UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INFORMÁTICOS

Advanced Evaluation Techniques for (Non)-Monotonic Reasoning Using Rules with Constraints

PH.D. THESIS

Joaquín Arias

Departamentamento de Lenguajes y Sistemas Informáticos e Ingenieria de Software

Escuela Técnica Superior de Ingenieros Informáticos

Advanced Evaluation Techniques for (Non)-Monotonic Reasoning Using Rules with Constraints

Submitted in partial fulfillment of the requirements for the degree of:
Doctor of Philosophy in Computer Science

Author: **Joaquín Arias**

Advisor: **Dr. Manuel Carro**

February 2020

Abstract of the Dissertation

Constraint Logic Programming (CLP) is a declarative paradigm that extends Logic Programming (LP) with constraint solving capabilities over arbitrary domains that can be combined to model the problem to be solved. CLP brings additional expressive power to LP since constraints can very concisely capture complex relationships. That makes it easier to write programs that solve problems where no effective algorithm exists, and/or to adapt these programs when the problem specifications change. Additionally, the shift from "generate-and-test" to "constrain-and-generate" reduces the search tree and brings additional performance. CLP has been used in planning, scheduling, resource allocation, logistics, circuit design, and verification, among others. However, a CLP top-down execution strategy may enter loops in the presence of left recursion and/or non-stratified negation while a bottom-up execution strategy limits the range of admissible constraint domains, the places where constraints can appear, and the type (or number) of models that can be returned.

This thesis contributes to the state of the art of Tabled Constraint Logic Programming (TCLP), which suspends subsumed calls checking entailment (improving termination properties of CLP programs), and of Constraint Answer Set Programming (CASP), which computes the stable model semantics of CLP programs with negation.

First, we extend the theoretical foundation of TCLP by proving the soundness and completeness of a top-down operational semantics of TCLP that uses a richer and more flexible answer management strategy and we extend the proof of termination including arbitrary constraint domains (e.g., the Herbrand domain) even if they are not constraint-compact. Then, taking advantage of these extended properties we designed and implemented a generic TCLP framework, called Mod TCLP, which provides a clear and simple interface that facilitates the integration of different constraint domains with a tabling engine. Mod TCLP fully implements the call and answer entailment check improving performance and termination w.r.t. Prolog, tabling, CLP, and previous TCLP implementations. We validated the expressiveness and flexibility of Mod TCLP integrating several constraint solvers (one of them written in C) and we evaluated its performance in several benchmarks. We also used Mod TCLP to incrementally compute lattice-based aggregates providing a framework, ATCLP, based on a new semantics that views the aggregates as constraints, and uses the entailment and join relations of the lattice to define the operations to compute and combine aggregates. Finally, we applied Mod TCLP to re-implement the fixpoint algorithm of a state-of-the-art abstract interpreter where the tabling engine computes the fixpoint and the TCLP interface invokes the abstract domain to compute the LUB of the abstract substitutions. The resulting code of the abstract interpreter is simpler and shorter than the initial one and, in most cases, the resulting implementation is faster.

On the other hand, this thesis extends a goal-directed non-monotonic reasoner to compute CASP programs without the grounding phase required by most CASP systems. The resulting reasoner, called s(CASP), computes, given a query, the (partial) stable models, that due to the presence of non-stratified negation in CASP programs, could be more than one, and the justification tree with the terms and rules that support the query. We prove, through several examples, the enhanced expressiveness of s(CASP) system w.r.t. Prolog, ASP, CLP, and other ASP systems featuring constraints. We briefly discuss the efficiency of s(CASP) (which in some benchmarks outperforms a mature, highly optimized ASP system) and present a more complex application to exploit its expressiveness. Then, we present a real application that exploits the expressiveness of s(CASP). We present the implementation of an automated reasoner that uses Event Calculus to model commonsense reasoning with a sound, logical basis. Previous attempts to mechanize reasoning using EC faced difficulties in treating continuous change in dense domains (e.g., time and other physical quantities), constraints between variables, default negation, and the uniform application of different inference methods, among others. We show how EC scenarios can be elegantly modeled using the goal-directed execution model of s(CASP) and how its expressiveness makes it possible to perform deductive reasoning tasks in domains featuring constraints involving dense time and fluents with continuous properties.

Together, these results envision advantages on several fronts: complex queries and non-trivial reasoning can be easier to express thanks to the higher-level of logic programming and constraints; fewer computations are needed thanks to the automatic reuse of previous inferences (which in some sense will automatically perform dynamic programming); queries and associated actions (if any) can be programmed using the same formalism. The use of the resulting tools, Mod TCLP and s(CASP), makes it easier the translation of problem requirements into code and minimizes the amount of re-engineering needed to comply with the requirements when they change.

Resumen de la Tesis Doctoral

La programación lógica con restricciones (CLP) es un paradigma de programación declarativa que extiende la programación lógica (LP) con capacidades para resolver restricciones sobre diferentes dominios que puede combinarse para modelar el problema a resolver. CLP aporta mayor expresividad a LP, ya que las restricciones pueden representar relaciones complejas de manera concisa. Esto simplifica el desarrollo de programas que resuelven problemas para los que no existe un algoritmo eficaz y/o la adaptación de dichos programas cuando las especificaciones del problema cambian. Además, al cambiar de ”generar y probar” a ”restringir y generar” se reduce el árbol de búsqueda y se incrementa la eficiencia. CLP se ha utilizado, entre otras aplicaciones, en planificación, asignación de recursos, logística, diseño de circuitos y verificación. Sin embargo, una estrategia de ejecución *top-down* de CLP puede entrar en bucle debido a la recursión por la izquierda y/o la presencia de negación no estratificada, mientras que una estrategia de ejecución *bottom-up* limita el rango de dominios de restricción admisibles, donde pueden aparecer las restricciones y el tipo (o número) de modelos que se pueden obtener.

Esta tesis contribuye al estado del arte de la programación lógica con restricciones y tabulación (TCLP), que suspende llamadas más particulares comprobando *entailment* (haciendo que los programas CLP terminen en más casos), y de la programación lógica con restricciones con conjuntos de respuestas (Constraint Answer Set Programming, CASP), que evalúa programas CLP con negación usando la semántica de modelos estables.

En primer lugar, ampliamos los fundamentos teóricos de TCLP, demostrando corrección y completitud de una semántica operacional top-down de TCLP que utiliza una estrategia de gestión de respuestas más rica y flexible y extendemos las prueba de terminación incluyendo algunos casos de dominios de restricciones, como el dominio de Herbrand, que no son *constraint-compact*. Después, aprovechando estas propiedades extendidas, diseñamos e implementamos un entorno genérico de TCLP, llamado Mod TCLP, que proporciona una interfaz clara y sencilla que facilita la integración de diferentes dominios de restricciones con el módulo de tabulación. Mod TCLP implementa de manera completa la comprobación mediante entailment de llamadas y respuestas mejorando el rendimiento y la terminación con respecto a Prolog, tabulación, CLP e implementaciones previas de TCLP. Validamos la expresividad y flexibilidad de Mod TCLP integrando diferentes resolutores de restricciones (uno de ellos escrito en C) y evaluamos su rendimiento con varios benchmarks. También usamos Mod TCLP para calcular agregados sobre retículos de manera incremental mediante un framework, ATCLP, que se basa en una nueva semántica, y ve los agregados como restricciones y usa el entailment y la relación *join* del retículo para definir los agregados. Finalmente, aplicamos Mod TCLP para re-implementar el algoritmo de punto fijo de un intérprete abstracto de última generación donde el modulo de tabulación calcula el punto fijo y la interfaz de TCLP invoca el dominio abstracto para calcular el LUB de las sustituciones abstractas. El código resultante del intérprete abstracto es más simple y corto que el inicial y, en la mayoría de los casos, la implementación resultante es más rápida.

Por otro lado, esta tesis extiende un razonador no monótono y goal-directed para evaluar programas CASP sin la fase de grounding requerida por la mayoría de los sistemas CASP. El razonador resultante, llamado s(CASP), calcula, a partir de una consulta, los modelos estables (parciales), que debido a la presencia de negación no estratificada en programas CASP podrían ser mas de uno, y el árbol de justificación con los términos y las reglas que soportan la consulta. Demostramos, mediante varios ejemplos, la mejora en la expresividad de s(CASP) con respecto a Prolog, ASP, CLP, y otros sistemas ASP con restricciones. Evaluamos brevemente la eficiencia de s(CASP) (que en algunos casos supera a un perfeccionado y altamente optimizado sistema ASP). A continuación, presentamos una aplicación real que se beneficia de la expresividad de s(CASP). Presentamos la implementación de un razonador automático que usa Event Caculus para modelar razonamiento de sentido común con una base lógica sólida. Intentos anteriores de automatizar el razonamiento utilizando la IA se enfrentaron a dificultades en el manejo de: cambios en dominios continuos y densos (por ejemplo, tiempo y cantidades físicas), restricciones entre variables, negación por defecto y una utilización homogénea de diferentes métodos de inferencia, entre otros. Mostramos cómo distintos escenarios de EC pueden ser modelados elegantemente usando el modelo de ejecución goal-directed de s(CASP) y cómo su expresividad permite realizar tareas de razonamiento deductivo en dominios que representan restricciones involucrando tiempo denso y fluentes con propiedades continuas.

En conjunto, estos resultados arrojan ventajas en varios frentes: preguntas complejas y razonamiento no trivial son más fáciles de expresar gracias al mayor nivel de programación y restricciones lógicas; es necesaria una menor cantidad de cómputo gracias a la reutilización automática de datos inferencias previas (que, en cierto sentido, implementa automáticamente programación dinámicas); las consultas y las acciones asociadas (si las hubiere) pueden ser programadas usando el mismo formalismo. El uso de las herramientas resultantes, Mod TCLP y s(CASP), facilita la traducción de los requisitos del problema en código y minimiza la cantidad de reingeniería que es necesaria para adecuar los requisitos cuando estos cambian.

A las mujeres más importantes de mi vida,
M^a Victoria, Barbara, Diana, Dafne,
y a mi padre.

Acknowledgments

This thesis collects the results of about five years of research, during which I have learned, among other things, about scientific research and constraint logic programming. The thesis would not have been possible without the help and advise of people I met during this journey and to whom I am grateful.

To remember all the people that helped me with this thesis, I should first try to decide when this journey started. Most likely it started in the summer of 2013 when Manuel gave me "The art of Prolog" (Sterling and Shapiro, 1994) together with an exercise statement: Let `A` and `B` be two sorted lists of pairs `V-Pr`, then the output is a sorted list of pairs `VC-PrC`, such as $\mathtt{VC}_k$=$\mathtt{VA}_i$+$\mathtt{VB}_j$ and $\mathtt{PrC}_k=\sum_{i,j}(\mathtt{PrA}_i*\mathtt{PrB}_j)$, e.g., given the query `?- sadd([1-0.3, 2-0.7], [2-0.25, 3-0.4, 4-0.35], C)` the answer would be `C=[3-0.075,4-0.295,5-0.385,6-0.245]`.[1] I was able to solve this exercise thanks to the Functional Programming course I attended the previous semester, and I enjoyed Prolog thanks to the Logic course I attended during my first semester in Mathematics & Computer Science. Therefore, I could say that this journey started in 2011. However, I decided to start a new degree as a consequence of the collapse of the Spanish real estate bubble (I worked as an architect for 10 years) and Lehman Brothers in 2008, and I chose Mathematics & Computer Science because I like programming since I was a child and I learned the use of *GoTo* (later I read Dijkstra's "Go To Statement Considered Harmful"). So it is not clear to me when it all really started.

In a similar way I do not clearly know which are the geographical limits of this journey because during the work on the thesis, I have attended conferences and courses in many cities around Europe and the USA (e.g., I stayed three months in Dallas). Everywhere I made friends who helped me improve the thesis and exploiting the visits. All in all I would like to thank everyone I have met on this long journey. As I could not name each and every one and I do not want to choose a threshold (see Example 4.3), my greatest thanks go to Manuel for the opportunity to work and learn with him these years and for his steady guidance, patience and support.

[1] The first item `3-0.075` combines $\mathtt{A}_1$ with $\mathtt{B}_1$, `3=(1+2)` and `0.075=(0.3*0.7)`. The second item `4-0.295` combines $\mathtt{A}_1$ with $\mathtt{B}_2$ and $\mathtt{A}_2$ with $\mathtt{B}_1$, `4=(1+3)=(2+2)` and `0.295=(0.3*0.4 + 0.7*0.25)`. The third item combines $\mathtt{A}_1$ with $\mathtt{B}_3$ and $\mathtt{A}_2$ with $\mathtt{B}_2$, and the last one combines $\mathtt{A}_2$ with $\mathtt{B}_3$.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter gives a brief introduction to (Constraint) Logic Programming (CLP) and motivates why it is relevant to improve its expressiveness and efficiency. It describes the current state of the art with respect to the research topics I was working on: Tabled Constraint Logic Programming (to improve termination properties of CLP) and Constraint Answer Set Programming (to allow non-monotonic reasoning). We also list the contributions of my thesis, where they have been published, and applications of these contributions. Finally, the structure of the thesis is outlined.

1.1 Motivation and Overview

High-level programming languages, in particular declarative programming languages, make it easier for programmers to write and/or maintain software by providing a language closer to the natural specification language of the problem to be solved. Intuitively, declarative programming states *what* are the properties of the solution to a problem, instead of *how* the problem has to be solved. The separation of control issues and the logical specification is expressed by Kowalski's equation from (Kowalski, 1979), $algorithm = logic + control$, where the logic component specifies the knowledge to be used in solving problems and the control determines the problem-solving strategy. More formally, the key idea of declarative programming languages (Lloyd, 1994), is that a program is a theory (in some suitable logic), and computation is deduction from the theory. A suitable logic should have a model theory, a proof theory, a soundness theorem (i.e., computed answers should be correct) and a completeness theorem (i.e., all correct answers should be computed).

Logic programming languages (LP) are declarative languages based on first-order logic where the programs are sets of sentences in logical form (facts and rules) that characterize the properties of the solution to a problem. Major logic programming language families include Prolog, Datalog, and Answer Set Programming (ASP), which differ not only in the control strategy used but also in their expressiveness. Prolog (Sterling and Shapiro, 1994) is query-driven and uses a top-down strategy to deduce the answers for a query following specific resolution steps. SLD-resolution (Emden and Kowalski, 1976; Robinson, 1965) is the strategy used in Prolog. It is however incomplete and some logical consequences of a program may never be found (i.e., it may enter infinite loops).

This problem is partially solved using tabling (Tamaki and Sato, 1986; Warren, 1992) which is complete and ensure termination for definite programs (i.e., programs without negation) with bounded term depth (i.e., programs which can only generate terms with a finite depth). These programs include the class of Datalog programs. Datalog is syntactically a subset of Prolog (e.g., non-interpreted functions are not allowed), often used as a query language for deductive databases. Due to its finiteness properties, queries on Datalog programs can be resolved by bottom-up computation.

ASP (Lifschitz, 2008) allows the representation of non-monotonic reasoning using negation. ASP is based on the stable model semantics of logic programming (Gelfond and Lifschitz, 1988) and uses a bottom-up strategy, which in principle always terminates, to compute stable models of programs with negation. However, ASP programs with variables have to be grounded (replaced by an equivalent program without variables), with a concomitant combinatorial explosion, before they are solved. There are many techniques which try to mitigate the impact of the grounding phase, and considerable research has been conducted on deriving top-down execution models (Baselice and Bonatti, 2010; Baselice et al., 2009; Dal Palù et al., 2009; Marple et al., 2017a) to avoid the grounding phase.

Constraint logic programming (CLP) (Jaffar and Maher, 1994) represents relationships among different parts of the solution of a problem as equations and/or constraints and provides a common operational and declarative semantics where different equation domains/solvers can be combined to model problems using the more adequate representation. It extends LP with variables that can belong to arbitrary constraint domains and with constraint solvers that can incrementally simplify and solve equations set up during program evaluation. CLP brings additional expressive power to LP, since constraints can very concisely capture complex relationships. Also, the shift from "generate-and-test" to "constrain-and-generate" code patterns reduces the search tree and therefore brings additional performance, even if constraint solving is in general more expensive than unification. This expressiveness has been used to model satisfiability and optimization constraints (Marriott and Stuckey, 1993) for solving combinatorial search problems that draws on a wide range of techniques from artificial intelligence, computer science, databases, programming languages, and operations research. Currently, CLP is applied with success to other problems, such as scheduling, planning, vehicle routing, networks, and stream data analysis.

However, CLP inherits previous drawbacks that are not fully addressed in current attempts: the integration of constraint and tabling execution avoid loops, but they are inefficient and the integration of different constraint solvers is not easy; the extension of ASP systems with constraints allows non-monotonic reasoning, but their execution is not straightforward due to the grounding phase.

1.2 State of the Art

This section introduces the main concepts related to the techniques used to integrate different constraint solvers with tabling engines and/or ASP models. Section 1.2.1 introduces tabling and the limitations of the systems that have been proposed to evaluate Constraint Logic Programs under tabled execution. Section 1.2.2 introduces ASP, the proposals to reduce the impact of the grounding phase, and the limitations of systems proposed to evaluate (Constraint) Answer Set Programming under the stable model semantics. More details related to the contributions of this thesis appear at the beginning of each chapter.

1.2.1 Tabled Constraint Logic Programming

The initial ideas for the combination of tabling and constraints originate in (Kanellakis et al., 1995), where a variant of Datalog featuring constraints was proposed. Datalog (Maier and Warren, 1988) is a syntactical subset of logic programming and is used to reason about the databases. The restrictions in the expressiveness of the language ensure programs to have finite interpretations and also, queries on Datalog programs are executed bottom-up rather than the top-down as in Prolog. The former has the advantage that it always terminates for Datalog programs, whereas the latter may get stuck in infinite loops. However, a goal-directed approach usually obtains the desired result much faster and uses less space.

Tabling (Tamaki and Sato, 1986; Warren, 1992) is a top-down execution strategy for logic programs that always terminates for calls / programs with the bounded term depth property (e.g., Datalog programs) and can improve efficiency for terminating programs which repeat computations, as it automatically implements a variant of dynamic programming. The tabling execution suspends repeated calls that could cause infinite loops and answers from non-looping branches are used to resume suspended calls that can, in turn, generate more answers. Only new answers are saved, and evaluation finishes when no new answers can be generated. Tabling has been successfully applied in a variety of contexts, including deductive databases, program analysis, semantic Web reasoning, and model checking (Charatonik et al., 2002; Dawson et al., 1996; Ramakrishna et al., 1997; Warren et al., 1988; Zou et al., 2005).

Just as Datalog is a syntactic subset of logic programming, Datalog^{D} (Revesz, 1993; Toman, 1995) is a syntactic subset of constraint logic programming. Datalog^{D} has been applied in Constraint Databases where assignments to atomic values are generalized to constraints applied to variables, which provides more compact representations and increased expressiveness. The time and space problems associated with the bottom-up evaluation of Datalog were worked around in (Toman, 1997b) where a top-down evaluation strategy featuring tabling was proposed to take advantages of the top-down computation of CLP. Tabled CLP generalizes Datalog^{D}, in the same way as tabled LP generalizes Datalog, and has been applied in other areas, including verification of timed automata and infinite systems (Charatonik et al., 2002), and abstract interpretation (Toman, 1997a).

XSB (Swift and Warren, 2012) was the first logic programming system that provided tabled CLP as a generic feature, instead of resorting to ad-hoc adaptations. This was done by extending XSB with attributed variables (Cui and Warren, 2000), one of the most popular mechanism to implement constraint solvers in Prolog. However, one of its drawbacks is that it only uses variant checking (even for goals with constraints), instead of entailment and therefore it does not suspend more particular calls nor discards more particular answers. This makes programs terminate in fewer cases than with entailment and takes longer in other cases. This is similar to what happens in tabled logic programs with and without subsumption (Swift and Warren, 2010).

A general framework for CHR under tabled evaluation is described in (Schrijvers et al., 2008). It takes advantage of the flexibility that CHR provides for writing constraint solvers, but it also lacks call entailment checking and enforces *total call abstraction*: all constraints are removed from calls before executing them, which can result in non-termination w.r.t. systems that use entailment. Besides, the need to change the representation between CHR and Herbrand terms takes a toll in performance.

Failure Tabled CLP (Gange et al., 2013) is an execution technique which has several points in common with TCLP. It executes CLP programs following a top-down execution where for recursive clauses it uses iterative deepening search. The key idea is that it can learn from failed derivations and uses interpolants instead of constraint projection to generate conditions (i.e., the invariant) for reuse. It will however not terminate in some cases even with the addition of counters to implement a mechanism akin to iterative deepening. This technique has been applied in verification where you try to over-approximate failures loosing completeness but ensuring correctness (i.e., successful derivations are corrects).

Last, a previous Tabled CLP framework featuring a more complete treatment of constraint projection and entailment (Chico de Guzmán et al., 2012) focused on adapting the implementation of a tabling algorithm to be used with constraints. As a result, and although the ideas therein were generic, they are not easily extensible. Adding new constraint domains to them is a difficult task that requires deep knowledge about the particular tabling implementation and the constraint solver. The modifications done to the tabling implementation for one particular constraint solver may very well be not useful for another constraint solver; in turn, constraint solvers had to be modified in order to make then aware of internal characteristics and capabilities of the tabling algorithm. These adaptations generate a *technical debt* that made using the full potential of TCLP challenging.

1.2.2 Constraint Answer Set Programming

Answer Set Programming (ASP) has emerged as a successful paradigm for developing intelligent applications. It uses the stable model semantics (Gelfond and Lifschitz, 1988) for programs with negation. ASP has attracted much attention due to its expressiveness and ability to incorporate non-monotonic reasoning, represent knowledge, and model combinatorial problems. However, most of the ASP systems require a ground phase to remove the variables. To mitigate the impact of the exponential increase in the size of the grounded programs, several approaches have been proposed. For example, in the case of large data sets, *magic set* techniques have been used to improve grounding for specific queries (Alviano et al., 2012). For programs that use uninterpreted function symbols, techniques such as *external sources* (Calimeri et al., 2007) have been proposed.

As we mentioned before, constraints have been used both to enhance expressiveness and to increase performance in logic programming. It is therefore natural to incorporate constraints in ASP systems. For Constraint Answer Set Programming (CASP) systems based on a bottom-up execution, the grounding is still an issue. The integration of constraints with ASP is not as seamless as in standard constraint logic programming (CLP), because during the grounding phase the variables disappear and, therefore, the constraints linking them. The loss of communication due to elimination of variables makes the execution methods for CASP systems complex and explicit hooks are needed in the language. For example, EZCSP (Balduccini and Lierler, 2017) provides a builtin to place constraints in the head of the rules that is used, during the grounding phase, to convert these constraints into an intermediate language that communicates the ASP solver and the constraint solver.

Moreover, variable domains induced by constraints can be unbound and, therefore, infinite (e.g., `X #> 0` with $\mathtt{X} \in \mathbb{N}$ or $\mathtt{X} \in \mathbb{Q}$). Even if they are bound, they can contain an infinite number of elements (e.g. `X #> 0` $\land$ `X #< 1` in $\mathbb{Q}$ or $\mathbb{R}$). The proposals to work around this issue limit the range of admissible constraint domains (e.g., discrete instead of dense), the places where constraints can appear, and the type (or number) of models that can be returned.

These problems have been attacked using different techniques:

- Translation-based methods (Balduccini and Lierler, 2017), which convert both ASP and constraints into a theory that is executed by an SMT solver. Once the input program is translated, they benefit from the features and performance of the target ASP and SAP solvers. However, the translation may result in a large propositional representation or weak propagation strength, negatively impacting scalability and performance respectively.

- Extensions of ASP systems with constraint propagators (Banbara et al., 2017; Janhunen et al., 2017) that generate and propagate new constraints during the search and thus continuously check for consistency using external solvers featuring, e.g., conflict-driven clause learning. However, they are restricted to finite domain solvers (hence, dense domains cannot be appropriately captured) and incrementally generate ground models, lifting the upper bounds for some parameters. This, besides being a performance bottleneck, falls short of capturing the true nature of variables in constraint programming.

On the other hand, to avoid the grounding phase, research has been conducted on devising top-down execution models for ASP (Baselice and Bonatti, 2010; Baselice et al., 2009; Dal Palù et al., 2009) that could be extended with constraints. One on them is s(ASP) (Marple et al., 2017a), a goal-directed, top-down, SLD resolution-like procedure that evaluates programs under the ASP semantics without a grounding phase either before or during execution. s(ASP) supports predicate logic programs (also with unsafe clauses and/or uninterpreted functions) and retains logical variables both during execution and in the answer sets.

1.3 Thesis Contributions and Impact

This section discusses the theoretical and technical contributions of this thesis, and it details where they have been published or presented. The main goal of this thesis is the improvement of a high-level language rooted in logic and constraints to provide customized solutions (using the more appropriated constraint domain) to different problems and to make it easier to write and maintain the programs. From the analysis of the State of the Art we have identified three main research challenges:

- The current results on correctness, completeness, and termination of Tabled Constraint Logic Programming are based on a bottom-up execution of Datalog and they only consider a constraint-compact constraint domain.

