longer works. There the number of valuations was 2^u, and we could just write them all. For CSP Completion, the number of possible completions of the assignment is $|D|^u$, where D is the domain of the CSP instance! But we are only allowed to be expensive in u, not in $|D|$, so the same strategy no longer works.

The natural question is whether we can show this formally. That is, can we formally show that CSP Completion does not allow "fpt-size" compilation under this additional parameterization? The parameter compilation framework cannot account for this, but it could be extended into a new framework where such a form of doubly parameterized problems can be analyzed.

This is precisely the goal of the thesis: to provide a new framework for parameterized compilability, extending Chen's parameter compilation system. Under this new setting, we shall develop tools to classify the complexity of CSP Completion parameterized by this additional parameter u and a variety of other problems.

Chapter 2

A new framework for parameterized compilability

We start by extending Chen's parameter compilation framework, the goal being to provide a complexity-theoretic account of compilation of fixed-parameter size, further extending the connection between compilability and parameterized complexity.

Our motivating question is the one we posed for CSP Completion: is it of any help to allow the compilation to be slightly bigger? What happens if we add as a second parameter the number u of variables left unassigned by the partial assignment in the input to CSP Completion? Does letting the compilation be "expensive in u" provide any benefit? This additional power makes SAT Completion efficiently compilable, but at first glance it does not seem to do the same for CSP. We would like an extension of the parameter compilation framework that can let us classify the compilation complexity of problems consisting of two parameters: the first one indicating the part that can be compiled and the second one indicating how much bigger the size of the compilation can get.

Recall that the original parameter compilation framework already contemplates the possibility of compilations having different sizes. Although **poly-comp-P** is the central class, the framework also consists of classes like **exp-comp-P** allowing exponential-size compilation. Intuitively, what we now want is a class like **fpt-comp-P**.

Section 2.1 addresses this issue, defining the **fpt-comp-C** classes as the natural extension of **poly-comp-C**. Section 2.2 presents an extended notion of reduction that can be used to show hardness results around these classes. Section 2.3 discusses how our notion of reduction can be used to prove conditional lower bounds on the hardness of our doubly parameterized problems. Unlike in the parameter compilation framework, where lower bounds rely on classical complexity-theoretic conjectures, the extended framework will rely on conjectures about the parameterized world. As in the parameter compilation framework, our notion of reduction will often achieve completeness with regards to some restricted subclasses, the **chopped-** classes, defined in Section 2.4.

2.1 The new fpt-comp-C classes

Our primary motivation in extending Chen's parameter compilation framework is to account for problems with two parameters. The first one should tell us what part of the input is available for precomputation, while the second parameter should be there to let the compilation be fixed-parameter-size with respect to it. When appending these two types of parameters to a problem, we will be talking about *doubly parameterized problems*.

Definition 2.1 (Doubly parameterized languages)**.** Let $Q \subseteq \Sigma^* \times \Sigma^*$ be a language, $\kappa : \Sigma^* \to \Sigma^*$ a polynomial-time function and $\lambda : \Sigma^* \to \Sigma^*$ a parameterization (a function fpt-computable with respect to itself). Such a triple (Q, κ, λ) is called a *doubly parameterized language*.

Note how κ is required to be polynomial-time computable, while λ is fpt-computable with respect to itself. The motivation behind this is that κ indicates what part of the input we are allowed to compile, and hence we should be able to access that information easily, while λ is there to bound the size of the compilation, so it does not matter if it is harder to compute.

In the remaining of this chapter, we will be dealing with the framework itself and not with any particular problems, so we will use lower case Greek letters to denote parameterizations. For convenience, the later chapters adopt the more readable notation presented in Remark 1.3.

Our idea is then that, when looking at a doubly parameterized language, on input x we will perform some expensive precomputation on $\kappa(x)$ whose outcome can be of size $f(\lambda(x)) \cdot \text{poly}(|\kappa(x)|)$ for some computable function $f : \Sigma^* \to \mathbb{N}$. We formalize this by saying that the compilation function is of fixed-parameter size with respect to λ, or simply "fpt-length in size".

Definition 2.2 (fpt-length functions)**.** Let $\mu : \Sigma^* \to \Sigma^*$ be a computable function and let $f : \Sigma^* \to \Sigma^*$ be a function. We say that f is of *μ-fpt-length*, *μ-fpt-bounded in size* or *fpt-bounded in $\mu(x)$* if there is a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a computable function $h : \Sigma^* \to \mathbb{N}$ such that for every $x \in \Sigma^*$,

$$|f(x)| \leq h(\mu(x)) \cdot p(|x|).$$

Remark 2.3. We will often work with functions that take more than one input and where we allow the output to be fpt-bounded by the i-th input. In these cases, for some $f : (\Sigma^*)^n \to \Sigma^*$, the previous definition lets us say something like "$f(x_1, \ldots, x_n)$ is π_i-fpt-length", meaning that the output of function f is of size $h(|x_i|) \cdot \text{poly}(|x_1| + \cdots + |x_n|)$, for some computable function h. ■

With the concept of fpt-length functions in hand we are ready to define complexity classes of the form **fpt-comp-C**, meaning that the compilation function is an fpt-length function. We are faced, however, with an important design question: Should **C** be a classical complexity class or a parameterized one? In other words, when thinking of efficient fpt-compilations, is the online phase carried out in polynomial time or can that running time be parameterized as well?

The first option, where the online computation is forced to remain "classical", leads to the following version of the **fpt-comp-** classes.

Definition 2.4 (**fpt-comp-C** classes for classical **C**)**.** Let **C** be a classical complexity class. We say that a problem (Q, κ, λ) *fpt-compiles to* **C** if there exists a π_2-fpt-length computable function $c : \Sigma^* \times \Sigma^* \to \Sigma^*$ and a language of pairs $Q_{+c} \in \mathbf{C}$ such that for all $x \in \Sigma^*$,

$$x \in Q \text{ if and only if } (x, c(\kappa(x), \lambda(x))) \in Q_{+c}.$$

We then say that (Q, κ, λ) *compiles to Q_{+c} via c*. We denote by **fpt-comp-C** the class of doubly parameterized problems fpt-compilable to **C**.

Intuitively, the previous definition is just extending the usual **poly-comp-C** in the natural way: the compilation function c acts on $\kappa(x)$ and the new parameter $\lambda(x)$ in such a way that x appended with $c(\kappa(x), \lambda(x))$ can be computed within the resources of **C**.

If one is more permissive and lets the online phase run in fpt-time with respect to the second parameter, the following classes appear.

Definition 2.5 (**fpt-comp-C** classes for parameterized **C**)**.** Let **C** be a parameterized complexity class. We say that a problem (Q, κ, λ) *fpt-compiles to* **C** if there exists a π_2-fpt-length computable function $c : \Sigma^* \times \Sigma^* \to \Sigma^*$ and a language of pairs $(Q_{+c}, \lambda \circ \pi_1) \in \mathbf{C}$ such that for all $x \in \Sigma^*$,

$$x \in Q \text{ if and only if } (x, c(\kappa(x), \lambda(x))) \in Q_{+c}.$$

We then say that (Q, κ, λ) *compiles to* Q_{+c} *via* c. We denote by **fpt-comp-C** the class of doubly parameterized problems fpt-compilable to **C**.

The notation $\lambda \circ \pi_1$ might result slightly confusing, but it is just making sure that the online phase of the computation is fpt-time with respect to $\lambda(x)$: since Q_{+c} is a language of pairs, with instances of the form $(x, c(\kappa(x), \lambda(x)))$, the parameter should be $\lambda(x)$, so we need to first project the first component of the pair and then apply λ.

Remark 2.6. The reader may argue that it might be desirable to consider the online phase of the computation to run in fpt-time with respect to some third parameter, possibly different from λ. Though this might result in interesting insights, we focus here on the scenario where λ is reused. The reason, as we will now see, is that it greatly simplifies our design decisions. Furthermore, it makes sense to just combine such a third parameter into λ: if we are willing to let the online phase run in time longer than polynomial, we are probably also willing to let the compilation be slightly bigger with respect to this same parameter, and vice versa. ■

Remark 2.7. Recall that throughout this work, when quantifying over complexity classes, we do so over classical classes that are closed under polynomial-time many-one reductions ($\leq_p$) or parameterized classes closed under fixed-parameter tractable many-one reductions ($\leq_{fpt}$). Furthermore, we assume that for every class **C** under consideration, $\mathbf{C} \supseteq \mathbf{P}$ (or, alternatively, $\mathbf{C} \supseteq \mathbf{FPT}$). ■

We now have two seemingly different notions of efficient fpt-compilation: **fpt-comp-P** and **fpt-comp-FPT**. Interestingly, we already know of an member of both of these classes.

Example 2.8 (SAT Completion). Recall the problem SAT Completion, taking as input a propositional formula φ and a partial assignment to it. Consider now the doubly parameterized problem (SAT Completion, φ, u), where $u : \Sigma^* \to \Sigma^*$ returns, given the partial assignment, the number of variables left unassigned by it. The first parameter gives us compilation access to the formula φ.

Observe that (SAT Completion, φ, u) $\in$ **fpt-comp-P**. As we argued, we can enumerate all possible bit-strings of length u in the compilation, which will be of size roughly $2^u \cdot \text{poly}(|\varphi|)$. Then, when the partial assignment arrives, we can iterate over all possible completions of it, which takes time linear in the size of the compilation!

Interestingly, we also have that (SAT Completion, φ, u) $\in$ **fpt-comp-FPT**, simply because, without any compilation, the online phase can iterate over all possible ways of completing the partial assignment. ■

Though in the case of SAT Completion both classes seem to capture the problem, at first sight the second definition seems to provide a more general and powerful notion of computation, since the online phase can be generally more expensive than in the classical setting. However, careful observation shows that whatever additional power is used by the parameterized online computation, one can always push this inside the compilation, so that the online phase can remain classical. More precisely, the two variants of the **fpt-comp**- classes are related as follows.

Proposition 2.9. *Let* **C** *be a classical complexity class. Then,*

$$\textbf{fpt-comp-C} = \textbf{fpt-comp-para-C}.$$

Proof. The forward inclusion is immediate. For the backwards one, let $(Q, \kappa, \lambda) \in$ **fpt-comp-para-C**. Then, there is a π_2-fpt-compilable function c and a language $(Q_{+c}, \lambda \circ \pi_1) \in$ **para-C** as in Definition 2.5. The latter in turn means that there is a computable $c' : \Sigma^* \to \Sigma^*$ such that there is some $Q'_{+c'} \in \mathbf{C}$ such that for all $x \in \Sigma^*$, $(x, c(\kappa(x), \lambda(x))) \in Q_{+c}$ if and only if $((x, c(\kappa(x), \lambda(x))), c'(\lambda(x))) \in Q'_{+c'}$.

It then suffices to argue that (Q, κ, λ) can be fpt-compiled to $Q'_{+c'}$, which is in **C**. One needs to argue that $c(\kappa(x), \lambda(x))$ and $c'(\lambda(x))$ are both fpt-length in $\lambda(x)$. We know that c is fpt-length with respect to λ, which means there is a polynomial $p : \mathbb{N} \to \mathbb{N}$ and a computable function $s_0 : \Sigma^* \to \mathbb{N}$ such that

$$|c(\kappa(x), \lambda(x)))| \leq s_0(\lambda(x)) \cdot p(|\kappa(x)|)$$

and, besides, let s_1 be a computable function bounding the size of c'. Then,

$$\begin{aligned}|c(\kappa(x), \lambda(x)))| + |c'(\lambda(x))| &\leq s_0(\lambda(x)) \cdot p(|\kappa(x)|) + s_1(\lambda(x)) \\ &\leq (s_0(\lambda(x)) + s_1(\lambda(x))) \cdot p(|\kappa(x)|)\end{aligned}$$

which is again fpt-bounded by λ. □

In light of this previous result, our notion of efficient fpt-compilation is the same regardless of whether the online phase of the computation is parameterized or not.

Corollary 2.10. **fpt-comp-P** = **fpt-comp-FPT**.

As a result, we are inclined to focus on the second version of the **fpt-comp-** classes. After all, if at some point we become interested in some **fpt-comp-C** for a classical **C**, we can simply study **fpt-comp-para-C** and transfer the results. Furthermore, classes like **fpt-comp-W**[1] dot not have a classical analogue (unless **FPT** = **W**[1]), so the parameterized world offers a richer variety of classes.

Note that our definition of **fpt-comp-C** is simultaneously a natural generalization of the **poly-comp-C** classes as well as of the usual parameterized classes, as demonstrated by the following result.

Proposition 2.11. *Let $Q \subseteq \Sigma^*$ be a language, let* **C** *be a classical complexity class and let* **D** *be a parameterized complexity class. Then,*

(i) if $(Q, \kappa) \in$ **poly-comp-C***, then $(Q, \kappa, \lambda) \in$* **fpt-comp-C** *for every parameterization λ;*

(ii) if $(Q, \kappa, \lambda) \in$ **fpt-comp-C** *and λ is constant, then $(Q, \kappa) \in$* **poly-comp-C***;*

(iii) if $(Q, \lambda) \in$ **D***, then for every polynomial-time computable κ, $(Q, \kappa, \lambda) \in$* **fpt-comp-D***;*

(iv) if $(Q, \kappa, \lambda) \in$ **fpt-comp-D** *and κ is constant, then $(Q, \lambda) \in$* **D***.*

Proof.

(i) If (Q, κ) admits a polynomial-size compilation, then it also admits fpt-size compilation, simply by ignoring the additional parameter λ.

(ii) If λ is constant, then (Q, κ, λ) compiles via some c into some $Q_{+c} \in \mathbf{C}$, with the size of c bounded by

$$|c(\kappa(x), \lambda(x))| \leq h(\lambda(x)) \cdot p(|(\kappa(x))|) \leq k \cdot p(|(\kappa(x))|)$$

for some computable h, some constant $k \in \mathbb{N}$ and some polynomial p. That means that c depends only on κ and that c is polynomially bounded. Since, furthermore, $Q_{+c} \in \mathbf{C}$, we have that $(Q, \kappa) \in$ **poly-comp-C**.

(iii) One can simply ignore the compilable part given by κ and solve the entire instance within the resources of **D**.

(iv) If κ is constant, then there is no parameter to compile. Indeed, by definition, there is a compilation function $c : \Sigma^* \times \Sigma^* \to \Sigma^*$ such that (Q, κ, λ) compiles some $(Q_{+c}, \lambda \circ \pi_1) \in \mathbf{C}$ and there is a polynomial p and a computable function $h : \Sigma^* \to \mathbb{N}$ such that

$$|c(\kappa(x), \lambda(x))| \leq h(\lambda(x)) \cdot p(|\kappa(x)|).$$

Since κ is constant, $p(|\kappa(x)|) = k \in \mathbb{N}$, for some constant k, and hence

$$|c(\kappa(x), \lambda(x))| \leq k \cdot h(\lambda(x))$$

which depends only on λ. Hence, λ suffices to compute c and solve every instance within the resources of **D**.

□

2.2 Reductions

To carry out the type of structural research we are after, our new complexity classes must be equipped with some notion of reduction that suits them. Recall that the **poly-comp**- classes are closed under poly-comp reductions ($\leq^{\text{poly}}_{\text{comp}}$), a specific case of fpt reductions. Our approach is precisely to extend the existing poly-comp reduction into a relation giving closure to the **fpt-comp**- classes.

Intuitively, the functions performing the reduction should then be computable in fpt-time, except for some possible expensive computation depending only on the compilable part. We now define the functions performing such reductions.

Definition 2.12 (fpt-compilable function)**.** A function $g : \Sigma^* \to \Sigma^*$ is said to be *fpt-compilable* with respect to κ and λ or simply (κ, λ)*-fpt-compilable* if there is a π_2-fpt-length computable function $c : \Sigma^* \times \Sigma^* \to \Sigma$ and a $(\lambda \circ \pi_1)$-fpt-time-computable $f : \Sigma^* \times \Sigma^* \to \Sigma^*$, such that for all $x \in \Sigma^*$,

$$g(x) = f(x, c(\kappa(x), \lambda(x))).$$

Intuitively, a reduction g between two doubly parameterized problems (Q, κ, λ) and (Q', κ', λ') should satisfy three requirements, for every input $x \in \Sigma^*$:

- in the computation of $g(x)$ one may carry out some expensive computation on the compilable part of the input, $\kappa(x)$;
- the compilable part of the reduced instance, $\kappa'(g(x))$, should be computable from the compilable part of the original instance, $\kappa(x)$, and be of size fpt-bounded by $\lambda(x)$;
- like in fpt reductions, the new parameter $\lambda'(g(x))$ should be bounded by some function depending solely on $\lambda(x)$.

It is easy to see that this is in fact a natural generalization of the poly-comp reductions from Definition 1.4.

Using fpt-compilable functions, the following definition captures the new fpt-comp reductions. Recall that $\mathcal{P}_{\text{fin}}(\Sigma^*)$ denotes the set of all finite subsets of Σ^*.

Definition 2.13 (fpt-comp reductions)**.** We say that (Q, κ, λ) *fpt-comp-reduces* to (Q', κ', λ'), written $(Q, \kappa, \lambda) \leq^{\text{fpt}}_{\text{comp}} (Q', \kappa', \lambda')$, if there is a (κ, λ)-fpt-compilable function $g : \Sigma^* \to \Sigma^*$ such that,

(i) for all $x \in \Sigma^*$,
$$x \in Q \Leftrightarrow g(x) \in Q';$$

(ii) there is a π_2-fpt-length computable function $t : \Sigma^* \times \Sigma^* \to \mathcal{P}_{\text{fin}}(\Sigma^*)$ such that for all $x \in \Sigma^*$, $\kappa'(g(x)) \in t(\kappa(x), \lambda(x))$;

(iii) there is a computable $s : \Sigma^* \to \mathcal{P}_{\text{fin}}(\Sigma^*)$ such that for all $x \in \Sigma^*$, $\lambda'(g(x)) \in s(\lambda(x))$.

If $(Q, \kappa, \lambda) \leq^{\text{fpt}}_{\text{comp}} (Q', \kappa', \lambda')$ and $(Q', \kappa', \lambda') \leq^{\text{fpt}}_{\text{comp}} (Q, \kappa, \lambda)$, we say that the problems are *fpt-comp-interreducible* and write $(Q, \kappa, \lambda) \equiv^{\text{fpt}}_{\text{comp}} (Q', \kappa', \lambda')$.

Observe that one could have followed a more restrictive approach, defining instead a notion of reduction where, apart from the compilation, the final wrapping function f runs in polynomial time instead of fpt time. This would seem the natural notion of reduction for the classical variant of the **fpt-comp-C** classes. However, in light of Proposition 2.9, once we prove that the **fpt-comp-C** classes for parameterized **C** are closed under our new fpt-comp reductions, we will immediately get a working notion of reduction for the classical variant of our classes.

Observe how this new notion of reduction is simultaneously a generalization of poly-comp reductions and fpt reductions.