- Implementations of Tabled Constraint Logic Programming did not fully use entailment to determine call/answer subsumption and/or did not provide a simple and well-documented interface to facilitate the integration of constraint solvers in existing tabling systems.

- For non-monotonic reasoning, current bottom-up implementations of Constraint Answer Set Programming restrict the range of admissible constrained variable domains (e.g., discrete instead of dense), or the type (or number) of models that can be returned. And none of the current top-down proposals integrate constraint solvers.

To address the aforementioned problems, we propose two frameworks: Mod TCLP, a modular framework that facilitates the integration of constraint solvers with tabling engines and s(CASP), a top-down non-monotonic reasoner that evaluates Constraint ASP programs.

1.3.1 Mod TCLP

The first main contribution is a modular framework of Tabled CLP, that implements extended theoretical foundations for generic constraint domains (including the Herbrand domain), and facilitates the integration of different constraint solvers in a tabling engine.

We have studied the role of projection and entailment in the termination, soundness, and completeness of TCLP systems to characterize the properties required by the constraint domain in order to, e.g., ensure termination. Then, we have generalized the theoretical foundations of Tabled CLP for arbitrary constraint solvers in the top-down operational semantics. Based on this theoretical foundations we designed, implemented and evaluated Mod TCLP, a framework that eases the integration of additional constraint solvers. Mod TCLP views constraint solvers as clients of the tabling system, which is generic w.r.t. the solver and only requires a clear interface from the latter. This work has been presented in the 18th International ACM Symposium on Principles and Practice of Declarative Programming (Arias and Carro, 2016) and published in the journal Theory and Practice of Logic Programming[1] (Arias and Carro, 2019a).

To demonstrate the expressiveness and flexibility provided by Mod TCLP, and the improvement in terms of performance (e.g., due to a more comprehensive answer management strategy design) w.r.t. Prolog, tabling and CLP:

- We integrated four constraint solvers: a previously existing constraint solver for difference constraints, written in C; the standard versions of Holzbaur's CLP(Q) and CLP(R), written in Prolog; and a new constraint solver for equations over finite lattices. And we evaluated the performance of our framework in several benchmarks using the aforementioned solvers.

- We used Mod TCLP to define a framework to incrementally compute aggregates for elements in a lattice. We use the entailment and join relations of the lattice to

[1]Theory and Practice of Logic Programming (TPLP) is ranked in Q1 in JCR.

define (and compute) aggregates and decide whether some atom is compatible with (entails) the aggregate. The semantics of the aggregates defined in this way is consistent with the LFP semantics of tabling with constraints. This work has been presented in the 21*st* International Symposium on Practical Aspects of Declarative Languages (Arias and Carro, 2019c) and is under submission for a journal version.

- We adapted the existing PLAI implementation in CiaoPP using Mod TCLP. The tabling engine is used to compute the fixpoint and the constraint interface computes the LUB of the abstract substitutions of different clauses. That provides, on one hand, a much simpler code as the fixpoint computation is taken care of by the underlying tabling machinery, and, in most cases, it also brings performance gains, since some crucial operations (such as branch switching and resumption) are executed by the tabling engine. This work has been presented at the 35*th* International Conference on Logic Programming and published in the journal Theory and Practice of Logic Programming (Arias and Carro, 2019b).

1.3.2 s(CASP)

The second main contribution is a novel top-down system to evaluate constraint answer set programs with the ability to express non-monotonic programs *à la* ASP and the possibility of expressing control in a way similar to traditional logic programming. This work has been (partially) done during my stay at the University of Texas at Dallas, under the supervision of Dr. Gopal Gupta and in collaboration with Kyle, Elmer, and Zhuo, and has been presented at the 34th International Conference on Logic Programming and published in the journal Theory and Practice of Logic Programming (Arias et al., 2018). I am the main contributor to this work. We have also reported a very substantial performance increase w.r.t. the original s(ASP) implementation and thanks to the possibility of writing pieces of code with control in mind, it can also beat state-of-the-art ASP systems in certain programs.

To demonstrate the expressiveness and flexibility of s(CASP) and the improvement in terms of performance (e.g., ASP programs can be written with control in mind), w.r.t. s(ASP), CLP, and mature ASP systems that it provides:

- We used s(CASP) to write programs/queries that cannot be written in [C]ASP without resorting to a complex, unnatural encoding. The resulting programs can use structures/functors directly and their answers are more expressive than those given by ASP. Additionally, the constraints and the goal-directed evaluation strategy of s(CASP) makes it possible to use directly algorithms that can not be immediately coded in CASP and to reduce the search space.

- We used s(CASP) as the underlying reasoning infrastructure to model and reason in Event Calculus. This reasoner shows how Event Calculus scenarios can be elegantly modeled in s(CASP) and how its expressiveness makes it possible to perform deductive and abductive reasoning tasks in domains featuring, for example, constraints involving dense time and fluents. This work has been presented at the 3rd International Workshop Datalog 2.0 (Arias et al., 2019b) and at the 29th International Symposium on Logic-based Program Synthesis and Transformation (Arias et al., 2019a).

1.4 Thesis Organization

This section summarizes the contents of the different chapters:

- Chapter 1 motivates the work presented in this thesis, describes the state of the art, and outlines the contributions of the thesis and its organization.
- Chapter 2 extends the theoretical foundations of Tabled TCLP with a top-down execution parametric w.r.t. constraint domains by proving soundness, completeness, and termination for generic constraint domains.
- Chapter 3 describes the design and implementation of Mod TCLP, gives examples integrating different constraint solvers, and evaluates its performance w.r.t. Prolog, tabling, and CLP.
- Chapter 4 presents the intended semantics and the implementation of a framework to incrementally compute aggregates based on Mod TCLP (extended to allow the combination of answers).
- Chapter 5 presents the re-implementation of the existing PLAI algorithm used by CiaoPP under Mod TCLP, reducing the complexity and the size of the code.
- Chapter 6 describes the design and the implementation of s(CASP), which improves and extends s(ASP) with constraints.
- Chapter 7 presents and evaluates a reasoner of Event Calculus written using s(CASP).
- Chapter 8 draws some conclusions and outlines some of the new research directions that can extend the research presented in this thesis.

Part I

Tabled Constraint Logic Programming

Chapter 2

Top-Down TCLP: Semantics, Correctness, Completeness, and Termination

Tabled Constraint Logic Programming (TCLP) basis was defined for a bottom-up evaluation of Datalog (a syntactic subset of Prolog), and constraint-compact constraint domains (e.g., the gap-order constraints). This chapter extends the theoretical foundations of TCLP supporting generic constraint domains (including the Herbrand domain used by Prolog), and giving a top-down operational semantics based on CLP. It also studies the role of entailment check and projection of the constraint stores in the design of TCLP systems (i.e., relaxing their precision impacts efficiency and correctness).

The theoretical basis of TCLP (Toman, 1997b) were established in the framework of bottom-up evaluation of Datalog systems and presents the basic operations (projection and entailment checking) that are necessary to ensure completeness w.r.t. the declarative semantics. Database evaluation in principle proceeds bottom-up, which ensures termination in this context. However, in order to speed up query processing and spend fewer resources, top-down evaluation is also applied, where tabling can be used to avoid loops. In this setting, TCLP is necessary to capture the semantics of the constraint database.

In this chapter, we complete previous work on conditions for termination of TCLP, we provide a richer, more flexible answer management mechanism, and with a study on how the implementation of a relaxed projection impacts soundness, completeness and termination. On the other hand, we generalize the design of a tabling implementation so that it can use the projection and entailment operations provided by a constraint solver presented to the tabling engine as a *server*, and we define a set of operations that the

```
dist(X, Y, D) :-
    dist(X, Z, D1),
    edge(Z, Y, D2),
    D is D1 + D2.
dist(X, Y, D) :-
    edge(X, Y, D).

?- dist(a,Y,D), D < K.
```

(a) LP version.

```
:- use_package(clpq).

dist(X, Y, D) :-
    D1 #> 0, D2 #> 0,
    D  #= D1 + D2,
    dist(X, Z, D1),
    edge(Z, Y, D2).
dist(X, Y, D) :-
    edge(X, Y, D).

?- D #< K, dist(a,Y,D).
```

(b) CLP version.

Figure 2.1: Left-recursive distance traversal in a graph.
Note: The symbols `#>` and `#=` are (in)equalities in CLP.

constraint solver has to provide to the tabling engine. These operations are natural to the constraint solver, and when they are not already present, they should be easy to implement by extending the solver.

In Section 2.1 we highlight some of the advantages of TCLP versus LP, tabling, and CLP. In Section 2.2 we present the syntax and some properties of CLP. In Section 2.3 we define a more general TCLP semantics based on the operational semantics of CLP under a top-down execution and in Section 2.4 we compare their behaviour using CLP trees and TCLP forests. In Section 2.5 we generalize previous soundness, completeness and termination proofs for this extended TCLP semantics. In Section 2.6 we explain the benefits of using entailment checking with a precise implementation of the projection.

2.1 Motivation

In order to highlight some of the advantages of TCLP versus LP, tabling, and CLP with respect to declarativeness and logical reading, we will compare how different versions of a program to compute distances between nodes in a graph behave under these three approaches. Each version will be adapted to a different paradigm, but trying to stay as close as possible to the original code, so that the additional expressiveness can be attributed to the semantics of the programming language rather than to differences in the code itself.

```
dist(X, Y, D) :-
    edge(X, Z, D1),
    dist(Z, Y, D2),
    D is D1 + D2.
dist(X, Y, D) :-
    edge(X, Y, D).
```

(a) LP version.

```
:- use_package(clpq).

dist(X, Y, D) :-
    D1 #> 0, D2 #> 0,
    D  #= D1 + D2,
    edge(X, Z, D1),
    dist(Z, Y, D2).
dist(X, Y, D) :-
    edge(X, Y, D).
```

(b) CLP version.

Figure 2.2: Right-recursive distance traversal in a graph.
Note: The symbols `#>` and `#=` are (in)equalities in CLP.

2.1.1 LP vs. CLP

The code in Fig. 2.1, left, is the Prolog version of a program used to find nodes in a graph within a distance `K` from each other.[1] Fig. 2.1, right, is the CLP version of the same code. The queries used to find the nodes `Y` from the node `a` within a maximum distance `K` appear in the figures themselves.

In the Prolog version, the distance between two nodes is calculated by adding variables `D1` and `D2`, corresponding to distances to and from an intermediate node, once they are instantiated. In the CLP version, addition is modeled as a constraint and placed at the beginning of the clause. Since the total distance is bound, this constraint is expected to prune the search in case it tries to go beyond the maximum distance `K`. These checks are not added to the Prolog version, since they would not be useful for termination: they would have to be placed after the calls to `edge/3` and `dist/3`, when it is too late to avoid infinite loops. In fact, none of the queries shown before terminates as left recursion makes the recursive clause enter an infinite loop even for acyclic graphs.

If we convert the program to a right-recursive version by swapping the calls to `edge/3` and `dist/3` (Fig. 2.2), the LP execution will still not terminate in a cyclic graph. The right-recursive version of the CLP program will however finish because the initial bound to the distance eventually causes the constraint store to become inconsistent, which provokes a failure in the search. This behavior is summarized in columns “LP” and “CLP” of Table 2.1.

Note that this transformation is easy in this case, but in other cases, such as language interpreters or tree / graph traversal algorithms, left (or double) recursion is much more natural. While there are techniques to remove left / double recursion, most Prolog compilers do not feature them. Therefore, we assume that the original source code is straightforwardly mapped to the low-level runtime system, and, if necessary, left /

[1]This is a typical query for the analysis of social networks (Swift and Warren, 2010).

Table 2.1: Termination properties comparison of LP, CLP, tabling and TCLP.

Graph		LP	CLP	TAB	TCLP
Without cycles	Left recursion	×	×	✓	✓
	Right recursion	✓	✓	✓	✓
With cycles	Left recursion	×	×	×	✓
	Right recursion	×	✓	×	✓

double recursion has to be manually removed by adding extra arguments implementing, for example, explicit stacks — precisely the kind of manual program transformation that we would like to avoid due to the difficulties that it brings with respect to maintenance and clarity.

2.1.2 LP vs. Tabling

Tabling records the first occurrence of each call to a tabled predicate (the *generator*) and its answers. In variant tabling, the most usual form of tabling, when a call equal up to variable renaming to a previous generator is found (a variant), its execution is suspended, and it is marked as a *consumer* of the generator. For example, `dist(a,Y, D)` is a variant of `dist(a,Z, D)` if `Y` and `Z` are free variables. When a generator finitely finishes exploring all of its clauses and its answers are collected, its consumers are resumed and are fed the answers of the generator. This may make consumers produce new answers that will in turn cause more resumptions.

Tabling is a complete strategy for all programs with the bounded term-depth property, which in turn implies that the Herbrand model is finite. Therefore, left- or right-recursive *reachability* terminates in finite graphs with or without cycles. However, the program in Fig. 2.1, left, has an infinite minimum Herbrand model for cyclic graphs: every cycle can be traversed an unbounded number of times, giving rise to an unlimited number of answers with a different distance each. The query `?- dist(a,Y, D), D<K` will therefore not terminate under variant tabling.

2.1.3 TCLP vs. Tabling and vs. CLP

The program in Fig. 2.1, right, can be executed with tabling and using constraint entailment to suspend calls which are more particular than previous calls and, symmetrically, to keep only the most general answers returned.

Entailment can be seen as a generalization of subsumption for the case of general constraints; in turn, subsumption was shown to enhance termination and performance in tabling (Swift and Warren, 2010). For example, the goal $G_0 \equiv$ `dist(a, Y, D)` is subsumed by $G_1 \equiv$ `dist(X, Y, D)` because the former is an instance of the latter ($G_0 \sqsubseteq G_1$). All the answers for G_1 where `X=a` are valid answers for G_0; on the other hand, all the answers for G_0 are also answers for G_1.

The main idea behind the use of entailment in TCLP is that more particular calls (consumers) can suspend and later reuse the answers collected by more general calls (generators). In order to make this entailment relationship explicit, we define a TCLP goal as $\langle g, c_g \rangle$ where g is the call (a literal) and c_g is the projection of the current constraint store onto the variables of the call. Then, $\langle$`dist(a, Y, D)`, `D` $<$ `150`$\rangle$ is entailed by the goal $\langle$`dist(a, Y, D)`, `D` $>$ `0` $\wedge$ `D` $<$ `75`$\rangle$ because `D` $>$ `0` $\wedge$ `D` $<$ `75`$\sqsubseteq$`D` $<$ `150`. We also say that the former (the generator) is more general than the latter (the consumer). All the solutions of the consumer are solutions of the generator or, in other words, the space of solutions of the consumer is a subset of that of the generator. However, not all the answers from a generator are valid for its consumers. For example `Y=b` $\wedge$ `D` $>$ `125` $\wedge$ `D` $<$ `135` is a solution for our generator, but not for our consumer, since the consumer call was made under a constraint store more restrictive than the generator. Therefore, the tabling engine should check and filter, via the constraint solver, that the answer from the generator is consistent w.r.t. the constraint store of the consumer.

The use of entailment in calls and answers enhances termination properties and can also increase speed (Section 3.3.1). The column "TCLP" in Table 2.1 summarizes the termination properties of `dist/3` under TCLP, and shows that a full integration of tabling and CLP makes it possible to find all the solutions and finitely terminate in all the cases. Our TCLP framework not only facilitates the integration of constraint solvers with the tabling engine thanks to its simple interface (Section 3.1.1), but also minimizes the effort required to execute existing CLP programs under tabling (Fig. 3.2), since the changes required to the source code are minimal.

2.2 Constraint Logic Programming

In Section 2.2.1 we present the syntax of constraint logic programs, and in Section 2.2.2 we define the properties of a constraint solver required to be used under a TCLP top-down execution (see Chapter 3).

2.2.1 Syntax of Constraint Logic Programs

A constraint logic program consists of clauses of the form:

$$h \text{ :- } c_h, l_1, \ \ldots, \ l_k.$$

where h is an atom, c_h is a constraint or conjunction of constraints, and l_i are literals. The head of the clause is h and the rest is called the body. The clauses where the body is always true, h :- *true*, are called facts and usually written omitting the body (h.). We will use L to denote the set of l_i in a clause. We will assume throughout this chapter that the program has been rewritten so that clause heads are linearized (all the variables are different) and all head unifications take place in c_h. We will assume that we are dealing with *definite programs*, i.e., programs where the literals in the body are always positive (non-negated) atoms. *Normal programs* require a different treatment.

A query to a CLP program is a clause without head ?- c_q, q_1, …, q_k, where c_q is a conjunction of constraints, q_i are the literals of the query. We denote the set of q_i as Q.

2.2.2 Constraint Solvers

We follow (Jaffar and Maher, 1994) in this section. Constraint logic programming introduces constraint solving methods in logic-based systems. During the evaluation of a CLP program, the *inference engine* generates constraints whose consistency with respect to the current constraint store is checked by the *constraint solver*. If the check fails, the engine backtracks to a previous state and takes a pending branch of the search tree. In the next sections we will review the fixpoint and operational semantics of CLP and will extend them to TCLP.

Definition 2.1. A *constraint solver*, CLP($\mathcal{X}$), is a (partial) executable implementation of a *constraint domain* $(\mathcal{D}, \mathcal{L})$. The parameter $\mathcal{X}$ stands for the 4-tuple (Σ, $\mathcal{D}$, $\mathcal{L}$, $\mathcal{T}$) where:

- Σ is a signature which determines the predefined predicates and function symbols and their arities.
- $\mathcal{D}$ is a Σ-structure: the constraint domain over which the computation is performed.
- $\mathcal{L}$ is the class of Σ-formulas: all the constraints that can be expressed with Σ.
- $\mathcal{T}$ is a first-order Σ-theory: an axiomatization of the properties of $\mathcal{D}$, which determines what constraints hold and what constraints do not hold. $\mathcal{D}$ and $\mathcal{T}$ should agree on satisfiability of constraints, and every unsatisfiability in $\mathcal{D}$ has to be detected by $\mathcal{T}$, i.e., for every constraint $c \in \mathcal{L}$, $\mathcal{D} \models c$ iff $\mathcal{T} \models c$.

A constraint can be a singleton constraint or a conjunction of simpler constraints. We denote constraints with lower case letters, e.g. c, and sets of constraints with uppercase letters, e.g. S.

Example 2.1.
The Herbrand domain CLP($\mathbb{H}$) used in logic programming is the constraint domain over finite trees, where Σ contains constants, function symbols, and the predicate `=/2`; $\mathcal{D}$ is the set of finite trees, where each node is labeled by a constant (if it does

no have children) or function symbol of arity n (if it has n children). $\mathcal{L}$ is the set of constraints generated by the primitive constraints (i.e., equality) between trees (terms). Typical constraints are `X=g(a)` and `X=f(Z, Y)` $\wedge$ `Z=a`.

Definition 2.2 (Valuation)**.** Let $S = \{X_1, \ldots, X_n\}$ be a set of variables. A *valuation* v is a mapping from variables in S to values in $\mathcal{D}$. We write $v = \{X_1 \mapsto d_1, \ldots, X_n \mapsto d_n\}$ to indicate that the value d_i is assigned to variable X_i. We determine the value assigned to a variable as $v(X_i) = d_i$.

Definition 2.3 (Solution of a constraint)**.** Let c be a constraint, $vars(c)$ the set of variables occurring in c, and v a valuation over $vars(c)$ on the constraint domain $\mathcal{D}$. Then v is a *solution* of c if $v(c)$, obtained by replacing each variable in c by the value assigned to it by v, holds in the constraint domain.

The minimal set of operations that we expect a constraint solver to support, in order to interface it successfully with our tabling system, are:

- Test for consistence or satisfiability. A constraint c is consistent on the constraint domain $\mathcal{D}$, denoted $\mathcal{D} \models c$, if it has a solution in $\mathcal{D}$.

- Test for entailment ($\sqsubseteq_{\mathcal{D}}$).[2] We say that a constraint c_0 is entailed by another constraint c_1 ($c_0 \sqsubseteq_{\mathcal{D}} c_1$) if any solution of c_0 is also a solution of c_1. We extend the notion of constraint entailment to a set of constraints: a set of constraints C_0 is entailed (or covered) by another set of constraints C_1 (and we write it as $C_0 \sqsubseteq_{\mathcal{D}} C_1$) if $\forall c_i \in C_0 \, \exists c_j \in C_1 . c_i \sqsubseteq_{\mathcal{D}} c_j$.

- An operation to compute the projection of a constraint c onto a set of variables S to obtain a constraint c_S involving only variables in S such that: any solution of c is also a solution of c_S, and a valuation v over S which is a solution of c_S is a partial solution of c (i.e., there exists an extension of v which is a solution of c). We denote the projection as $Proj(S, c)$.

2.3 Top-down Semantics

In this section we extend the theoretical basis of Tabled Constraint Logic Programming for a top-down evaluation. In Section 2.3.1 we generalize the fixpoint semantics of (Schrijvers et al., 2008; Toman, 1997b) with the S-semantics (Falaschi et al., 1989; Jaffar and Maher, 1994). In Section 2.3.2 we define the operational semantics of TCLP, based on the operational semantics of CLP.

[2] We may omit the subscript $\mathcal{D}$ if there is no ambiguity.

2.3.1 Fixpoint Semantics

The canonical model of a Prolog program is the minimal Herbrand model. Similarly, the logical semantics of a CLP program P over a constraint domain $\mathcal{D}$ is the least $\mathcal{D}$-model / $\mathcal{D}$-S-model, which we define next.

Definition 2.4 ($\mathcal{D}$-interpretation)**.** Let L be a set of literals. Then a $\mathcal{D}$-interpretation is the set of the valuations for the literals in L on the constraint domain $\mathcal{D}$. It is a subset of the Herbrand base $B = \{v(l) \mid l \in L$ is a literal, v is a valuation on $\mathcal{D}\}$.

Definition 2.5 ($\mathcal{D}$-model)**.** Let P be a program. A $\mathcal{D}$-model of P is a $\mathcal{D}$-interpretation such that all the clauses of P are consistent on $\mathcal{D}$ under this interpretation.

We can define the least $\mathcal{D}$-S-model of a program using the S-semantics (Falaschi et al., 1989; Jaffar and Maher, 1994) for languages with constraints (Gabbrielli and Levi, 1991). It differs from the standard model (Emden and Kowalski, 1976) essentially due to the presence of variables in interpretations and models.

Definition 2.6 ($\mathcal{D}$-S-interpretation)**.** Let the pair (l, c) be a constraint literal where l is a literal and $c \in \mathcal{D}$ a constraint such that $vars(c) \subseteq vars(l)$. A $\mathcal{D}$-S-interpretation is a set of constraint literals.

Definition 2.7 ($\mathcal{D}$-S-model)**.** Let P be a program. A $\mathcal{D}$-S-model of P is a $\mathcal{D}$-S-interpretation such that all the clauses of P are consistent on $\mathcal{D}$ under this interpretation.

The CLP fixpoint semantics (resp., S-semantics) is defined as usual as the smallest fixpoint of the immediate consequence operators $T_P^{\mathcal{D}}$ (resp., $S_P^{\mathcal{D}}$) where all the operations behave as defined in the constraint domain $\mathcal{D}$. Function $T_P^{\mathcal{D}}$ maps $\mathcal{D}$-interpretations onto $\mathcal{D}$-interpretations:

Definition 2.8 (Operator $T_P^{\mathcal{D}}$ (Jaffar and Maher, 1994))**.** Let P be a CLP program and I a $\mathcal{D}$-interpretation. The immediate consequence operator $T_P^{\mathcal{D}}$ is defined as:

$$T_P^{\mathcal{D}}(I) = I \cup \{\, v(h) \mid \begin{array}{l} h \text{ :- } c_h, l_1, \ldots, l_k \text{ is a clause of } P, \\ v \text{ is a valuation on } \mathcal{D} \text{ s.t. } \mathcal{D} \models v(c_h), \\ v(l_i) \in I,\ 0 < i \leq k \,\} \end{array}$$

The function $S_P^{\mathcal{D}}$ does the same over $\mathcal{D}$-S-interpretations:

Definition 2.9 (Operator $S_P^{\mathcal{D}}$ (Falaschi et al., 1989; Toman, 1997b))**.** Let P be a CLP program and I a $\mathcal{D}$-S-interpretation. The immediate consequence operator $S_P^{\mathcal{D}}$ is defined as:

$$
\begin{aligned}
S_P^{\mathcal{D}}(I) = I \cup \{\ (h,c)\ \mid\quad & h \text{ :- } c_h, l_1, \ldots, l_k \text{ is a clause of } P,\\
& (a_i,c_i) \in I,\ 0 < i \leq k,\\
& c' = Proj(vars(h),\ c_h \wedge \textstyle\bigwedge_{i=1}^{k}(a_i = l_i \wedge c_i)),\\
& \mathcal{D} \vDash c',\\
& \text{if } c' \sqsubseteq c'' \text{ for some } (h,c'') \in I \text{ then } c = c'' \text{ else } c = c'\ \}
\end{aligned}
$$

Note that, unlike the $T_P^{\mathcal{D}}$ operator, $S_P^{\mathcal{D}}$ may not add a pair *(literal, constraint)* when a more general constraint is already present in the interpretation being enlarged. However, to guarantee monotonicity, it does not remove existing more particular constraints. The operational semantics (Section 2.3.2) will however be able to do that.

We denote the least fixpoint of a function f by *lfp*(f). This fixpoint exists for $T_P^{\mathcal{D}}$ and $S_P^{\mathcal{D}}$ because both are monotonic functions on complete lattices: the set of $\mathcal{D}$-models of a program P forms a complete lattice under the subset ordering, with a unique least $\mathcal{D}$-model, and the set of $\mathcal{D}$-S-models of a program P forms a complete lattice under $\leq_S$, the ordering on S-interpretations defined in (Falaschi et al., 1989), with a unique least $\mathcal{D}$-S-model. We can take as semantics $lfp(T_P^{\mathcal{D}})$ or $lfp(S_P^{\mathcal{D}})$, because $[S_P^{\mathcal{D}}(I)]_{\mathcal{D}} = T_P^{\mathcal{D}}([I]_{\mathcal{D}})$ and consequently $[lfp(S_P^{\mathcal{D}})]_{\mathcal{D}} = lfp(T_P^{\mathcal{D}})$. The S-semantics, returned by $lfp(S_P^{\mathcal{D}})$, allows the intensional definition of models, and the standard semantics can be easily obtained from it: if X is a $\mathcal{D}$-S-interpretation, a $\mathcal{D}$-interpretation is the set of valuations such that $[X]_{\mathcal{D}} = \{v(l) \mid (l,c) \in X, \mathcal{D} \vDash v(c)\}$.

2.3.2 Operational Semantics

In this section we extend the operational semantics for CLP programs under a top-down execution scheme (Jaffar and Maher, 1994) to TCLP programs. The operational semantics is given in terms of a transition system which computes the least model defined by the CLP fixpoint semantics (Section 2.3.1). The evaluation of a query is a sequence of steps from one state to the next.

Definition 2.10. A *state* is a tuple $\langle R,\ c\rangle$ where:

- R, the resolvent, is a multiset of literals and constraints that contains the collection of as-yet-unseen literals and constraints of the program. For brevity, when the set is a singleton we will write its only element, e.g., t instead of $[t]$.
- c, the constraint store, is a constraint or conjunction of constraints. It is acted upon by the constraint solver.

In (Jaffar and Maher, 1994) the constraint store c is divided in a collection of *awake* constraints and a collection of *asleep* constraints. This separation is ultimately motivated by implementation issues.

Given a query (Q, c_q), the initial state of the evaluation is $\langle Q,\, c_q\rangle$. Every transition step between states resolves literals of the resolvent against the clauses of the program and adds constraints to the constraint store. A derivation is *successful* if it is finite and the final state has the form $\langle \emptyset,\, c\rangle$ (i.e., the resolvent becomes empty). The answer for the query is $Proj(vars(Q), c)$.

The transitions due to constraint handling are deterministic (there is only one possible offspring per node), while the transitions due to literal matching are non-deterministic (there are as many offsprings as clauses match with the node literal). This is usually represented as a search tree, constructed following the definition below. The order in which the literals/constraints are selected is decided by the computation rule.

Definition 2.11 (CLP tree)**.** Let P be a CLP definite program and (Q, c_q) a query. A CLP tree of (Q, c_q) for P, denoted by $\mathcal{T}_P(Q, c_q)$, is a tree such that:

1. the root of $\mathcal{T}_P(Q, c_q)$ is $\langle Q,\, c_q\rangle$, the initial state.
2. the nodes of $\mathcal{T}_P(Q, c_q)$ are labeled with its corresponding state $\langle L,\, c\rangle$.
3. the child/children of a node $\langle l \cup L,\, c\rangle$, where l is a literal selected by the computation rule, is/are:
    - a node/nodes $\langle body(h_i) \cup L,\, c \wedge (l = h_i)\rangle$ obtained by resolution of l against the matching clause(s) h_i :- $body(h_i)$ in P where $l = h_i$ is an abbreviation for the conjunction of equations between the arguments of l and h_i. There is one node for each matching clause. Matching clauses are assumed to be renamed apart.
    - or a leaf node *fail* if there are no clauses in P which matching heads for the literal l.
4. the child of a node $\langle c' \cup L,\, c\rangle$, where c' is a constraint selected by the computation rule, is:
    - the node $\langle L,\, c \wedge c'\rangle$, such that $\mathcal{D} \models c \wedge c'$.
    - or a leaf node *fail* if $\mathcal{D} \not\models c \wedge c'$.
5. a leaf node $\langle \emptyset,\, c\rangle$ is the final state of a successful derivation, where c is the final constraint store.
6. the set of answers of $\mathcal{T}_P(Q, c_q)$ (i.e., the answers to the query (Q, c_q)), denoted by $Ans(Q, c_q)$, is the set of constraints c'_i obtained as the projection of the final constraint stores c_i onto $vars(Q)$:

$$Ans(Q, c_q) = \{c'_i \;\mid\; c'_i = Proj(vars(Q), c_i).\langle \emptyset,\, c_i\rangle \in \mathcal{T}_P(Q, c_q)\}$$

In a TCLP program (a CLP program executed under tabling), the set of tabled predicates is denoted by Tab_P. In Def. 2.12 we define the forest (a set of trees) generated during the computation of a TCLP program where there are two main phases: the call entailment phase (Def. 2.12.3c) and the answer entailment phase (Def. 2.12.3g). The call entailment phase checks if a new goal is entailed/subsumed by a previous goal (called its generator).[3] The set of generators is denoted by Gen_P.

Tabled literals (literals which match heads of tabled predicates) are not resolved against program clauses. Instead, they are resolved consuming the answer constraints from a generator; this is termed *answer resolution*. The answer constraints of a generator are collected in the answer entailment phase in such a way that an answer which is entailed by another more general answer is discarded/removed. As a result, the forest obtained as the derivation of a query w.r.t. a TCLP program is the set containing the tree corresponding to the initial query and the trees corresponding to the generators (see some example in Section 2.4).

Definition 2.12 (TCLP forest)**.** Let P be a TCLP definite program, Tab_P the set of tabled predicates, and (Q, c_q) a query. A TCLP forest of (Q, c_q) for P, denoted as $\mathcal{F}_P(Q, c_q)$, the set of TCLP trees such that:

1. the initial tree, $\tau_P(Q, c_q)$, is the TCLP tree of the query, and the rest of trees, $\tau_P(g_i, c_{g_i})$ are the TCLP trees of the generators $(g_i, c_{g_i}) \in Gen_{(Q,c_q)}$:

$$\mathcal{F}_P(Q, c_q) = \{\tau_P(Q, c_q),\ \tau_P(g_i, c_{g_i}), \dots\} \text{ with } i \geq 0$$

2. the set of generators of $\mathcal{F}_P(Q, c_q)$, denoted as $Gen_{(Q,c_q)}$, is ordered with respect to the “age” of the generator such that for a given generator (g_i, c_{g_i}) with $g_i \in Tab_P$, there are no younger generators, denoted as (g_j, c_{g_j}) with $j < i$, which are more general:

$$Gen_{(Q,c_q)} = \{(g_i, c_{g_i}) \mid \forall j < i, \nexists (g_j, c_{g_j}) \in Gen_{(Q,c_q)}.(g_i, c_{g_i}) \sqsubseteq_{\mathcal{D}} (g_j, c_{g_j})\ \}$$

3. A TCLP tree, denoted by $\tau_P(Q, c_q)$ or $\tau_P(g_i, c_{g_i})$, is similar to a CLP tree where:
   (a) the root of the TCLP tree $\tau_P(Q, c_q)$ is $\langle Q,\ c_q\rangle$, the initial state.
   (b) the root of a TCLP tree $\tau_P(g, c_g)$ is the state $\langle g,\ c_g\rangle$ corresponding to a generator $(g, c_g) \in Gen_{(Q,c_q)}$. The child/children of this node is/are:
      - a node/nodes $\langle body(h_i),\ c \wedge (g = h_i)\rangle$ obtained by resolution of g against matching clauses h_i :- $body(h_i)$ in P, where $g = h_i$ is an abbreviation for the conjunction of equations between the corresponding arguments of g and h_i. There is one child node for each matching clause.
      - or a leaf node *fail* if there are no clauses in P with matching heads for the literal g.

   (c) the child/children of a node $\langle t \cup L,\ c\rangle$ which is not the root of a generator and where t is a tabled literal ($t \in Tab_P$) selected by the computation rule, is/are obtained by answer resolution consuming the answers c_i such that:

[3]Note that this entailment check includes subsumption in the Herbrand domain.

  - $c_i \in Ans(g, c_g)$, where $Ans(g, c_g)$ is the set of answers of the oldest generator $(g, t) \in Gen_{(Q, c_q)}$, such that g and t are equal upon variable renaming, and $c \wedge (t = g) \sqsubseteq_{\mathcal{D}} c_g$, where $t = g$ is an abbreviation for the conjunction of equations between corresponding arguments of t and g, i.e., (g, c_g) is more general than (t, c). In this case the goal (t, c) is marked as a consumer of (g, c_g).
  - or $c_i \in Ans(t, c')$, where $c' = Proj(vars(t), c)$ and $Ans(t, c')$ is the set of answers of a new TCLP $\mathcal{F}_P(t, c')$ which is created and added to the current forest. In this case the goal (t, c') is marked as a generator and is added to $Gen_{(Q, c_q)}$, the set of generators of $\mathcal{F}_P(Q, c_q)$.

  resulting:
  - a node/nodes $\langle c_i \cup L, c\rangle$, one for each answer c_i.
  - or a leaf *fail* if there are no answer c_i.

(d) the transitions for non tabled literal and for constraints are as in the CLP tree (Def. 2.11.3 and Def. 2.11.4).

(e) a leaf node $\langle \emptyset, c\rangle$ is the final state of a successful derivation, where c is the final constraint store.

(f) the set of answers of $\tau_P(Q, c_q)$, the TCLP tree of the query, are obtained as in the CLP tree (Def. 2.11.6).

(g) the set of answers of $\tau_P(g, c_g)$, the TCLP tree of the generator (g, c_g), denoted by $Ans(g, c_g)$, is the set of more general (w.r.t. $\sqsubseteq_{\mathcal{D}}$) constraints c'_i obtained as the projection of the final constraint stores c_i onto $vars(L)$:

$$Ans(g, c_g) = \{c'_i \mid \quad c'_i = Proj(vars(g), c_i).\langle \emptyset, c_i\rangle \in \tau_P(g, c_g), \\ \forall j \neq i,\ \nexists c'_j \in Ans(g, c_g).c'_i \sqsubseteq_{\mathcal{D}} c'_j\ \}$$

4. the set of the answers of the forest $\mathcal{F}_P(Q, c_q)$, denoted by $Ans(Q, c_q)$, is the set of answers of $\tau_P(Q, c_q)$.

The order in which we search in the TCLP forest for a previous generator during the call entailment phase (Def. 2.12.3c) does not impact the completeness, soundness, or termination properties of the execution, but it can change its efficiency. Generators are naturally sorted from more particular ones (older) to more general (younger) ones — note that a younger, more particular call would be a consumer. Searching for a generator for a call can be performed in any direction. Starting at older, more particular generators, may need to examine several generators and perform potentially expensive entailment checks before finding one that suits the needs of the call. On the other hand, starting at younger, more general generators, should allow us to locate a suitable generator faster. However, this more general generator would have more answers associated which need to be filtered than what a more particular generator would have. Therefore, there is no definitive general strategy: either more generators have to be traversed, or more answers have to be filtered.

The answer management strategy used in the answer entailment phase (Def. 2.12.3g)

```
edge(a, b, 50).
edge(b, a,  D) :-
     D #> 25,
     D #< 35.
```

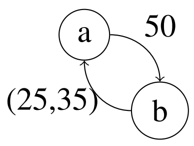


Figure 2.3: Code of a cycled graph. (25,35) is the open interval from 25 to 35.

aims at keeping only the more general answers by discarding/removing more particular answers. This is specified by the quantification $\forall j \neq i$. Simpler answer management strategies are possible: the implementations in (Chico de Guzmán et al., 2012; Cui and Warren, 2000), following (Toman, 1997b), only discard answers which are more particular than a previous one, i.e., they implement $\forall j < i$, and keep previously saved answers. A third possibility is to remove previous answers that are more particular than a new one, implementing $\forall j > i$. The choice among them does not impact soundness or completeness properties. However, discarding and removing redundant answers can greatly increase the efficiency of the implementation (see Chapter 3).

2.4 CLP Trees and TCLP Forests

Prolog and CLP follow a depth-first search strategy with chronological backtracking. The computation rule selects constraints and literals from the resolvent from left to right. Literals are resolved against the clauses of the program, selected from top to bottom. When a literal unifies with a clause head, it is substituted by the body of the clause after applying the unifier obtained from the literal-head unification. If a derivation branch fails because there are no more matching clauses or the constraint store is inconsistent, the evaluation backtracks to the youngest literal that has a candidate matching clause. Depth-first search is incomplete and in general not all answers can be computed. Moreover, there are programs with finite derivations for which logically equivalent programs produce infinite derivations. The use of TCLP can work around this issue in many cases.

We will show the CLP trees and the TCLP forests for the query `?- D #< 150, dist(a, Y, D)` for two logically equivalent versions of the `dist/3` program: with left recursion (Fig. 2.1, right) and with right recursion (Fig. 2.2, right). We use the graph in Fig. 2.3, where the length of one of the edges is defined with constraints.

Fig. 2.4 and Fig. 2.5 (top) are the CLP trees of the right- / left-recursive programs respectively. Fig. 2.5 (bottom) and Fig. 2.6 are the TCLP forest of the left- / right-recursive programs respectively. In these figures, the nodes of the trees represent the states (Def. 2.10) of the computation. A state is a tuple $\langle R, c \rangle$, where R is a sequence of goals, $[g_1, g_2, \ldots, g_n]$ and c is a conjunction of constraints. The numbers attached to each state indicates the order in which they are created.

On the one hand, Fig. 2.4 shows a finite CLP tree which finds all the answers and Fig. 2.5 (top) shows an infinite CLP tree caused by the left recursion. On the other hand, Fig. 2.5 (bottom) and Fig. 2.6 show that the TCLP forest for both programs are finite and all the answers to the query are found, since the combination of tabling and entailment makes it terminate with left recursion as well.

2.4.1 CLP Tree of `dist/3` with Right Recursion

Fig. 2.4 shows the CLP tree of the query using the version of `dist/3` with right recursion (Fig. 2.2, right). We see that the evaluation of the recursive clause generates similar states (s1, s4, s7 and s10), but in each iteration the domain of the constrained variable $D2_i$ is reduced. As a consequence, the constraint store in state s13 is inconsistent and the evaluation of this derivation fails. The pending branches are evaluated upon backtracking. We explain now how we obtain some of the states; the rest are obtained similarly, so we will skip them:

- **s1** the initial state is the representation of the query.
- **s2i/ii** are obtained by resolving the literal `dist(a, Y, D)` against the two clauses of the program. The constraints $Y_1{=}Y \wedge D_1{=}D$ are added to the constraint store.
- **s3** is obtained from the leftmost state s2i by adding the constraints of the resolvent $[D1_1 \#{>} 0,\ D2_1 \#{>} 0,\ D_1 \#{=} D1_1{+}D2_1]$ to the constraint store.
- **s4** is obtained by resolving the literal $edge(a, Z_1, D1_1)$. The constraint $Z_1{=}b \wedge D1_1{=}50$ reduces the domain[4] of $D2_1$ to $D2_1 > 0 \wedge D2_1 < 100$.
- **s7** is obtained by resolving the literal $edge(b, Z_2, D1_2)$. The constraint $Z_2{=}a \wedge D1_2 > 25 \wedge D1_2 < 35$ reduces the domain of $D2_2$ to $D2_2 > 0 \wedge D2_2 < 75$.
- **s10** is obtained by resolving the literal $edge(a, Z_3, D1_3)$. The constraint $Z_3{=}b \wedge \{D1_3{=}50$ reduces the domain of $D2_3$ to $D2_3 > 0 \wedge D2_3 < 25$.
- **s13** is obtained by resolving the literal $edge(b, Z_4, D1_4)$. The constraint $Z_4{=}a \wedge D1_4 > 25 \wedge D1_4 < 35$ is inconsistent with the current constraint store, $D < 150 \wedge D > 125{+}D1_4{+}D2_4 \wedge D2_4 > 0 \wedge \ldots$. Its child is a `fail` node.
- **s14** is obtained, upon backtracking to the state s11b by resolving the literal $edge(b, Y, D2_3)$. However, it is also a failed derivation because the resulting constraint store is inconsistent.

[4]We are considering a linear constraint solver over the rational numbers that from $D < 150 \wedge D{=}D1_1{+}D2_1 \wedge D1_1{=}50$ it infers that $D2_1 < 100$

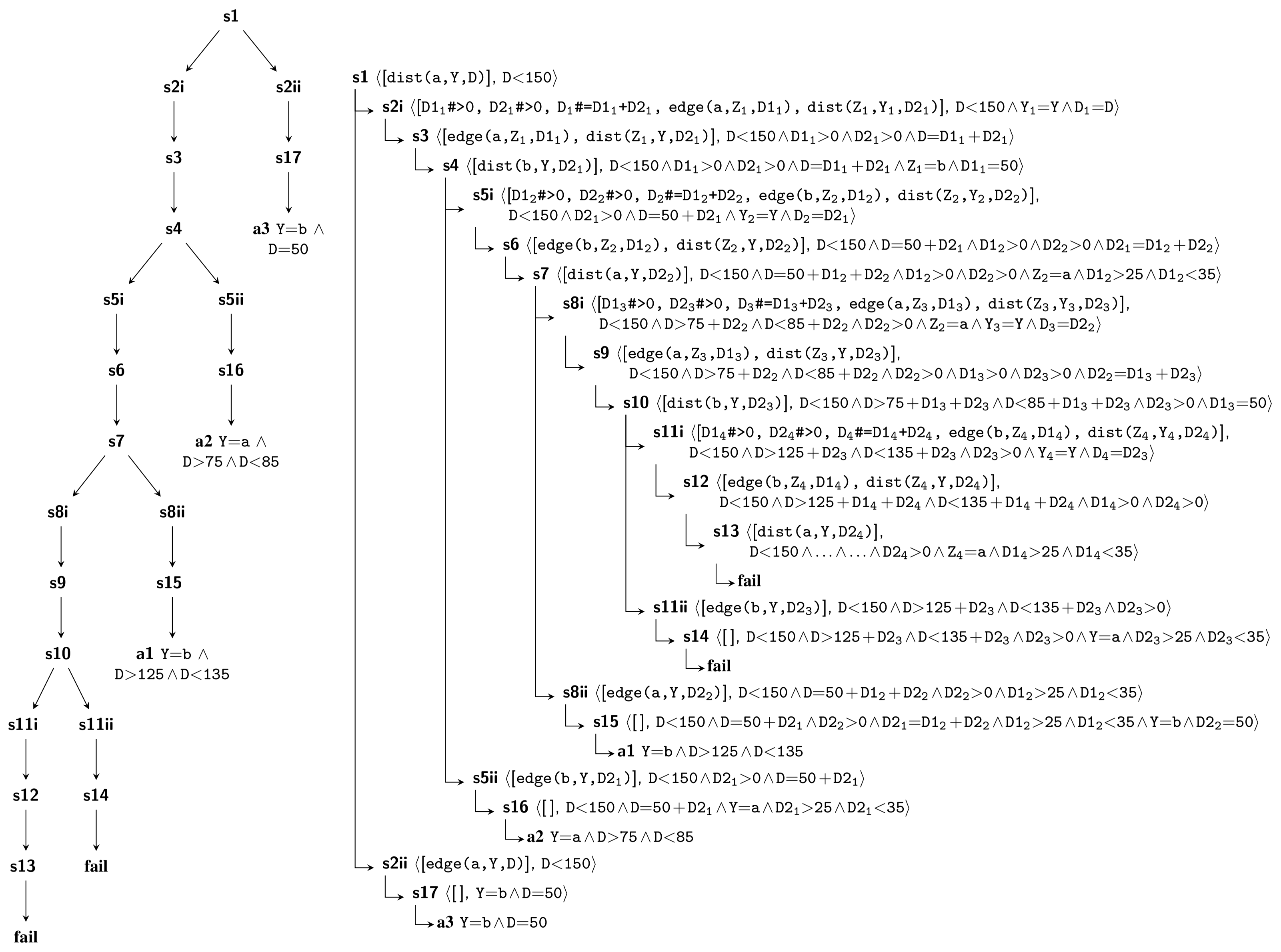

s1
s2i
s2ii
s3
s17
s4
a3 Y=b ∧ D=50
s5i
s5ii
s6
s16
s7
a2 Y=a ∧ D>75 ∧ D<85
s8i
s8ii
s9
s15
s10
a1 Y=b ∧ D>125 ∧ D<135
s11i
s11ii
s12
s14
s13
fail
fail
s1 ⟨[dist(a,Y,D)], D<150⟩
s2i ⟨[D1₁#>0, D2₁#>0, D₁#=D1₁+D2₁, edge(a,Z₁,D1₁), dist(Z₁,Y₁,D2₁)], D<150∧Y₁=Y∧D₁=D⟩
s3 ⟨[edge(a,Z₁,D1₁), dist(Z₁,Y,D2₁)], D<150∧D1₁>0∧D2₁>0∧D=D1₁+D2₁⟩
s4 ⟨[dist(b,Y,D2₁)], D<150∧D1₁>0∧D2₁>0∧D=D1₁+D2₁∧Z₁=b∧D1₁=50⟩
s5i ⟨[D1₂#>0, D2₂#>0, D₂#=D1₂+D2₂, edge(b,Z₂,D1₂), dist(Z₂,Y₂,D2₂)], D<150∧D2₁>0∧D=50+D2₁∧Y₂=Y∧D₂=D2₁⟩
s6 ⟨[edge(b,Z₂,D1₂), dist(Z₂,Y,D2₂)], D<150∧D=50+D2₁∧D1₂>0∧D2₂>0∧D2₁=D1₂+D2₂⟩
s7 ⟨[dist(a,Y,D2₂)], D<150∧D=50+D1₂+D2₂∧D1₂>0∧D2₂>0∧Z₂=a∧D1₂>25∧D1₂<35⟩
s8i ⟨[D1₃#>0, D2₃#>0, D₃#=D1₃+D2₃, edge(a,Z₃,D1₃), dist(Z₃,Y₃,D2₃)], D<150∧D>75+D2₂∧D<85+D2₂∧D2₂>0∧Z₂=a∧Y₃=Y∧D₃=D2₂⟩
s9 ⟨[edge(a,Z₃,D1₃), dist(Z₃,Y,D2₃)], D<150∧D>75+D2₂∧D<85+D2₂∧D2₂>0∧D1₃>0∧D2₃>0∧D2₂=D1₃+D2₃⟩
s10 ⟨[dist(b,Y,D2₃)], D<150∧D>75+D1₃+D2₃∧D<85+D1₃+D2₃∧D2₃>0∧D1₃=50⟩
s11i ⟨[D1₄#>0, D2₄#>0, D₄#=D1₄+D2₄, edge(b,Z₄,D1₄), dist(Z₄,Y₄,D2₄)], D<150∧D>125+D2₃∧D<135+D2₃∧D2₃>0∧Y₄=Y∧D₄=D2₃⟩
s12 ⟨[edge(b,Z₄,D1₄), dist(Z₄,Y,D2₄)], D<150∧D>125+D1₄+D2₄∧D<135+D1₄+D2₄∧D1₄>0∧D2₄>0⟩
s13 ⟨[dist(a,Y,D2₄)], D<150∧...∧...∧D2₄>0∧Z₄=a∧D1₄>25∧D1₄<35⟩
fail
s11ii ⟨[edge(b,Y,D2₃)], D<150∧D>125+D2₃∧D<135+D2₃∧D2₃>0⟩
s14 ⟨[], D<150∧D>125+D2₃∧D<135+D2₃∧D2₃>0∧Y=a∧D2₃>25∧D2₃<35⟩
fail
s8ii ⟨[edge(a,Y,D2₂)], D<150∧D=50+D1₂+D2₂∧D2₂>0∧D1₂>25∧D1₂<35⟩
s15 ⟨[], D<150∧D=50+D2₁∧D2₂>0∧D2₁=D1₂+D2₂∧D1₂>25∧D1₂<35∧Y=b∧D2₂=50⟩
a1 Y=b∧D>125∧D<135
s5ii ⟨[edge(b,Y,D2₁)], D<150∧D2₁>0∧D=50+D2₁⟩
s16 ⟨[], D<150∧D=50+D2₁∧Y=a∧D2₁>25∧D2₁<35⟩
a2 Y=a∧D>75∧D<85
s2ii ⟨[edge(a,Y,D)], D<150⟩
s17 ⟨[], Y=b∧D=50⟩
a3 Y=b∧D=50

Figure 2.4: CLP tree of `?- D #< 150, dist(a, Y, D)` with right recursion.

s15 is a final state of a successful derivation, obtained upon backtracking to the state s8b by resolving the literal `edge(a, Y, D2`$_2$`)`. The constraint `Y=a` $\wedge$ `D2`$_3$ `> 25` $\wedge$ `D2`$_3$ `< 35` is consistent with the constraint store.

a1 is the first answer `Y=a` $\wedge$ `D > 125` $\wedge$ `D < 315`, projected onto the variables of the query (`vars(Q)={Y, D}`).

s16 is a final state obtained upon backtracking to the state s5b.

a2 is the second answer, `Y=a` $\wedge$ `D > 75` $\wedge$ `D < 85`.

s17 is a final state obtained upon backtracking to the state s2ii.

a3 is the third and last answer, `Y=b` $\wedge$ `D=50`.