Proposition 2.14. *Let (Q, κ, λ) and (Q', κ', λ') be doubly parameterized languages. Then,*

(i) if $(Q, \kappa, \lambda) \leq^{\text{fpt}}_{\text{comp}} (Q', \kappa', \lambda')$ and κ is constant, then $(Q, \lambda) \leq_{\text{fpt}} (Q', \lambda')$;

(ii) if $(Q, \lambda) \leq_{\text{fpt}} (Q', \lambda')$ and κ' is constant, then $(Q, \kappa, \lambda) \leq^{\text{fpt}}_{\text{comp}} (Q', \kappa', \lambda')$;

(iii) if $(Q, \kappa, \lambda) \leq^{\text{fpt}}_{\text{comp}} (Q', \kappa', \lambda')$ and λ is constant, then $(Q, \kappa) \leq^{\text{poly}}_{\text{comp}} (Q', \kappa')$;

(iv) if $(Q, \kappa) \leq^{\text{poly}}_{\text{comp}} (Q', \kappa')$ and λ' is constant, then $(Q, \kappa, \lambda) \leq^{\text{fpt}}_{\text{comp}} (Q', \kappa', \lambda')$.

Proof.

(i) If κ is constant, the compilation phase of the reduction becomes useless, and it is straightforward to verify that the remaining conditions of fpt-comp reductions are precisely the conditions of fpt reductions.

(ii) The fpt-comp reduction is just the existing fpt reduction. Since κ' is constant, it can always be obtained from κ, which is the only additional requirement we need.

(iii) If λ is constant, that means the running time of the reduction as well as the size of the compilation are polynomial and hence there is a poly-comp reduction.

(iv) The fpt-comp reduction is the existing poly-comp reduction. Since λ' is constant, we can ensure that it is bounded by some constant value, which can be expressed as a function depending exclusively on λ. The rest of the requirements coincide.

□

Crucially, our new **fpt-comp-C** classes are closed under this new notion of reduction.

Proposition 2.15. *For every parameterized complexity class* **C***, the class* **fpt-comp-C** *is closed under fpt-comp reductions.*

Proof. Let $(Q, \kappa, \lambda) \leq^{\text{fpt}}_{\text{comp}} (Q', \kappa', \lambda')$ and suppose $(Q', \kappa', \lambda') \in$ **fpt-comp-C**. We need to show that $(Q, \kappa, \lambda) \in$ **fpt-comp-C**. Let $g(x) = f(x, c(\kappa(x), \lambda(x)))$, s and t be as in the definition of fpt-comp reductions. Furthermore, since we assume that $(Q', \kappa', \lambda') \in$ **fpt-comp-C**, there is a π_2-fpt-length $c' : \Sigma^* \times \Sigma^* \to \Sigma^*$ such that (Q', κ', λ') compiles into some $(Q'_{+c'}, \lambda \circ \pi_1) \in \mathbf{C}$.

Given these components, we have that

$$\begin{aligned} x \in Q &\iff g(x) = f(x, c(\kappa(x), \lambda(x))) \in Q' \\ &\iff (g(x), c'(\kappa'(g(x)), \lambda'(g(x)))) \in Q'_{+c'}. \end{aligned}$$

The proof has two steps. First, we give a π_2-fpt-length $d : \Sigma^* \times \Sigma^* \to \Sigma^*$ such that (Q, κ, λ) fpt-compiles into $(Q_{+d}, \lambda \circ \pi_1)$. Second, we show that $(Q_{+d}, \lambda \circ \pi_1) \leq_{\text{fpt}} (Q'_{+c'}, \lambda \circ \pi_1) \in \mathbf{C}$. Since $\mathbf{C}$ is a parameterized class that is closed under fpt-reductions, it follows that $(Q_{+d}, \lambda \circ \pi_1) \in \mathbf{C}$, and hence we have that $(Q, \kappa, \lambda) \in$ **fpt-comp-C**.

The function $d(\kappa(x), \lambda(x))$ should output both $c(\kappa(x), \lambda(x))$ and well as the value $c'(\kappa'(g(x)), \lambda'(g(x)))$. The issue is that $g(x) = f(x, c(\kappa(x), \lambda(x)))$ depends on x, while d does not have access to the entire x. However, we do know that $\kappa'(g(x)) \in t(\kappa(x), \lambda(x))$ and $\lambda'(g(x)) \in s(\lambda(x))$. Hence, the idea is as follows: for each pair $(k, \ell) \in t(\kappa(x), \lambda(x)) \times s(\lambda(x))$, compute $c'(k, \ell)$, and store it in a table together with (k, ℓ). The output of $d(\kappa(x), \lambda(x))$ is this table, together with $c(\kappa(x), \lambda(x))$.

It now suffices to verify that d is π_2-fpt-length. Since c is π_2-fpt-length, c' is π_2-fpt-length, and $\lambda' \circ g$ is fpt-bounded by $\lambda(x)$, it suffices to check that the number of entries in the table is fpt-bounded by $\lambda(x)$, which is indeed the case, since

$$|t(\kappa(x), \lambda(x)) \times s(\lambda(x))| = |t(\kappa(x), \lambda(x))| \cdot |s(\lambda(x))|$$

and both t and s are fpt-bounded by $\lambda(x)$, as desired.

We now have that (Q, κ, λ) compiles into $(Q_{+d}, \lambda \circ \pi_1)$. Consider now the fpt-reduction that maps $(x, d(\kappa(x), \lambda(x)))$ to $(g(x), c'(\kappa'(g(x))), \lambda'(g(x)))$. Note that $g(x) = f(x, c(\kappa(x), \lambda(x)))$ is fpt-computable with respect to $\lambda(x)$, and $c(\kappa(x), \lambda(x))$ is part of the output of d. As for c', we can check its right value in the table computed by d. The relation between parameters λ and λ' is correctly regulated by the fpt-comp-reduction, so we have $(Q_{+d}, \lambda \circ \pi_1) \leq_{\text{fpt}} (Q'_{+c'}, \lambda \circ \pi_1) \in \mathbf{C}$, as desired. □

Another positive indicator of the robustness of our notion of fpt-comp reducibility is that multiple reductions can be chained together, preserving transitivity.

Proposition 2.16. *The relation of fpt-comp reducibility is transitive.*

Proof. Suppose that $(Q_1, \kappa_1, \lambda_1) \leq^{\text{fpt}}_{\text{comp}} (Q_2, \kappa_2, \lambda_2)$ via some function $g(x) = f(x, c(\kappa_1(x), \lambda_1(x)))$ and bounding functions s and t as in the definition of fpt-comp reductions. Similarly, suppose $(Q_2, \kappa_2, \lambda_2) \leq^{\text{fpt}}_{\text{comp}} (Q_3, \kappa_3, \lambda_3)$ via some $g'(x) = f'(x, c'(\kappa_2(x), \lambda_2(x)))$ and bounding functions s' and t' as in the definition. We claim that the function $g''(x) := \pi_2(x, g'(g(x))) = g'(g(x))$ works as a fpt-comp reduction witnessing $(Q_1, \kappa_1, \lambda_1) \leq^{\text{fpt}}_{\text{comp}} (Q_3, \kappa_3, \lambda_3)$.

Correctness of the reduction is clear: for every $x \in \Sigma^*$,

$$x \in Q_1 \Leftrightarrow g(x) \in Q_2 \Leftrightarrow g'(g(x)) \in Q_3.$$

As for the running time and the size of the function, π_2 is computable in polynomial time, so that presents no issue. We just have to make sure that $g'(g(x))$ is λ_1-fpt-length. This is justified by the fact that $\lambda_2(g(x)) \in s(\lambda_1(x))$, so every time something is $\lambda_2(g(x))$-fpt-bounded, it is also $\lambda_1(x)$-fpt-bounded. It then suffices to see that to compute $g'(g(x))$, one first computes $g(x)$, which is by definition fpt-bounded in $\lambda_1(x)$. Then, $g'(g(x))$ is fpt-bounded in $\lambda_2(g(x))$, but we just argued this is in turn fpt-bounded in $\lambda_1(x)$, so it is fine.

Finally, we need to make sure the parameterizations κ_3 and λ_3 are appropriately bounded by κ_1 and λ_1. For bounding λ_3, consider the function $s'' : \Sigma^* \to \mathcal{P}_{\text{fin}}(\Sigma^*)$ defined as

$$s''(x) := \bigcup \{s'(y) \mid y \in s(x)\}.$$

Indeed, we have that $\lambda_3(g'(g(x))) \in s''(\lambda_1(x))$: simply recall that $\lambda_3(g'(x)) \in s'(\lambda_2(x))$, so we get $\lambda_3(g'(g(x))) \in s'(\lambda_2(g(x))) \subseteq s''(\lambda_1(x))$, since $\lambda_2(g(x)) \in s(\lambda_1(x))$. Observe that s'' is computable, since s and s' are computable, and that that is the only requirement it must fulfill.

As for bounding κ_3 in terms of κ_1 and λ_1, define

$$t''(x, y) := \bigcup \{t'(x', y') \mid x' \in t(x, y), y' \in s(y)\}$$

and note indeed that since $\kappa_3(g'(x)) \in t'(\kappa_2(x), \lambda_2(x))$, we get that

$$\kappa_3(g'(g(x))) \in t'(\kappa_2(g(x)), \lambda_2(g(x))) \subseteq t''(\kappa_1(x), \lambda_1(x))$$

since $\kappa_2(g(x)) \in t(\kappa_1(x))$ and $\lambda_2(g(x)) \in \lambda_1(x)$. Finally, note that t'' is obviously computable and that for any input (x, y) it is π_2-fpt-length. This is because both $t(x, y)$ and $s(y)$ are fpt-bounded by $|y|$, so t'' cannot be much bigger. □

We already mentioned that, in fact, compilation itself can be seen as a form of reduction: one reduces a language Q to a language Q_{+c} consisting of the input padded with a compilation. That is, (Q, κ, λ) fpt-comp reduces to $(Q_{+c}, \epsilon, \lambda \circ \pi_1)$, where there is no need to compile anything, since the compilation is already in the instance. It comes as no surprise that **fpt-comp-C** can be characterized in terms of fpt-comp reductions in this sense.

Proposition 2.17. *Let (A, κ, λ) be a doubly parameterized language and let* **C** *be a parameterized complexity class. It holds that $(A, \kappa, \lambda) \in$* **fpt-comp-C** *if and only if there is some $(B, \mu) \in$* **C** *such that $(A, \kappa, \lambda) \leq^{\text{fpt}}_{\text{comp}} (B, \epsilon, \mu)$, where recall that ϵ denotes the empty parameterization.*

Proof. The forward direction is a consequence of Definition 2.5: if $(A, \kappa, \lambda) \in$ **fpt-comp-C**, then there is a π_2-fpt-size computable function $c : \Sigma^* \times \Sigma^* \to \Sigma^*$ and a language of pairs $(Q_{+c}, \lambda \circ \pi_1) \in$ **C** such that for every $x \in \Sigma^*$, $x \in Q$ if and only if $(x, c(\kappa(x), \lambda(x))) \in Q_{+c}$. Then, the mapping $x \mapsto (x, c(\kappa(x), \lambda(x)))$ is an fpt-comp reduction witnessing

$$(A, \kappa, \lambda) \leq^{\text{fpt}}_{\text{comp}} (Q_{+c}, \epsilon, \lambda \circ \pi_1)$$

where we have nothing to compile on the right-hand side.

For the backwards direction, if there exists some $(B, \mu) \in$ **C** such that $(A, \kappa, \lambda) \leq^{\text{fpt}}_{\text{comp}} (B, \epsilon, \mu)$, then clearly $(B, \epsilon, \mu) \in$ **fpt-comp-C**, which is closed under fpt-comp reductions, and hence $(A, \kappa, \lambda) \in$ **fpt-comp-C**. □

2.3 Methodology theorems for lower bounds

The new **fpt-comp-** classes, equipped with fpt-comp reductions, are going to be our basic tool to show hardness results. The idea, once again, is to classify (parameterized) compilability problems by reducing to and from some canonical problem for each class. The canonical problems, analogous Chen's framework, are of the form (Q, len, λ), where the compilable part of the input is its length and (Q, λ) is some hard parameterized problem. As we discussed in Section 1.4, this goes back to lower bounds via advice classes as devised by Selman and Kautz and the $\|\not\hookrightarrow$**C** classes of Cadoli et al. Having compilability access to the length of the input is equivalent to having computable polynomial-size or fpt-sized advice. So, if (Q, λ) is a hard parameterized problem but (Q, len, λ) is efficiently compilable, that should provoke some collapse involving advice classes.

Parameterized advice is defined as follows[1].

Definition 2.18 (fpt-size advice)**.** Let **C** be a parameterized complexity class. We say that a parameterized problem (Q, λ) is in the class **C**/fpt if there exists a π_2-fpt-length function $\alpha : \Sigma^* \times \Sigma^* \to \Sigma^*$ and a language of pairs $(Q_{+\alpha}, \lambda \circ \pi_1) \in$ **C** such that for all $x \in \Sigma^*$,

$$x \in Q \text{ if and only if } (x, \alpha(1^{|x|}, \lambda(x))) \in Q_{+\alpha}.$$

Intuitively, parameterized advice behaves just like the usual polynomial-size advice, with the corresponding upgrades: the advice can be slightly bigger (of fpt-size in $\lambda(x)$) and the computation can also be carried out in time fpt in $\lambda(x)$. The latter explains why the language of pairs $Q_{+\alpha}$ takes as parameterization $\lambda \circ \pi_1$: we project to get x and then apply λ.

The intuition for lower bounds is captured by the following methodology theorem, which lets us prove conditional hardness for parameterized uncompilability via fpt-comp reductions.

[1] Parameterized advice classes where first defined by Chen [Che05]. Here we follow the similar definition of De Haan [Haa19, Definition 15.1].

Theorem 2.19 (General methodology theorem)**.** *Let* $\mathbf{C}$ *and* $\mathbf{C}'$ *be parameterized complexity classes and let* (A, λ) *be a hard problem for* $\mathbf{C}$ *under fpt reductions. If* $(A, \text{len}, \lambda) \leq^{\text{fpt}}_{\text{comp}} (B, \kappa, \mu)$ *for some* $(B, \kappa, \mu) \in$ **fpt-comp-**$\mathbf{C}'$*, then* $\mathbf{C} \subseteq \mathbf{C}'/\text{fpt}$.

Proof. By the closure of **fpt-comp-**$\mathbf{C}$ under fpt-comp reductions (Proposition 2.15), we have

$$(A, \text{len}, \lambda) \in \textbf{fpt-comp-}\mathbf{C}'$$

which means there is a π_2-fpt-length computable $c : \Sigma^* \times \Sigma^* \to \Sigma^*$ and a language $(A_{+c}, \lambda \circ \pi_1) \in \mathbf{C}'$ such that for all $x \in \Sigma^*$, $x \in A$ if and only if $(x, c(1^{|x|}, \lambda(x))) \in A_{+c}$. Hence, $(A, \lambda) \in \mathbf{C}'/\text{fpt}$, since c is precisely a function giving fpt-advice. Since (A, λ) is $\mathbf{C}$-hard, it immediately follows that $\mathbf{C} \subseteq \mathbf{C}'/\text{fpt}$. □

Example 2.20. The methodology theorem implies that if one manages to show a reduction of the form $(\textsc{Clique}, \text{len}, k) \leq^{\text{fpt}}_{\text{comp}} (A, \kappa, \lambda)$, then we can claim that $(A, \kappa, \lambda) \notin$ **fpt-comp-FPT** unless $\mathbf{W}[1] \subseteq \mathbf{FPT}/\text{fpt}$, since $(\textsc{Clique}, k)$ is $\mathbf{W}[1]$-complete. ■

Interestingly, in some cases we can prove an even stronger collapse, not involving advice classes. This is reminiscent of the $\rightsquigarrow\mathbf{C}$ classes of Cadoli et al. and the reduction for Formula Inference (Theorem 1.6), where there is no compilation involved. In our parameterized framework, these extreme cases can be phrased as follows.

Theorem 2.21 (Strong methodology theorem)**.** *If* (A, λ) *is* $\mathbf{C}$*-hard for some parameterized class* $\mathbf{C}$*,* $(A, \epsilon, \lambda) \leq^{\text{fpt}}_{\text{comp}} (B, \kappa, \mu)$ *and* $(B, \kappa, \mu) \in$ **fpt-comp-**$\mathbf{C}'$*, then* $\mathbf{C} \subseteq \mathbf{C}'$.

Proof. By the closure of **fpt-comp-**$\mathbf{C}$ under $\leq^{\text{fpt}}_{\text{comp}}$ reductions, we have

$$(A, \epsilon, \lambda) \in \textbf{fpt-comp-}\mathbf{C}'$$

which means there is a π_2-fpt-length computable $c : \Sigma^* \times \Sigma^* \to \Sigma^*$ and a language $(A_{+c}, \lambda \circ \pi_1) \in \mathbf{C}'$ such that for all $x \in \Sigma^*$, $x \in A$ if and only if $(x, c(\epsilon, \lambda(x))) \in A_{+c}$. Hence, $(A, \lambda) \in \mathbf{C}'$, since compiling the empty string does not give any additonal power. Since (A, λ) is $\mathbf{C}$-complete, it immediately follows that $\mathbf{C} \subseteq \mathbf{C}'$. □

2.4 The new chopped-C classes

The methodology theorems in the previous section suggest that given a $\mathbf{C}$-complete language (Q, λ) for some class $\mathbf{C}$ (believed to be) beyond **FPT**, the problem (Q, len, λ) is a hard problem.

Unfortunately, even if (Q, λ) is $\mathbf{C}$-complete under fpt reductions, we cannot show that (Q, len, λ) is hard for **fpt-comp-**$\mathbf{C}$ under fpt-comp reductions. The same happened in the parameter compilation framework, where one could not show that (Q, len) is complete for **poly-comp-**$\mathbf{C}$ even if Q is $\mathbf{C}$-complete.

In an attempt to rephrase our methodology theorems in terms of hardness for some class, we take the pragmatic approach of defining, for each **fpt-comp-**$\mathbf{C}$, a subclass consisting of the closure around the problems of $\mathbf{C}$ that get compilability access to the length of the input.

Definition 2.22 (**chopped-**$\mathbf{C}$ classes for parameterized $\mathbf{C}$)**.** Let $\mathbf{C}$ be a parameterized complexity class. We define **chopped-**$\mathbf{C}$ as the closure under fpt-comp reductions of all languages of the form (Q, len, λ), such that $(Q, \lambda) \in \mathbf{C}$. That is,

$$\textbf{chopped-}\mathbf{C} = \bigcup_{(Q,\lambda) \in \mathbf{C}} [(Q, \text{len}, \lambda)]^{\leq^{\text{fpt}}_{\text{comp}}}.$$

The following two propositions are immediate consequences of our definition.

Proposition 2.23. *For every parameterized class* **C**, **chopped-C** $\subseteq$ **fpt-comp-C**.

Proof. Observe that if $(Q, \kappa, \lambda) \in$ **chopped-C** for some **C**, then there is some (Q_0, len, μ) such that $(Q_0, \mu) \in \mathbf{C}$ and $(Q, \kappa, \lambda) \leq^{\text{fpt}}_{\text{comp}} (Q_0, \text{len}, \mu)$. Since clearly $(Q_0, \text{len}, \mu) \in$ **fpt-comp-C** and the class is closed under fpt-comp reductions, $(Q, \kappa, \lambda) \in$ **fpt-comp-C**. □

Proposition 2.24. *For every parameterized class* **C**, *the class* **chopped-C** *is closed under fpt-comp reductions.*

Proof. If $(Q_1, \kappa_1, \lambda_1) \leq^{\text{fpt}}_{\text{comp}} (Q_2, \kappa_2, \lambda_2) \in$ **chopped-C**, then by definition of **chopped-C**, $(Q_2, \kappa_2, \lambda_2) \leq^{\text{fpt}}_{\text{comp}} (Q_0, \text{len}, \lambda_0)$ for some $(Q_0, \lambda_0) \in \mathbf{C}$. Hence, by transitivity of fpt-comp reductions, $(Q_1, \kappa_1, \lambda_1) \leq^{\text{fpt}}_{\text{comp}} (Q_0, \text{len}, \lambda_0)$, and so it is contained in the closure around $(Q_0, \text{len}, \lambda_0)$, which is a subset of **chopped-C**. □

As it happened for **poly-comp-P**, **chopped-FPT** coincides with the already known **fpt-comp-FPT**.

Proposition 2.25. **chopped-FPT** = **fpt-comp-FPT**.

Proof. The forward inclusion is a consequence of Proposition 2.23. For the other direction, let $(A, \kappa, \lambda) \in$ **fpt-comp-FPT**. Then, there is an π_2-fpt-length computable c such that for some $(Q_{+c}, \lambda \circ \pi_1) \in$ **FPT**, $x \in A$ if and only if $(x, c(\kappa(x), \lambda(x))) \in Q_{+c}$.

Fix some problem $(B, \mu) \in$ **FPT** and some instances $y_0 \notin B$ and $y_1 \in B$. Then, consider the mapping g such that $g(x) = y_1$ if $x \in A$ and $g(x) = y_0$ otherwise. Observe that g can be computed in fpt-time after compiling c, since we can obtain $(x, c(\kappa(x), \lambda(x)))$ and decide in fpt-time whether it is in Q_{+c}. Furthermore, note that for all $x \in \Sigma^*$, $|g(x)| \leq \max\{y_0, y_1\}$, meaning that the size of $g(x)$ is bounded by a constant, and so we can always feasibly obtain $\text{len}(g(x))$. This means that

$$(A, \kappa, \lambda) \leq^{\text{fpt}}_{\text{comp}} (B, \text{len}, \mu)$$

and since $(B, \mu) \in$ **FPT** we have $(B, \text{len}, \mu) \in$ **chopped-FPT** and hence $(A, \kappa, \lambda) \in$ **chopped-FPT**, as desired. □

Does this coincidence extend to other classes? The answer is no, at least not as long as they are strictly beyond **FPT**. That is, it does not seem like we can do without the **chopped-** classes. In fact, this also happens in the classical setting of the parameter compilation framework, though this was not proven in Chen's original paper.

Proposition 2.26. *Let* **C** *be a classical complexity class. If* $\mathbf{P} \subsetneq \mathbf{C}$, *then* **chopped-C** $\subsetneq$ **poly-comp-C**.

Proof. We already know that **chopped-C** $\subseteq$ **poly-comp-C**, so we only need to prove that the inclusion is strict. Recall the background assumption that **C** is closed under polynomial-time reductions, meaning that $\mathbf{P} \subseteq \mathbf{C}$. Hence, we assume that **chopped-C** = **poly-comp-C** and derive $\mathbf{C} \subseteq \mathbf{P}$, concluding $\mathbf{C} = \mathbf{P}$, which contradicts the assumption of the proposition.

Let A be some language in **C**. Clearly, $(A, \epsilon) \in$ **poly-comp-C** and we assumed **poly-comp-C** = **chopped-C**, so we get that (A, ϵ) must reduce to (B, len) for some $B \in \mathbf{C}$ via some mapping g. Note that this reduction must be computable in polynomial time, since the compilation function has only access to ϵ, so whatever it outputs it can be hardwired in the reduction. Furthermore, there is a polynomial-size computable function $t : \Sigma^* \to \mathcal{P}_{\text{fin}}(\Sigma^*)$ such that $\text{len}(g(x)) \in t(\epsilon)$. But clearly $t(\epsilon)$ is always some finite fixed set of strings. This means that all the positive instances of A are mapped to some finite subset of B in polynomial time. But this essentially means that we can solve A in polynomial time! Hence, $A \in \mathbf{P}$, and since A was arbitrary, $\mathbf{C} \subseteq \mathbf{P}$. As we argued above, this gives $\mathbf{C} = \mathbf{P}$. Contradiction. □

The same argument with minor modifications to account for fpt reductions yields an identical result for parameterized classes.

Proposition 2.27. *Let* **C** *be a parameterized complexity class. If* **FPT** $\subsetneq$ **C** *, then* **chopped-C** $\subsetneq$ **fpt-comp-C**.

Hence, in particular, unless **P** = **NP**, **chopped-NP** $\subsetneq$ **poly-comp-NP**. Similarly, unless **FPT** = **W**[1], **chopped-W**[1] $\subsetneq$ **fpt-comp-W**[1]. In short, we cannot do without the **chopped-** classes. We can, however, phrase our methodology theorem (Theorem 2.19) in terms of hardness for these classes, as we intended.

Theorem 2.28 (General methodology theorem, rephrased)**.** *Let* **C** *and* **C′** *be parameterized complexity classes. If* (Q, κ, λ) *is* **chopped-C***-hard and* $(Q, \kappa, \lambda) \in$ **fpt-comp-C′** *(that is, if* **chopped-C** $\subseteq$ **fpt-comp-C′***), then* **C** $\subseteq$ **C′**/fpt.

Proof. Let $(A, \mu) \in$ **C**. We then have that $(A, \text{len}, \mu) \in$ **chopped-C**, and since (Q, κ, λ) is **chopped-C**-hard, we must have a reduction witnessing $(A, \text{len}, \mu) \leq^{\text{fpt}}_{\text{comp}} (Q, \kappa, \lambda)$. But since $(Q, \kappa, \lambda) \in$ **fpt-comp-C′** and **fpt-comp-** classes are closed under fpt-comp reductions, we have $(A, \text{len}, \mu) \in$ **fpt-comp-C′**. As we argued in Theorem 2.19, having compilation access to the length of the input is equivalent to having fpt-size advice, so $(A, \mu) \in$ **C′**/fpt. □

Remark 2.29. The reformulation of the general methodology theorem in terms of the **chopped-** complexity classes lets us observe that the lower bound provided by the methodology theorem is not tight, in the following sense: if **C** $\subseteq$ **C′**/fpt it does not necessarily follow that **chopped-C** $\subseteq$ **fpt-comp-C′**. This is because even if all problems in **C** allow for fpt-size advice, this advice need not be computable, so the advice would not work as valid compilation. In other words, fpt-size compilation can be seen as fpt-size advice, but the converse is not true. ■

In this way, the **chopped-** classes capture the general methodology theorem (Theorem 2.19). Analogously, one could capture the strong methodology theorem (Theorem 2.21) with a simpler class, obtained around the closure of problems where nothing is compiled.

Definition 2.30 (**simple-C** classes for parameterized **C**)**.** Let **C** be a parameterized complexity class. We define **simple-C** as the closure under fpt-comp reductions of all languages of the form (Q, ϵ, λ), such that $(Q, \lambda) \in$ **C**, and where recall that ϵ denotes the empty parameterization. That is,

$$\textbf{simple-C} = \bigcup_{(Q,\lambda)\in\textbf{C}} [(Q, \epsilon, \lambda)]^{\leq^{\text{fpt}}_{\text{comp}}}.$$

The **chopped-C** classes would correspond to the parameterized variant of the $\|\!\rightsquigarrow$**C** classes of Cadoli et al., while **simple-C** would be the parameterized analogue of $\rightsquigarrow$**C**. So much so that, in fact, the **fpt-comp-** and **simple-** classes coincide!

Proposition 2.31. *For every parameterized complexity class* **C**, **simple-C** = **fpt-comp-C**.

Proof. The forward inclusion is clear: if $(A, \kappa, \lambda) \in$ **simple-C**, that means $(A, \kappa, \lambda) \leq^{\text{fpt}}_{\text{comp}} (B, \epsilon, \mu)$ for some $(B, \mu) \in$ **C**, and so $(B, \epsilon, \mu) \in$ **fpt-comp-C**. Since **fpt-comp-C** is closed under fpt-comp reductions, $(A, \kappa, \lambda) \in$ **fpt-comp-C**.

For the backwards inclusion, it suffices to use the characterization of **fpt-comp-C** in terms of fpt-comp reductions (Proposition 2.17): some (A, κ, λ) being an element of **fpt-comp-C** means reducing to some language $(A_{+c}, \lambda \circ \pi_1) \in$ **C**, where c is the compilation function. As a result,

$$(A, \kappa, \lambda) \leq^{\text{fpt}}_{\text{comp}} (A_{+c}, \epsilon, \lambda \circ \pi_1)$$

which entails $(A, \kappa, \lambda) \in$ **simple-C**. □

Theorem 2.32 (Strong methodology theorem, rephrased)**.** *Let* $\mathbf{C}$ *and* $\mathbf{C}'$ *be parameterized complexity classes. If* (Q, κ, λ) *is* **fpt-comp-C***-hard and* $(Q, \kappa, \lambda) \in$ **fpt-comp-C**$'$ *(that is, if* **fpt-comp-C** $\subseteq$ **fpt-comp-C**$'$*), then* $\mathbf{C} \subseteq \mathbf{C}'$*.*

Proof. We use the fact that **fpt-comp-C** = **simple-C**. Let $(A, \mu) \in \mathbf{C}$. We then have that $(A, \epsilon, \mu) \in$ **simple-C**, and since (Q, κ, λ) is **simple-**C-hard, we must have a reduction witnessing $(A, \epsilon, \mu) \leq^{\text{fpt}}_{\text{comp}} (Q, \kappa, \lambda)$. But since $(Q, \kappa, \lambda) \in$ **fpt-comp-C**$'$ and **fpt-comp-** classes are closed under fpt-comp reductions, we have $(A, \epsilon, \mu) \in$ **fpt-comp-C**$'$. As we argued in Theorem 2.21, having compilation access to the emtpy string is like having no compilation power, so the problem must be solved within the resourced of $\mathbf{C}'$. Hence, $(A, \mu) \in \mathbf{C}'$. □

Remark 2.33. Unlike in the case of the general methodology theorem (see Remark 2.29), we called the strong methodology theorem *strong* precisely because the lower bound is tight. That is, if $\mathbf{C} \subseteq \mathbf{C}'$, then it also follows that **fpt-comp-C** $\subseteq$ **fpt-comp-C**$'$. ■

Finally, the following observation is key to prove hardness results around the **chopped-** classes. We need this proposition to easily identify hard problems for the **chopped-** classes to that we can reduce to and from some existing problem.

Proposition 2.34. *If* (A, λ) *is* $\mathbf{C}$*-hard, then* (A, len, λ) *is* **chopped-C***-hard.*

Proof. We need to show that every (B, κ, μ) in **chopped-C** reduces to (A, len, λ). Since (B, κ, μ) is in **chopped-C**, that means that $(B, \kappa, \mu) \leq^{\text{fpt}}_{\text{comp}} (C, \text{len}, \nu)$ for some $(C, \nu) \in \mathbf{C}$. Since (A, λ) is $\mathbf{C}$-hard, we have that $(C, \nu) \leq_{\text{fpt}} (A, \lambda)$. Then, this same reduction achieves $(C, \text{len}, \nu) \leq^{\text{fpt}}_{\text{comp}} (A, \text{len}, \lambda)$. By transitivity, $(B, \kappa, \mu) \leq^{\text{fpt}}_{\text{comp}} (A, \text{len}, \lambda)$, as desired. □

The same result holds for **fpt-comp-C** classes.

Proposition 2.35. *If* (A, λ) *is* $\mathbf{C}$*-hard, then* (A, ϵ, λ) *is* **fpt-comp-C***-hard.*

Proof. We need to show that every (B, κ, μ) in **simple-C** reduces to (A, ϵ, λ). Since (B, κ, μ) is in **simple-C**, that means that $(B, \kappa, \mu) \leq^{\text{fpt}}_{\text{comp}} (C, \epsilon, \nu)$ for some $(C, \nu) \in \mathbf{C}$. Since (A, λ) is $\mathbf{C}$-hard, we have that $(C, \nu) \leq_{\text{fpt}} (A, \lambda)$. Then, this same reduction achieves $(C, \epsilon, \nu) \leq^{\text{fpt}}_{\text{comp}} (A, \epsilon, \lambda)$. By transitivity, $(B, \kappa, \mu) \leq^{\text{fpt}}_{\text{comp}} (A, \epsilon, \lambda)$, as desired. □

Chapter 3

Compilability around W[1]

The framework presented in the previous chapter, consisting of the **fpt-comp**- and **chopped**- classes, together with the notion of fpt-comp reducibility, is designed to prove hardness results on the parameterized compilability of computational problems. So far we only studied its general properties and justified its robustness. We are now ready to put the framework to use. The goal is to answer the question that sparked our interest in the first place: can CSP Completion be fpt-compiled when adding as an extra parameter the number of variables left undefined by the partial assignment? As it turns out, our brand new toolkit lets us show that this problem is **chopped**-**W**[1]-complete, and hence not fpt-compilable unless **W**[1] ⊆ **FPT**/fpt.

As warm-up, Section 3.1 presents a simple fpt-comp reduction exploiting the "superinstance technique" while preserving parameter dependencies for the **W**[1]-complete Weighted q-SAT Completion. The complexity of CSP Completion is discussed in Section 3.2, where we first show that, when parameterized by the number of undefined variables in the partial assignment, the problem is **W**[1]-complete. Section 3.2.2 finally addresses the opening question of this thesis via a superinstance reduction from Clique, showing that the problem is **chopped**-**W**[1]-complete. Furthermore, we note that the existing reduction also preserves low primal treewidth, meaning that CSP Completion remains **chopped**-**W**[1]-hard when parameterized by that measure. Finally, Section 3.3 applies our framework to a constrained form of Clique, Clique Completion, previously studied by De Haan [Haa19]. His results immediately translate into **chopped**-**W**[1]-completeness in our framework and present and alternative route to prove hardness for CSP Completion.

3.1 A simple hardness result: Weighted q-SAT Completion

To get familiar with hardness proofs in our new framework, we start thinking of fpt-comp reductions for problems in **W**[1]. Of course, not every problem is amenable to compilation, simply because it is unclear what part of the input we should preprocess. Usually, the instance must be made up of more than one element; otherwise, compiling the entire input would trivially solve every problem. Our approach is that given some **W**[1]-complete problem, we can define its "completion variant", which is just the same problem but adding some side constraints imposing that the solutions must include certain elements and exclude certain others. In this way, one can naturally think of compiling the main part of the instance but not the side constraints.

In this section we follow this approach for the canonical **W**[1]-complete problem Weighted q-SAT. In the classical setting, SAT Completion was the "completion variant" of SAT. In the parameterized world we are instead interested in weighted versions of SAT.

Weighted q-SAT Completion	
Instance	A q-CNF formula φ, a partial assignment α and a natural number k.
Question	Is there a satisfying assignment extending α that only sets k more variables to true?

In parameterized complexity, we would study a problem like (Weighted 3SAT Completion, k), where abusing notation k is referring to weight k of the remaining part of the assignment. For compilability purposes, we will be interested in the problem (Weighted 3SAT Completion, φ, k) instead, meaning that we can compile φ but not the partial assignment. The compilation can then be of size fpt-bounded by k.

It is not difficult to see that in terms of parameterized complexity, Weighted q-SAT Completion is just as hard as Weighted q-SAT.

Proposition 3.1. *For every $q \geq 2$,* (Weighted q-SAT Completion, k) *is* $\mathbf{W}[1]$*-complete.*

Proof. Hardness is obvious. For membership, simply "plug in" the partial assignment, getting an input to Weighted q-SAT. □

Unfortunately, compilation does not seem to make things easier in this context.

Theorem 3.2. *For every $q \geq 2$,* (Weighted q-SAT Completion, φ, k) *is* **chopped-W**$[1]$*-complete.*

Proof. For the moment, we focus on the case $q \geq 3$, since it is the one clearly showcasing the superinstance technique. We postpone the proof of $q = 2$ for later in this chapter, since it follows a different strategy.

It suffices to show that (Weighted 3SAT Completion, φ, k) is **chopped-W**$[1]$-complete. For any $q > 3$, hardness follows immediately from the trivial observation that a 3CNF formula is also a q-CNF formula, since we only require that every clause contains *at most* q literals. For membership, note that since we know from the previous proposition that each Weighted q-SAT Completion is in $\mathbf{W}[1]$, we have that (Weighted q-SAT Completion, len, k) $\in$ **chopped-W**$[1]$, and clearly

$$(\text{Weighted } q\text{-SAT Completion}, \varphi, k) \leq^{\text{fpt}}_{\text{comp}} (\text{Weighted } q\text{-SAT Completion}, \text{len}, k)$$

since the length of the full instance is at most $O(|\varphi|)$.

We hence show that (Weighted 3SAT Completion, φ, k) is **chopped-W**$[1]$-hard. The idea is to reduce from (Independent Set, len, k), which is **chopped-W**$[1]$-complete (an immediate consequence of Proposition 2.34). This is going to be a superinstance reduction, since we are mapping all instances of the same size to a single reduced instance that is then configured by the side constraints (in this case the partial assignment).

The idea is that every graph of n nodes is sent to the following 3-CNF formula φ_n, consisting of variables $x_1, \ldots, x_n$, one per node in the original graph, plus some auxiliary variables $c_{i,j}$, one per pair $(i, j) \in [n]^2$, with $i \neq j$:

$$\varphi_n \coloneqq \bigwedge_{\substack{i,j \in [n] \\ i \neq j}} (\neg c_{i,j} \vee \neg x_i \vee \neg x_j)$$

The idea is then that each $c_{i,j}$ will be made true if and only if there is actually an edge from i to j in the graph. This information is not visible to the compilation function, but it is accessible to the wrapping function finishing the reduction and defining the partial assignment. Furthermore, since all the fresh $c_{i,j}$ will be assigned some value, we can safely set weight k for the rest of the assignment.

Note that this can all be written into a (len, k)-fpt-compilable function such that $g(x) = f(x, c(|G|, k))$, for some computable c and f as in the definition of fpt-comp reductions. Indeed, since

the formula is at most polynomially larger than φ, we can simply forget about c and let f do all the work. Furthermore, the bounding functions are just $s(k) = \{k\}$ and $t(\text{len}(x), k) = \{\varphi_1, \ldots, \varphi_{|\text{len}(x)|}\}$, which make the construction fit into the definition of fpt-comp reductions. □

It is worth noting that in the proof of the previous result, the formula φ is an *anti-monotone* 3-CNF formula: all the variables occur negated. This means the completeness result holds for also for Weighted Antimonotone q-SAT Completion, which is defined just like Weighted q-SAT Completion but for antimonotone formulas.

Theorem 3.3. *For every $q \geq 2$, the problem* (Weighted Antimonotone q-SAT Completion, φ, k) *is complete for* **chopped-W**[1].

3.2 The case of CSP Completion

We now address the opening question of this thesis: the parameterized compilability of CSP Completion. Recall that the problem is defined analogously to SAT Completion, as a constrained form of CSP.

CSP Completion	
Instance	An instance $I = \langle X, D, C \rangle$ of CSP and a partial assignment $\alpha : X \to D$.
Question	Is there an extension of α into a satisfying assignment for I?

We imagine the inputs to CSP Completion coming in two phases: first the instance I, which we can preprocess, and then the partial assignment α, which we cannot. We denote by (CSP Completion, I) the paramterization where we have compilability access to the CSP instance but not to the partial assignment.