2.4.2 CLP Tree of `dist/3` with Left Recursion

Fig. 2.5 (top) shows the CLP tree of the query to `dist/3` with left recursion (Fig. 2.1, right). We see that the recursive clause also generates similar states (s1, s3, s5, ...) but in this example the domain of the constrained variable $D1_i$ remains unchanged, and the evaluation therefore enters a loop. As before, we only explain how we obtain some of the states:

s3 is obtained from the leftmost state s2i. The domain of `D1`$_1$ is `D1`$_1$ `> 0` $\wedge$ `D1`$_1$ `< 150`.

s5 is obtained from the leftmost node s4i. The domain of `D1`$_2$ is `D1`$_2$ `> 0` $\wedge$ `D1`$_2$ `< 150`.

... the evaluation enters a loop.

Although the program that generates this CLP tree is logically equivalent to the previous one, this tree is infinite and no answers are found.

2.4.3 TCLP Forest of `dist/3` with Left Recursion

Fig. 2.5 (bottom) shows the TCLP forest for the query we have been using with the `dist/3` program written using left recursion (Fig. 2.1, right), where the set of tabled predicates is `Tab`$_P$`={dist/3}`. The main point is that at state s3 the tabling engine detects that the evaluation of $\langle$`dist(a, Z`$_1$`,D1`$_1$`), D1`$_1$ `> 0` $\wedge$ `D1`$_1$ `< 150`$\rangle$ entails the generator $\langle$`dist(a, V0, V1), V1 < 150`$\rangle$ and therefore it suspends the execution and waits until another generator feeds the suspended goal with answers. The evaluation of the state s2ii generates the first answer a1 upon backtracking. Then, the tabling engine resumes the consumer with a1 and generates a2 which is used to generate a3. Finally,

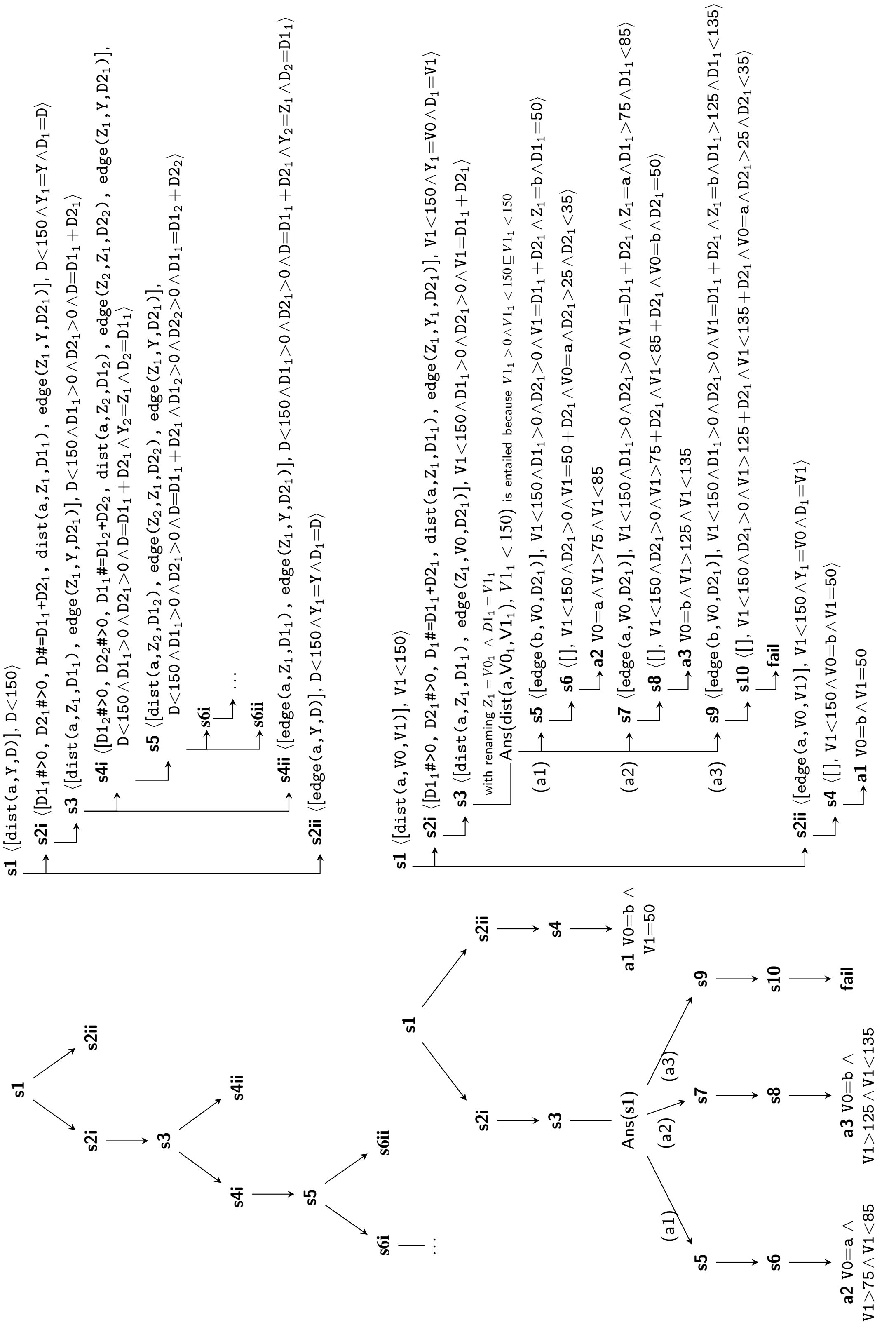

s1 ⟨[dist(a,Y,D)], D<150⟩
s2i ⟨[D1₁#>0, D2₁#>0, D#=D1₁+D2₁, dist(a,Z₁,D1₁), edge(Z₁,Y,D2₁)], D<150∧Y₁=Y∧D₁=D⟩
s3 ⟨[dist(a,Z₁,D1₁), edge(Z₁,Y,D2₁)], D<150∧D1₁>0∧D2₁>0∧D=D1₁+D2₁⟩
s4i ⟨[D1₂#>0, D2₂#>0, D1₁#=D1₂+D2₂, dist(a,Z₂,D1₂), edge(Z₂,Z₁,D2₂), edge(Z₁,Y,D2₁)], D<150∧D1₁>0∧D2₁>0∧D=D1₁+D2₁∧Y₂=Z₁∧D₂=D1₁⟩
s5 ⟨[dist(a,Z₂,D1₂), edge(Z₂,Z₁,D2₂), edge(Z₁,Y,D2₁)], D<150∧D1₁>0∧D2₁>0∧D=D1₁+D2₁∧D1₂>0∧D2₂>0∧D1₁=D1₂+D2₂⟩
s6i
...
s6ii
s4ii ⟨[edge(a,Z₁,D1₁), edge(Z₁,Y,D2₁)], D<150∧D1₁>0∧D2₁>0∧D=D1₁+D2₁∧Y₂=Z₁∧D₂=D1₁⟩
s2ii ⟨[edge(a,Y,D)], D<150∧Y₁=Y∧D₁=D⟩
s1 ⟨[dist(a,V0,V1)], V1<150⟩
s2i ⟨[D1₁#>0, D2₁#>0, D₁#=D1₁+D2₁, dist(a,Z₁,D1₁), edge(Z₁,Y₁,D2₁)], V1<150∧Y₁=V0∧D₁=V1⟩
s3 ⟨[dist(a,Z₁,D1₁), edge(Z₁,V0,D2₁)], V1<150∧D1₁>0∧D2₁>0∧V1=D1₁+D2₁⟩
with renaming Z₁ = V0₁ ∧ D1₁ = V1₁
Ans(dist(a,V0₁,V1₁), V1₁ < 150) is entailed because V1₁ > 0 ∧ V1₁ < 150 ⊑ V1₁ < 150
(a1) s5 ⟨[edge(b,V0,D2₁)], V1<150∧D1₁>0∧D2₁>0∧V1=D1₁+D2₁∧Z₁=b∧D1₁=50⟩
s6 ⟨[], V1<150∧D2₁>0∧V1=50+D2₁∧V0=a∧D2₁>25∧D2₁<35⟩
a2 V0=a∧V1>75∧V1<85
(a2) s7 ⟨[edge(a,V0,D2₁)], V1<150∧D1₁>0∧D2₁>0∧V1=D1₁+D2₁∧Z₁=a∧D1₁>75∧D1₁<85⟩
s8 ⟨[], V1<150∧D2₁>0∧V1>75+D2₁∧V1<85+D2₁∧V0=b∧D2₁=50⟩
a3 V0=b∧V1>125∧V1<135
(a3) s9 ⟨[edge(b,V0,D2₁)], V1<150∧D1₁>0∧D2₁>0∧V1=D1₁+D2₁∧Z₁=b∧D1₁>125∧D1₁<135⟩
s10 ⟨[], V1<150∧D2₁>0∧V1>125+D2₁∧V1<135+D2₁∧V0=a∧D2₁>25∧D2₁<35⟩
fail
s2ii ⟨[edge(a,V0,V1)], V1<150∧Y₁=V0∧D₁=V1⟩
s4 ⟨[], V1<150∧V0=b∧V1=50⟩
a1 V0=b∧V1=50
s1
s2i
s2ii
s3
s4i
s4ii
s5
s6i
s6ii
...
s1
s2i
s3
Ans(s1)
(a1)
(a2)
(a3)
s5
s6
a2 V0=a ∧ V1>75∧V1<85
s7
s8
a3 V0=b ∧ V1>125∧V1<135
s9
s10
fail
s2ii
s4
a1 V0=b ∧ V1=50

Figure 2.5: CLP tree (top) and TCLP forest (bottom) of
`?- D #< 150, dist(a, Y, D)` with left recursion.

the evaluation fails after consuming a3 and, since all the clauses have been evaluated and there are no more consumers to be resumed or answers to be consumed, the generator is marked as complete and all the answers are returned. We explain below how some of the states are obtained. The rest of the states are obtained similarly, so we skip them for brevity:

s0 We omit the representation of the TCLP tree for the query $\mathcal{T}_P(\texttt{dist}(\texttt{a},\texttt{Y},\texttt{D}),\texttt{D} < 150)$ and its answer resolution.

s1 the initial state of the TCLP tree $\mathcal{T}_P(\texttt{dist}(\texttt{a},\texttt{V0}_1,\texttt{V1}_1),\texttt{V1}_1 < 150)$ is the renamed generator. (Def. 2.12.3c).

s2i/ii are obtained by resolving the literal `dist(a, V0, V1)` against the two clauses of the program.

s3 is obtained from the leftmost state s2i by adding the constraints to the constraint store as in the CLP tree.

Ans(s1) the tabled literal $\texttt{dist(a,Z}_1\texttt{,D1}_1\texttt{)}$ has to be resolved by answer resolution (Def. 2.12.3c) using the answer from the current TCLP tree $\mathcal{T}_P(\texttt{dist}(\texttt{a},\texttt{V0}_1,\texttt{V1}_1),\texttt{V1}_1 < 150)$ because, after renaming, the projection of the current constraint store onto the variables of the literal entails the projected constraint store of the generator: $\texttt{V1}_1 > 0 \land \texttt{V1}_1 < 150 \sqsubseteq \texttt{V1}_1 < 150$. Since the current TCLP tree is under construction and depends on itself, this branch derivation is suspended.

s4 is a final state of a successful derivation. It is obtained, upon backtracking to the state s2ii, by resolving with `edge(a, V0, V1)`. The equations $\texttt{V0=b} \land \texttt{V1=50}$ are consistent with the constraint store.

a1 is the first answer, $\texttt{V0=b} \land \texttt{V1=50}$ (Def. 2.12.3g). Since is the first one, it is also the more general one.

s5 is obtained from the state s3 (because there are no more branches) by answer resolution consuming a1 (Def. 2.12.3g).

s6 is a final state obtained by resolving the literal $\texttt{edge(b, V0, D2}_1\texttt{)}$.

a2 is the second answer, $\texttt{V0=a} \land \texttt{V1} > 75 \land \texttt{V1} < 85$. It is neither more particular nor more general than a1.

s7 is obtained from the state s3 by consuming a2.

s8 is a final state.

a3 is the third answer, $\texttt{V0=b} \land \texttt{V1} > 125 \land \texttt{V1} < 135$. It is neither more particular nor more general than a1 or a2.

s9 is obtained from the state s3 by consuming, a3.

s10 is a failed derivation because the resulting constraint store is inconsistent, $\texttt{V1} < 150 \land \ldots \land \texttt{V1} > 125 + \texttt{D2}_1 \land \texttt{D2}_1 > 25$. Its child is a fail node.

Note that the CLP execution entered a loop when resolving the state s3. Under TCLP, answer resolution avoids looping and the resulting TCLP forest is finite and complete (i.e., the leaves of the trees are either fail nodes or answers).

2.4.4 TCLP Forest of `dist/3` with Right Recursion

Fig. 2.6 shows the TCLP forest corresponding to querying the right recursive `dist/3` program (Fig. 2.2, right). This example is useful to show how the algorithm works with mutually dependent generators[5] and to see why not all the answers from a generator may be directly used by its consumers.

Unlike the left-recursive version, which shows only one TCLP tree (Fig. 2.5, bottom), Fig. 2.6 has two TCLP trees (one for each generator). That is because the left recursive version only sought paths from the node a, but the right recursive version creates a new TCLP tree at the state s4 to collect the paths from the node b, since `edge(a, b)` had been previously evaluated at state s3. As before we only explain how we obtain some of the states:

s1 the TCLP tree $\mathcal{T}_P(\texttt{dist}(\texttt{a}, \texttt{V0}, \texttt{V1}), \texttt{V1} < 150)$ is created.

s4 is obtained by resolving the literal $\texttt{edge(a, Z}_1\texttt{,D1}_1\texttt{)}$.

Ans(s5) the tabled literal $\texttt{dist(b, V0, D2}_1\texttt{)}$ is a new generator and a new TCLP tree $\mathcal{T}_P(\texttt{dist}(\texttt{b}, \texttt{V2}, \texttt{V3}), \texttt{V3} > 0 \land \texttt{V3} < 100)$ is created (Def. 2.12.3c).

s5 is the root node of the new TCLP tree.

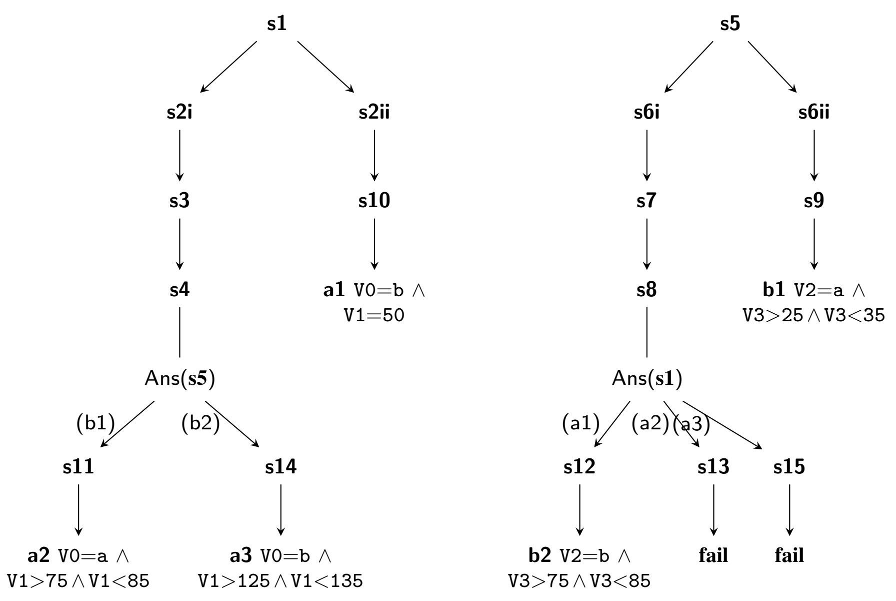
s6i/ii are obtained by resolving the literal `dist(b, V2, V3)` against the clauses of the program.

s8 is obtained by resolving the literal $\texttt{edge(b, Z}_1\texttt{,D1}_1\texttt{)}$.

In the state s8, the call $\langle \texttt{dist(a, V2, D2}_1\texttt{)}, \texttt{D2}_1 > 0 \land \texttt{D2}_1 < 75 \rangle$ is suspended because it entails the former generator $\langle \texttt{dist(a, V0}_1\texttt{,V1}_1\texttt{)}, \texttt{V1}_1 < 150 \rangle$.

Ans(s1) the tabled literal $\texttt{dist(a, V2, D2}_1\texttt{)}$ is resolved with answer resolution (Def. 2.12.3g) using the answers from the previous TCLP tree

[5]I.e., generators which consume answers from each other.



s1 $\langle$[dist(a,V0,V1)], $V1<150\rangle$

↳ **s2i** $\langle$[$D1_1$#>0, $D2_1$#>0, $V1_1$#=$D1_1$+$D2_1$, edge(a,Z_1,$D1_1$), dist(Z_1,Y_1,$D2_1$)], $V1<150 \wedge Y_1=V0 \wedge D_1=V1\rangle$

↳ **s3** $\langle$[edge(a,Z_1,$D1_1$), dist(Z_1,V0,$D2_1$)], $V1<150 \wedge D1_1>0 \wedge D2_1>0 \wedge V1=D1_1+D2_1\rangle$

↳ **s4** $\langle$[dist(b,V0,$D2_1$)], $V1<150 \wedge D1_1>0 \wedge D2_1>0 \wedge V1=D1_1+D2_1 \wedge Z_1=b \wedge D1_1=50\rangle$

with renaming $V0=V2 \wedge D2_1=V3$

Ans(dist(b,V2,V3), $V3>0 \wedge V3<100$)

(b1) ↳ **s11** $\langle$[], $V1<150 \wedge D2_1>0 \wedge V1=50+D2_1 \wedge V0=a \wedge D2_1>25 \wedge D2_1<35\rangle$

↳ **a2** $V0=a \wedge V1>75 \wedge V1<85$

(b2) ↳ **s14** $\langle$[], $V1<150 \wedge D2_1>0 \wedge V1=50+D2_1 \wedge V0=b \wedge D2_1>75 \wedge D2_1<85\rangle$

↳ **a3** $V0=b \wedge V1>125 \wedge V1<135$

↳ **s2ii** $\langle$[edge(a,V0,V1)], $V1<150 \wedge Y_1=V0 \wedge D_1=V1\rangle$

↳ **s10** $\langle$[], $V1<150 \wedge V0=b \wedge V1=50\rangle$

↳ **a1** $V0=b \wedge V1=50$

s5 $\langle$[dist(b,V2,V3)], $V3>0 \wedge V3<100\rangle$

↳ **s6i** $\langle$[$D1_1$#>0, $D2_1$#>0, D_1#=$D1_1$+$D2_1$, edge(b,Z_1,$D1_1$), dist(Z_1,Y_1,$D2_1$)], $V3>0 \wedge V3<100 \wedge Y_1=V2 \wedge D_1=V3\rangle$

↳ **s7** $\langle$[edge(b,Z_1,$D1_1$), dist(Z_1,V2,$D2_1$)], $V3>0 \wedge V3<100 \wedge D1_1>0 \wedge D2_1>0 \wedge V3=D1_1+D2_1\rangle$

↳ **s8** $\langle$[dist(a,V2,$D2_1$)], $V3>0 \wedge V3<100 \wedge D1_1>0 \wedge D2_1>0 \wedge V3=D1_1+D2_1 \wedge Z_1=a \wedge D1_1>25 \wedge D1_1<35\rangle$

with renaming $V2=V0_1 \wedge D2_1=V1_1$

Ans(dist(a,$V0_1$,$V1_1$), $V1_1<150$) is entailed because $V1_1>0 \wedge V1_1<75 \sqsubseteq V1_1<150$

(a1) ↳ **s12** $\langle$[], $V3>0 \wedge V3<100 \wedge D2_1>0 \wedge V3>25+D2_1 \wedge V3<35+D2_1 \wedge V2=b \wedge D2_1=50\rangle$

↳ **b2** $V2=b \wedge V3>75 \wedge V3<85$

(a2) ↳ **s13** $\langle$[], $V3>0 \wedge V3<100 \wedge D2_1>0 \wedge V3>25+D2_1 \wedge V3<35+D2_1 \wedge V2=a \wedge D2_1>75 \wedge D2_1<85\rangle$

↳ **fail**

(a3) ↳ **s15** $\langle$[], $V3>0 \wedge V3<100 \wedge D2_1>0 \wedge V3>25+D2_1 \wedge V3<35+D2_1 \wedge V2=b \wedge D2_1>125 \wedge D2_1<135\rangle$

↳ **fail**

↳ **s6ii** $\langle$[edge(b,V2,V3)], $V3>0 \wedge V3<100 \wedge Y_1=V2 \wedge D_1=V3\rangle$

↳ **s9** $\langle$[], $V3>0 \wedge V3<100 \wedge V2=a \wedge V3>25 \wedge V3<35\rangle$

↳ **b1** $V2=a \wedge V3>25 \wedge V3<35$

Figure 2.6: TCLP forest of `?- D #< 150, dist(a, Y, D)` with right recursion.

$\mathcal{T}_P(\mathtt{dist(a,V0_1,V1_1)},\mathtt{V1_1} < \mathtt{150})$ because the renamed projection[6] of the current constraint store onto the variable of the literal entails the projected constraint store of the generator: $(\mathtt{V1_1>0} \wedge \mathtt{V1_1<75}) \sqsubseteq \mathtt{V1_1<150}$. Since the initial TCLP forest is under construction and depends on itself, the current branch derivation is suspended.

This suspension also causes the former generator to suspend at the state s4.

s9 is a final state obtained upon backtracking to the state s6ii.

b1 is the first answer of the second generator.

At this point the suspended calls can be resumed by consuming the answer b1 or by evaluating s2ii. The algorithm first tries to evaluate s2ii and then it will resume s4 consuming b1.

s10 is a final state obtained upon backtracking to the state s2ii.

a1 is the first answer of the first generator: $\mathtt{V0=b} \wedge \mathtt{V1=50}$.

s11 is a final state obtained from the state s4 by consuming b1.

a2 is the second answer of the first generator: $\mathtt{V0=a} \wedge \mathtt{V1>75} \wedge \mathtt{V1<85}$.

s12 is a final state obtained from the state s8 by consuming a1.

b2 is the second answer of the second generator.

s13 is a failed derivation obtained from s8 by consuming a2. It fails because the constraints $\mathtt{V0=a} \wedge \mathtt{V1>75} \wedge \mathtt{V1<85}$ are inconsistent with the current constraint store. Note that the projection of the constraint store of s8 onto V1 is $\mathtt{V1>0} \wedge \mathtt{V1<75}$. Its child is a `fail` node.

s14 is a final state obtained from the state s4 by consuming b2.

a3 is the third answer of the first generator: $\mathtt{V0=b} \wedge \mathtt{V1>125} \wedge \mathtt{V1<135}$.

s15 is a failed derivation obtained from s8 by consuming a3. Its child is a `fail` node.

This example illustrates why left recursion reduces the execution time and memory requirements when using tabling / TCLP: left recursion will usually create fewer generators. We have also seen that using answers from a more general call, as in the answer resolution of state s8 (i.e., the constraint store of the consumer $\mathtt{V1_1>0} \wedge \mathtt{V1_1<75}$

[6]The projection of $\mathtt{V3>0} \wedge \mathtt{V3<100} \wedge \mathtt{D1_1>0} \wedge \mathtt{D2_1>0} \wedge \mathtt{V3=D1_1+D2_1} \wedge \mathtt{Z_1=a} \wedge \mathtt{D1_1>25} \wedge \mathtt{D1_1<35}$ onto $\mathtt{D2_1}$ is $\mathtt{D2_1>0} \wedge \mathtt{D2_1<75}$. After renaming $\mathtt{D2_1=V1_1}$, the resulting projection is $\mathtt{V1_1>0} \wedge \mathtt{V1_1<75}$.

is more particular than the constraint store of the generator `V1`$_1$ `< 150`), makes it necessary to filter the correct ones (i.e., answer resolution for `a2` and `a3` failed). This is not required in variant tabling because the answers from a generator are always valid for its consumers.

2.5 Theorems and Proofs

In this section we prove the soundness and completeness of TCLP (Section 2.5.1), and present some results on termination properties (Section 2.5.2) for general constraint solvers.

2.5.1 Soundness and Completeness

(Toman, 1997b) proves soundness and completeness of SLG^C for TCLP Datalog programs by reduction to soundness and completeness of bottom-up evaluation. It is possible to extend these results to prove soundness and completeness of our proposal: they only differ in the answer management strategy and the construction of the TCLP forest. The strategy used in SLG^C only discards answers which are more particular than a previous answer, while in our proposal we in addition remove previously existing more particular answers (Def. 2.12.3g). The result of this is that only the most general answers are kept. In SLG^C, the generation of the forest is modeled as the application of rewriting rules. In our proposal, the TCLP forest is defined as a transition system (Def. 2.12), where the different cases in the definition can be seen as rules which make the TCLP forest evolve.