Let us make some obvious observations regarding the classical complexity and classical compilability of this problem.

Proposition 3.4. CSP Completion *is* **NP**-*complete.*

Proof. Hardness is obvious from the fact that CSP is **NP**-complete. For membership, note that one can always "plug in" the partial assignment to get a simplified instance to CSP. □

Proposition 3.5. (CSP Completion, I) *is* **chopped-NP**-*complete.*

Proof. For membership, note how obviously (CSP Completion, I) $\leq^{\text{poly}}_{\text{comp}}$ (CSP Completion, len). For hardness, we can reduce (3SAT, len) to (CSP Completion, I). The idea is again the same superinstance reduction from 3SAT (as in Theorem 1.7), where we build all the possible clauses over three literals together with an additional flag that is assigned some value by the partial assignment, and then translate into CSP. □

3.2.1 Parameterized complexity of CSP Completion

Since CSP Completion is hard both in terms of classical complexity and compilability, we turn to parameterized complexity. We consider the extra parameterization u, giving us the number of variables left undefined by the partial assignment. Clearly, (SAT Completion, u) $\in$ **FPT**. Unfortunately, (CSP Completion, u) is likely not in **FPT**. We show this by noting that different parameterizations of CSP and CSP Completion are all **W**[1]-complete. Notationally, for the following theorem, given a CSP instance $I = \langle X, D, C \rangle$, we denote by $|X|$ the parameterization giving us the number of variables and by $|C|$ the one giving us the number of constraints.

Theorem 3.6. *The following sequence of reductions holds:*

$$(\text{CSP Completion}, u) \leq_{\text{fpt}} (\text{CSP}, |X|) \leq_{\text{fpt}} (\text{CSP}, |C|) \leq_{\text{fpt}} (\text{Clique}, k) \leq_{\text{fpt}} (\text{CSP Completion}, u).$$

Proof.

(i) $(\text{CSP Completion}, u) \leq_{\text{fpt}} (\text{CSP}, |X|)$

Given a CSP instance and a partial assignment, instantiate the partial assignment by reducing the relation set of each constraint, getting an equivalent CSP instance. The reduced instance is now defined over u variables, since the others already received some value, so the reduction is correct and preserves parameter dependencies.

(ii) $(\text{CSP}, |X|) \leq_{\text{fpt}} (\text{CSP}, |C|)$

Observe that, without loss of generality, a CSP instance over $|X|$ variables can never have more than $2^{|X|}$ constraints. To see this, note how all constraints can be normalized as follows:

- first, fix some ordering $x_1, \ldots, x_n$ of the variables in X and for every constraint $(X_i, R_i) \in C$, sort the variables in the tuple X_i according to the order $x_1, \ldots, x_n$;
- if two constraints $c_1 = (X_1, R_1)$ and $c_2 = (X_2, R_2)$ are such that $X_1 = X_2$ and $R_1 \neq R_2$, then we can simply add the constraint $(X_1, R_1 \cap R_2)$ and remove c_1 and c_2.

After this normalization, there can only be one constraint per possible subset of variables, and there are only $2^{|X|}$ of those. So, without loss of generality, $|C| \leq 2^{|X|}$, which keeps the new parameter, $|C|$, bounded in terms of the old one, $|X|$.

(iii) $(\text{CSP}, |C|) \leq_{\text{fpt}} (\text{Clique}, k)$

Consider an instance $I = \langle X, D, C \rangle$ to CSP. We build a graph $G_I = (V_I, E_I)$ for Clique as follows. Let $C = \{c_1, \ldots, c_m\}$ be the set of constraints, where each constraint is the form $c_i = (X_i, R_i)$ for some $X_i \subseteq X$ and $R_i \subseteq D^{|X_i|}$. The nodes of our graph are all the possible pairs between a constraint and each of the tuples in its relation set. That is,

$$V_I := \bigcup_{(X_i, R_i) \in C} \{(X_i, t) \mid t \in R_i\}.$$

Then, the edge relation E_I is defined such that nodes (X_i, t) and (X_j, t') are connected if and only if they are not conflicting. Two nodes are conflicting if the tuples t and t' disagree on the value assigned to some shared variable. Finally, let $k := |C|$. It is easy to see that I is satisfiable if and only if G_I has a clique of size $|C|$. Furthermore, it is also straightforward to check that the size of the graph is at most quadratic in the size of the original CSP instance.

(iv) $(\text{Clique}, k) \leq_{\text{fpt}} (\text{CSP Completion}, u)$

This is just the usual encoding of Clique into CSP. Take a graph $G = (V, E)$, and define a CSP instance consisting of variables $X := \{x_1, \ldots, x_k\}$ and domain $D := V$, with the intended meaning that x_i is interpreted by the vertex that happens to be the i-th element of the clique. Then, for every $i \neq j$, we add a constraint imposing $(x_i, x_j) \in E$. Since G does not have self-loops, we do not need to worry about x_i and x_j being assigned to the same vertex. There are only $k(k-1)/2$ constraints, so we can feasibly construct the graph. Though we are technically reducing to CSP Completion, we will leave the partial assignment empty, meaning that the undefined number of variables is $u = |X| = k$, which makes this a correct fpt reduction.

□

Since (Clique, k) is $\mathbf{W}[1]$-complete, the previous sequence of reductions immediately implies our desired parameterized complexity classification.

Corollary 3.7. *The parameterized problems* (CSP, $|X|$), (CSP, $|C|$) *and* (CSP Completion, u) *are all complete for* $\mathbf{W}[1]$.

3.2.2 Parameterized compilability of CSP Completion

At this point, CSP Completion has been proven intractable in terms of classical complexity (**NP**-complete), classical compilability (**chopped-NP**-complete) and parameterized complexity ($\mathbf{W}[1]$-complete). Can it be of any help to combine both the compilation and the second parameterization? Unfortunately, it seems like this is not the case. To prove hardness for (CSP Completion, I, u), we give a superinstance reduction from the **chopped-**$\mathbf{W}[1]$-complete problem (Clique, len, k).

Theorem 3.8. (Clique, len, k) $\leq^{\text{fpt}}_{\text{comp}}$ (CSP Completion, I, u).

Proof. Let (G, k) be an instance of Clique of $|G| = n$ vertices. We will build a "superinstance", such that all instances of Clique over n vertices looking for cliques of size k get mapped to the same CSP instance. Then, the partial assignment will "configure" the superinstance to represent the graph G at hand.

We start by defining the CSP instance $I_{n,k} = \langle X_{n,k}, D_{n,k}, C_{n,k}\rangle$. First, define the set $P := \{(a, b) \mid a, b \in [n], a \neq b\}$ of possible edges on a graph of n nodes.

For the set $X_{n,k}$ of variables, define

$$X_{n,k} := \{e_{i,j} \mid i, j \in [k], i \neq j\} \cup \{a_{i,j,e} \mid i, j \in [k], i \neq j, e \in P\}.$$

Intuitively, the variable $e_{i,j}$ will be the edge connecting the i-th vertex in the clique to the j-th vertex in the clique. The variables $a_{i,j,e}$ will take binary values, indicating whether the edge e is really in the graph received as input, and hence a possible option of the edge going from the i-th vertex to the j-th vertex of the clique.

For the domain, consider the set $D_{n,k} := P \cup \{0, 1\}$. The values in P will be the potential edges of the graph, while the values 0 and 1 will be used to represent what edges are available.

Finally, define three types of constraints:

1. Vertices are forward-fixed: for all $i, j, \ell \in [k]$, $i \neq j$, $i \neq \ell$, $j \neq \ell$,
$$(e_{i,j}, e_{i,\ell}) \in \{((a, b), (a, c)) \mid a, b, c \in [n], a \neq b, a \neq c\}$$
2. Vertices are backwards-fixed: for all $i, j \in [k]$, $i \neq j$,
$$(e_{i,j}, e_{j,i}) \in \{((a, b), (b, a)) \mid a, b \in [n], a \neq b\}$$
3. Availability: for all $i, j \in [k], i \neq j, e \in P$,
$$(e_{i,j}, a_{i,j,e}) \in \{(e', 1) \mid e' \in P\} \cup \{(e', 0) \mid e' \in P \setminus \{e\}\}$$

Now, on a specific input (G, k), with $G = (V, E)$ and $|V| = n$, consider the partial variable assignment $\alpha_{n,k}$ that maps for all $i, j \in [k]$, $i \neq j$ and $e \in P$,

$$\begin{cases} a_{i,j,e} \mapsto 1 & \text{if } e \in E \\ a_{i,j,e} \mapsto 0 & \text{if } e \notin E \end{cases}$$

and take the CSP Completion instance $(I_{n,k}, \alpha_{n,k})$, consisting of the CSP instance we just constructed together with the partial assignment just defined.

Hence, on input $x = (G, k)$, the reduction map $g(x) = f(x, c(n, k))$ simply ignores the function c and uses f to build the CSP superinstance $I_{n,k}$ as described above. As for the parameters,

- the second parameter u is the number of undefined variables, which in this case is $k(k-1)/2$, since after defining the partial assignment, the only variables left are the ones of the form $e_{i,j}$, defining the edges of the clique, which is just a complete graph of size k. Hence, consider the function $s(k) := \{k(k-1)/2\}$, which is trivially computable and fpt-bounded in k;
- for the compilable parameter, that is the CSP superinstance, we just build all the superinstances $I_{m,k}$, for $m \leq |\text{len}(x)|$. That is, define $t(\text{len}(x), k) := \{I_{m,k} \mid m \leq |\text{len}(x)|\}$ and note indeed that since $n \leq |\text{len}(x)|$, $I_{n,k} \in t(\text{len}(x), k)$.

Observe that for every instance (G, k) of Clique, $(G, k) \in$ Clique if and only if $(I_{n,k}, \alpha_{n,k}) \in$ CSP Completion. This completes the reduction. □

This shows our desired classification.

Theorem 3.9. $(\text{CSP Completion}, I, u)$ *is* **chopped-W**$[1]$*-complete.*

Proof. For membership, Corollary 3.7 implies $(\text{CSP Completion}, u) \in \mathbf{W}[1]$, and hence by definition of the **chopped**- classes, $(\text{CSP Completion}, \text{len}, u) \in$ **chopped-W**$[1]$. Since clearly

$$(\text{CSP Completion}, I, u) \leq^{\text{fpt}}_{\text{comp}} (\text{CSP Completion}, \text{len}, u)$$

we have $(\text{CSP Completion}, I, u) \in$ **chopped-W**$[1]$.

Hardness follows directly from the previous reduction (Theorem 3.8). □

3.2.3 Primal treewidth

Based on the previous result, our methodology theorem immediately implies that $(\text{CSP Completion}, I, u) \notin$ **fpt-comp-FPT** unless $\mathbf{W}[1] \subseteq$ **FPT**/fpt. This suggests that u is not a very good parameterization. What other parameters could prove useful?

The immediate one that comes to mind is *primal treewidth*. Given a CSP instance $I = \langle X, D, C \rangle$, the primal graph of I, $G_I = (V_I, E_I)$, is a graph having as vertices the variables in I, $V_I = X$, such that two nodes are connected if and only if the corresponding variables appear together in some constraint in I. The primal treewidth of I, denoted $\text{ptw}(I)$, is the treewidth of the primal graph G_I of I.

It turns out that, if the domain of the CSP instance is bounded, CSP parameterized by primal treewidth is in **FPT**! This is because there is an fpt-algorithm that processes each bag of the tree decomposition of the CSP instance in time $O(|D|^{\text{ptw}(I)})$, where $|D|$ is the size of the domain [GSS02].

It is then natural to ask whether primal treewidth can be of any help for CSP Completion. Perhaps by using the additional power of compilation, one may be able to have fpt compilation without bounding the size of the domain. In other words, we would like to study the problem $(\text{CSP Completion}, I, \text{ptw})$, meaning that we compile the CSP instance (and not the partial assignment), and the size of the compilation can be fpt-bounded by $\text{ptw}(I)$.

Unforturnately, not even such a powerful compilation can help. In fact, we do not need to go very far to prove such a hardness result. The reduction from Clique in Theorem 3.8 has its treewidth bounded by k!

Theorem 3.10. $(\text{Clique}, \text{len}, k) \leq^{\text{fpt}}_{\text{comp}} (\text{CSP Completion}, I, \text{ptw})$.

Proof. Consider the same superinstance reduction as in Theorem 3.8. We just need to argue that the primal treewidth of this superinstance is bounded by some function of k. To do this, we build a good possible tree decomposition of the primal graph of the superinstance and reason about its width.

Consider the following labelled tree T. It consists of a root node, labelled by the bag $E := \{e_{i,j} \mid i, j \in [k], i \neq j\}$. Then, for every variable of the form $a_{i,j,e}$, consider a leaf node that is reached from the root node. Each such leaf node is labelled by the bag $\{a_{i,j,e}\} \cup E$.

Note how this labelled tree T is a tree decomposition of the primal graph of the superinstance. First, it is a tree. Second, every variable appears in some bag: every $e_{i,j}$ variable appears everywhere, and every $a_{i,j,e}$ variable appears exactly once, in some leaf. Third, for every two variables connected in the primal graph, they appear together in some node of the tree. Indeed, since in the primal graph two variables are connected if they appear together in some constraint, we can easily see that all edges of the primal graph are covered by our tree decomposition. Finally, for every variable x of the CSP superinstance, the subgraph obtained form the nodes in which x is represented forms a subtree. Indeed, if x is of the form $e_{i,j}$, then the subtree is the entire tree itself. If x is of the form $a_{i,j,e}$, then it is just a leaf node, which is trivially a tree.

To complete the proof, note every node has at most $k^2 - k + 1$ variables in the bag. Hence, the width of T is $k^2 - k$. We safely conclude that the primal treewidth of the superinstance is $O(k^2)$, which means that the previous reduction is a valid fpt-comp reduction from $(\textsc{Clique}, \text{len}, k)$. □

Corollary 3.11. $(\text{CSP } \textsc{Completion}, I, \text{ptw})$ *is hard for* **chopped-W**$[1]$.

The reader may notice that this time we only proved hardness and not completeness for **chopped-W**$[1]$. The reason is simple: we cannot prove membership in **chopped-W**$[1]$ because, in fact, we do not even know if $(\text{CSP } \textsc{Completion}, \text{ptw}) \in \mathbf{W}[1]$. This is also unknown for the original unconstrained CSP. It is definitely the case that (CSP, ptw) is $\mathbf{W}[1]$-hard, since

$$(\textsc{Clique}, k) \leq_{\text{fpt}} (\text{CSP}, \text{ptw}) \leq_{\text{fpt}} (\text{CSP } \textsc{Completion}, \text{ptw})$$

following the simple reduction described in Theorem 3.6.iv, but it is not known whether (CSP, ptw) fpt-reduces into $(\textsc{Clique}, k)$.

We come back to the issue of treewidth measures in CSP in Chapter 5.

3.3 The problem Clique Completion

The other well-known $\mathbf{W}[1]$-complete problem is of course $(\textsc{Clique}, k)$. Its completion variant is defined as follows.

Clique Completion	
Instance	An undirected graph $G = (V, E)$, two subsets $I, O \subseteq V$ and a value $k \in \mathbb{N}$.
Question	Is there a $(k + \lvert I\rvert)$-clique in G containing all the vertices in I and none of the ones in O?

For our purposes, the problem of interest is $(\textsc{Clique Completion}, \langle G, k\rangle, k)$, meaning that we can compile the graph together with the size of the extension of the clique.

Firstly, we note that the parameterized complexity of Clique Completion stays the same.

Proposition 3.12. $(\textsc{Clique Completion}, k)$ *is* $\mathbf{W}[1]$*-complete.*

Proof. Hardness is clear. For membership, note how one can "plug in the constraints". If a vertex is forced to be out of the clique then simply delete it, and if a vertex is enforced to be inside the clique, then delete all the vertices that are not in its neighborhood. This yields a graph without side constraints that has a k-clique if and only if the constrained instance had a clique of size $k + |I|$. □

Interestingly, the parameterized compilability of Clique Completion was already studied by De Haan in his doctoral dissertation [Haa19], under the name of Constrained Clique. His hardness result was for the class $\|\!\rightsquigarrow\mathbf{W}[1]$, one of the classes defined by Chen in his first framework for parameterized compilation [Che05], extending the CDLS setting, but the reduction he provides from the problem Multicolored Clique immediately gives us **chopped-W**$[1]$-completeness in our framework.

Theorem 3.13 (De Haan [Haa19, Proposition 15.55])**.** *The problem* (Clique Completion, $\langle G, k\rangle, k$) *is* **chopped-W**[1]*-complete.*

Though the problem Clique Completion is interesting in its own right, it is particularly useful in that it lets us complete our picture of compilability around **W**[1]. Firstly, it can be used to complete the proof of Theorem 3.2, where we showed **chopped-W**[1]-completeness for Weighted q-SAT Completion, since our proof strategy did not work for the case $q = 2$.

Proof of the case $q = 2$ in Theorem 3.2. The proof strategy presented for $q \geq 3$ adapting the classic superinstance technique breaks for the case $q = 2$. This is because every time we reduce from Weighted q-SAT Completion parameterizing by the length of the input, we are forced to introduce auxiliary variables to act as flags, but this yields a $(q+1)$-CNF formula. For the case $q = 2$, we cannot reduce from Weighted 1-SAT because this problem is in **FPT** when parameterized by the weight of the assignments.

Instead, we reduce from (Clique Completion, $\langle G, k\rangle, k$), which we now know is **chopped-W**[1]-complete. Since we now have access to the actual graph, our reduction can be the usual fpt reduction: on input $G = (V, E)$, build the formula

$$\bigwedge_{\substack{(i,j)\notin E \\ i\neq j}} (\neg x_i \vee \neg x_j)$$

over variables $x_1, \ldots, x_n$, where $n = |V|$. The constraints of Clique Completion can be plugged in into a partial assignment, so that the remaining part of the assignment is required to have weight k.

Furthermore, note that the formula is still a 2-CNF antimonotone formula, so the result we had for Weighted Antimonotone q-SAT Completion (Theorem 3.3) also holds now for $q = 2$. □

Furthermore, the astute reader may have noticed that De Haan's hardness result for Clique Completion opens an alternative route to **chopped-W**[1]-completeness for CSP Completion.

Alternative proof of Theorem 3.9. Membership in **chopped-W**[1] is proved in the same way, but we interchange the reduction from Theorem 3.8 showing

$$(\text{Clique}, \text{len}, k) \leq^{\text{fpt}}_{\text{comp}} (\text{CSP Completion}, I, u)$$

for a simpler reduction showing

$$(\text{Clique Completion}, \langle G, k\rangle, k) \leq^{\text{fpt}}_{\text{comp}} (\text{CSP Completion}, I, u).$$

Crucially, because we can now have access to the actual graph G, we no longer need to build a superinstance. This means the the usual reduction from Clique into CSP (as in Theorem 3.6.iv) almost does the trick. The only issue is with the in/out constraints of Clique Completion. The in constraints can be modelled using the partial assignment, but the out constraints cannot. Instead, we do the following.

On input $(\langle G = (V, E), k\rangle, I, O)$ with vertices $V = \{v_1, \ldots, v_n\}$ we consider as variables

$$X := \{x_1, \ldots, x_k\} \cup \{r_1, \ldots, r_n\}$$

where we have the usual k variables interpreting the clique, together with n auxiliary variables to indicate restrictions. The idea is that the r_i-variables are binary and indicate whether vertex v_i is in the set O of out constraints. The domain is $D := V \cup \{0, 1\}$.