The lemma and theorems and their proofs are adapted taking in consideration these differences. First we prove that answer resolution using entailment is correct w.r.t. SLD resolution; and although only the most general answers are kept, answer resolution using entailment is complete w.r.t. SLD resolution. Then we use these results to prove soundness and completeness of TCLP with entailment w.r.t. the least fixed point semantics.

Lemma 2.1 (Application of derivations with more general constraint stores)**.** *Let $\langle [l_i, \dots, l_k], cs_i\rangle \rightsquigarrow \langle [l_{i+1}, \dots, l_k], cs_{i+1}\rangle$ be a derivation and $\langle l_i, c\rangle$ a goal with $cs_i \sqsubseteq c$. Then:*

$$\exists \langle l_i, c\rangle \rightsquigarrow \langle \emptyset, c'\rangle .\ cs_{i+1} = cs_i \wedge c'$$

Proof. We will see that there exists a derivation $\langle l_i, c\rangle \rightsquigarrow \langle \emptyset, c'\rangle$ that follows the same steps as $\langle [l_i, \dots, l_k], cs_i\rangle \rightsquigarrow \langle [l_{i+1}, \dots, l_k], cs_{i+1}\rangle$:

(1) if $\langle [l_i, \ldots, l_k], cs_i\rangle$ is resolved against a clause l_i :- c_h, then its resulting constraint store is $cs_{i+1} = cs_i \wedge c_h$. Since $cs_i \sqsubseteq c$, we can apply the same rule to $\langle l_i, c\rangle$ and its resulting constraint store is $c' = c \wedge c_h$. Since $cs_i \sqsubseteq c$, we have $cs_i \Leftrightarrow cs_i \wedge c$. Therefore, $cs_{i+1} = cs_i \wedge c \wedge c_h$ (expanding cs_i) and $cs_{i+1} = cs_i \wedge c'$ (contracting $c \wedge c_h$).

(2) if $\langle [l_i, \ldots, l_k], cs_i\rangle$ is resolved against a clause l_i :- $c_h, a_1, \ldots, a_m$, the next state is $\langle [a_1, \ldots, a_m, l_{i+1}, \ldots, l_k], cs_i \wedge c_h\rangle$ (resp. $\langle [a_1, \ldots, a_m], c \wedge c_h\rangle$). By induction, since $cs_i \sqsubseteq true$ (resp. $c \sqsubseteq true$), there exist m derivations $\langle a_j, true\rangle \rightsquigarrow \langle \emptyset, c'_{a_j}\rangle$ such that the resulting constraint store of the path is $cs_{i+a1} = cs_i \wedge c_h \wedge \bigwedge_{j=1}^{m} c'_{a_j}$ (resp. $c' = c \wedge c_h \wedge \bigwedge_{j=1}^{m} c'_{a_j}$). Since $cs_i \sqsubseteq c$, we have $cs_i \Leftrightarrow cs_i \wedge c$. Therefore, $cs_{i+1} = cs_i \wedge c \wedge c_h \wedge \bigwedge_{j=1}^{m} c'_{a_j}$ (expanding cs_i) and $cs_{i+1} = cs_i \wedge c'$ (contracting $c \wedge c_h \wedge \bigwedge_{j=1}^{m} c'_{a_j}$). □

Corollary 2.1. *[Correctness of answer resolution using entailment] As an immediate consequence of Lemma 2.1, using answer resolution with entailment (Def. 2.12.3c) gives correct results. Answer resolution of $\langle [l_i, \ldots, l_k], cs_i\rangle$ consumes an answer c' from a previous derivation $\langle l_i, c\rangle \rightsquigarrow \langle \emptyset, c'\rangle$ where $\langle l_i, c\rangle$ is the generator of the derivation and, by the definition of generator, $cs_i \sqsubseteq c$. When $\mathcal{D} \models cs_i \wedge c'$ (Def. 2.12.3d), it generates the state $\langle [l_{i+1}, \ldots, l_k], cs_i \wedge c'\rangle$.*

Corollary 2.2. *[Completeness of answer resolution using entailment] Recall that $Ans(l, c)$ is the set containing the more general answers for a generator goal $\langle l, c\rangle$ (Def. 2.12.3g), and if there are two goals $\langle l, c_a\rangle$ and $\langle l, c_b\rangle$ with $c_a \sqsubseteq c_b$, only the answers for the more general goal c_b need to be kept. Therefore, for any derivation of a generator $\langle l_i, c\rangle \rightsquigarrow \langle \emptyset, c_i\rangle$ we have that $\exists c'_i \in Ans(l_i, c').c_i \sqsubseteq c'_i$ for some c' s.t. $c \sqsubseteq c'$. Let us take a (partial) clause derivation $\langle [l_i, \ldots, l_k], c\rangle \rightsquigarrow \langle [l_{i+1}, \ldots, l_k], c \wedge c_i\rangle$. If $c'_i \in Ans(l_i, c')$ for some c' s.t. $c \sqsubseteq c'$ (which is the entailment condition necessary to use the saved answer constraints), then $c_i \sqsubseteq c'_i$. If we use c'_i to perform answer resolution with $\langle l_i, c\rangle$, we have $\langle [l_i, \ldots, l_k], c\rangle \rightsquigarrow \langle [l_{i+1}, \ldots, l_k], c \wedge c'_i\rangle$. Given that $c_i \sqsubseteq c'_i$, we have that $c \wedge c_i \sqsubseteq c \wedge c'_i$, and any answer returned by clause resolution is contained in some answer returned by answer resolution with entailment. The same reasoning can be applied to the derivation of l_{i+1} and so on. Therefore, answer resolution with entailment does not lose answers w.r.t. clause resolution even if not all the goals and answers are memorized.*

Theorem 2.1 (Soundness w.r.t. the fixpoint semantics)**.** *Let P be a TCLP definite program and (q, c_q) a query. Then for any answer c' of the TCLP forest $\mathcal{F}_P(q, c_q)$*

$$c' \in Ans(q, c_q) \ \Rightarrow \exists (q, c) \in lfp(S_P^{\mathcal{D}}(\emptyset)).\ c' = c_q \wedge c$$

I.e., all the answers derived from the forest construction are also derived from the bottom-up computation.

Proof. For any answer $c' \in Ans(q, c_q)$ there exists a successful derivation $\langle q, c_q\rangle \rightsquigarrow \langle \emptyset, c'\rangle$. Since $c_q \sqsubseteq true$, by Lemma 2.1 there exists $\langle q, true\rangle \rightsquigarrow \langle \emptyset, c\rangle$. $c' = c_q \wedge c$. We know that for any successful derivation $\langle q, true\rangle \rightsquigarrow \langle \emptyset, c\rangle$ against the clauses of the program there is an answer derived from the bottom-up computation $(q,c) \in lfp(S_P^{\mathcal{D}}(\emptyset))$. Therefore, by Corollary 2.1 if answer resolution is used instead of clause resolution, the result is also correct and for any answer $c' \in Ans(q, c_q)$ there exists $(q,c) \in lfp(S_P^{\mathcal{D}}(\emptyset))$. $c' = c_q \wedge c$. □

Theorem 2.2 (Completeness w.r.t. the fixpoint semantics)**.** *Let P be a TCLP definite program and $(h, true)$ a query. Then for every (h,c) in $lfp(S_P^{\mathcal{D}})$:*

$$(h,c) \in lfp(S_P^{\mathcal{D}}(\emptyset)) \;\Rightarrow\; \exists c' \in Ans(h, true).\; c \sqsubseteq c'$$

I.e., all the answers derived from the bottom-up computation are also derived by the forest construction or entailed by answers inferred in the forest.

Proof. We know that for any answer derived from the bottom-up computation $(h,c) \in lfp(S_P^{\mathcal{D}}(\emptyset))$ there exists a successful derivation $\langle h, true\rangle \rightsquigarrow \langle \emptyset, c\rangle$ against the clauses of the program. By Corollary 2.2 if answer resolution is used instead of clause resolution, the results is also complete. Therefore, since the answer management strategy only keeps the more general answers (Def. 2.12.3g), we have that $\exists c' \in Ans(h, true).\; c \sqsubseteq c'$. □

2.5.2 Termination

The next definition is a fundamental property of some constraint domains that plays a key role in the termination of the evaluation of queries to TCLP programs (Toman, 1997b).

Definition 2.13 (Constraint-compact)**.** Let $\mathcal{D}$ be a constraint domain, and D the set of all constraints expressable in $\mathcal{D}$. Then $\mathcal{D}$ is constraint-compact iff:

– for every finite set of variables S, and
– for every subset $C \subseteq D$ such that $\forall c \in C. vars(c) \subseteq S$,

there is a finite subset $C_{fin} \subseteq C$ such that $\forall c \in C. \exists c' \in C_{fin}. c \sqsubseteq_{\mathcal{D}} c'$

This definition establishes sufficient conditions for the termination of any TCLP evaluation under a compact constraint domain. Intuitively speaking, a constraint domain $\mathcal{D}$ is constraint-compact if for any (potentially infinite) set of constraints C expressable in $\mathcal{D}$, there is a *finite* set of constraints $C_{fin} \subseteq C$ that covers (in the sense of $\sqsubseteq_{\mathcal{D}}$) C. In other words, C_{fin} is as general as C. Additionally, in a constraint-compact constraint domain, if an infinite set of constraints is unsatisfiable, then there is a finite subset which is unsatisfiable, therefore guaranteeing the existence of finite unsatisfiability proofs.

Example 2.2.
The gap-order constraints (Revesz, 1993) is a constraint-compact constraint domain generated from the set $\mathcal{C}_{<Z} = \{x < u : u \in A\} \cup \{u < x : u \in A\} \cup \{x + k < y : k \in Z^+\}$ where A is a finite set of constants. First, we see that the set $C_{x<u}$ (resp. $C_{u<x}$) of possible constraints of the form $x < u$ (resp. $u < x$), for a finite set of variables S, is finite, because A and S are finite. Therefore, it is trivial to define the finite set which covers them ($C_{x<u} \cup C_{u<x}$). Second, for every pair of variables $x, y \in S$, the set $C_{x+k<y}$ of possible constraint of the form $x+k < y$, $k \in Z^+$ can be covered by a finite subset of itself. Since S is finite, we only have to check it for two given x, y; we can repeat the same process for every pair of variables, since there is only a finite number of them. Although for a given pair of variables x, y one can generate an infinite number of constraints $x + k_i < y$ choosing different $k_i \in Z^+$, the constraint $x + k_0 < y$ having the smallest k_0 among all the k_i ($\forall k_i . k_0 \leq k_i$) subsumes all the rest of the constraints ($x + k_i < y \sqsubseteq x + k_0 < y$). Note that k_0 always exists, since $k_i \in Z^+$, which has a minimum. Therefore, the infinite set $C_{x+k<y}$ has a finite subset $C_{fin} = \{x + k_0 < y\}$ which covers it ($C_{x+k<y} \sqsubseteq C_{fin}$).

Example 2.3.
The Herbrand constraint domain is not constraint-compact. Take the infinite set of constraints $C = \{X = a, X = f(a), X = f(f(a)), \ldots\}$. No finite subset of C using only constraints in C can cover C.

(Toman, 1997b) proves termination of TCLP Datalog programs under a top-down strategy when the constraint domain is constraint-compact. In that case, the evaluation will suspend the exploration of a call whose constraint store is less general or comparable to a previous call. Eventually, the program will generate a set of call constraint stores which can cover any infinite set of constraints in the constraint domain, therefore finishing evaluation.

Most TCLP applications require constraint domains which are not constraint-compact because constraint-compact constraint domains are not very expressive. Therefore, we refined the termination theorem (Theorem 23 in (Toman, 1997b)) for Datalog programs with constraint-compact constraint domains to cover cases where a program, during the evaluation, generates only a constraint-compact subset of all constraints expressable in the constraint domain.

Theorem 2.3 (Termination)**.** *Let P be a TCLP($\mathcal{D}$) definite program and (Q, c_q) a query. Then the TCLP execution terminates iff:*

- *For every literal g, the set C_g is constraint-compact, where C_g is the set of all the constraint stores c_i, projected and renamed w.r.t. the arguments of g, s.t. $\langle g, c_i \rangle$ is in the forest $\mathcal{F}(Q, c_q)$.*
- *For every goal $\langle g, c_g \rangle$, the set $A_{\langle g, c_g \rangle}$ is constraint-compact, where $A_{\langle g, c_g \rangle}$ is the set of all the answer constraints c', projected and renamed w.r.t. the arguments of g, s.t.*

c' is a successful derivation in the forest $\mathcal{F}(Q, c_q)$.

Proof. (Toman, 1997b) proves termination by observing that the SLG^C rewriting rules can be applied only finitely many times. We extend this proof to ensure that the TCLP forest generated is finite and therefore the program execution terminates.

1. The execution can only generate a finite number of literals because they are linearized (unifications take place in the constraints in the body) and the number of predicates in the program is finite.
2. The execution can only generate a finite number of TCLP forests $\tau_P(g, c_g)$ because the number of possible literals is finite (by point 1 the number of literals is finite) and for each literal g, the set C_g of its possible active constraint stores is constraint-compact. That means that, for every subset of active constraint stores $C \sqsubseteq C_g$, there exists a finite subset, $C_{fin} \subseteq C$ of possible more general calls, such that $\forall c \in C. \exists c' \in C_{fin}. c \sqsubseteq_{\mathcal{D}} c'$. Therefore, at some point every new call will be entailed by some previous generator. This is checked in Def. 2.12.3c.
3. The set of answers $Ans(g, c_g)$ (Def. 2.12.3g) is finite because the set A_{fin} of possible more general answer constraints is finite. Similar justification to that in point 2.
4. The number of children from a node resolved against clauses in P (Def. 2.12.3b and 2.12.3d) is finite. The number of clauses in P is finite.
5. The number of children from a node resolved by answer resolution (Def. 2.12.3c) is finite because, by point 3, the set of answer $Ans(g, c_g)$ is finite.

□

The intuition here is that for every subset C from the set of all possible constraint stores C_g that can be generated when evaluating a call to P, if there is a finite subset $C_{fin} \subseteq C$ that covers (i.e., is as general as) C, then, at some point, any call will be entailed by previous calls, thereby allowing its suspension to avoid loops. Similarly, for every subset A from the set of all possible answer constraints $A_{\langle g, c_g \rangle}$ that can be generated by a call, if there is a finite subset $A_{fin} \subseteq A$ that covers A, then, at some point, any answer will be entailed by a previous one, ensuring that the class of answers $Ans(g, c_g)$ which entail any other possible answer returned by the program is finite.[7]

Example 2.4.
The Herbrand domain (with constants and function symbols) and syntactic equality is not constraint-compact, and therefore termination of TCLP($\mathbb{H}$) programs is not guaranteed. However, in the case of programs which have only constants, the number of constraints that can be generated is finite, and therefore termination is ensured. Termination is also ensured (even with variant tabling) when a program can only generate terms with a

[7]Note that a finite answer set does not imply a finite domain for the answers: the set of answers `Ans(Q, `c_q`)={V > 5}` is finite, but the answer domain of `V` is infinite.

```
p(X) :-
    Y = f(X),
    p(Y).
p(a).
```

(a) Program which finishes under TCLP($\mathbb{H}$).

```
nat(X) :-
    X #= Y+1,
    nat(Y).
nat(0).
```

(b) Natural numbers in TCLP($\mathbb{Q}$).

```
nat(X) :-
    X #= Y+1,
    nat(Y).
nat(0).
nat(X) :- X #> 1000.
```

(c) Describing infinitely many numbers in TCLP($\mathbb{Q}$).

Figure 2.7: TCLP programs under $\mathbb{H}$ and $\mathbb{Q}$.

bounded depth. In this case, the number of distinct terms (and therefore of equality constraints) that can be generated is finite as well.

Example 2.5.
Fig. 2.7, left, shows a program which loops in Prolog and under *variant* tabling. The unification explicitly appear in the body to make them apparent in the constraint stores. Although CLP($\mathbb{H}$) is not constraint-compact, the constraints generated by that program under the query `?- p(X)` can make it finish. Let examine its behavior from two points of view:

Compactness of call/answer constraint stores The set of all the constraint stores generated for the predicate `p/1` under the query $\langle$`p(X), true`$\rangle$ is $C_{\texttt{p(V)}} = \{\texttt{true}, \texttt{V} = \texttt{f(X)}, \texttt{V} = \texttt{f(f(X))}, \ldots\}$.[8] It is constraint-compact because for every subset C there is a finite set, e.g. $C_{fin} = \{\texttt{true}\}$, that covers C. The set of all answer constraint for the query, $A_{\langle \texttt{p(V)},\, \texttt{true}\rangle} = \{\texttt{V} = \texttt{a}\}$, is also constraint-compact because it is finite. Since they both are constraint-compact the execution terminates.

Suspension due to entailment The first recursive call is $\langle$`p(`$\texttt{Y}_1$`),` $\texttt{Y}_1$`=f(X)`$\rangle$ and its projected and renamed constraint store is entailed by the initial store: `V=f(X)` $\sqsubseteq$ `true`. Therefore, TCLP evaluation suspends the recursive call, shifts execution to the second clause, and generates the answer `X=a`. This answer is given to the suspended recursive call, results in the inconsistent constraint store $\texttt{Y}_1\texttt{=f(X)} \wedge \texttt{Y}_1\texttt{=a}$, and the execution terminates.

Example 2.6.
Using the previous example (Fig. 2.7, left), under the query `?- p(a)`, the set of all the generated constraint stores is $C_{\texttt{p(V)}} = \{\texttt{V} = \texttt{a}, \texttt{V} = \texttt{f(a)}, \texttt{V} = \texttt{f(f(a))}, \ldots\}$. It is not constraint-compact and the execution does not terminate. Let us examine its behavior:

[8]The syntax $C_{\texttt{p(V)}}$ means that (i) we are projecting all the calls to predicate `p/1` on the variables that call, and (ii) we are renaming these variables to be `V` in all the calls. We could associate with every constraint store the names of the variables in the call in order to be able to compare different constraints stores (which is unnecessary after projection if there is only one variable in the call, but it would be needed if more than one variable is involved). In order to avoid such an overload, and without loss of generality, we preferred to project and rename to a unique set of variables.

The first recursive call is $\langle$`p(Y`$_1$`)`, `X=a` $\wedge$ `Y`$_1$`=f(X)`$\rangle$ and the projection of its constraint store, `Y`$_1$`=f(a)`, is not entailed by the initial one after renaming `V=f(a)` $\not\sqsubseteq$ `V=a`. Then this call is evaluated and produces the second recursive call, $\langle$`p(Y`$_2$`)`, `X=a` $\wedge$ `Y`$_1$`=f(X)` $\wedge$ `Y`$_2$`=f(f(X))`$\rangle$. Its projected constraint store, `Y`$_2$`=f(f(a))`, is not entailed by any of the previous constraint stores, and so on with the rest of the recursive calls. Therefore, the evaluation loops without terminating.

Example 2.7.
Fig. 2.7, center, shows a program which generates all the natural numbers using TCLP($\mathbb{Q}$). Although CLP($\mathbb{Q}$) is not constraint-compact, the constraint stores generated by that program for the query `?- X #< 10, nat(X)` are constraint-compact and the program finitely finishes. Let us look at its behavior from two points of view:

Compactness of call/answer constraint stores The set of all active constraint stores generated for the predicate codenat/1 under the query $\langle$`nat(X)`, `X < 10`$\rangle$ is $C_{\texttt{nat(V)}} = \{\texttt{V} < 10, \texttt{V} < 9, \ldots, \texttt{V} < -1, \texttt{V} < -2, \ldots\}$. It is constraint-compact because every subset $C \in C_{\texttt{nat(V)}}$ is covered by $C_{fin} = \{\texttt{V} < 10\}$. The set of all possible answer constraint for the query, $A_{\langle \texttt{nat(V)},\, \texttt{V < 10}\rangle} = \{\texttt{V} = 0, \ldots, \texttt{V} = 9\}$, is also constraint-compact because it is finite. Therefore, the program terminates.

Suspension due to entailment The first recursive call is $\langle$`nat(Y`$_1$`)`, `X < 10` $\wedge$ `Y`$_1$`=X+1`$\rangle$ and the projection of its constraint store after renaming is entailed by the initial one since `V < 9` $\sqsubseteq$ `V < 10`. Therefore, TCLP evaluation suspends in the recursive call, shifts execution to the second clause and generates the answer `X=0`. This answer is given to the recursive call, which was suspended, produces the constraint store `X < 10` $\wedge$ `Y`$_1$`=X+1` $\wedge$ `Y`$_1$`=0`, and generates the answer `X=1`. Each new answer `X`$_n$`=n` is used to feed the recursive call. When the answer `X=9` is given, it results in the (inconsistent) constraint store `X < 10` $\wedge$ `Y`$_1$`=X+1` $\wedge$ `Y`$_1$`=9` and the execution terminates.

Example 2.8.
The program in Fig. 2.7, center, does not terminate for the query `?- X #> 0, X #< 10, nat(X)`. Let us examine its behaviour:

The constraint stores are not compact The set of all constraint stores generated by the query $\langle$`nat(X)`, `X > 0` $\wedge$ `X < 10`$\rangle$ is $C_{\texttt{nat(V)}} = \{\texttt{V} > 0 \wedge \texttt{V} < 10, \texttt{V} > -1 \wedge \texttt{V} < 9, \ldots, \texttt{V} > -\texttt{n} \wedge \texttt{V} < (10-\texttt{n}), \ldots\}$, which it is not constraint-compact. Note that `V` is, in successive calls, restricted to a sliding interval `[k, k+10]` which starts at `k=0` and decreases `k` in each recursive call. No finite set of intervals can cover any subset of the possible intervals.

The evaluation loops The first recursive call is $\langle$`nat(Y`$_1$`)`, `X > 0` $\wedge$ `X < 10` $\wedge$ `Y`$_1$`=X+1`$\rangle$ and the projection of its constraint store is not entailed by the initial one after renaming since (`V > -1` $\wedge$ `V < 9`) $\not\sqsubseteq$ (`X > 0` $\wedge$ `X < 10`).

Then this call is evaluated and produces the second recursive call, $\langle$`nat(Y`$_2$`)`, `X > 0` $\wedge$ `X < 10` $\wedge$ `Y`$_1$`=X+1` $\wedge$ `Y`$_2$`=Y`$_1$`+1`$\rangle$. Again, the projection of its constraint store, `Y`$_2$ `> -2` $\wedge$ `Y`$_2$ `< 8`, is not entailed by any of the previous constraint stores, and so on. The evaluation therefore loops.

Example 2.9.
Let us examine again Fig. 2.7, center, with the query `?- nat(X).` Now, the set of all constraint stores generated by the query $\langle$`nat(X), true`$\rangle$ is $C_{\texttt{nat(V)}} = \{\texttt{true}\}$ which is finite and constraint-compact. However, the answer constraint set $A_{\langle\texttt{nat(V), true}\rangle} = \{\texttt{V} = \texttt{0}, \texttt{V} = \texttt{1}, \ldots, \texttt{V} = \texttt{n}, \ldots\}$ is not constraint-compact, and therefore, the program does not terminate.

Call suspension with an infinite answer constraint set The first recursive call is $\langle$`nat(Y`$_1$`), X = Y`$_1$`+1`$\rangle$ and the projection of its constraint store[9] is entailed by the initial store. Therefore, the TCLP evaluation suspends the recursive call, shifts execution to the second clause, and generates the answer `X=0`. This answer is used to feed the suspended recursive call, resulting in the constraint store `X=Y`$_1$`+1` $\wedge$ `Y`$_1$`=0` which generates the answer `X=1`. Each new answer `X=n` is used to feed the suspended recursive call. Since the projection of the constraint stores on the call variables is `true`, the execution tries to generate infinitely many natural numbers.

Example 2.10.
Unlike the situation that happens in pure Prolog/variant tabling, adding new clauses to a program under TCLP can make it terminate.[10] As an example, Fig. 2.7, right, is the same as Fig. 2.7, center, with the addition of the clause `nat(X) :- X #> 1000`. Let us examine its behavior under the query `?- nat(X)`:

Compactness of call/answer constraint stores The set of all constraint stores generated remains $C_{\texttt{nat(V)}} = \{\texttt{true}\}$. But the new clause makes the answer constraint set becomes $A_{\langle\texttt{nat(V), true}\rangle} = \{\texttt{V} = \texttt{0}, \texttt{V} = \texttt{1}, \ldots, \texttt{V} = \texttt{n}, \ldots, \texttt{V} > \texttt{1000}, \texttt{V} > \texttt{1001}, \ldots, \texttt{V} > \texttt{n}, \ldots\}$, which is constraint-compact because a constraint of the form `V > n` entails infinitely many constraints, i.e. it covers the infinite set `{V=n+1, ...,V > n+1,...}`. Therefore, since both sets are constraint-compact, the program terminates.