Hence, our CSP instance consists first of $O(k^2)$ constraints of the form

$$(x_i, x_j) \in E \text{ for every } i, j \in [k], i \neq j$$

as usual, followed by constraints of the form

$$(x_i, r_j) \in \{(v, 0) \mid v \in V\} \cup \{(v, 1) \mid v \in V \setminus \{v_j\}\}$$

for every $i \in [k]$ and every $j \in [n]$. Observe that this CSP instance can be easily computed from $\langle G, k \rangle$, and the partial assignment to the CSP can now easily encode the I and O sets: the elements in I can be modeled by a partial assignment, and for every $v_i \in V$, $r_i \mapsto 1$ if $v_i \in O$, and $r_i \mapsto 0$ otherwise. After the partial assignments, the only variables left unassigned are at most the k variables $x_1, \ldots, x_k$, meaning that the new parameter u fulfills that $u \leq k$, as desired.

It is easy to see that the instance of Clique Completion has a clique if and only if the CSP Completion instance can be satisfied, giving correctness to the reduction. □

In fact, it is not difficult to see that the previous construction also preserves low primal treewidth, as in Theorem 3.10, which means the previous redution can also be used to prove Corollary 3.11: (CSP Completion, I, ptw) is **chopped-W**[1]-hard.

Chapter 4

Compilability around W[2]

The previous chapter applied our new parameterized compilability framework to the completion variants of three **W**[1]-complete problems: Weighted q-SAT Completion, CSP Completion and Clique Completion. Of course, our framework was defined in greater generality and can be applied beyond **W**[1]. The natural candidate to continue our research is the next level of the Weft hierarchy: **W**[2].

This chapter studies the compilability of completion variants of two classical **W**[2]-complete problems: Dominating Set and Hitting Set. Section 4.1 introduces the problems (four in total) and the new types of constraints. We show that these completion variants of Dominating Set and Hitting Set remain **W**[2]-complete and hence ask for the possibility of compiling the main part of the input, but not the side constraints. Section 4.2 shows a web of fpt-comp reductions and poly-comp reductions relating all four new problems to two canonical problems in **W**[2]: Weighted CNF SAT and Weighted Monotone CNF SAT. Such a web of reductions gives **chopped**-**NP**-completeness for all of the problems under consideration. Unfortunately, not all of the reductions can be made into proper fpt-comp reductions and, for the parameterized setting, we can only show **chopped**-**W**[2]-completeness for one problem (Hitting Set Completion), while the other three remain **chopped**-**W**[1]-hard for the moment.

4.1 The problems Hitting Set and Dominating Set Completion

This chapter focuses on two canonical **W**[2]-complete problems: Hitting Set and Dominating Set [DF13, Corollary 23.2.2]. The former is a simple set-theoretic puzzle consisting of some universe set and some subsets of it, where the task is to find a subset of the universe that "hits" at least one element in each subset. The formal statement is as follows.

Hitting Set	
Instance	A universe $U = \{u_1, \ldots, u_n\}$, a set of sets $S = \{S_1, \ldots, S_m\} \subseteq \mathcal{P}(U)$ and a natural number k.
Question	Is there a *hitting set* $H \subseteq U$ of size $\lvert H\rvert = k$? That is, a subset H such that for every $i \in [m]$, $S_i \cap H \neq \emptyset$.

When referring to the **W**[2]-completeness of Hitting Set we are referring to the problem (Hitting Set, k), where we parameterize by the desired size of the hitting set.

Example 4.1. Consider the Hitting Set instance consisting of the universe $U = \{u_1, u_2, u_3, u_4\}$ and sets $S_1 = \{u_1, u_2\}$, $S_2 = \{u_1, u_3, u_4\}$ and $S_3 = \{u_2, u_3\}$, together with the value $k = 2$. It is easy to see that there is no singleton hitting set, but $\{u_1, u_2\}$, $\{u_1, u_3\}$ or $\{u_2, u_4\}$ are possible hitting sets of two elements. ■

We now define two completion variants of Hitting Set, considering side constraints.

Simple Hitting Set Completion (S-HS-C)	
Instance	An instance $\langle U, S, k\rangle$ of Hitting Set together with sets $I, O \subseteq U$.
Question	Is there a hitting set $H \subseteq U$ of size $k + \lvert I\rvert$ such that $I \subseteq H$ and $H \cap O = \emptyset$?

The problem requires that some elements appear in the hitting set while others cannot. Observe that this imposes no constraints on the elements of S. The following more general version of the problem adds such a third type of constraint, specifying that even if a particular element appears in some specific set, hitting it cannot suffice to hit the set.

Hitting Set Completion (HS-C)	
Instance	An instance $\langle U, S, k\rangle$ of Hitting Set together with sets $I, O \subseteq U$ and a set $A \subseteq S \times U$.
Question	Is there a hitting set $H \subseteq U$ of size $k + \lvert I\rvert$ such that $I \subseteq H$, $H \cap O = \emptyset$ and for every $i \in [m]$, $H \cap (S_i \setminus \{u \in U \mid (u, S_i) \in A\}) \neq \emptyset$?

Observe how the third type of constraint is encoded: in the set A, a pair (u, S_i) indicates that even if $u \in S_i$, u is not available and something else must be hit.

It is not difficult to see that the parameterized complexity of these new versions of Hitting Set remains the same.

Proposition 4.2. *The problems*

- (Simple Hitting Set Completion, k)
- (Hitting Set Completion, k)

are both $\mathbf{W}[2]$*-complete.*

Proof. Hardness is clear since Hitting Set reduces into both of the completion variants by having no constraints. For membership, note that Simple Hitting Set Completion reduces into Hitting Set Completion, again by having no constraints of the third type, and we can quickly reduce Hitting Set Completion to Hitting Set: on input $(\langle U, S, k\rangle, I, O, A)$, we first delete, for each $(u, S_i) \in A$, the occurrence of u in S_i; then, delete all the elements in O and all its occurrences in sets in S; finally, delete all sets hit by something in I and delete the elements in I. Since the hitting set had to be of size $k + \lvert I\rvert$, the value of k can remain the same. It is straightforward to see that this way of "plugging in" the constraints gives an equivalent instance. Membership then follows from the closure of $\mathbf{W}[2]$ under fpt reductions. □

Note, of course, that the previous reductions are also polynomial-time reductions, and hence **NP**-completeness of Hitting Set transfers to Simple Hitting Set Completion and Hitting Set Completion.

The other problem we are interested in is Dominating Set. Given a graph $G = (V, E)$, a *dominating set* $D \subseteq V$ is a set of vertices such that for every $v \in V$, either $v \in D$ or v is adjacent to some $d \in D$. That is, every vertex is either dominated or adjacent to a dominated vertex. This can be alternatively formulated by requiring that for every $v \in V$, $N_G(v) \cap D \neq \emptyset$.

Dominating Set	
Instance	An undirected graph $G = (V, E)$ and a natural number k.
Question	Is there a dominating set $D \subseteq V$ for G of size k?

Example 4.3. Consider the graph in Figure 4.1.a, which has a dominating set of two elements marked in gray. The graph has no dominating set of smaller size. ■

We now present two completion variants for Dominating Set. The first one is analogous to how we defined Clique Completion: some vertices must be in, some must be out.

Simple Dominating Set Completion (S-DS-C)	
Instance	An instance of Dominating Set consisting of a graph $G = (V, E)$ and a number k, together with sets $I, O \subseteq V$.
Question	Is there a dominating set $D \subseteq V$ in G of size $k + \lvert I\rvert$ such that $I \subseteq D$ while $D \cap O = \emptyset$?

Observe that, by the way we defined the problem, the vertices in O cannot be in the dominating set and hence they must be dominated by some adjacent node. We now extend this problem with a third type of constraint: we add a third set $S \subseteq V$ of vertices such that they also cannot be in the dominating set, but unlike the vertices in O, these do not have to be dominated by anything else.

Dominating Set Completion (DS-C)	
Instance	An instance of Dominating Set consisting of a graph $G = (V, E)$ and a number k, together with sets $I, O, S \subseteq V$.
Question	Is there a dominating set D for the induced subgraph $G[V \setminus S]$ such that D is of size $k + \lvert I\rvert$, $I \subseteq D$ and $D \cap O = \emptyset$?

Example 4.4. In Figure 4.1.b we can see the same graph as in Example 4.3, but with added constraints $I = \{f\}$, $O = \{e\}$ and $S = \{c\}$, meaning that the graph no longer has dominating sets of two elements. The only possible dominating set of three elements has been marked in gray. ■

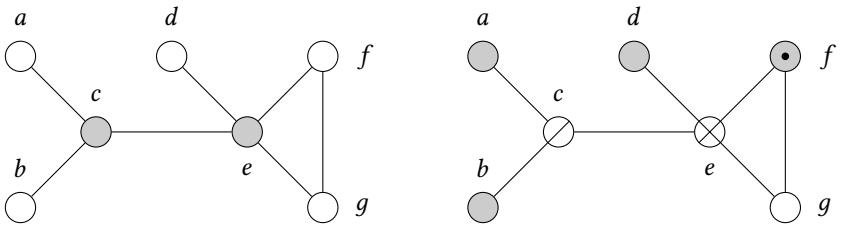
Once again, these additional constraints do not increase the parameterized complexity of the problem.

Proposition 4.5. *The problems*

- (Simple Dominating Set Completion, k)
- (Dominating Set Completion, k)

are both $\mathbf{W}[2]$*-complete.*

Proof. Hardness is clear, since Dominating Set reduces into both of the completion variants by having no constraints. For membership, note that Simple Dominating Set Completion reduces into Dominating Set Completion, again by having no constraints of the third type, so it suffices to show that Dominating Set Completion is in $\mathbf{W}[2]$. However, unlike in the analogous proofs for Clique Completion (Proposition 3.12) and Hitting Set Completion (Proposition 4.2), it is not immediately clear how one can "plug in" the constraints to get an instance of Dominating Set. This is because if some vertex has to be included in the dominating set, we cannot erase its neighbourhood since a vertex in such neighbourhood might be needed to dominate something else outside of it.



(a) Graph without constraints. (b) Same graph with side constraints.

Figure 4.1: On the left-hand side, the graph has a dominating set of two elements, $\{c, e\}$, highlighted in gray. On the right-hand side, the Dominating Set Completion instance adds side constraints: the vertices with a dot ($\odot$) must be included (they are in I), the ones crossed out ($\otimes$) cannot be in the dominating set but must be dominated (they are in O), and the ones with a single line ($\oslash$) cannot dominate and do not have to be dominated (they are the ones in S). The smallest possible dominating set satisfying the constraints is $\{a, b, d, f\}$.

Instead, we reduce Dominating Set Completion to Simple Hitting Set Completion, which we just showed is $\mathbf{W}[2]$-complete. On input $(\langle G = (V, E), k\rangle, I, O, S)$ we construct the following instance $(\langle U, S', k'\rangle, I', O')$ for Simple Hitting Set Completion. It consists of a universe $U := V \cup \{s_v \mid v \in V\}$ such that the sets in S' are just the neighborhoods of each vertex v, together with a corresponding flag vertex s_v. That is,

$$S' := \{N_G(v) \cup \{s_v\} \mid v \in V\}$$

with the intention that for every v, you can hit either v or something in the neighborhood. The size $k' := k$ of the solution sets stays the same. The flags s_v are there to encode the constraint set S of vertices that cannot be dominated but do not have to be dominated anyway. That is, define

$$I' := I \cup \{s_v \mid v \in S\} \qquad \text{and} \qquad O' := O \cup \{s_v \mid v \notin S\}.$$

Observe how this has the intended effect and gives correctness to the reduction: if a vertex v is in S and hence cannot be dominated but does not have to be, that means that the element s_v is in the hitting set, implying that the set $N_G(v) \cup \{s_v\}$ is automatically hit; if, on the other hand, $v \notin S$, then we must hit something in $N_G(v)$, since hitting s_v is no longer an option. Since the flag variables are all included in $I \cup O$, the number of elements we can additionally add to the hitting set is $k' = k$, so the reduction is correct and fits the problem statement. □

4.2 A web of reductions

We consider now the possibility of compiling the main part of the instance in the previous completion problems. That is, we are talking about problems like (Hitting Set Completion, $\langle U, S, k\rangle, k$) or (Dominating Set Completion, $\langle G, k\rangle, k$), where we can compile the Hitting Set instance or the graph, respectively, but not the side constraints. We will now show fpt-comp and poly-comp reductions between the newly defined completion problems, with the aim of obtaining new parameterized compilability classifications. To establish such a web of reductions, we make explicit two new problems that will help complete the picture.

Weighted CNF SAT	
Instance	A CNF formula φ and a natural number k.
Question	Is there a satisfying assignment for φ of weight k?

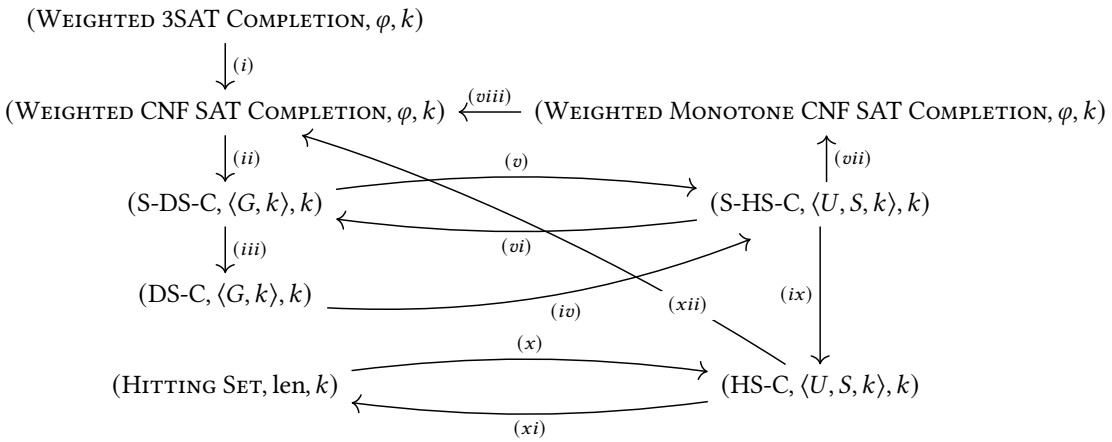
The problem (Weighted CNF SAT, k) is in fact the canonical $\mathbf{W}[2]$-complete problem [DF13, Theorem 23.2.1]. We shall also use the following restricted version, concerning monotone formulas. Recall that a Boolean formula in CNF is monotone if no literal occurs negated.

Weighted Monotone CNF SAT	
Instance	A monotone CNF formula φ and a natural number k.
Question	Is there a satisfying assignment for φ of weight k?

It is a classical result of parameterized complexity that (Weighted Monotone CNF SAT, k) is also $\mathbf{W}[2]$-complete [DF13, Corollary 23.2.2]. We define the completion variants of the previous two problems, Weighted CNF SAT Completion and Weighted Monotone CNF SAT Completion in the usual way, by adding a partial assignment to the input, as we did for Weighted q-SAT Completion in Section 3.1.

We are now ready to present fpt-comp and poly-comp reductions between all four new problems and the previous two.

Theorem 4.6. *The following web of reductions holds*



where all arrows denote fpt-comp reductions that are also poly-comp reductions (by dropping the last parameterzations), except for reduction (xii), which is only a poly-comp reduction.

Proof.

(i) (Weighted 3SAT Completion, φ, k) $\leq^{\text{fpt}}_{\text{comp}}$ (Weighted CNF SAT Completion, φ, k)
Every 3-CNF formula is also a CNF formula, so the reduction trivially leaves the formula unchanged.

(ii) (Weighted CNF SAT Completion, φ, k) $\leq^{\text{fpt}}_{\text{comp}}$ (S-DS-C, $\langle G, k\rangle, k$)
We invoke here the fpt reduction from Weighted CNF SAT to Dominating Set by Downey and Fellows [DF13, Lemma 23.2.1]. As noted in their Remark 23.2.1, such a reduction has the

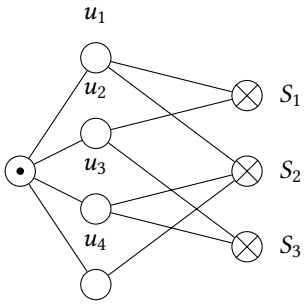


Figure 4.2: Incidence graph construction for the Hitting Set instance of Example 4.1 as described in reduction (vi). Nodes with a dot (⊙) must be in the dominating set and nodes that are crossed out (⊗) cannot in the set (but must me dominated by something else).

additional property that satisfying assignments are in one-to-one correspondence with the dominating sets of the graph obtained in the reduction. This entails, in particular, that partial assignments to the Boolean formula can be turned into input/output constraints for the graph, in such a way that we can get a valid input for Simple Dominating Set Completion. Since the original reduction was from formulas to graphs, $\langle G, k\rangle$ can be computed from φ, preserving the required dependencies, and the usual parameter dependency of fpt reductions ensures that this is an fpt-comp reduction.

(iii) $(\text{S-DS-C}, \langle G, k\rangle, k) \leq^{\text{fpt}}_{\text{comp}} (\text{DS-C}, \langle G, k\rangle, k)$
This is straightforward: every instance of Simple Dominating Set Completion is also an instance of Dominating Set Completion, simply by having no constraints of the third type.

(iv) $(\text{DS-C}, \langle G, k\rangle, k) \leq^{\text{fpt}}_{\text{comp}} (\text{S-HS-C}, \langle U, S, k\rangle, k)$
We can reuse the construction employed to reduce Dominating Set Completion to Simple Hitting Set Completion in Proposition 4.5. Note how the dependencies are clearly respected and the value of k stays the same, making this a correct fpt-comp reduction.

(v) $(\text{S-DS-C}, \langle G, k\rangle, k) \leq^{\text{fpt}}_{\text{comp}} (\text{S-HS-C}, \langle U, S, k\rangle, k)$
Again, we can reuse the construction from Proposition 4.5, except that we no longer need the auxiliary s_v variables, since there are no constraints of the third type.

(vi) $(\text{S-HS-C}, \langle U, S, k\rangle, k) \leq^{\text{fpt}}_{\text{comp}} (\text{S-DS-C}, \langle G, k\rangle, k)$
We describe an incidence graph to represent the Hitting Set instance. On input $(\langle U = \{u_1, \ldots, u_n\}, S = \{S_1, \ldots, S_m\}, k\rangle, I, O)$ consider a bipartite graph having as vertices $u_1, \ldots, u_n$ on one side and $S_1, \ldots, S_m$ on the other side, such that u_i is connected to S_j if and only if $u_i \in S_j$. All the vertices in U are themselves connected to a single node u_0. Then, as constraints, consider the sets $I' := I \cup \{u_0\}$ and $O' := O \cup S$. That is, the vertices $S_1, \ldots, S_m$ cannot be in the dominating set and must hence be dominated by some vertex in U. Finally, set $k' := k$. As an example, Figure 4.2 represents the incidence graph construction for the Hitting Set instance of Example 4.1.

Note indeed how the graph and the value of k can be obtained from the main part of the instance, in such a way that the side constraints depend solely on the previous side constraints. Hence, dependencies are preserved and since the only available vertices to add to the dominating set are $u_1, \ldots, u_n$, there is a one-to-one correspondence between hitting sets and dominating sets, giving correctness to the reduction.

(vii) $(\text{S-HS-C}, \langle U, S, k\rangle, k) \leq^{\text{fpt}}_{\text{comp}} (\textsc{Weighted Monotone CNF SAT Completion}, \varphi, k)$

On input $(\langle U = \{u_1, \ldots, u_n\}, S = \{S_1, \ldots, S_m\}, k\rangle, I, O)$, consider a Boolean formula over variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_m$ such that

$$C_i := \bigvee_{\substack{j \in [n] \\ u_j \in S_i}} x_j$$

and ask for assignments of weight $k' = k$. The side constraints can be immediately turned into a partial assignment, and correctness is straightforward to verify.

(viii) $(\text{W-Monotone-CNF-SAT-C}, \varphi, k) \leq^{\text{fpt}}_{\text{comp}} (\textsc{Weighted CNF SAT Completion}, \varphi, k)$

This follows from the trivial observation that a monotone CNF formula is also a CNF formula, so keeping the instance intact suffices for the reduction.

(ix) $(\text{S-HS-C}, \langle U, S, k\rangle, k) \leq^{\text{fpt}}_{\text{comp}} (\text{HS-C}, \langle U, S, k\rangle, k)$

Once again, an instance of Simple Hitting Set Completion is also an instance of Hitting Set Completion with no constraints of the third type.

(x) $(\textsc{Hitting Set}, \text{len}, k) \leq^{\text{fpt}}_{\text{comp}} (\textsc{Hitting Set Completion}, \langle U, S, k\rangle, k)$

We map all the instances of Hitting Set having $n = |U|$ elements and $m = |S|$ sets to a superinstance consisting of m sets, each of them a copy of U. Then, the constraints of the third type let us configure the superinstance, crossing out elements from the sets so that we get the actual S in the input. Observe that the superinstance can be computed easily by knowing $|U|$ and $|S|$, the value of k stays the same and this all makes an fpt-comp reduction. As usual, the bounding function $t(\text{len}(x), k)$ is simply generating all the possible superinstances up to size $|\text{len}(x)|$.

(xi) $(\text{HS-C}, \langle U, S, k\rangle, k) \leq^{\text{fpt}}_{\text{comp}} (\textsc{Hitting Set}, \text{len}, k)$

We already saw in Proposition 4.2 how to reduce Hitting Set Completion to Hitting Set, and, clearly the size of such a reduced instance is at most polynomially larger than the original input to Hitting Set Completion, so the parameter dependencies hold.

(xii) $(\text{HS-C}, \langle U, S, k\rangle) \leq^{\text{poly}}_{\text{comp}} (\textsc{Weighted CNF SAT}, \varphi)$

Consider an input to Hitting Set Completion consisting of an universe $U = \{u_1, \ldots, u_n\}$, sets $S = \{S_1, \ldots, S_m\}$ and constraint sets I, O, A. We build a CNF formula over $n + m \cdot n$ variables, as follows. First, take n variables $x_1, \ldots, x_n$, which will encode the hitting set, matching $u_1, \ldots, u_n$. Then, consider variables $y_{i,j}$, with $i \in [m]$, $j \in [n]$, and the clauses

$$C_1 = (y_{1,1} \vee \cdots \vee y_{1,n})$$

$$C_2 = (y_{2,1} \vee \cdots \vee y_{2,n})$$

$$\vdots$$

$$C_m = (y_{m,1} \vee \cdots \vee y_{m,n})$$

together with clauses of the form

$$y_{i,j} \rightarrow x_j$$

written as $\neg y_{i,j} \vee x_j$, for every $i \in [m]$, $j \in [n]$. Finally, the weight of the satisfying assignment should be $k + m$ and the partial assignment will do the following: for every $u_i \in I$, map $x_i \mapsto 1$; for every $u_i \in O$, map $x_j \mapsto 0$, and for every $(u_i, S_j) \in A$, map $y_{i,j} \mapsto 0$. Everything else is

left unassigned. This partial assignment carves the specific instance of Hitting Set into the formula. The idea is then that a hitting set becomes an assignment to the variables $x_1, \ldots, x_n$, and additionally, for each clause C_i (which corresponds to the set S_i), we make true some $y_{i,j}$ such that x_j is made true, meaning that "set S_i is hit by u_j".

Indeed, if there is a hitting set of size k, then mapping that into the variables $x_1, \ldots, x_n$ plus choosing some $y_{i,j}$ for every C_i makes for a satisfying assignment of weight $k + m$. On the other hand, suppose there is no hitting set of size k, but imagine for contradiction that there was a satisfying assignment of weight $k + m$. Such an assignment makes true at least one variable $y_{i,j}$ per clause C_i, and there are m of them, so that leaves a budget of at most k for the variables $x_1, \ldots, x_n$. Clearly, the assignment to those variables is immediately an hitting set on $u_1, \ldots, u_n$, which means there is a hitting set of size at most k, and thus also exactly of size k. Contradiction.

It is straightforward to check that none of the reductions (i)-(xi) use the extra power of fpt-comp reductions (that is, the parameters do not blow up and the computation can be carried out in polynomial time), which means that they are all poly-comp reductions as well. □

Remark 4.7 (Why not an fpt-comp reduction?). We shall briefly address the elephant in the room: why is reduction (xii) a poly-comp reduction and not an fpt-comp reduction like the rest? The reason is that the weight of the satisfying assignments in the formula is $k + m$, where m is the number of sets $|S|$. The issue is of course that we cannot bound $k + m$ with some computable function $f(k)$ depending solely on k as the definition of fpt-comp reductions requires. Surprisingly, we could not come up with an alternative reduction avoiding this issue!

What is more interesting, in all attempts to reduce from some (Q, len, λ) for some $\mathbf{W}[2]$-hard (Q, λ), we ran into the same issue: the new parameter tended to depend on something other than just λ. The fact that this barrier appears in every attempt to prove **chopped**-$\mathbf{W}[2]$-hardness for Simple Dominating Set Completion, Dominating Set Completion and Simple Hitting Set Completion suggests that there is some form of structural gap present here between the usual compilation problems coming from $\mathbf{W}[1]$ and the ones coming from $\mathbf{W}[2]$.

We suspect that the solution may be found in the additional properties of fpt-comp reductions. Observe that, in fact, none of the fpt reductions presented so far used any of the extra power given to this type of reduction. None of them used the compilation power, none of them produced compilation of fpt-size that were not also of polynomial size and none of them had a third parameter that blew up. In other words, all the reductions we encountered so far are also poly-comp reductions! This leads to the intuition that, in order to show **chopped**-$\mathbf{W}[2]$-hardness for all of the new completion variants, some of this extra power must be harnessed. ■

At first glance, the web of reductions in Theorem 4.6 might be somewhat overwhelming, but it neatly gives us our desired complexity classifications. Firstly, in classical terms.

Corollary 4.8. *The problems*

- (Weighted CNF SAT Completion, φ)
- (Weighted Monotone CNF SAT Completion, φ)
- (Simple Hitting Set Completion, $\langle U, S, k \rangle$)
- (Hitting Set Completion, $\langle U, S, k \rangle$)
- (Simple Dominating Set Completion, $\langle G, k \rangle$)
- (Dominating Set Completion, $\langle G, k \rangle$)

are all **chopped**-**NP**-*complete.*

Proof. Look at the web of reductions on Theorem 4.6. All six of the problems are interreducible under poly-comp reductions. Then, by reduction (x), (Hitting Set Completion, $\langle U, S, k\rangle$) is **chopped-NP**-hard, and hence so are all the rest. By reduction (xi), (Hitting Set Completion, $\langle U, S, k\rangle$) is in **chopped-NP**, and thus so are all the rest. □

For the parameterized case, the results differ a bit, since we cannot prove **chopped-W**[2]-hardness except for Hitting Set Completion.

Corollary 4.9. *The problems*

- (Weighted CNF SAT Completion, φ, k)
- (Weighted Monotone CNF SAT Completion, φ, k)
- (Simple Hitting Set Completion, $\langle U, S, k\rangle, k$)
- (Hitting Set Completion, $\langle U, S, k\rangle, k$)
- (Simple Dominating Set Completion, $\langle G, k\rangle, k$)
- (Dominating Set Completion, $\langle G, k\rangle, k$)

are all interreducible under fpt-comp reductions, they are all in **chopped-W**[2] *and they are all* **chopped-W**[1]*-hard. Furthermore,* (Hitting Set Completion, $\langle U, S, k\rangle$) *is* **chopped-W**[2]*-complete.*

Proof. Look at the web of reductions on Theorem 4.6. All the problems (with the exception of Hitting Set Completion) are interreducible under fpt-comp reductions, and reduction (i) from Weighted 3SAT Completion immediately gives **chopped-W**[1]-hardness (since we proved in Theorem 3.2 that Weighted 3SAT Completion is **chopped-W**[1]-complete). For membership in **chopped-W**[2], observe that they all reduce into Hitting Set Completion, which in turn reduces into (Hitting Set, len, k) via reduction (xi), giving membership. As for (Hitting Set Completion, $\langle U, S, k\rangle, k$), reduction (x) gives hardness for **chopped-W**[2]. □

Chapter 5

Inference problems for CSP

In Chapter 1 (Section 1.4), we motivated compilability based on different inference problems for propositional logic. In these problems, one receives as input some large propositional formula together with a query (another formula, often in some specific format) and is asked for logical entailment between the two: does every valuation making the first formula true also satisfy the second one? The question was then whether preprocessing the first formula could be of any use to handle the online queries faster.

We noted that inference problems immediately pointed at a structural gap in classical complexity: the problems Term Inference and Formula Inference are both **coNP**-complete yet seem to be fundamentally different in their structure. While Term Inference is efficiently compilable (a member of **poly-comp-P**), Formula Inference is not (unless **P** = **NP**). This difference invited us to survey the CDLS framework and its reductions, preserving compilability dependencies between the different parts of the instances. Furthermore, we concluded that Clause Inference, which sits naturally in between the constrained structure of Term Inference and the generality of Formula Inference, is also unlikely to be compiled efficiently (it is **chopped-NP**-complete, and hence not in **poly-comp-P** unless $\mathbf{PH} = \Sigma_2^{\mathrm{p}}$) —rather disappointing news overall.

It is then only natural to ask how these problems fit in our extension of the parameter compilation framework. Are they fpt-compilable under some reasonable additional parameterization, hence avoiding their classical uncompilability? Rather interestingly, this happens to be the case! In [Bov+16], Bova, De Haan, Lodha and Szeider studied several parameterizations for fpt-sized compilations of Clause Inference, achieving positive results for some treewidth-related measures that cannot, without compilation power, give fpt algorithms[1]. They even showed that these treewidth-related parameterizations perform well in practical implementations.

The motivating question for this chapter is whether those positive parameterizations can be extended to the realm of CSP. The chapter defines several inference problems for CSP, reminiscent of Term Inference, Clause Inference, and Formula Inference in propositional logic, and looks at the possibility of extending the existing positive parameterizations of Clause Inference to this setting. Since CSP can be seen as a general case of SAT, there is no doubt that these new problems will be intractable classically speaking. Nevertheless, it might be possible to transfer some of the positive parameterizations of Bova et al. to these new problems.

Section 5.1 reviews the ideas behind the existing positive results for Clause Inference, focusing on one particular parameterization: incidence treewidth modulo equivalence. Section 5.2 introduces three new inference problems involving CSP (Weak CSP Inference, Strong CSP Inference and General CSP Inference) and suggests the natural analogues of the treewidth parameterizations

[1] Though technically speaking these results were framed inside the parameterized extension of the CDLS framework defined by Chen [Che05] prior to the parameter compilation framework, they immediately transfer to our new setting.

of Bova et al. for the context of CSP. Section 5.2.1 addresses the intractability of these problems in terms of classical complexity and classical compilability, while Sections 5.2.2 and 5.2.3 finish by (unfortunately) showing hardness for Strong CSP Inference and General CSP Inference when additionally parameterized by the new treewidth measures.

5.1 Existing positive results for Clause Inference

The central insight of Bova et al. is that treewidth-related measures, which often yield fpt-time algorithms for SAT, can be extended to Clause Inference in a very efficient way with the help of compilation. It is worth noting that these are, to the best of our knowledge, the only existing positive results for parameterized compilation! We really know very little about what makes for good parameterized compilation algorithms and, in fact, as we will soon see, the techniques of Bova et al. are not genuinely new parameterizations, but rather refinements on parameters that already yield efficient fpt algorithms.

Recall that the treewidth of a graph G, denoted $\mathrm{tw}(G)$, is a measure of how close G is from being a tree. The intuition is that many intractable graph-theoretic problems become easy when the input graph is a tree. Hence, a good strategy to solve these problems is often to compute a tree decomposition of the graph and solve the problem via dynamic programming on that tree decomposition. This strategy can be extended to problems that are not graph-theoretic in nature, as long as the instances induce some good graph representation. This is the case of SAT, where one often works with the so-called incidence graphs and primal graphs of CNF formulas.

Definition 5.1 (Incidence and primal graphs and treewidths)**.** Let φ be a CNF formula consisting of clauses $C_1, \ldots, C_m$ over variables $x_1, \ldots, x_n$.

The *incidence graph* of φ is the bipartite graph having as nodes $x_1, \ldots, x_n$ on one side and $C_1, \ldots, C_m$ on the other, such that x_i is connected to C_j if and only if x_i occurs (possibly negated) in C_j. We denote by $\mathrm{itw}(\varphi)$ the *incidence treewidth* of φ, which is defined as the treewidth of the incidence graph of φ.

The *primal graph* of φ is the graph having as nodes $x_1, \ldots, x_n$ such that x_i is adjacent to x_j if and only if x_i and x_j occur together (negated or otherwise) in some clause of φ. We denote by $\mathrm{ptw}(\varphi)$ the *primal treewidth* of φ, which is defined as the treewidth of the primal graph of φ.

The following simple relation between incidence treewidth and primal treewidth will prove helpful.

Lemma 5.2 ([Sze03, Lemma 4])**.** *For every CNF formula φ,* $\mathrm{itw}(\varphi) \leq \mathrm{ptw}(\varphi) + 1$.

Interestingly, (SAT, itw) is in **FPT** [Sze03]. And, by the previous lemma, so is (SAT, ptw), since we can guarantee that there is an fpt-reduction from (SAT, ptw) to (SAT, itw). In fact, the dynamic programming technique on tree decompositions is so powerful that it extends to Clause Inference! In particular, (Clause Inference, itw) $\in$ **FPT** [Sze03], meaning that we parameterize by the incidence treewidth of the main formula, not that of the query.

If these parameterizations already give positive results for Clause Inference, why use compilation? The idea of Bova et al. is that, with the power of compilation, one can use the same dynamic programming techniques on some really efficient tree decomposition, generally not computable in fpt-time. In their work, they define up to four different treewidth-related measures, showing positive results for two of them and negative results for the other two. For our purposes, we focus solely on the primary treewidth measure providing efficient compilation: incidence treewidth modulo equivalence.

Definition 5.3 (Incidence treewdith modulo equivalence [Bov+16])**.** Let φ be a CNF formula. The *incidence treewidth modulo equivalence* of φ, denoted $\mathrm{itw}_{/\equiv}(\varphi)$, is the minimum incidence treewidth over all CNF formulas logically equivalent to φ. That is, taking $\equiv$ to denote logical equivalence,

$$\mathrm{itw}_{/\equiv}(\varphi) := \min_{\psi \equiv \varphi} \mathrm{itw}(\psi).$$

The goal is then to study the complexity of the problem (Clause Inference, φ, $\mathrm{itw}_{/\equiv}$), meaning that on input (φ, C) we compile φ and the size of the compilation can be of length fpt-bounded by $\mathrm{itw}_{/\equiv}(\varphi)$. Intuitively, one needs compilation because computing the equivalent formula attaining minimal treewidth cannot be done in time fpt-bounded in $\mathrm{itw}_{/\equiv}(\varphi)$. However, it can be done by some compilation function where the computation can be as expensive as desired!

This actually poses a small technical issue. In our framework (Definition 2.1), we define parameterizations to be functions that are fpt-time-computable with respect to themselves. This is precisely so that one can use as a parameter measures like treewidth, which are expensive to compute in general, but which can be obtained in fpt-time with respect to themselves. At the same time, letting parameterizations be computable instead of just fpt-time-computable with respect to themselves would be too lose, since, in particular, one would not be able to show that **FPT** = **para-P** (since the compilation function would not work in fpt-time if computing the parameter is very expensive). This all means that, technically speaking, $\mathrm{itw}_{/\equiv}$ is not a legal parameterization in our framework, since it is not fpt-time-computable with respect to itself.

For the purposes of this chapter, we let this small detail slide. The reasons are two. First, although it is true that allowing just computable parameterizations would slightly break our framework, $\mathrm{itw}_{/\equiv}$ has the nice property that it can be computed by the compilation function. That is, the parameter can be computed from the compilable part of the instance, which is reasonable to a certain extent. Furthermore, the negative results we will later show, which do use our framework, are for the usual treewidth (without modulo equivalence), which is fpt-computable, so it does not pose that big of a problem overall.

Applied to Clause Inference, the idea is then that on input (φ, C), where the question is whether $\varphi \models C$, we can use the compilation phase to obtain the equivalent formula $\psi \equiv \varphi$ attaining minimal treewidth and then execute the usual dynamic programming procedure to decide whether $\psi \models C$, with the online phase running in fpt-time parameterized by $\mathrm{itw}_{/\equiv}(\varphi)$. This is summarized in the following theorem, corresponding to Proposition 3, Corollary 1 and Theorem 1 of [Bov+16], which we rephrase in terms of our framework.

Theorem 5.4 (Bova et al. [Bov+16])**.** *The parameterized problem* (Clause Inference, $\mathrm{itw}_{/\equiv}$) $\notin$ **FPT** *unless* **P** = **NP**. *However,* (Clause Inference, φ, $\mathrm{itw}_{/\equiv}$) $\in$ **fpt-comp-FPT**.

Further work shows that one can apply this technique to a simpler notion of treewidth, called *incidence treewidth modulo backbones*, for which the same positive compilation applies. The main difference is that this alternative parameter can be more easily measured in practice and is smaller than incidence treewidth modulo equivalence. In fact, practical implementations using this parameterization perform surprisingly well, as exhibited by the experimental results of Bova et al. themselves.

Observe that the idea behind incidence treewidth modulo equivalence also makes sense in the context of CSP since the notions of incidence graph and primal graph can be defined analogously. Like in Definition 5.1, we can consider the incidence treewidth of a CSP instance I, $\mathrm{itw}(I)$, to be the treewidth of its incidence graph, which is the bipartite graph having variables on one side and constraints on the other, such that a variable and a constraint are connected if the variable occurs in that constraint. Similarly, the primal treewidth of a CSP instance I, $\mathrm{ptw}(I)$, can be defined as the treewidth of the primal graph of I, which is the graph having as nodes the variables in I such that two variables are connected if and only if they appear together in some constraint.