First search, then consume The first recursive call $\langle$`nat(Y`$_1$`), X = Y`$_1$`+1`$\rangle$ is suspended and the TCLP evaluation shifts to the second clause which generates the answer `X=0`. Then, instead of feeding the suspended call, the evaluation continues the search and shifts to the added clause, `nat(X) :- X #> 1000`, and generates the

[9] The equation in the body of the clause `X=Y`$_1$`+1` defines a relation between the variables but, since the domain of `X` is not restricted, its projection onto `Y`$_1$ returns no constraints (i.e., `Proj(Y`$_1$`, X=Y`$_1$`+1)= true`).

[10] This depends on the strategy used by the TCLP engine to resume suspended goals. An implementation that gathers all the answers for goals that can produce results first, and then these answers are used to feed suspended goals, makes the exploration of the forests proceed in a breadth-first fashion.

answer `X > 1000`. Since no more clauses remain to be explored, the answer `X=0` is used, generating `X=1`. Then `X > 1000` is used, resulting in the constraint store $\texttt{X=Y}_1\texttt{+1} \wedge \texttt{Y}_1 > \texttt{1000}$, which generates the answer `X > 1001`. However, during the answer entailment phase, `X > 1001` is discarded because $\texttt{X > 1001} \sqsubseteq \texttt{X > 1000}$. Then, one by one each answer `X=n` is used, generating `X=n+1`. But when the answer `X=1000` is used, the resulting answer `X=1001` is discarded, during the answer entailment phase, because $\texttt{X=1001} \sqsubseteq \texttt{X > 1000}$. At this point the evaluation terminates because there are no more answers to be consumed. The resulting set of answers is `Ans(nat(X),true) = {X=0, X > 1000, X=1, ...,X=1000}`.

2.6 The role of the Projection in TCLP

The detection of more particular calls and answers is performed during the call and answer entailment phases, respectively and perform entailment check against the projected constraint store of previous calls/answers. Allegedly due to performance issues and implementation issues, some frameworks (Cui and Warren, 2000; Schrijvers et al., 2008) did not implement precise projection. Given that in some cases approximate projections can be more efficient and/or considerably easier to implement, it is worth revisiting how relaxing projection impacts soundness, completeness, and termination. Three variants of projection can be distinguished:

Precise projection ($c \equiv c_s$) The projected constraint c_s is equivalent to the constraint store c. Any solution of c is also a solution of c_s, and a valuation v over S which is a solution of c_s is a partial solution of c.

Over-approximated projection ($c \sqsubseteq c_s$) The projected constraint c_s is more general than the constraint store c. Any solution of c is also a (partial) solution of c_s.

Under-approximated projection ($c \sqsupseteq c_s$) The projected constraint c_s is more particular than the constraint store c. A valuation v over S which is a solution of c_s is a (partial) solution of c.

We use s, t to denote the constraint store before a call; s', s'', t' to denote the constraint store after the execution of a call (i.e., the constraint answer); s_p, s'_p, s''_p, t'_p to denote the projection of its respective constraint store/answer; and $\tilde{s}$, $\tilde{t}$ to denote the constraint answer obtained from its generators by answer resolution. The execution of a generator g is denoted by $\{s_p\}g\{s''\}\{s''_p\}$ which means that the execution of g with the constraint store s_p generates the constraint store s'' which is projected to obtain s''_p. The answer resolution of the generator and its consumers are denoted by $\{s\}g\{\tilde{s}\}$ and $\{t\}c\{\tilde{t}\}$, where $\tilde{s} = s''_p \wedge s$ and $\tilde{t} = s''_p \wedge t$. The resolution against the clauses of the generator and its consumers are denoted by $\{s\}g\{s'\}\{s'_p\}$ and $\{t\}c\{t'\}\{t'_p\}$.

Table 2.2: Soundness and completeness comparison of precise, over- and under-approximation (‘$\equiv$’, ‘$\sqsubseteq$’ and ‘$\sqsupseteq$’) for the entailment phases.

Call Answer	$\equiv$	$\sqsubseteq$	$\sqsupseteq$
$\equiv$	$s' \equiv s'_p \equiv \tilde{s}$ $t' \equiv t'_p \equiv \tilde{t}$	$s' \equiv s'_p \equiv \tilde{s}$ $t' \equiv t'_p \equiv \tilde{t}$	$s' \equiv s'_p \sqsupseteq \tilde{s}$ $t' \equiv t'_p \equiv \tilde{t}$
$\sqsupseteq$	$s' \sqsupseteq s'_p \equiv \tilde{s}$ $t' \sqsupseteq t'_p \equiv \tilde{t}$	$s' \sqsupseteq s'_p \equiv \tilde{s}$ $t' \sqsupseteq t'_p \equiv \tilde{t}$	$s' \sqsupseteq s'_p \sqsupseteq \tilde{s}$ $t' \sqsupseteq t'_p \equiv \tilde{t}$
$\sqsubseteq$ $\begin{cases} t \sqsupseteq s''_p \\ t \not\sqsupseteq s''_p \end{cases}$	$s' \sqsubseteq s'_p \equiv \tilde{s}$ $\begin{cases} t' \sqsubseteq t'_p \equiv \tilde{t} \\ t' \sqsubseteq t'_p \sqsubseteq \tilde{t} \end{cases}$	$s' \sqsubseteq s'_p \equiv \tilde{s}$ $\begin{cases} t' \sqsubseteq t'_p \equiv \tilde{t} \\ t' \sqsubseteq t'_p \sqsubseteq \tilde{t} \end{cases}$	$s' \sqsubseteq s'_p \sqsupseteq \tilde{s}$ $\begin{cases} t' \sqsubseteq t'_p \equiv \tilde{t} \\ t' \sqsubseteq t'_p \sqsubseteq \tilde{t} \end{cases}$

To check soundness and completeness, we compare s', t' (the answer constraint obtained by resolution against the clauses), versus s'_p, t'_p (the projection of s', t'), and versus $\tilde{s}$, $\tilde{t}$ (the answer constraint obtained by answer resolution). Table 2.2 summarizes the results of these comparisons depending on which projection is used in the call/answer entailment phase. For brevity, we comment only three of them:

$\equiv / \equiv$ Precise projection ‘$\equiv$’ in the call and answer entailment phase. This is the optimal option (used in all the interfaces presented in Chapter 3) because it guarantees soundness and completeness ($s' \equiv s'_p \equiv \tilde{s}$ and $t' \equiv t'_p \equiv \tilde{t}$) and it does not enlarge the search space.

$\sqsubseteq / \equiv$ Over-approximate projection ‘$\sqsubseteq$’ for the calls and precise projection ‘$\equiv$’ for the answers. Generators would be then bound to produce more possible answers with a relaxed constraint store, which can turn terminating queries into non-terminating ones. In the case of termination, it may produce more answers than necessary given the initial generator constraints.

Example 2.11.
Call abstraction (Schrijvers et al., 2008) is an extreme example, where the constraint store associated to the tabled call is not taken into account to execute it (i.e., the projection of a constraint store is always the constraint `true`, note that $c \sqsubseteq true$ for any constraint c). As we mentioned before, this loses many benefits of tabling with

constraints since it has to compute all the possible results for an unrestricted call and then filter them through the call-time constraint store.

$\sqsubseteq$ / $\sqsubseteq$ Over-approximate projection '$\sqsubseteq$' for calls and answers. From our point of view this option is relevant because applications such as abstract interpretation explicitly over-approximate the results in order to ensure termination and completeness w.r.t. to the real program execution. The over-approximate projection makes the result obtained applying answer resolution to a generator sound w.r.t. the result obtained repeating the execution, i.e., $s'_p \equiv s''_p$. Applying answer resolution to a consumer, whose constraint store before the call is more general that the answer constraint of its generator, $t \sqsupseteq s''_p$, is also sound w.r.t. the result obtained executing the consumer, i.e., $t'_p \equiv \tilde{t}$. However, in cases where the constraint store of consumer is not more general than the answer constraint of its generator, applying answer resolution is not sound w.r.t. the results obtained execution the consumer, because extraneous answers may have been introduced by the over-approximate projection of the answer constraint of the generator, i.e., $t'_p \sqsubseteq \tilde{t}$. As a result, accepting loss of precision, the over-approximate projection for answers may increase performance because a more general answer would entail more number of answers, without losing completeness.

2.7 Discussion

We have extended the theoretical basis of tabled constraint logic programming for a top-down execution. We have characterized the properties that the constraint solver should holds in order to guarantee soundness and completeness. Moreover, for non constraint-compact constraint solvers we define the condition of programs/queries to ensure termination.

Additionally, for constraint domain without a precise implementation of the projection of constraint stores, we evaluate how relaxing the projection impacts soundness, completeness and termination.

Chapter 3

Design and Implementation of Mod TCLP

Based on the operational semantics of TCLP presented in the previous chapter, we designed and implemented Mod TCLP, a modular framework implemented in Ciao Prolog, that facilitates the integration of generic constraint solvers (even written in C) into the tabling engine. We validated its flexibility and performance by integrating several constraint solvers, and we evaluated the benefits of a new (more complex) answer management strategy that not only discards more particular answers but also removes them from the answer store.

As we mentioned in Section 2.1, the combination of CLP and tabling brings several advantages: it enhances termination properties, increases speed in a range of programs, and provides additional expressiveness. It has been applied in several areas, including constraint databases (Kanellakis et al., 1995; Toman, 1997b), verification of timed automata and infinite systems (Charatonik et al., 2002), and abstract interpretation (Toman, 1997a).

The theoretical basis of TCLP (Toman, 1997b) establishes the basic operations (projection and entailment checking) that are necessary to ensure completeness w.r.t. the declarative semantics. However, previous implementation, such as XSB (Cui and Warren, 2000) and TCHR (Schrijvers et al., 2008), did not fully use these two operations, likely due to performance issues and also to implementation difficulty; and from the point of view of interfacing / adding additional CLP solvers to existing systems:

- The framework in (Cui and Warren, 2000) requires the constraint solver to provide the `projection/1` and `entail/2` predicates, which are used to discard more particular answers, but only in one direction. It also requires the implementation of the predicate `abstract/3`, which has to take care of the call abstraction. However, it is not clear if this predicate is part of the constraint solver or of the user program.

- The TCHR framework described in (Schrijvers et al., 2008) provides interesting hooks: `projection(PredName)` specifies that predicate `PredName/1` determines how projection is to be performed, which makes it possible to, for example, ignore arguments; `canonical_form(PredName)` modifies the answer store to a canonical form as defined by `PredName/2`, so that identical answers can be detected (e.g. using `sort/2` the constraints `[leq(1,X),leq(X,3)]` and `[leq(X,3),leq(1,X)]` are reduced to the same canonical form); and `answer_combination(PredName)`, if specified, applies `PredName/3` in such a way that two answers can be merged into one.

In this chapter we present the design and implementation of a generic framework based on a previous Tabled TCLP framework (Chico de Guzmán et al., 2012) that features a complete treatment of constraint projection and entailment. In this generic framework (termed Mod TCLP), we provide a richer, more flexible answer management mechanism and design a tabling implementation so that it can use the projection and entailment operations provided by a constraint solver presented to the tabling engine as a *server*. To facilitate the integration, we define a set of operations that the constraint solver has to provide to the tabling engine. These operations are natural to the constraint solver, and when they are not already present, the implementation of the solver could be easily extended to provide them.

We have validated the expressiveness and efficiency of its implementation in Ciao Prolog (Hermenegildo et al., 2012) by interfacing four non-trivial constraint solvers. As a result, we provide four different TCLP system instances that we have experimentally evaluated with several benchmarks against tabling, CLP, and Prolog. Additional performance comparisons between them are also provided.

3.1 The Mod TCLP Framework

In this section we describe the Mod TCLP framework, the operations required by the interface, and the program transformation that we use to compile programs with tabled constraints (Section 3.1.1). We also provide a sketch of its implementation and we describe step-by-step some executions at the level of the TCLP libraries (Sections 3.1.2 and 3.1.3). In Section 3.1.4 we present the implementation of the TCLP interface for Holzbaur's CLP($\mathbb{Q}$) solver and in Section 3.1.5 we present an optimization, the *Two-Step* projection.

`store_projection(+Vars,-ProjStore)` Returns in `ProjStore` a representation of the projection of the current constraint store onto the list of variables `Vars`.

`call_entail(+ProjStore,+ProjStore`$_{gen}$`)` Succeeds if the projection of the current constraint store, `ProjStore`, entails the projected store, `ProjStore`$_{gen}$, of a previous generator. It fails otherwise.

`answer_compare(+ProjStore,+ProjStore`$_{ans}$`,-Res)` Returns `Res='=<'` if the projected store of the current answer, `ProjStore`, entails the projected store of a previous answer, `ProjStore`$_{ans}$, or `Res='>'` if `ProjStore` is entailed by `ProjStore`$_{ans}$ and they are not equal. It fails otherwise.

`apply_answer(+Vars,+ProjStore)` Adds the projected constraint store `ProjStore` of the answer to the current constraint store and succeeds if the resulting constraint store is consistent.

Figure 3.1: Generic interface specification.

3.1.1 Design of the Generic Interface

Mod TCLP provides a generic interface (Fig. 3.1) designed to facilitate the integration of different constraint solvers. The predicates of the interface use extensively two objects: `Vars`, the list of constrained variables, provided by the tabling engine to the constraint solver, and `ProjStore`, a representation of the projected constraint store, opaque to the tabling engine, and which should be self-contained and independent (e.g., with fresh variables) from the *main* constraint store. For example, the constraint solver CLP($\mathbb{D}_{\leq}$) (Section 3.2.1) is written in C and the projection of a constraint store is a C structure whose representation is its memory address and length.

To implement these predicates, the constraint solver has to support the (minimal) set of operations defined in Section 2.2.1: projection, test for entailment, and test for consistency. The predicates that the constraint solver must provide in order to make its interaction with the tabling engine possible are:

- `store_projection(+Vars,-ProjStore)`, that is invoked before the call and the answer entailment phases:
  - It is used before the call entailment phase to generate the representation of the goal as a tuple $\langle$`G`, `ProjStore`$\rangle$, where `ProjStore` represents the projection of the constraint store at the moment of the call onto `Vars`, the variables in `G`. Although a generic implementation should include the Herbrand constraints of the call in the constraint store, our implementation does not consider Herbrand constraints to be part of the constraint store by default. Instead, calls are syntactically compared using variant checking, but

the programmer can also choose to use subsumption, if required, by using a package described below. There are some reasons for that decision: on the one hand, programmers (even using tabling) are used to this behavior; on the other hand, there are data structures highly optimized (Ramakrishnan et al., 1995) to save and retrieve calls together with their input / output substitutions which perform variant checking on the fly while taking advantage of the WAM-level representation of substitutions.

– Similarly, before the answer entailment phase, the projection of the Herbrand constraints onto the variables of the goal is directly taken care of by their WAM-level representation. We use variant checking to detect when the Herbrand constraints associated to two calls are equal. Therefore, an answer constraint is internally represented by a tuple $\langle \mathtt{S}, \mathtt{ProjStore} \rangle$ where `S` captures the Herbrand constraints of the variables of the goal and `ProjStore` represents the projection of the rest of the answer constraint onto `Vars`, the variables of the answer.

- $\mathtt{call\_entail(+ProjStore,+ProjStore}_{gen}\mathtt{)}$ is invoked during the call entailment phase to check if a new call, represented by $\langle \mathtt{G}, \mathtt{ProjStore} \rangle$, entails a previous generator, represented by $\langle \mathtt{G}_{gen}, \mathtt{ProjStore}_{gen} \rangle$, where `G` is a variant of $\mathtt{G}_{gen}$. The predicate succeeds if $\mathtt{ProjStore} \sqsubseteq \mathtt{ProjStore}_{gen}$, and fails otherwise. If Herbrand subsumption checking is needed, our implementation provides a package which transforms calls to tabled predicates so that suspension is based on entailment in $\mathbb{H}$. This transformation moves Herbrand constraint handling away from the level of the WAM by creating attributed variables (Cui and Warren, 2000) that carry the constraints — i.e., the unifications. Later on, a Herbrand constraint solver is used to check subsumption.[1]

- $\mathtt{answer\_compare(+ProjStore,+ProjStore}_{ans}\mathtt{,-Res)}$ is invoked during the answer entailment phase to check a new answer, represented by $\langle \mathtt{S}, \mathtt{ProjStore} \rangle$, against a previous one, represented by $\langle \mathtt{S}_{ans}, \mathtt{ProjStore}_{ans} \rangle$, when the Herbrand constraints `S` and $\mathtt{S}_{ans}$ are equal. The predicate compares `ProjStore` and $\mathtt{ProjStore}_{ans}$ and returns `'=<'` in its last argument when $\mathtt{ProjStore} \sqsubseteq \mathtt{ProjStore}_{ans}$, `'>'` when $\mathtt{ProjStore} \sqsupset \mathtt{ProjStore}_{ans}$, and fails otherwise. This bidirectional entailment check, which is used to discard / remove more particular answers, is a potentially costly process, but it brings considerable advantage from saved resumptions (Section 3.3.3): when an answer is added to a generator, consumers are resumed by that answer. These consumers in turn generate more answers and cause further resumptions in cascade. Reducing the number of redundant answers reduces the number of redundant resumptions, and we have experimentally observed that it results in important savings in execution time.

[1] If there are several constraint domains involved, such as e.g. CLP($\mathbb{H}$) and CLP($\mathbb{Q}$), we assume that we can distinguish them appropriately at run-time and the entailment is determined as $\mathtt{variant(G,G_{gen})} \wedge \mathtt{ProjStore} \sqsubseteq_{\mathbb{H}} \mathtt{ProjStore_{gen}} \wedge \mathtt{ProjStore} \sqsubseteq_{\mathbb{Q}} \mathtt{ProjStore_{gen}}$

- `apply_answer(+Vars,+ProjStore)` is invoked to consume an answer from a generator. In variant tabling, since consumers are variants of generators, answer substitutions from generators can always be applied to consumers. That is not the case when using entailment in TCLP: consumers may be called in the realm of a constraint store more restrictive than that of their generators, and answers from the generator have to be filtered to discard those which are inconsistent with the constraint store at the time of the call to the consumer. In our implementation, an answer is represented by ⟨`S`, `ProjStore`⟩, where `S`, the set of Herbrand constraints, is applied by the tabling engine and `ProjStore`, the projection of the constraint answer, is added to the constraint store of the consumer by `apply_answer/2`, which succeeds iff the resulting constraint store is consistent.

The design of the interface assumes that external constraint solvers are compatible with Prolog operational semantics so that when Prolog backtracks to a previous state, the corresponding constraint store is transparently restored. That can be done by adding a Prolog layer which uses the trail to store `undo` information that is used to reconstruct the previous constraint store when Prolog backtracks (this is a reasonable, minimal assumption for any integration of constraint solving and logic programming). The TCLP interface can use any constraint solver which follows this design, because the suspension and resumption mechanisms of the tabling are based on the trailing mechanism of Prolog. When a consumer suspends, backtracking takes place, the memory stacks are frozen, and the variable bindings are saved on untrailing. They are reinstalled upon consumer resumption.

However, the entailment operations `call_entail/2` and `answer_compare/3` need to know the correspondence among variables in `ProjStore` and in $\texttt{ProjStore}_{gen}$ (resp., $\texttt{ProjStore}_{ans}$). To this end, projections are (conceptually) a pair `(VarList, Store)`, where `VarList` is a list of fresh variables in `Store` that correspond to `Vars`, the variables on which the projection was originally made. Different, independent constraint stores can then be compared by means of these lists. This list is also necessary to apply the `ProjStore` of an answer to the global store: it is used to determine the correspondence of variables between the global and the projected store.

The actual implementation may differ among constraint solvers. For example, the TCLP($\mathbb{Q}$) interface (Fig. 3.6) uses a list of fresh variables following the same order as those in `Vars`, while the TCLP($\mathbb{D}_{\leq}$) interface (Section 3.2.1) uses a vector containing the index in the matrix corresponding to every variable in `Vars`, again following the same order.

3.1.2 Implementation Sketch

We summarily describe now the implementation of Mod TCLP, including the global table where generators, consumers, and answers are saved. We also present the transformation performed to execute tabled predicates and a (simplified) flowchart showing the interactions between the tabling engine and the constraint solver through the generic interface.

3.1.2.1 Global Table

Tries are the data structure of choice for the call / answer global table (Ramakrishnan et al., 1995). In variant tabling, every generator $\texttt{G}_{gen}$ is uniquely associated (modulo variable renaming) to a leaf from where the Herbrand constraints for every answer hangs. Generators are identified in Mod TCLP by the projection of the constraint store on the variables of the generator, i.e., with a tuple $\langle \texttt{G}_{gen}, \texttt{ProjStore}_{gen} \rangle$. We store generators in a trie where each leaf is associated to a call pattern $\texttt{G}_{gen}$ and a list with a frame for each projected constraint store $\texttt{ProjStore}_{gen_i}$. Each frame identifies: (i) the projected constraint store $\texttt{ProjStore}_{gen_i}$, (ii) the answer table where the generator's answers Ans($\texttt{G}_{gen}$, $\texttt{ProjStore}_{gen_i}$) are stored, and (iii) the list of its consumers.

Answers are represented by a tuple $\langle \texttt{S}_{ans}, \texttt{ProjStore}_{ans} \rangle$ and are stored in a trie where each leaf points to the Herbrand constraints $\texttt{S}_{ans}$ and to a list with the projected constraint stores $\texttt{ProjStore}_{ans_i}$ corresponding to answers whose Herbrand constraints are a variant of $\texttt{S}_{ans}$. The answers are stored in order of generation, since (as we mentioned before) it is not clear that other orders eventually pay off in terms of speeding up the entailment check of future answers.

3.1.2.2 TCLP Directives and Program Transformation

Executing a CLP program under the TCLP framework only needs to enable tabling and import a package that implements the bridge CLP / tabling instead of the regular constraint solver. Fig. 3.2a, shows the TCLP version of the left recursive distance traversal program in Fig. 2.1, right. The constraint interface remains unchanged, and the program code does not need to be modified to be executed under TCLP. The directive `:- use_package(tabling)` initializes the tabling engine and the directive `:- use_package(t_clpq)` imports TCLP($\mathbb{Q}$), the TCLP interface for the CLP($\mathbb{Q}$) solver (Section 3.1.4). To select another TCLP interface (more examples in Section 3.2) we just have to import the corresponding package. Finally, the directive `:- table dist/3` specifies that the predicate `dist/3` should be tabled.

Fig. 3.2b, shows the transformation applied to the program `dist/3` which we used in Section 2.1 and we reproduce in Fig. 3.2a. The original entry point to the predicate

```
:- use_package(tabling).
:- use_package(t_clpq).

:- table dist/3.
dist(X, Y, D) :-
    D1 #> 0, D2 #> 0,
    D  #= D1 + D2,
    dist(X, Z, D1),
    edge(Z, Y, D2).
dist(X, Y, D) :-
    edge(X, Y, D).
```

(a) TCLP version (with directives)

```
dist(A, B, C) :-
    tabled_call(dist_aux(A,B,C)).

dist_aux(X, Y, D) :-
    D1 #> 0, D2 #> 0,
    D  #= D1 + D2,
    dist(X, Z, D1),
    edge(Z, Y, D2),
    new_answer.
dist_aux(X, Y, D) :-
    edge(X, Y, D),
    new_answer.
```

(b) Program transformation

Figure 3.2: TCLP version of dist/3 and its transformation.

is rewritten to call an auxiliary predicate through the meta-predicate tabled_call/1. The auxiliary predicate corresponds to the original one with a renamed head and with an additional new_answer/0 at the end of the body to collect the answers. An internal global stack, called PTCP, is used to identify the generator under execution when new_answer/0 is invoked.

3.1.2.3 Execution Flow

Fig. 3.3 shows a (simplified) flowchart to illustrate how the execution of a tabled call proceeds. The calls to predicates in the interface with the constraint solver have a grey background. We explain next the steps of an execution, using the labels in the nodes.

0. A call to a tabled predicate Call starts the tabled execution invoking tabled_call/1 (Fig. 3.4), which takes the control of the execution.

1. call_lookup_table/3 returns in Gen a reference to the trie leaf corresponding to the current call pattern Call and in Vars a list with the constrained variables of Call.

2. The tabling engine calls store_projection/2, which returns in ProjStore the projection onto Vars of the current constraint store.

3. The tabling engine uses member/2 to retrieve in ProjStore_G the projected constraint stores from the list of frames associated to Gen. If it succeeds, the execution continues in step 5. If it fails, it may be because Gen is the first occurrence of this call pattern, or because it does not entail any of the previous generators (and it is therefore a new generator).