Unfortunately, treewidth measures do not yield efficient fpt-algorithms for CSP, or at least not unconditionally. As we discussed in Section 3.2.3, an instance I of CSP can be solved in fpt-time, assuming that the domain size is bounded. That is, the problem $(\mathrm{CSP}_{|D|\leq d}, \mathrm{ptw})$ is in **FPT** for every fixed $d \in \mathbb{N}$, meaning that we consider the domain size to be bounded in advance. However, as part of that discussion, we already saw that the additional power of compilation does not prove helpful in avoiding the need to bound the domain size. Recall from Corollary 3.11 that (CSP Completion, I, ptw)

is **chopped-W**[1]-hard and hence not in **fpt-comp-FPT** unless **W**[1] $\subseteq$ **FPT**/fpt. The next section extends this hardness result to different inference problems for CSP, where we showcase that the hardness pointed at in Corollary 3.11 extends to two of those inference problems as well.

5.2 Inference problems for CSP

We start by defining three inference problems involving CSP instances. Like in propositional logic, we use the symbol $\models$ to indicate semantic entailment: every assignment satisfying the left-hand side must also satisfy the right-hand side.

Recall that we looked at three different problems in propositional logic: Term Inference, Clause Inference, and Formula Inference. For Term Inference, the translation into CSP is immediate.

Weak CSP Inference	
Instance	A CSP instance I and a partial assignment α, written as a constraint consisting of a single tuple in its relation space.
Question	Does $I \models \alpha$?

If entailing a partial assignment is the CSP equivalent of entailing a conjunction of literals, then the following more general form of inference is in some sense analogous to Clause Inference.

Strong CSP Inference	
Instance	A CSP instance I and an additional constraint C, defined over the same variables and domain as I.
Question	Does $I \models C$?

Finally, one may think that following this pattern the CSP analogue for Formula Inference is the problem of deciding whether one CSP instance entails another. However, this turns out to be not a very interesting problem since it can be reduced to solving multiple instances of Strong CSP Inference, one per constraint in the second CSP instance. This is analogous to how the problem of whether a Boolean formula entails another CNF formula can be decomposed into multiple queries to Clause Inference. Instead, we need to introduce some form of disjunction that can achieve greater expressive power. For convenience, given two CSP instances I and J defined over the same variables and domain, we say that a partial assignment α satisfies $I \vee J$ if it satisfies all the constraints in I or all the constraints in J (or both).

General CSP Inference	
Instance	CSP instances $I, J_1, \ldots, J_m$ defined over the same variables and domain.
Question	Does $I \models J_1 \vee \cdots \vee J_m$?

We can now define some suitable parameterizations to study these problems' parameterized complexity and compilability. We denote by (Weak CSP Inference, itw) the parameterized problem where we do not have compilation power, and the running time can be fpt-bounded by itw, while (Weak CSP Inference, I, itw) is the problem Weak CSP Inference where we compile the instance I on the left-hand side and the size of this compilation can be of fpt-size in ptw(I). Similarly, we extend this notation to the other two inference problems.

We will be interested in a four-step analysis for each of the three previous problems (following the flowchart in Figure 1.1). For example, for Strong CSP Inference, we shall ask:

1. What is the classical complexity of Strong CSP Inference?
2. What is the parameterized complexity of (Strong CSP Inference, itw)?
3. What is the classical compilability complexity of (Strong CSP Inference, I)?
4. What is the parameterized compilability complexity of (Strong CSP Inference, I, itw)?

The first question can be quickly answered already: all three new problems are precisely as hard under the lens of classical structural complexity.

Proposition 5.5. *The problems* Weak, Strong *and* General CSP Inference *are all* **coNP***-complete.*

Proof. Membership in **coNP** is immediate. For hardness, see that

$$\begin{aligned}\overline{\textsc{Clique}} &\leq_{\mathrm{p}} \textsc{Weak CSP Inference}\\ &\leq_{\mathrm{p}} \textsc{Strong CSP Inference}\\ &\leq_{\mathrm{p}} \textsc{General CSP Inference}\end{aligned}$$

and recall that $\overline{\textsc{Clique}}$ is **coNP**-complete, by virtue of Clique being **NP**-complete. Recall the usual reduction from Clique to CSP (as described, for example, in Theorem 3.6.iv), and add an empty constraint (an empty assignment) to complete the instance of the CSP inference problem. Indeed, the graph does not have a k-clique if and only if the CSP instance entails the empty constraint. For the other two problems, observe that an instance of Weak CSP Inference is already an instance of Strong CSP Inference, and similarly, an instance of Strong CSP Inference is an instance of General CSP Inference. □

The previous hardness results are, of course, totally expected. It is then only natural to ask, analogous to Bova et al., whether adding treewidth as a parameter can make things easier. Unfortunately, it does not look like it —at least not without compilation power.

Proposition 5.6. *The problems* Weak, Strong *and* General CSP Inference *are all* **coW**$[1]$*-hard for all four of the following parameterizations:*

- *incidence treewidth (*itw*);*
- *primal treewidth (*ptw*);*
- *incidence treewidth modulo equivalence (*$\mathrm{itw}_{/\equiv}$*);*
- *primal treewidth modulo equivalence (*$\mathrm{ptw}_{/\equiv}$*).*

Proof. The proof is identical to the previous one showing **coNP**-hardness. It suffices to show that $(\overline{\textsc{Clique}}, k)$, which is **coW**$[1]$-complete, reduces to (Weak CSP Inference, ptw). Observe that the primal graph of the CSP instance is just the complete graph of size k, so the primal treewidth is $k-1$, ensuring that the new parameter is bounded by the old one.

For hardness under the other three parameterizations, it suffices to note that for every CSP instance I, $\mathrm{itw}(I) \leq \mathrm{ptw}(I)+1$ (from Lemma 5.2), and obviously $\mathrm{itw}_{/\equiv}(I) \leq \mathrm{itw}(I)$ and $\mathrm{ptw}_{/\equiv}(I) \leq \mathrm{ptw}(I)$, which gives **coW**$[1]$-hardness in all cases.

For the other two problems, we use again the fact that they are of ascending complexity. That is,

$$\begin{aligned}(\textsc{Weak CSP Inference}, \mathrm{ptw}) &\leq_{\mathrm{fpt}} (\textsc{Strong CSP Inference}, \mathrm{ptw})\\ &\leq_{\mathrm{fpt}} (\textsc{General CSP Inference}, \mathrm{ptw}).\end{aligned}$$

□

Fortunately, the one positive preliminary observation we can make is that Weak CSP Inference is easy to compile, just as it happened for Term Inference.

Proposition 5.7. (Weak CSP Inference, I) $\in$ **poly-comp-P**.

Proof. The compilation uses the same technique as for Term Inference. Given a CSP instance I, list all the satisfying assignments and check whether all of them send some variable always to the same value. Store all such variables together with the values they are assigned to in the output of the compilation. Now, when given a particular partial assignment, it suffices to check whether it matches part of the one we compiled. □

The following sections continue the analysis of Strong CSP Inference and General CSP Inference, which, rather naturally, turn out to be hard to compile. We shall show that the two problems are hard to compile even if we allow the compilation to be parameterized by primal treewidth. This implies that parameterized by primal or incidence treewidth modulo equivalence, like for Clause Inference, cannot help since those measures are smaller than primal treewidth.

5.2.1 Classical compilability for Strong and General CSP Inference

Our next task is to study the problems (Strong CSP Inference, I) and (General CSP Inference, I), where we are allowed to compile the left-hand side of the instance. Naturally, since these problems can be conceptualized as a general version of Clause Inference and Formula Inference, it comes as no surprise that they are not compilable.

We now show directly how these problems are indeed hard for classical compilation. First, we need a simple technical lemma.

Lemma 5.8. *Let φ be a Boolean formula, let C be a clause and let C' be the clause C after removing all literals of variables not occurring in φ. Then, $\varphi \models C$ if and only if $\varphi \models C'$.*

Proof. The backwards direction is immediate: if $\varphi \models C'$, then it must also entail C since C can only be at most a larger clause.

For the forward direction, suppose for contradiction that there is a variable y occuring in C as literal ℓ_y but not in φ, such that $C = C_0 \vee \ell_y$. Now, suppose for contradiction that $\varphi \models C$ but $\varphi \not\models C_0$. Then, there is a valuation α such that $\alpha \models \varphi$ but $\alpha \not\models C_0$. This means that $\alpha \models \ell_y$. But now consider the valuation α' that switches the value assigned to y. Since y does not occur in φ, $\alpha' \models \varphi$, and since $\varphi \models C$, it is also the case that $\alpha' \models C$. However, $\alpha' \not\models \ell_y$, so it must be that $\alpha' \models C_0$. But then $\alpha \models C_0$. Contradiction. □

Theorem 5.9. *The following two poly-comp reductions hold:*

(i) (Clause Inference, φ) $\leq^{\text{poly}}_{\text{comp}}$ (Strong CSP Inference, I)*;*

(ii) ($\overline{\text{Clique}}, \epsilon$) $\leq^{\text{poly}}_{\text{comp}}$ (General CSP Inference, I).

Proof.

(i) Given an instance (φ, C) of Clause Inference over variables $x_1, \ldots, x_n$, we do the following: convert φ to an equisatisfiable 3CNF formula in the usual way. It is easy to see that $\varphi \models C$ if and only if the translation entails the same clause. Now, convert the 3CNF encoding of φ into a CSP instance in the usual way (include one constraint per clause). On top of this, we will add a new variable x_ℓ and we will extend the domain of the CSP instance with the set of all possible literals, $\{x_1, \ldots, x_n, \neg x_1, \ldots, \neg x_n\}$.

This new variable x_ℓ will be interpreted with the literal that is made true in the entailed clause. Hence, we need to add the following constraints:

$$(x_\ell, x_1) \in \{(x_1, 1), (\neg x_1, 0)\} \cup \{(x_i, b), (\neg x_i, b) \mid i \neq 1, b \in \{0, 1\}\}$$
$$(x_\ell, x_2) \in \{(x_2, 1), (\neg x_2, 0)\} \cup \{(x_i, b), (\neg x_i, b) \mid i \neq 2, b \in \{0, 1\}\}$$
$$\vdots$$
$$(x_\ell, x_n) \in \{(x_n, 1), (\neg x_n, 0)\} \cup \{(x_i, b), (\neg x_i, b) \mid i \neq n, b \in \{0, 1\}\}$$

Now, if the clause being checked is $C = \ell_1 \vee \cdots \vee \ell_m$, we can add as the entailed constraint

$$x_\ell \in \{\ell_1, \ldots, \ell_m\}$$

which completes the reduction. This is indeed a poly-comp reduction since the new CSP instance can be obtained directly from the formula φ. The only problem would be that C contained variables that did not appear in φ, but these variables can be easily erased, as justified by the previous lemma (Lemma 5.8).

(ii) We give a reduction from the complement of Clique. The idea is first to transform an instance $(G = (V, E), k)$ of Clique into a CSP instance $I_{G,k}$ in the usual way (like in Theorem 3.6.iv). Recall that each constraint in $I_{G,k}$ will be of the form $((x_i, x_j), E)$. For each such constraint C, take its complement, $((x_i, x_j), \overline{E})$. Then, consider the General CSP Inference instance consisting of the trivial CSP instance on the left-hand side, made true by every assignment, and the disjunction of all the complements of each constraint in $I_{G,k}$ on the other. The graph will not have a k-clique if and only if the trivial CPS entails that one of the constraints is always violated. Observe that the compilable part of the new instance is just some trivial CSP that is always true, so it can be obtained from ϵ, making this a correct poly-comp reduction.

□

Remark 5.10. It is interesting to note that instead of reducing $\overline{(\text{Clause Inference}, \varphi)}$ into (Strong CSP Inference, I), we could have also reduced directly from $\overline{\text{SAT Completion}}$ instead and get the same hardness result. This is because, given an input (φ, α) to SAT Completion, if we write α as a conjunction of literals, then $(\varphi, \alpha) \notin$ SAT Completion if and only if $\varphi \models \neg\alpha$, since $\neg\alpha$ can be immediately rewritten as a clause. Then, the idea is the same as in the reduction from Clause Inference: introduce an extra variable that takes the value of the literal in the "assignment" clause that is made false. ■

The following corollary immediately follows.

Corollary 5.11.

(i) (Strong CSP Inference, I) *is* **chopped-coNP**-*complete and hence not in* **poly-comp-P** *unless* $\mathbf{PH} = \Sigma_2^{\mathrm{p}}$;

(ii) (General CSP Inference, I) *is* **poly-comp-coNP**-*complete and hence not in* **poly-comp-P** *unless* $\mathbf{P} = \mathbf{NP}$.

Remark 5.12. Note that we can get a weaker lower bound for General CSP Inference simply by the fact that (Strong CSP Inference, I) $\leq_{\text{comp}}^{\text{poly}}$ (General CSP Inference, I), just by seeing the extra constraint as a full CSP instance. However, this would only entail that (General CSP Inference, I) $\notin$ **poly-comp-P** unless **PH** collapses at the second level, which may not imply $\mathbf{P} = \mathbf{NP}$. ■

5.2.2 Parameterized compilability for Strong CSP Inference

At this point, we ruled out three possible routes towards the tractability of Strong CSP Inference: the problem is **coNP**-complete (Proposition 5.5), compiling the left-hand side cannot help unless **PH** collapses (Corollary 5.11) and the standard treewidth measures do not seem to yield fpt-time algorithms (Proposition 5.6).

We now combine the parameterized complexity and compilation approaches, considering the problem (Strong CSP Inference, I, ptw), where we compile the CSP instance, and the compilation can be of size fpt-bounded by its primal treewidth. Recall that this approach worked for Clause Inference. Unfortunately, the following reduction showcases that the same approach is likely to fail for CSP.

Theorem 5.13. $(\overline{\text{Clique}}, \text{len}, k) \leq^{\text{fpt}}_{\text{comp}}$ (Strong CSP Inference, I, ptw).

Proof. Observe that the existing reduction from Clause Inference used in Theorem 5.9.i does not work anymore, since (Clause Inference, ptw) $\in$ **FPT** [Sze03], so it does not entail **chopped-coW**[1]-hardness.

The proof uses instead the reduction we already described in Theorem 3.10 showing that

$$(\text{Clique}, \text{len}, k) \leq^{\text{fpt}}_{\text{comp}} (\text{CSP Completion}, I, \text{ptw})$$

by taking complements. That is, the existing reduction also shows that

$$(\overline{\text{Clique}}, \text{len}, k) \leq^{\text{fpt}}_{\text{comp}} (\overline{\text{CSP Completion}}, I, \text{ptw})$$

so it would suffice to show that that

$$(\overline{\text{CSP Completion}}, I, \text{ptw}) \leq^{\text{fpt}}_{\text{comp}} (\text{Strong CSP Inference}, I, \text{ptw}).$$

Let (I, α) be an input of CSP Completion. We want to encode that "every complete assignment satisfying I disagrees with α at some point". The idea is to use an auxiliary variable x_α, such that under some complete assignment β satisfying I, x_α is interpreted by a pair (x, v), such that $\beta(x) = v \neq \alpha(x)$, hence pointing at a variable on which α and β disagree.

We now describe this reduction in more detail. Given the CSP instance $I = \langle X, D, C\rangle$, we define a new instance $I' = \langle X', D', C'\rangle$ as follows. We add an extra variable, such that $X' := X \cup \{x_\alpha\}$ and we expand the domain with all pairs of old variables and values: $D' := D \cup (X \times D)$. Now, assuming $X = \{x_1, \ldots, x_n\}$, we add the following constraints:

$$(x_\alpha, x_1) \in \{((x_1, v), v) \mid v \in D\} \cup \{((x_i, v), v') \mid i \neq 1, v, v' \in D\}$$

$$\vdots$$

$$(x_\alpha, x_n) \in \{((x_n, v), v) \mid v \in D\} \cup \{((x_i, v), v') \mid i \neq n, v, v' \in D\}$$

and the entailed constraint

$$x_\alpha \in \{(x, v) \mid x \in \text{dom}(\alpha), v \in D \setminus \{\alpha(x)\}\}.$$

See that $(I, \alpha) \notin$ CSP Completion if and only if I' entails the extra constraint, so the reduction works.

It now suffices to argue that this is indeed an fpt-comp reduction. Indeed, the reduction is computable in polynomial time, and the new CSP instance I' can be directly obtained from the old I.

As for the primal treewidth, suppose I has primal treewidth k for some optimal tree decomposition T. It suffices to add x_α to every bag of T to get a tree decomposition of the new instance, ensuring that the new treewidth is at most $k + 1$, so it qualifies as an fpt-comp reduction. □

Hardness follows immediately.

Corollary 5.14. (Strong CSP Inference, I, ptw) *is* **chopped-coW**$[1]$*-hard.*

Of course, the general methodology theorem then implies that if the problem is in **fpt-comp-FPT**, then **coW**$[1] \subseteq$ **FPT**/fpt. But since **FPT**/fpt is closed under complementation, we still get the usual conditional lower bound: (Strong CSP Inference, I, ptw) $\notin$ **fpt-comp-FPT** unless **W**$[1] \subseteq$ **FPT**/fpt.

The same comment as at the end of Section 3.2.3 applies here. We cannot show membership in **chopped-coW**$[1]$ because we do not know whether (Strong CSP Inference, ptw) is itself a member of **coW**$[1]$, which eventually boils down to the problem of whether (CSP, ptw) is in **W**$[1]$.

5.2.3 Parameterized compilability for General CSP Inference

The reader must have noticed that our previous hardness result (Corollary 5.14) immediately implies that (General CSP Inference, I, ptw) is not in **fpt-comp-FPT** unless **W**$[1] \subseteq$ **FPT**/fpt, simply because instances of Strong CSP Inference are also instances of General CSP Inference, so hardness is transferred. However, the problem General CSP Inference is so general that having an efficient fpt-compilation would have even greater consequences than for Strong CSP Inference. This is analogous to how we can easily show that Formula Inference is not in **poly-comp-P** unless **P** = **NP**, while for Clause Inference we can only show the collapse of the polynomial hierarchy to the second level.

We can prove that General CSP Inference is hard for **fpt-comp-coW**$[1]$, reusing a reduction we already presented. Recall that in light of Proposition 2.31 this is the same as showing **simple-coW**$[1]$-hardness since **fpt-comp-C** = **simple-C** for every **C**.

Theorem 5.15. $(\overline{\text{Clique}}, \epsilon, k) \leq^{\text{fpt}}_{\text{comp}}$ (General CSP Inference, I, ptw).

Proof. Since we have nothing to compile, we will be mapping an instance (G, k) of Clique to a sequence of CSP instances $J_1, \ldots, J_m$ in such a way that there is no k-clique if and only if $\top \models J_1 \vee \cdots \vee J_m$.

Consider the graph (G, k) and take the usual CSP encoding. For each constraint, consider the same constraint where the relation is its complement. The instances $J_1, \ldots, J_m$ are instances consisting of individual relations. This is essentially the same reduction that we presented in Theorem 5.9.ii. Observe that the reduction works and that it can be obtained in polynomial time. As for the primal treewidth, note that the left-hand side of the instance is a trivially true CSP, and so its treewdith is constant, so trivially bounded by k, as desired. □

Corollary 5.16. (General CSP Inference, I, ptw) *is* **fpt-comp-coW**$[1]$*-hard.*

Therefore, if (General CSP Inference, I, ptw) $\in$ **fpt-comp-FPT**, we get that **coW**$[1] \subseteq$ **FPT**, and since **FPT** is closed under complementation, **W**$[1] \subseteq$ **FPT**. This in turn entails that the Exponential Time Hypothesis (ETH) fails, a stronger claim that the inclusion **W**$[1] \subseteq$ **FPT**/fpt derived in previous sections.