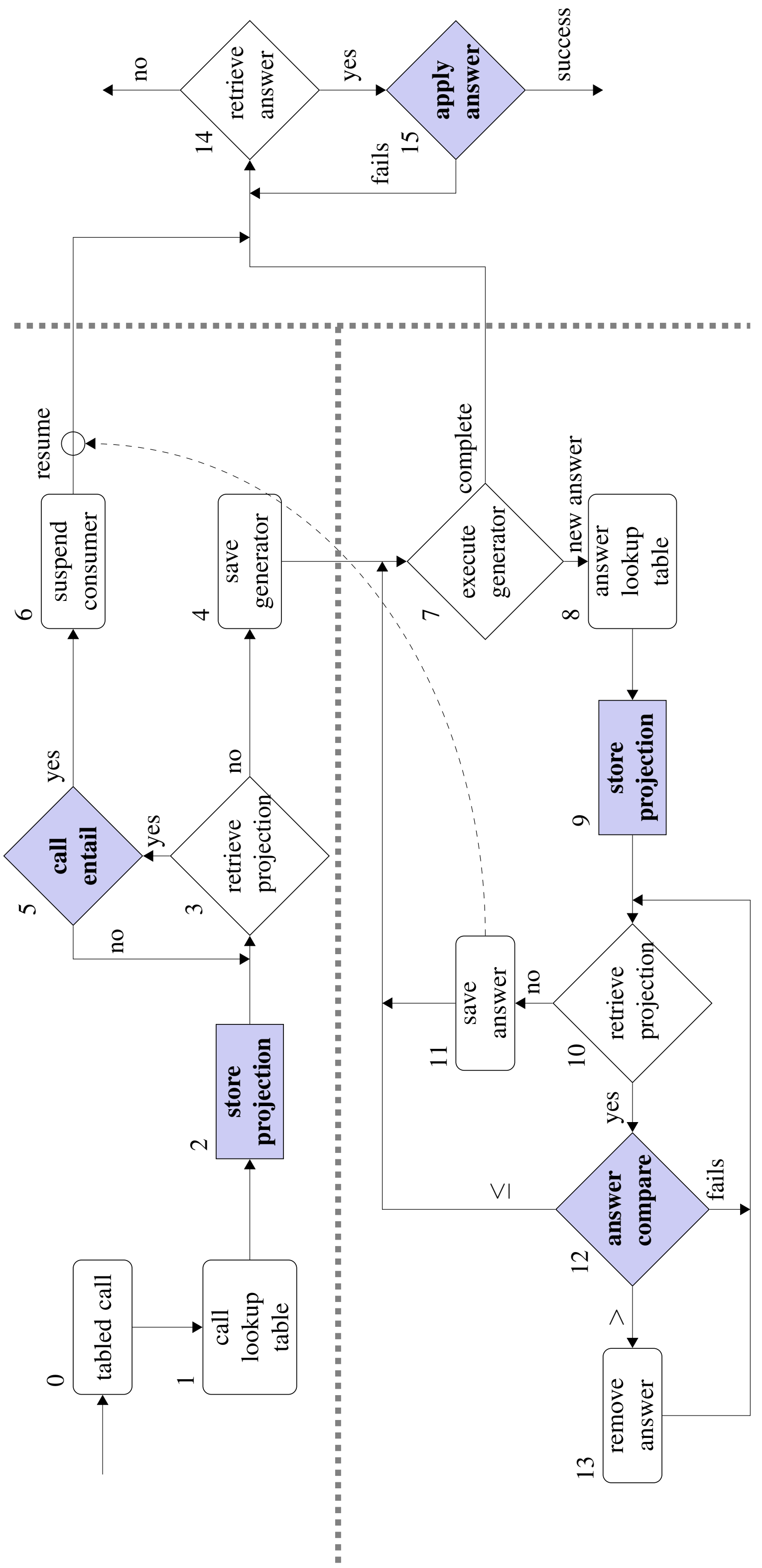


Figure 3.3: Flowchart of the execution algorithm of Mod TCLP.

```
tabled_call(Call) :-
    call_lookup_table(Call, Vars, Gen),
    store_projection(Vars, ProjStore),
    (
       projstore_Gs(Gen, List_GenProjStore),
       member(ProjStore_G, List_GenProjStore),
       call_entail(ProjStore, ProjStore_G) ->
       suspend_consumer(Call)
    ;
       save_generator(Gen, ProjStore_G, ProjStore),
       execute_generator(Gen, ProjStore_G),
    ),
    answers(Gen, ProjStore_G, List_Ans),
    member(Ans, List_Ans),
    projstore_As(Ans, List_AnsProjStore),
    member(ProjStore_A, List_AnsProjStore),
    apply_answer(Vars, ProjStore_A).
```

Figure 3.4: Implementation of `tabled_call/1`.

4. The tabling engine calls `save_generator/3` to add a new frame to `Gen`, identifying the new call as a generator. The projected store `ProjStore` is saved in this new frame and the answer table and the consumer list are initialized. From this point on, the generator is identified by ⟨`Gen`, `ProjStore_G`⟩ and the execution continues in step 7.

5. The constraint solver checks if the current store `ProjStore` entails the retrieved projected constraint store `ProjStore_G` using `call_entail/2`. In that case, `Call` is suspended in step 6. Otherwise, the tabling engine tries to retrieve another projected constraint store in step 3.

6. If the generator is not complete, the tabling engine suspends the execution of `Call` with `suspend_consumer/1` and adds `Call` to the list of consumers of the generator. Execution then continues by backtracking over the youngest generator. Otherwise, `Call` continues the execution in step 14. A suspended consumer is resumed when its generator produces new answers, and also continues in step 14.

7. The generator ⟨`Gen`, `ProjStore_G`⟩ is executed with `execute_generator/2`, which calls the renamed tabled predicate, and its reference is pushed onto the PTCP stack. If the execution reaches the end of a clause, a new answer has been found and `new_answer/0` continues the execution in step 8.

8. This is the entry point for `new_answer/0` (Fig. 3.5). The tabling engine calls `answer_lookup_table/2`, which retrieves a reference to the generator in execution from the PTCP stack. A reference to the Herbrand constraints of the current answer in the generator's answer table is returned in `Ans`, and the list of variables from the call that are now/still constrained is returned in `Vars`.

```
new_answer :-
    answer_lookup_table(Vars, Ans),
    store_projection(Vars, ProjStore),
    (
      projstore_As(Ans, List_AnsProjStore),
      member(ProjStore_A, List_AnsProjStore),
      answer_compare(ProjStore, ProjStore_A, Res),
      (
          Res == '=<'
       ;
          Res ==  '>',
          remove_answer(ProjStore_A),
          fail
      ), !
    ;
      save_answer(Ans, ProjStore)
    ), !,
    fail.

new_answer :-
    complete.
```

Figure 3.5: Implementation of `new_answer/0`.

9. The tabling engine invokes `store_projection/2`. This returns in `ProjStore` the projection of the current constraint store onto the constrained variables of the answer, `Vars`.

10. The tabling engine retrieves from `Ans` the list of projected constraint stores in `List_AnsProjStore` and calls `member/2` to return the stores one at a time in `ProjStore_A`. If it succeeds, the execution continues in step 12; otherwise, it continues in step 11. Failure can happen because all projected constraint stores were already retrieved from `List_AnsProjStore` or because `Ans` is the first answer with these Herbrand constraints.

11. The tabling engine adds `ProjStore` to the list of projected constraint stores (`List_AnsProjStore`) of the corresponding `Ans` with `save_answer/2`, and resumes one by one the consumers of the current generator which were suspended in step 6. Since `new_answer/0` always fails, the execution backtracks to complete the execution of the generator (step 7).

12. The constraint solver checks if the current store `ProjStore` entails the retrieved projected constraint store `ProjStore_A` using `answer_compare/3`. If this is the case, it returns `Res='=<'`, which makes `new_answer/0` discard the current answer, and the generator is re-executed in step 7. If `ProjStore` is entailed by `ProjStore_A` and they are not equal, it returns `Res='>'` and `ProjStore_A` is removed in step 13.

Otherwise, it fails and the execution continues in step 10, where the tabling engine tries to retrieve another projected constraint store.

13. The tabling engine marks the more particular answer as removed using `remove_answer/1`. Then the execution continues in step 10.

14. Once the generator has exhausted all the answers and does not have more dependencies, it is marked as complete using `complete/0` and the generator's reference is popped from the PTCP stack. The tabling engine retrieves answers ⟨`Ans`, `ProjStore_A`⟩ from the generator ⟨`Gen`, `ProjStore_G`⟩ using `member/2`. If it succeeds and the answer is not marked as removed, the answer will be applied in step 15. Otherwise, the execution backtracks to retrieve another answer.

15. Applying the Herbrand constraints `Ans` always succeeds, because the generator and its consumers have the same call pattern. Then the constraint solver adds the projected constraint store of the answer to the current constraint store with `apply_answer/2`, and checks if the resulting constraint store is consistent. If so, execution continues; otherwise the execution goes back to step 14.

In Section 3.1.3, we examine step by step the execution of a program using the TCLP($\mathbb{Q}$) interface described in Section 3.1.4.

3.1.3 Step by Step Execution of `dist/3` under TCLP($\mathbb{Q}$)

The trace below shows the step by step execution of the TCLP version of the left recursive distance traversal program in Fig. 3.2a, with the query `?- D #< 150, dist(a, Y, D)` using the graph in Fig. 2.3. In this example we are using the TCLP($\mathbb{Q}$) interface (Section 3.1.4). Each step is annotated with the labels used in Fig. 3.3. The execution starts with the query `?- D #< 150, dist(a, Y, D)`:

0 the constraint `D #< 150` in the query is added to the current store (state `s0`). Then ⟨`dist(a, Y, D)`, `D < 150`⟩ is called and the tabling engine takes the control of the execution calling `tabled_call(dist_aux(a,Y,D))`.

1 `call_lookup_table/3` initializes and saves (after renaming) `dist_aux(a,V0, V1)`, because it is the first occurrence, and returns `Vars=[D]` and `Gen=$1`, where `$1` is the reference for this generator.

2 `store_projection([D],ProjStore)` returns `ProjStore=([V1],[V1 #< 150])`.

3 `member/2` fails because the list of projected constraint stores associated to `Gen=$1` is empty.

4 `save_generator/3` saves `([V1], [V1 #< 150])` in the list of projected constraint stores associated to `Gen=$1` (state `s1`).

7 `execute_generator/2` evaluates the generator against the first clause of `dist_aux/3` and adds the body of the clause to the resolvent of the state `s2i`. Then the constraints of the resolvent, `[D1 #> 0, D2 #> 0, D #= D1+D2]`, are added to the constraint store (state s3) and $\langle$`dist(a, Z, D1)`, `D < 150` $\wedge$ `D1 > 0` $\wedge$ `D2 > 0` $\wedge$ `D=D1+D2`$\rangle$ is called.

0 the tabling engine reenters the tabled execution with `tabled_call(dist_aux(a,Z, D1))`.

1 `call_lookup_table(dist_aux(a,Z,D1),Vars,Gen)` returns `Vars=[D1]` and `Gen=$1`, the reference to the previous generator, `dist_aux(a,V0, V1)`.

2 `store_projection([D1],ProjStore)` returns `ProjStore=([V1], [V1 #> 0, V1 #< 150])`. For clarification, the projection of the current constraint store `D < 150` $\wedge$ `D1 > 0` $\wedge$ `D2 > 0` $\wedge$ `D=D1+D2` onto `D1` is `D1 > 0` $\wedge$ `D1 < 150`.

3 `member/2` retrieves the projected constraint store `ProjStore_G=([V1], [V1 #< 150])`.

5 `call_entail/2` succeeds because (`D < 150` $\wedge$ `D1 > 0` $\wedge$ `D2 > 0` $\wedge$ `D=D1+D2`) $\sqsubseteq$ `D1 < 150`.

6 `suspend_consumer/1` suspends the current call `dist_aux(a,Z, D1)` (state s3), waiting for the answer of the current TCLP tree, Ans(`s1`).

7 The evaluation of the generator backtracks to evaluate the other clause (state s2ii). Now the current constraint store is `D #< 150` and the call $\langle$`edge(a, Y, D)`, `D < 150`$\rangle$ unifies with `edge(a, b, 50)` (state s4). The first answer is found and `new_answer/0` is invoked to collect the answer.

8 `answer_lookup_table/2` stores the Herbrand constraints[2] of the answer, `Y=b` $\wedge$ `D=50`, returning `Vars=[]` and `Ans=$a1`, where `$a1` is the reference for this answer.

9 `store_projection/2` returns `ProjStore=([], [])`.

10 `member/2` fails because the list of projected constraint stores associated to `$a1` is empty.

11 `save_answer/2` saves `([], [])` in the list of the answer constraints associated to `$a1` (state `a1`). The first answer is collected.

[2]In solvers written in Prolog and implemented using attributed variables, such as CLP($\mathbb{Q}$) and CLP($\mathbb{R}$), it is usual that variables lose their association with the constraints where they appeared when these variables become ground. As ground terms do not have attributes attached, `D=50` is handled as part of the Herbrand constraints.

14 the tabling engine resumes the goal suspended at state s3 and `member/2` retrieves the Herbrand constraints `Y=b` $\wedge$ `D=50` and the answer constraint `([], [])`.

15 `apply_answer/2` adds the answer to the current constraint store (state s5).

7 the execution continues resolving $\langle$`edge(b, Y, D2), ...` $\wedge$ `D2 < 100`$\rangle$ which unifies with the clause `edge(b, a, D2) :- D2 #> 25, D2 #< 35` (state s6). The second answer is found.

8 `answer_lookup_table/2` stores the Herbrand constraints of the answer, `V0=a`, returning `Vars=[D]` and `Ans=$a2`.

9 `store_projection/2` returns `ProjStore=([V1],[V1 #> 75,V1 #< 85])`.

10 `member/2` fails because the list of projected constraint stores associated to `Ans=$a2` is empty.

11 `save_answer/2` saves `([V1], [V1 #> 75, V1 #< 85])` in the list of answer constraints associated to `$a2` (state a2). The second answer is collected.

14, 15, 7 the tabling engine resumes the suspended goal at state s3 and consumes the second answer following the same steps as with the first one and generating the states s7 and s8. The third answer has been found.

8, 9, 10, 11 the answer is collected and `([V1], [V1 #> 125, V1 #< 135])` is saved in the list of answer constraint associated to `$a3` (state a3).

14, 15 the tabling engine resumes the suspended goal at state s3 and consumes the third answer.

7 the execution fails resolving $\langle$`edge(b, V0, D2`$_1$`), V1 < 150` $\wedge$ `...` $\wedge$ `D1`$_1$` < 135`$\rangle$ (states s9 and s10)

14, 15 the generator has exhausted all the answers and it does not have any more dependencies, so `complete/0` marks the generator as complete. The query retrieves the answers from the generator one by one and returns them.

3.1.4 Implementation of the TCLP($\mathbb{Q}$) Interface

Fig. 3.6 shows the interface for Holzbaur's CLP($\mathbb{Q}$) solver (Holzbaur, 1995) as an example of integration of a constraint solver with Mod TCLP. This CLP($\mathbb{Q}$) implementation already provides most of the functionality required by the tabling engine, and therefore the TCLP($\mathbb{Q}$) interface actually acts as a bridge to existing predicates.

A Mod TCLP constraint interface starts with the declaration `:- active_tclp`. It makes the compiler adjust the program transformation according to the available interface predicates and instructs the tabling engine to activate the TCLP framework. The functionality required by the interface is implemented as follows:

- `store_projection(+Vars,-st(F,Proj))` calls the CLP($\mathbb{Q}$) predicate `clpqr_dump_constraints(+Vars,-F,-Proj)` to perform the projection. It returns in `Proj` the projection of the current store onto the list of variables `Vars`. The variables in `Proj` are fresh and are contained in the list `F`, following the same order as those in `Vars`. `F` is used to restore the association between the variables in `Vars` and the constraints in `Proj`, as mentioned in Section 3.1.1.
- `call_entail(+st(F,Proj),+st(FGen,ProjGen))` calls the auxiliary predicate `check_entailment(F,FGen,Proj,ProjGen)` which success if `Proj` $\sqsubseteq$ `ProjGen`. First, `check_entailment/4` unifies `F` and `FGen`, resp. the variables of the projection of the current store and the variables of the generator's projection. Then, the CLP($\mathbb{Q}$) predicate `clpq_meta(+Proj)` makes `Proj` part of the current constraint store by executing it. This does not interact with the current store, because the variables in `F` and `FGen` are fresh. And finally, `clpq_entailed(+ProjGen)` success if `ProjGen` is entailed by the current constraint store (i.e., `Proj` $\sqsubseteq$ `ProjGen`).
- `answer_compare(+st(F,Proj),+st(FAns,ProjAns),Res)` calls the predicate `check_entailment(F,FAns,Proj,ProjAns)` to check if `Proj` $\sqsubseteq$ `ProjAns`. If it is the case, `answer_compare/3` returns `'=<'` in `Res`. Otherwise, it calls `check_entailment(FAns,F,ProjAns,Proj)` to check if `ProjAns` $\sqsubset$ `Proj`. If it is the case, it returns `'>'` in `Res`, otherwise it fails (i.e., there is no entailment in any direction).
- `apply_answer(+Vars,+st(FAns,ProjAns))` unifies `FAns`, the variables of `ProjAns` with `Vars`, those in the pattern of the resumed call. Then, it uses the CLP($\mathbb{Q}$) predicate `clpq_meta(+ProjAns)` to add the answer constraint store `ProjAns` to the current constraint store. If the resulting constraint store is consistent, execution continues, and it fails otherwise.

The TCLP interface for CLP($\mathbb{R}$) is similar to that of CLP($\mathbb{Q}$). CLP($\mathbb{R}$) uses floating-point numbers and its performance is better than that of CLP($\mathbb{Q}$), which uses exact fractions. However, floating-point rounding errors make CLP($\mathbb{R}$) (and TCLP($\mathbb{R}$)) inappropriate for some applications, as entailment is unsound and therefore termination can be compromised.

3.1.5 Two-Step Projection

The design we have presented strives for simplicity. There is however an improvement that can be used to obtain more performance / reduce memory usage, at the cost of a

```
:- active_tclp.

store_projection(Vars, st(F,Proj) ) :-
                              clpqr_dump_constraints(Vars, F, Proj).
call_entail(st(F,Proj), st(FGen,ProjGen) ) :-
                           check_entailment(F, FGen, Proj, ProjGen).
answer_compare(st(F,Proj), st(FAns,ProjAns), =<) :-
                        check_entailment(F, FAns, Proj, ProjAns), !.
answer_compare(st(F,Proj), st(FAns,ProjAns),  >) :-
                           check_entailment(FAns, F, ProjAns, Proj).
apply_answer(Vars, st(FAns,ProjAns) ) :-
                                    Vars = FAns, clpq_meta(ProjAns).
check_entailment(Vars1, Vars2, Proj1, Proj2) :-
              Vars1 = Vars2, clpq_meta(Proj1), clpq_entailed(Proj2).
```

Figure 3.6: The Mod TCLP interface for CLP($\mathbb{Q}$) is a bridge to existing predicates.

slightly more complex design. We present it now, with the understanding that it does not change the general ideas we have presented so far.

`store_projection/2` is usually the most expensive operation in the TCLP interface, but it is only mandatory when a call is a generator, which we can determine from entailment checking.[3] We have however placed `store_projection/2` before entailment checking because constraint solvers can often use the projection operation to compute some information needed by the entailment check. Instead of recomputing this information, the projection is divided in two parts: an initial operation `early_call_projection(+Vars,-EarlyProj)`, executed before the entailment phase, that returns in `EarlyProj` the information needed to check entailment, and a second operation `final_call_projection(+Vars,+EarlyProj,-ProjStore)` that is executed after the entailment phase if the entailment check fails (and the call would then be a generator). If it is executed, this operation returns the projected constraint store in `ProjStore` using the information in `EarlyProj`.

For symmetry, a similar mechanism is used with answers. Instead of using `store_projection/2` (step 10 in Fig. 3.3), two specialized versions are used: `early_ans_projection/2` and `final_ans_projection/3`, respectively called before and after the answer entailment check

Example 3.1.
The TCLP($\mathbb{Q}$) interface in Fig. 3.6 uses `store_projection/2` to project the constraint store of every new call. But, since `clpq_entailed/1` does not need the projection of the current constraint store to check entailment w.r.t. the projected constraint store of a previous generator, the execution of the projection can be

[3]For efficiency, we can check entailment using the current constraint store A instead of its projection onto a set of variables S because $A \sqsubseteq B \iff Proj(S,A) \sqsubseteq B$, where $S = vars(B)$, as in our case.

delayed:

```
early_call_projection(Vars, st(Vars,_)).
call_entail(st(Vars,_), st(FGen,ProjGen)) :-
                            Vars = FGen, clpq_entailed(ProjGen).
final_call_projection(_,st(Vars,_), st(F,Proj)) :-
                          clpqr_dump_constraints(Vars, F, Proj).
```

The performance impact of implementing the *Two-Step* projection is evaluated in Section 3.3.4, using the TCLP($\mathbb{Q}$) interface.

3.2 Other TCLP Interfaces

The design we presented brings more flexibility to a system with tabled constraints at a reasonable cost in implementation effort. To support this claim we present the implementation of the TCLP interface for a couple of additional solvers: a constraint solver for difference constraints (Section 3.2.1) completely written in C and ported from (Chico de Guzmán et al., 2012), and a solver for constraints over finite lattices (Section 3.2.2).

3.2.1 Difference Constraints

Difference constraints CLP($\mathbb{D}_{\leq}$) is a simple but relatively powerful constraint system whose constraints are generated from the set $\mathcal{C}_{\mathbb{D}_{\leq}} = \{X - Y \leq d : X, Y, d \in \mathbb{Z}\}$, where X and Y are variables, and d is a constant.

A system of difference constraints can be modeled with a weighted graph, and it is satisfiable if there are no cycles with negative weight. A solver for this constraint system can be based on shortest-path algorithms (Frigioni et al., 1998) where the constraint store is represented as an $n \times n$ matrix `A` of distances. The projection of a constraint store `A` onto a set of variables `V` extracts a sub-matrix `A'` containing all pairs (v_1, v_2) s.t. $v_1, v_2 \in V$. For efficiency, a projection can be represented as a vector of length $|V|$ containing the index of each v_i in `A`. For example, if the indexes in `A` of the variables `[X, Y, Z, T, W]` are `(1, 2, 3, 4, 5)`, the projection onto the set of variables `[T, X, Y]` is represented with the vector `(4, 1, 2)`. The implementation uses attributed variables to map Prolog variables onto their representation in the matrix by having as attribute the index of each variable in the matrix. Therefore, calculating projection is fast.

The TCLP($\mathbb{D}_{\leq}$) interface (Fig. 3.7) showcases that, as we mentioned in Section 3.1.1, the representation of the projected constraint store depends on the constraint solver. In this case, the projected constraint store is represented by a triple `st(Id, Ln, Proj)`

```
early_ans_projection(Vars, st(Id,Ln,_) ) :-
                              diff_project_index(Vars,(Id,Ln)).
answer_compare(st(Id,Ln,_), st(_,_,ProjAns), =<) :-
                             diff_entailed((Id,Ln),ProjAns), !.
answer_compare(st(Id,Ln,_), st(_,_,ProjAns),  >) :-
                                 diff_entails((Id,Ln),ProjAns).
final_ans_projection(_, st(Id,Ln,_), st(Id,Ln,Proj) ) :-
                                 diff_projection((Id,Ln),Proj).
```

Figure 3.7: The answer entailment check of the Mod TCLP($\mathbb{D}_{\leq}$) interface.

where `Id` is the memory address of the vector with the indexes of the constrained variables of the call / answer, `Ln` is its length (the number of constrained variables), and `Proj` is the memory address of a copy of the sub-matrix which represents the projected constraint store. The indexes of the vector `Id` follow the same order as the variables in `Vars` and are used to restore the association between `Vars` and `Proj` when they have to be compared or applied.

TCLP($\mathbb{D}_{\leq}$) checks entailment using `diff_entailed((Id,Ln),ProjGen)` and `diff_entails((Id,Ln),ProjGen)`, where `Id` and `Ln` identify the position of the variables in the matrix `A` (the current constraint store), and `ProjGen` is the memory address of a sub-matrix which represent the projection of a previous generator. Note that the indexes of the sub-matrix from 1 to n follows the order of the indexes in `Id`, i.e., the k^{th} column/row of the sub-matrix correspond to the variable identified by the k^{th} index in `Id`.

Therefore, the TCLP($\mathbb{D}_{\leq}$) interface increases performance and reduces memory footprint using the *Two-Step* projection (Section 3.1.5) because it only makes a copy of the sub-matrix `Proj` when the entailment phase fails and the current call / answer becomes a generator / new answer. Fig. 3.7 shows the implementation of the projection and answer comparison operations using the *Two-Step* projection in the answer entailment check.

3.2.2 Constraints over Finite Lattices

A lattice is a triple $(\mathbb{S}, \sqcup, \sqcap)$ where $\mathbb{S}$ is a set of points and join $(\sqcup)$ and meet $(\sqcap)$ are two internal operations that follow the commutative, associative and absorption laws. $(\mathbb{S}, \sqsubseteq)$ is a poset where $\forall a,b \in \mathbb{S} \,.\, a \sqsubseteq b$ if $a = a \sqcap b$ or $b = a \sqcup b$ and $\exists \bot, \top \in \mathbb{S}$ such that $\forall\, a \in \mathbb{S} \,.\, \bot \sqsubseteq a \sqsubseteq \top$.

In the system of constraints over finite lattices CLP($\mathbb{L}at$), the constraints between points in the lattice arise from (1) the topological relationship of the lattice elements and (2) any additional operations between the elements in the lattice. These two classes of constraints are handled by two different layers.