Chapter 6

Conclusion

We began this work wondering about what compilation looks like beyond polynomial size. We were interested in whether combining compilability with parameterized tractability could become a powerful tool for efficient precomputation. Although the approach seemed to work for a problem like SAT Completion, we suspected the general case of CSP Completion was more difficult.

Our goal was to develop a hardness theory extending Chen's parameter compilation framework that could let us classify computational problems in terms of the complexity of their parameterized compilability. This framework, presented in Chapter 2, studies doubly parameterized problems, extends the framework of parameterized complexity, and classifies problems around our newly defined **fpt-comp-C** (Definition 2.5) and **chopped-C** (Definition 2.22) classes, proving hardness results with the aid of fpt-comp reductions (Definition 2.13).

Our work not only developed the theoretical framework to establish hardness results for parameterized compilability but also applied it to specific problems. Chapter 3 and Chapter 4 both applied our framework to different doubly parameterized problems coming from canonical problems from the parameterized classes $\mathbf{W}[1]$ and $\mathbf{W}[2]$.

In Chapter 3 we looked at the problems Weighted q-SAT Completion, CSP Completion and Clique Completion. In all three cases, we concluded that while classical compilability cannot help, parameterized compilability cannot do much either. In particular, we addressed the opening questions of the thesis, regarding the difference between SAT Completion and CSP Completion. While the former is fpt-compilable when parameterized by the number of variables left unassigned by the partial assignment, the same parameterization does not work for CSP Completion unless $\mathbf{W}[1] \subseteq \mathbf{FPT}/\text{fpt}$ (Theorem 3.9).

In Chapter 4 we applied the framework to problems in the class $\mathbf{W}[2]$. We defined completion variants for the problems Hitting Set and Dominating Set and studied their compilability. We concluded that the new Simple Hitting Set Completion, Hitting Set Completion, Simple Dominating Set Completion and Dominating Set Completion all remain $\mathbf{W}[2]$-complete when parameterized by the size of the hitting sets and dominating sets, respectively (Proposition 4.2 and Proposition 4.5) and that they are all **chopped-NP**-complete when compiling the main part of the instance but not the side constraints (Corollary 4.8). For parameterized compilability, we were able to show some hardness results in the form of **chopped-W**$[1]$-hardness for these problems, and we were able to show **chopped-W**$[2]$-completeness for Hitting Set Completion. We come back to this in the discussion about future work.

Finally, Chapter 5 studied the power of treewidth for compilation of CSP instances in the context of inference problems, where traditionally compilability has been well studied. Unlike for propositional logic, unfortunately, these treewidth measures do not give positive compilability results (unless $\mathbf{W}[1] \subseteq \mathbf{FPT}/\text{fpt}$ or $\mathbf{FPT} = \mathbf{W}[1]$, depending on the problem), as shown in Corollaries 5.14 and 5.16.

Tables 6.1, 6.2 and 6.3 below summarize all our complexity results for all of the problems covered through this thesis. Note that some of the classifications are from the literature but have been nevertheless included for the sake of a more complete picture of what the complexity map looks like. See the theorems in question for the appropriate references to the literature.

Future work

We believe the existing framework can be used to further study the power of parameterized compilability. Beyond the application of this framework to higher and higher complexity classes, our summary table points to two aspects in which our work is incomplete. Firstly, on the full power of the reductions employed. Secondly, on the lack of positive results. We believe further research in parameterized compilability must tackle these issues sooner or later.

The power of fpt-comp reductions

We already noted in Remark 4.7 that essentially all of our fpt-comp reductions are also poly-comp reductions, in the sense that we do not use any of the additional power available to us. In fact, we do not use the additional compilation power even when they are poly-comp reductions.

Recall that fpt-comp reductions are more potent than polynomial-time reductions in that (i) the reduction can be carried out in fpt-time, (ii) although the parameter dependencies must be preserved, the new parameter can grow faster than polynomial, and (iii) the reduction can perform some expensive computation on the compilable part of the input. However, we did not use any of these features.

We do not think these features should be removed from fpt-comp reductions, though. We believe that our minds are just too used to thinking in terms of polynomial-time reductions that harnessing this extra power is counterintuitive. However, as we discussed in the context of Hitting Set Completion and Dominating Set Completion in Theorem 4.6, it seems like using this extra power is key to obtaining reductions showing hardness results for problems beyond $\mathbf{W}[1]$. At the time of writing, it remains an open problem to show that (Dominating Set Completion, $\langle G, k\rangle, k$) is **chopped-W**$[2]$-hard.

Positive results

Our focus on this thesis was to provide a hardness theory for parameterized compilability. It comes as no surprise that if all you have is a hammer, everything looks like a nail. Indeed, due to scope and time constraints, we could not focus on new parameterizations for positive results; instead, all of our theorems are negative in that they show conditional hardness for the problems under inspection.

We can make a couple of remarks. First, finding positive results seems like an arduous task. As we discussed in Chapter 1, parameterized compilability can be seen as a last resort against intractability once other approaches have failed. In this sense, it is rare that after having a problem that is hard in classical terms, in terms of classical compilability and in terms of parameterized complexity, the problem suddenly becomes tractable for some simple parameterization. In fact, the only positive parameterized compilability results we are aware of are the ones by Bova et al. [Bov+16] for Clause Inference discussed in Section 5.1, and one could argue these are not genuinely "new" results. After all, they are variations on existing parameterization that already yield fpt algorithms. Surely, they can perform well in practice, but it does not seem like the full power of compilability is being exploited here.

Secondly, note that parameterizations that are "bad" in parameterized complexity tend to remain "bad" in parameterized compilability. In all of the completion variants we defined from $\mathbf{W}[1]$-complete and $\mathbf{W}[2]$-complete problems, we reused the existing parameters for which parameterized hardness holds. In all cases, the extra power of compilation seems useless, suggesting that positive parameterizations must look beyond the obvious.

Problem	Parameterized complexity		Classical compilability		Parameterized compilability	
	Parameter	*Complexity*	*Compilable part*	*Complexity*	*Extra parameter*	*Complexity*
Weighted q-SAT Completion	k	**W**[1]-complete (Proposition 3.1)	φ	**chopped-NP**-complete	k	**chopped-W**[1]-complete (Theorem 3.2)
W-Antimonotone-q-SAT-C	k	**W**[1]-complete	φ	**chopped-NP**-complete	k	**chopped-W**[1]-complete (Theorem 3.3)
CSP Completion	u	**W**[1]-complete (Theorem 3.6)	I	**chopped-NP**-complete	u	**chopped-W**[1]-complete (Theorem 3.9)
	ptw	**W**[1]-hard (Section 3.2.3)			ptw	**chopped-W**[1]-hard (Corollary 3.11)
Clique Completion	k	**W**[1]-complete (Proposition 3.12)	$\langle G, k\rangle$	**chopped-NP**-complete	k	**chopped-W**[1]-complete (Theorem 3.13)

Table 6.1: Summary of results from Chapter 3.

Problem	Parameterized complexity		Classical compilability		Parameterized compilability	
	Parameter	*Complexity*	*Compilable part*	*Complexity*	*Extra parameter*	*Complexity*
Weighted CNF SAT Completion	k	**W**[2]-complete	φ	**chopped-NP**-complete (Corollary 4.8)	k	**chopped-W**[1]-hard (Corollary 4.9)
W-Monotone-CNF-SAT-C	k	**W**[2]-complete	φ	**chopped-NP**-complete (Corollary 4.8)	k	**chopped-W**[1]-hard (Corollary 4.9)
Simple Hitting Set Completion	u	**W**[2]-complete (Proposition 4.2)	$\langle U, S, k\rangle$	**chopped-NP**-complete (Corollary 4.8)	k	**chopped-W**[1]-hard (Corollary 4.9)
Hitting Set Completion	k	**W**[2]-complete (Proposition 4.2)	$\langle U, S, k\rangle$	**chopped-NP**-complete (Corollary 4.8)	k	**chopped-W**[2]-complete (Corollary 4.9)
Simple Dominating Set Completion	k	**W**[2]-complete (Proposition 4.5)	$\langle G, k\rangle$	**chopped-NP**-complete (Corollary 4.8)	k	**chopped-W**[1]-hard (Corollary 4.9)
Dominating Set Completion	k	**W**[2]-complete (Proposition 4.5)	$\langle G, k\rangle$	**chopped-NP**-complete (Corollary 4.8)	k	**chopped-W**[1]-hard (Corollary 4.9)

Table 6.2: Summary of results from Chapter 4.

Problem	Parameterized complexity		Classical compilability		Parameterized compilability	
	Parameter	*Complexity*	*Compilable part*	*Complexity*	*Extra parameter*	*Complexity*
Term Inference	ptw	**FPT**	φ	**poly-comp-P** (Section 1.4)	-	-
Clause Inference	ptw	**FPT**	φ	**chopped-coNP**-complete (Corollary 1.12)	$\text{itw}_{/\equiv}$	**fpt-comp-FPT** (Theorem 5.4)
Formula Inference	-	-	φ	**poly-comp-coNP**-complete (Theorem 1.6)	-	-
Weak CSP Inference	ptw	**coW**[1]-hard (Proposition 5.6)	I	**poly-comp-P** (Proposition 5.7)	-	-
Strong CSP Inference	ptw	**coW**[1]-hard (Proposition 5.6)	I	**chopped-coNP**-complete (Corollary 5.11)	ptw	**chopped-coW**[1]-hard (Corollary 5.14)
General CSP Inference	ptw	**coW**[1]-hard (Proposition 5.6)	I	**poly-comp-coNP**-complete (Corollary 5.11)	ptw	**fpt-comp-coW**[1]-hard (Corollary 5.16)

Table 6.3: Summary of results from Chapter 5.

Bibliography

[ADF95] Karl A. Abrahamson, Rodney G. Downey, and Michael R. Fellows. "Fixed-parameter tractability and completeness IV: On completeness for **W**[**P**] and **PSPACE** analogues". In: *Annals of pure and applied logic* 73.3 (1995), pp. 235–276.

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[BC19] Christoph Berkholz and Hubie Chen. "Compiling existential positive queries to bounded-variable fragments". In: *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 2019, pp. 353–364.

[BHM09] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of Satisfiability*. Vol. 185. IOS press, 2009.

[Bov+16] Simone Bova et al. *Positive and Negative Results for Parameterized Compilability*. Tech. rep. AC-TR-16-003. Algorithms and Complexity Group, TU Wien, 2016. URL: http://www.ac.tuwien.ac.at/files/tr/ac-tr-16-003.pdf.

[CDS96] Marco Cadoli, Francesco M. Donini, and Marco Schaerf. "Is intractability of nonmonotonic reasoning a real drawback?" In: *Artificial intelligence* 88.1-2 (1996), pp. 215–251.

[Cad+97] Marco Cadoli et al. "On compact representations of propositional circumscription". In: *Theoretical Computer Science* 182.1-2 (1997), pp. 183–202.

[Cad+99] Marco Cadoli et al. "The size of a revised knowledge base". In: *Artificial Intelligence* 115.1 (1999), pp. 25–64.

[Cad+02] Marco Cadoli et al. "Preprocessing of Intractable Problems". In: *Information and computation* 176.2 (2002), pp. 89–120. ISSN: 0890-5401.

[Che05] Hubie Chen. "Parameterized compilability". In: *IJCAI'05*. 2005, pp. 412–417.

[Che15] Hubie Chen. "Parameter Compilation". In: *10th International Symposium on Parameterized and Exact Computation*. 2015, p. 127.

[Che+21] Hubie Chen et al. "Semantic width and the fixed-parameter tractability of constraint satisfaction problems". In: *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*. 2021, pp. 1726–1733.

[Cyg+15] Marek Cygan et al. *Parameterized Algorithms*. Springer, 2015.

[DF13] Rodney G. Downey and Michael R. Fellows. *Fundamentals of Parameterized Complexity*. Springer, 2013.

[FG06] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Springer Berlin Heidelberg, 2006. ISBN: 9783540299530.

[FG03] Jörg Flum and Martin Grohe. "Describing parameterized complexity classes". In: *Information and Computation* 187.2 (2003), pp. 291–319.

[Gog+95] Goran Gogic et al. “The comparative linguistics of knowledge representation”. In: *IJCAI (1)*. 1995, pp. 862–869.

[GSS02] Georg Gottlob, Francesco Scarcello, and Martha Sideri. “Fixed-parameter complexity in AI and nonmonotonic reasoning”. In: *Artificial Intelligence* 138.1-2 (2002), pp. 55–86.

[GNR08] Jiong Guo, Rolf Niedermeier, and Daniel Raible. “Improved algorithms and complexity results for power domination in graphs”. In: *Algorithmica* 52.2 (2008), pp. 177–202.

[Haa19] Ronald de Haan. *Parameterized Complexity in the Polynomial Hierarchy*. Springer, 2019.

[IP01] Russell Impagliazzo and Ramamohan Paturi. “On the complexity of k-SAT”. In: *Journal of Computer and System Sciences* 62.2 (2001), pp. 367–375.

[IPZ01] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. “Which problems have strongly exponential complexity?” In: *Journal of Computer and System Sciences* 63.4 (2001), pp. 512–530.

[KL80] Richard M. Karp and Richard J. Lipton. “Some connections between nonuniform and uniform complexity classes”. In: *Proceedings of the twelfth annual ACM symposium on Theory of computing*. 1980, pp. 302–309.

[Kne+06] Joachim Kneis et al. “Parameterized power domination complexity”. In: *Information Processing Letters* 98.4 (2006), pp. 145–149.

[Mar07] Dániel Marx. “Can you beat treewidth?” In: *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS’07)*. IEEE. 2007, pp. 169–179.

[Nie06] Rolf Niedermeier. *Invitation to fixed-parameter algorithms*. Oxford University Press, 2006.

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994. ISBN: 0201530821.

[SK96] Bart Selman and Henry Kautz. “Knowledge compilation and theory approximation”. In: *Journal of the ACM (JACM)* 43.2 (1996), pp. 193–224.

[Sze03] Stefan Szeider. “On fixed-parameter tractable parameterizations of SAT”. In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2003, pp. 188–202.

[Yap83] Chee K. Yap. “Some consequences of non-uniform conditions on uniform classes”. In: *Theoretical computer science* 26.3 (1983), pp. 287–300.

Compendium of problems

Satisfiability and CSP

Weighted q-SAT	
Instance	A q-CNF formula φ and a natural number k.
Question	Is there a satisfying assignment that only sets k variables to true?

Weighted q-SAT Completion	
Instance	A q-CNF formula φ, a partial assignment α and a natural number k.
Question	Is there a satisfying assignment extending α that only sets k more variables to true?

Weighted Antimonotone q-SAT	
Instance	An antimontone q-CNF formula φ and a natural number k.
Question	Is there a satisfying assignment that only sets k variables to true?

Weighted Antimonotone q-SAT Completion	
Instance	An antimonotone q-CNF formula φ, a partial assignment α and $k \in \mathbb{N}$.
Question	Is there a satisfying assignment extending α that only sets k more variables to true?

Weighted CNF SAT	
Instance	A CNF formula φ and a natural number k.
Question	Is there a satisfying assignment that only sets k variables to true?

Weighted CNF SAT Completion	
Instance	A CNF formula φ, a partial assignment α and a natural number k.
Question	Is there a satisfying assignment extending α that only sets k more variables to true?

Weighted Monotone CNF SAT	
Instance	A montone CNF formula φ and a natural number k.
Question	Is there a satisfying assignment that only sets k variables to true?

Weighted Monotone CNF SAT Completion	
Instance	A monotone CNF formula φ, a partial assignment α and a number $k \in \mathbb{N}$.
Question	Is there a satisfying assignment extending α that only sets k more variables to true?

CSP	
Instance	An instance $I = \langle X, D, C \rangle$ to CSP.
Question	Is there satisfying assignment for I?

CSP Completion	
Instance	An instance $I = \langle X, D, C \rangle$ to CSP and a partial assignment $\alpha : X \to D$.
Question	Is there an extension of α into a complete satisfying assignment for I?

Inference problems

Term Inference	
Instance	A propositional formula φ and a term $T = \ell_1 \wedge \cdots \wedge \ell_k$.
Question	Does $\varphi \models T$?

Clause Inference	
Instance	A propositional formula φ and a clause $C = \ell_1 \vee \cdots \vee \ell_k$.
Question	Does $\varphi \models C$?

Formula Inference	
Instance	Two propositional formulas φ and ψ.
Question	Does $\varphi \models \psi$?

Weak CSP Inference	
Instance	A CSP instance I and a partial assignment α, written as a constraint consisting of a single tuple in its relation space.
Question	Does $I \models \alpha$?

Strong CSP Inference	
Instance	A CSP instance I and an additional constraint C, defined over the same variables and domain as I.
Question	Does $I \models C$?

General CSP Inference	
Instance	CSP instances $I, J_1, \ldots, J_m$ defined over the same variables and domain.
Question	Does $I \models J_1 \vee \cdots \vee J_m$?

Graph problems and hitting sets

Clique	
Instance	An undirected graph $G = (V, E)$ and a value $k \in \mathbb{N}$.
Question	Is there a k-clique in G?

Clique Completion	
Instance	An undirected graph $G = (V, E)$, two subsets $I, O \subseteq V$ and a value $k \in \mathbb{N}$.
Question	Is there a $(k + \lvert I\rvert)$-clique in G containing all the vertices in I and none of the ones in O?

Hitting Set	
Instance	A universe $U = \{u_1, \ldots, u_n\}$, a set of sets $S = \{S_1, \ldots, S_m\} \subseteq \mathcal{P}(U)$ and a natural number k.
Question	Is there a *hitting set* $H \subseteq U$ of size $\lvert U\rvert = k$? That is, a subset H such that for every $i \in [m]$, $S_i \cap H \neq \emptyset$.

Simple Hitting Set Completion (S-HS-C)	
Instance	An instance $\langle U, S, k\rangle$ of Hitting Set together with sets $I, O \subseteq U$.
Question	Is there a hitting set $H \subseteq U$ of size $k + \lvert I\rvert$ such that $I \subseteq H$ and $H \cap O = \emptyset$?

Hitting Set Completion (HS-C)	
Instance	An instance $\langle U, S, k\rangle$ of Hitting Set together with sets $I, O \subseteq U$ and a set $A \subseteq S \times U$.
Question	Is there a hitting set $H \subseteq U$ of size $k + \lvert I\rvert$ such that $I \subseteq H$, $H \cap O = \emptyset$ and for every $i \in [m]$, $H \cap (S_i \setminus \{u \in U \mid (u, S_i) \in A\}) \neq \emptyset$?

Dominating Set	
Instance	An undirected graph $G = (V, E)$ and a natural number k.
Question	Is there a dominating set $D \subseteq V$ for G of size k?

Simple Dominating Set Completion (S-DS-C)	
Instance	An instance of Dominating Set consisting of a graph $G = (V, E)$ and a number k, together with sets $I, O \subseteq V$.
Question	Is there a dominating set $D \subseteq V$ in G of size $k + \lvert I\rvert$ such that $I \subseteq D$ while $D \cap O = \emptyset$?

Dominating Set Completion (DS-C)	
Instance	An instance of Dominating Set consisting of a graph $G = (V, E)$ and a number k, together with sets $I, O, S \subseteq V$.
Question	Is there a dominating set D for the induced subgraph $G[V \setminus S]$ such that D is of size $k + \lvert I\rvert$, $I \subseteq D$ and $D \cap O = \emptyset$?