The external layer is concerned with the lattice topology and implements the constraint $Y \sqsubseteq X$ with $X, Y \in \mathbb{S}$ and the projection operation for variable elimination using Fourier's algorithm (Marriott and Stuckey, 1998): the projection of $X \sqsubseteq d \wedge Y \sqsubseteq X$ onto Y is $Y \sqsubseteq d$. This layer provides entailment checking and the operation to add a projected constraint store to the current constraint store.

Further constraints on variables can be imposed by relationships derived from internal operations other than those in the lattice. Compare, for example, $Y \sqsubseteq X$ with $Y \sqsubseteq X \wedge Y = X \oplus X$ for some operation $\oplus$ among elements of the lattice: the additional information can be helpful to simplify (or prove inconsistent) the constraint store. In the lattice solver, a second layer implements these additional operations (if they exist) and communicates with the topology-related layer.

We have used this solver to implement a constraint tabling-based abstract interpreter (Section 3.3.6), where the points of the lattice are the elements of the abstract domain. The lattice implementation provides at least the operators $\sqcup$ and $\sqcap$ and the operations among the elements of the lattice, which are the counterparts of the operations in the concrete domain, as described above.

3.3 Experimental Evaluation

In this section we evaluate the performance of our framework using the four constraint systems and interfaces we have summarily described ($\mathbb{Q}$, $\mathbb{R}$, $\mathbb{D}_{\leq}$ and $\mathbb{L}at$).

In Section 3.3.1, we quantify the performance benefits of TCLP versus LP, tabling, and CLP using the `dist/3` program presented in Section 2.1 with the TCLP($\mathbb{Q}$) interface. Then we explore the impact and advantages of a more flexible modular framework. In Section 3.3.2 we evaluate the performance impact of the increased overhead w.r.t. previous implementations with less flexibility (i.e., the previous TCLP implementation of (Chico de Guzmán et al., 2012)), and in Section 3.3.3 we evaluate the benefits of a more comprehensive answer management strategy, which is easier to implement due to the flexibility of the new framework.

In Section 3.3.4, we evaluate the performance benefits of the *Two Step* projection using TCLP($\mathbb{Q}$). These benefits are due to the reduction in the number of projections executed during the evaluation. In 3.3.5, we compare the expressiveness and performance of the TCLP($\mathbb{D}_{\leq}$), TCLP($\mathbb{R}$), and TCLP($\mathbb{Q}$) interfaces. In this case, the expressiveness of CLP($\mathbb{R}/\mathbb{Q}$) comes with an overhead (which is higher in CLP($\mathbb{Q}$) due to its higher precision), but in certain problems this expressiveness can bring greats benefits using TCLP (additionally, in some problems the precision could be determinant).

And finally, in Section 3.3.6, we use tabling and TCLP($\mathbb{L}at$) to implement and benchmark a simple abstract interpreter. This evaluation shows the benefits brought not only

Table 3.1: Run time (ms) of LP, CLP($\mathbb{Q}$), tabling and TCLP($\mathbb{Q}$) for `dist/3`. '–' means no termination.

Graph		LP	CLP($\mathbb{Q}$)	Tab	TCLP($\mathbb{Q}$)
Without cycles	Left recursion	–	–	2311	**1286**
	Right recursion	> 5 min.	5136	3672	**2237**
With cycles	Left recursion	–	–	–	**742**
	Right recursion	–	10992	–	**1776**

by the entailment check instead of variant checking, but also by the integration of CLP with tabling.

The Mod TCLP framework is implemented in Ciao Prolog and it is available, including the libraries and interfaces presented in this chapter, as part of the Ciao Prolog distribution at `http://www.ciao-lang.org`. All the experiments were performed on a Mac OS-X 10.9.5 machine with a 2.66 GHz Intel Core 2 Duo processor and the benchmarks are available at `http://www.cliplab.org/papers/tplp2018-tclp/`. Times are given in milliseconds.

3.3.1 Absolute Performance of TCLP vs. LP vs. Tabling vs. CLP

Let us recall Table 2.1, where we used the `dist/3` program (see Fig. 2.1 and 2.2) to support the use of TCLP due to its better termination properties. We now want to check whether, for those cases where LP or CLP also terminate, the performance of TCLP is competitive and for those cases where only TCLP terminates, whether its performance is reasonable. We have used a graph of 35 nodes without cycles (775 edges) and a graph of 49 nodes with cycles (785 edges) and timed the results — see Table 3.1.

As we already saw in Table 2.1, TCLP not only terminates in all cases, but it is also faster than the rest of the frameworks due to the combination of tabling (which avoids entering loops and caches intermediate results) and constraint solving. It also suggests, in line with the experience in tabling, that left-recursive implementations are usually faster and preferable, as they avoid work by "suspending first" and reusing answers when they are ready.

3.3.2 The Cost of Modularity: Mod TCLP vs. Original TCLP

The original TCLP implementation (Chico de Guzmán et al., 2012) was deeply intertwined with the tabling engine and had a comparatively low overhead. Since it was done on the same platform as ours (Ciao Prolog) and shares several components and low-level implementation decisions, it seems a fair and adequate baseline to evaluate the performance cost of the added modularity. We will evaluate both frameworks using exactly the same implementation of difference constraints (Section 3.2.1) and two benchmarks:

`truckload(P,Load,Dest,Time)` (Cui and Warren, 2000; Schrijvers et al., 2008): it solves a shipment problem given a maximum `Load` for a truck, a destination `Dest`, and a list of packages to ship (1 to `P`.) We set `P=30, Dest=chicago` and use `Load` as parameter to vary its complexity. `truckload/4` does not need tabling, but tabling speeds it up.

`step_bound(Init,Dest,Steps,Limit)`: it is a left-recursive graph reachability program similar to `dist/3` that constrains the total number (`Limit`) of edge traversals. `step_bound/4` needs tabling in the case of graphs with cycles, as it is the case of the graph we will use in this evaluation.

Table 3.2 shows that `truckload/4` incurs a nearly three-fold increase in execution time with respect to the initial non-modular TCLP($\mathbb{D}_{\leq}$) implementation. This is mainly due to the overhead of the control flow. In the original implementation, execution did not leave the level of C, as the tabling engine called directly the constraint solver, also written in C. However, in Mod TCLP, the tabling engine (in C) calls the interface level (written in Prolog), which calls back the constraint solver (in C). The additional overhead is the price we pay to make it much easier to plug in additional constraint solvers, which in the original TCLP needed ad-hoc, low level wiring.

However, `step_bound/4` is less efficient in the original TCLP($\mathbb{D}_{\leq}$) implementation than in Mod TCLP, and cannot be executed in CLP($\mathbb{D}_{\leq}$) due to the cycles in the graph. The reason behind this improvement is the enhanced answer management strategy whose implementation was made possible by our modular design. We will explore this point in the next section.

3.3.3 Improved Answer Management Strategies

The modular design of Mod TCLP makes it possible to implement alternatives for internal operations more easily. In particular, the solver interface can include the `answer_compare/3` operation which determines whether a new answer entails, is entailed by, or none of them, some previous answer. This can be used to decide whether

Table 3.2: Run time (ms) of CLP($\mathbb{D}_{\leq}$), original TCLP($\mathbb{D}_{\leq}$) and Mod TCLP($\mathbb{D}_{\leq}$) for `truckload/4` and `step_bound/4`. '–' means no termination.

	CLP($\mathbb{D}_{\leq}$)	Orig. TCLP($\mathbb{D}_{\leq}$)	Mod TCLP($\mathbb{D}_{\leq}$)
`truckload(300)`	40452	**2903**	7268
`truckload(200)`	4179	**1015**	2239
`truckload(100)`	145	**140**	259
`step_bound(30)`	-	2657	**1469**
`step_bound(20)`	-	2170	**1267**
`step_bound(10)`	-	917	**845**

to add or not a new answer and remove or not an existing answer. This is undoubtedly expensive in general, but as advanced in Section 3.1.1, it holds promise for improving performance. To validate this intuition, we executed again `truckload/4` and `step_bound/4` with TCLP($\mathbb{D}_{\leq}$) under four different answer management strategies:

$\emptyset$ all the answers are stored.

$\leftarrow$ checks if new answers entail previous answers. If so, the new answer is discarded. That is the strategy used in the original TCLP framework.

$\rightarrow$ checks if new answers are entailed by previous answers. If so, the previous answers are flagged as removed and ignored, and the new answer is stored.

$\leftrightarrow$ checks entailment in both directions, discarding new answers and removing more particular answers.

The results in Table 3.3 confirm that, in the examples studied, and despite the cost of these strategies, the computation time is reduced. The "$\leftrightarrow$" strategy proves to be the best one, although by a small margin in some cases.

On the other hand, the worst strategy is '$\emptyset$', which for the `truckload/4` program increases the execution time several orders of magnitude for large cases, while for the `step_bound/4` program the execution does not terminate because it runs out of memory when trying to generate infinitely many repeated answers. While `truckload/4` behaves similarly for the other strategies, `step_bound/4` varies drastically (i.e., '$\rightarrow$' runs out of memory for the largest case).

Part of the reasons for these differences can be inferred from Table 3.4, where, for each benchmark and strategy, we show how many of the generated answers were saved, discarded before being inserted, or removed after insertion. Note that these results

Table 3.3: Run time (ms) of answer management strategies under Mod TCLP($\mathbb{D}_{\leq}$) for `truckload/4` and `step_bound/4`.

	Mod TCLP($\mathbb{D}_{\leq}$)			
	$\emptyset$	$\leftarrow$	$\rightarrow$	$\leftrightarrow$
`truckload(300)`	742039	7806	7780	**7268**
`truckload(200)`	11785	2314	2354	**2239**
`truckload(100)`	300	263	263	**259**
`step_bound(30)`	–	8450	–	**1469**
`step_bound(20)`	–	6859	38107	**1267**
`step_bound(10)`	–	2846	8879	**845**

```
:- table sd/3.

sd(X,Y,D) :-
      edge(X,Y,D0),
      D #>= D0.
sd(X,Y,D) :-
      sd(X,Z,D1),
      edge(Z,Y,D2),
      D #>= D1+D2.
```

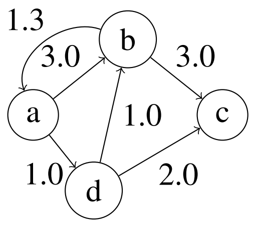


Figure 3.8: Code of `sd/3` a shortest-distance program.

are independent of the constraint solver used (i.e., executing the same programs using CLP($\mathbb{Q}$) or CLP($\mathbb{R}$) instead of CLP($\mathbb{D}_{\leq}$) generates the same answers).

For `truckload/4`, the '$\rightarrow$' and the '$\leftarrow$' strategies generate, discard / remove, and return a similar number of answers, which means that their impact in execution time is not very important. It is notwithstanding interesting to note that there is no slowdown when using the more complex strategy, '$\leftrightarrow$'. For `step_bound/4`, '$\rightarrow$' generates many more candidate answers than either of the other two — in excess of one million for `step_bound(30)` — but '$\leftarrow$' also generates one order of magnitude more candidates answers than '$\leftrightarrow$'. Note that the number of generated answers is not always the same since, as discussed before, fewer saved answers wake up fewer consumers.

As an additional example of the usefulness of obtaining the most general correct answer, Fig. 3.8 shows a graph and the program `sd/3`, used in (Cui and Warren, 2000) to calculate the "shortest distance" between the nodes in the graph. For a query such as `?- sd(X, Y, Dist)` the system reported in (Cui and Warren, 2000) returns a sequence of n answers of the form `Dist #>=` $\mathtt{N}_k$. Each $\mathtt{N}_k$ is the current achievable shortest distance

Table 3.4: Number of answers: saved (*Sav.*), discarded (*Dis.*), removed (*Rem.*) and returned to the query (*Ret.*) for each answer management strategy.

Answer strategy		# Sav.	# Dis.	# Rem.	# Ret.
$\emptyset$	`truckload(300)`	448538	0	0	14999
	`truckload(200)`	52349	0	0	1520
	`truckload(100)`	2464	0	0	58
$\leftarrow$	`truckload(300)`	67503	9971	0	41
	`truckload(200)`	16456	1325	0	23
	`truckload(100)`	1525	52	0	6
	`step_bound(30)`	44549	716826	0	252
	`step_bound(20)`	37548	599259	0	242
	`step_bound(10)`	15625	242351	0	165
$\rightarrow$	`truckload(300)`	75272	0	9460	30
	`truckload(200)`	17568	0	1298	18
	`truckload(100)`	1490	0	49	9
	`step_bound(30)`	>1145690	0	>1074071	–
	`step_bound(20)`	946309	0	891078	441
	`step_bound(10)`	294728	0	276867	221
$\leftrightarrow$	`truckload(300)`	48524	6596	1740	5
	`truckload(200)`	13550	1046	240	5
	`truckload(100)`	1343	45	10	3
	`step_bound(30)`	9697	74528	4571	25
	`step_bound(20)`	9352	71658	4371	25
	`step_bound(10)`	6650	56935	3019	25

```
fib(N,F) :-
     ⋮
    F #= F1 + F2,
    fib(N1, F1),
    fib(N2, F2).
```

(a) Version using $\mathbb{Q}$ and $\mathbb{R}$.

```
fib(N,F) :-
     ⋮
    fib(N1, F1),
    fib(N2, F2),
    F #= F1 + F2.
```

(b) Version using $\mathbb{D}_{\leq}$.

Figure 3.9: Code of `fib/2` under TCLP (it runs *forwards* and *backwards*).

from X to Y, such as $N_1 > \cdots >$ N_n, and the later N_n is the shortest distance from X to Y. E.g., for the query `?- sd(a, c, Dist)` it returns `Dist #>= 6.0` and `Dist #>= 3.0`. While using Mod TCLP under the '$\leftrightarrow$' strategy the evaluation of the query `?- sd(a, c, Dist)` only returns the answer `Dist #>= 3.0` (the most general) which corresponds to the tightest bound for the shortest distance between the nodes a and c.

3.3.4 Improved Two-Step Projection

The design of Mod TCLP makes it possible to postpone the projection during the call / answer entailment phase using the *Two-Step* projection. As we advanced in Section 3.1.5, it holds promise for performance improvements. To validate this intuition, we use two benchmarks:

`fib(N,F)` (Fig. 3.9) the doubly recursive Fibonacci program run *backwards*. It is well-known that tabling reduces `fib/2` complexity from exponential to linear. In addition, CLP makes it possible to run exactly the same program *backwards* to find the index of some Fibonacci number by generating a system of equations whose solution is the index of the given Fibonacci number (e.g., for the query `?- fib(N, 89)`, the answer is `N=11`). Under CLP, the size of this system of equations grows exponentially with the index of the Fibonacci number. However, under TCLP, entailment makes redundant equations not to be added and solving them becomes less expensive. Additionally, entailment makes it possible to terminate (with failure) even when the query does not contain a non Fibonacci number, e.g., `fib(N,` 10^{314}`)`.

`dist(X,Y,D)` (Fig. 3.2a) the program already used in Sections 2.1 and 3.3.1.

We executed each of them with Mod TCLP($\mathbb{Q}$) and the two designs for the call projection we discussed earlier:

One-Step: The projection of the call is executed before the call entailment phase (Fig. 3.6). Note that CLP($\mathbb{Q}$) does not need this projection to check entailment of the current call constraint store w.r.t. another constraint store.

***Two-Step*:** The projection of the call is executed using `final_call_projection/3` and, therefore, it is only executed when the call turns out to be a generator.

The results in Table 3.5 (top) confirm that, in the examples studied, the *Two-Step* design reduces the computation time, although only by a small margin in the case of `dist/3` with left recursion. That is because, as we see in Table 3.5 (bottom), using the *Two-Step* projection, `dist/3` with left recursion executes the projection of a call only once while using *One-Step* it executes the projection twice and therefore we only save the execution of one projection. Since they are executed early in the evaluation, the constraint store is small and their execution is faster than in the case of `dist/3` with right recursion. Note that using the *Two-Step* projection, `dist/3` with right recursion executes up to 8 times fewer call projections, and as consequence its execution has better performance.

On the other hand, `fib/2` reduces drastically the computation time using *Two-Step* projection because, during the execution, call entailment is checked many times (although the ratio of *useless* projections is similar to that of `dist/3` with left recursion). Note that the Fibonacci number F_{1500} and 10^{314} have the same size (315 digits), and the run-time and number of projections, for `fib(N,` F_{1500}`)` and for `fib(N,` 10^{314}`)`, are similar. That is because the work needed to find the index of a Fibonacci number is similar to the work needed to confirm whether a number is or is not a Fibonacci number.

3.3.5 Comparison of Mod TCLP($\mathbb{R}$, $\mathbb{Q}$ vs $\mathbb{D}_{\leq}$)

This section highlights that the modularity of TCLP makes it possible to choose the most adequate constraint solver for the specific problem, and that decision should not always be based solely on the performance of the constraint solver, but also on its expressiveness and/or precision. Since TCLP, unlike CLP, uses entailment checking extensively to decide whether to suspend and save / discard answers or not, the performance of entailment is more relevant than in CLP. It also makes its soundness (which can be challenged by e.g. numerical accuracy) critical, as incorrect entailment results can lead to non-termination or to unexpected termination.

We use the doubly recursive Fibonacci programs in Fig. 3.9 which runs *forwards* and *backwards*, already used in Section 3.3.4. We have run this benchmark using $\mathbb{R}$, $\mathbb{Q}$ and $\mathbb{D}_{\leq}$. Due to the characteristics of $\mathbb{D}_{\leq}$ (Section 3.2.1), the program for this constraint system is slightly different from the ones for $\mathbb{Q}$ and $\mathbb{R}$ (Section 3.1.4). In these two, constraints are placed before the recursive calls (see Fig. 3.9a). However, $\mathbb{D}_{\leq}$ can have at most two variables per constraint, and therefore we had to move the constraint `F #= F1+F2` to the end of the clause (Fig. 3.9b). This can be detrimental to the performance of Mod TCLP($\mathbb{D}_{\leq}$), as value propagation in the constraints is less effective.

Table 3.6 shows the experimental results. First, note that the Mod TCLP($\mathbb{D}_{\leq}$) version

Table 3.5: Comparative table of *One-Step* and *Two-Step* projection design. Note: F_n is the n^{th} Fibonacci number, and 10^{314} is not a Fibonacci number.

(a) Run time (ms) of Mod TCLP($\mathbb{Q}$) for `fib/2` and `dist/3`.

		Mod TCLP($\mathbb{Q}$)		
		One-Step	Two-Step	Ratio
`fib(N,`F_{1500}`)`		206963	**126461**	1.63
`fib(N,`F_{1000}`)`		89974	**55183**	1.63
`fib(N,`F_{500}`)`		22133	**13612**	1.63
`fib(N,`10^{314}`)`		205638	**125670**	1.63
`dist/3 right rec.`	Without cycles	2855	**2506**	1.14
	With cycles	2399	**1850**	1.30
`dist/3 left rec.`	Without cycles	1436	**1428**	1.01
	With cycles	776	**772**	1.01

(b) Number of call projections for `fib/2` and `dist/3`.

		Mod TCLP($\mathbb{Q}$)		
		One-Step	Two-Step	Ratio
`fib(N,`F_{1500}`)`		1129497	**565500**	2.00
`fib(N,`F_{1000}`)`		502997	**252000**	2.00
`fib(N,`F_{500}`)`		126497	**63500**	1.99
`fib(N,`10^{314}`)`		1126499	**563252**	2.00
`dist/3 right rec.`	Without cycles	1563	**181**	8.64
	With cycles	2144	**443**	4.84
`dist/3 left rec.`	Without cycles	2	**1**	2.00
	With cycles	2	**1**	2.00

Table 3.6: Run time (ms) of Mod TCLP($\mathbb{R}$,$\mathbb{Q}$ and $\mathbb{D}_{\leq}$) for `fib/2`.

	Mod TCLP($\mathbb{R}$)	Mod TCLP($\mathbb{Q}$)	Mod TCLP($\mathbb{D}_{\leq}$)
`fib(N, 832040)`	**25**	61	147
`fib(N, 28657)`	**16**	40	69
`fib(N, 610)`	**8**	19	24
`fib(N, 89)`	**5**	12	13

is slower than any of the other two. While the implementation of CLP($\mathbb{D}_{\leq}$) is comparatively faster than CLP($\mathbb{R}$) and CLP($\mathbb{Q}$), moving the `F #= F1+F2` to the end of the clause (which is necessary to satisfy the instantiation requirements of $\mathbb{D}_{\leq}$) reduces its usefulness to prune the generation of redundant constraints.

Additionally, note that although the solvers for $\mathbb{R}$ and $\mathbb{Q}$ are practically the same, Mod TCLP($\mathbb{R}$) is fastest in all cases, since it uses directly CPU floating point numbers while CLP($\mathbb{Q}$) implements rational numbers by software. However, there is a drawback: floating point arithmetic is not accurate, and when CLP($\mathbb{R}$) approximates its results, it can cause (depending on the particular program) non-termination. That would be the case for a query such as `?- fib(N, 23416728348467685)`, which terminates correctly with Mod TCLP($\mathbb{Q}$) but it does not (in under five minutes) with Mod TCLP($\mathbb{R}$), since the termination condition never holds.

3.3.6 Abstract Interpretation: Tabling vs. TCLP($\mathbb{L}at$)

We compare here tabling and TCLP using two versions of a simple abstract interpreter (Cousot and Cousot, 1977). The interpreter executes the programs to be analyzed on an abstract domain, collecting the possible values at every point until a fixpoint is reached. The result of the execution is a safe approximation of the run-time values of the variables in the concrete domain. The abstract domain we have used in this example is the *signs* abstract domain (Fig. 3.10).

The two versions of the abstract interpreter we have used are:

Tabling This version is a simple abstract interpreter written using tabling. This ensures termination, as the abstract domain is finite.

TCLP This version is based on the previous abstract interpreter, but it uses the TCLP($\mathbb{L}at$) constraint solver interface (Section 3.2.2) to operate on the abstract domain and set up constraints over the variables. The main differences with the *tabling* version is that TCLP uses constraint entailment instead of variant

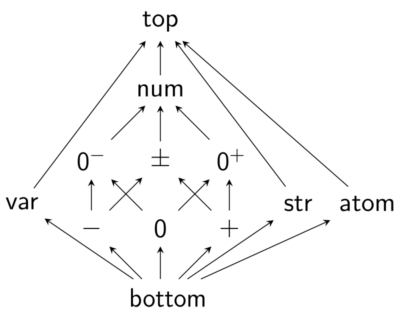


Figure 3.10: Lattice of the *Signs* abstract domain.

$$t(x_1,\ x_2,\ \ldots,\ x_n) \;=\; \texttt{if}\ x_1 \le x_2\ \texttt{then}\ x_2 \\ \texttt{else}\ t(t(x_1-1,\ x_2,\ \ldots,\ x_n),\ \ldots,\ t(x_n-1,\ x_1,\ \ldots,\ x_{n-1}))$$

```
takeuchi(X1, X2, ..., _Xn, R) :- X1 < X2, R = X2.
takeuchi(X1, X2, ..., _Xn, R) :- X1 =:= X2, R = X2.
takeuchi(X1, X2, ..., Xn, R) :- X1 > X2,
        N1 is X1 - 1, takeuchi(N1, X2, ..., Xn, R1),
        N2 is X2 - 1, takeuchi(N2, X3, ..., X1, R2),
        ...,
        Nn is Xn - 1, takeuchi(Nn, X1, ..., Xn-1, Rn),
        takeuchi(R1, R2, ..., Rn, R).
```

Figure 3.11: The n-dimensional Takeuchi function and code of `takeuchi/m`.

checking for loop detection and, therefore, it can also use the answers to more general goals to avoid computing more particular goals.

We applied our abstract interpreter to two programs:

`takeuchi/m` (Fig. 3.11): a Prolog implementation of the n-dimensional generalization of the Takeuchi function (Knuth, 1991). The program is parametric on the number of input arguments n and it returns the result in its last argument.

`sentinel/m` (Fig. 3.12): a variant of a synthetic program presented in (Genaim et al., 2001). It receives as input its first argument (the `Sentinel`) and the next n arguments $\mathtt{A}_1,\ \ldots,\ \mathtt{A}_n$ is a ring-ordered[4] series of numbers. The outputs are the arguments $\mathtt{B}_1,\ \ldots,\ \mathtt{B}_n$, which correspond to a circular shift of $\mathtt{A}_1,\ \ldots,\ \mathtt{A}_n$ such that on success $\mathtt{B}_i < \mathtt{B}_{i+1}$ for all $i < n$ and: if `Sentinel=0`, the first half of $\mathtt{B}_i$ are negative and the second half are positive; if `Sentinel` < `0`, $\mathtt{B}_i$ < `Sentinel` for all i; and if `Sentinel` > `0`, $\mathtt{B}_i$ > `Sentinel` for all i.

[4] I.e., there is a j such that $\mathtt{A_j} < \mathtt{A_{j+1}},\ \mathtt{A_{j+1}} < \mathtt{A_{j+2}},\ \ldots,\ \mathtt{A_n} < \mathtt{A_1},\ \mathtt{A_1} < \mathtt{A_2},\ \ldots,\ \mathtt{A_{j-2}} < \mathtt{A_{j-1}}$.

```
sentinel(Sentinel, A1, ..., An, B1, ..., Bn) :- Sentinel =:= 0,
        ring(A1, ..., An, B1, ..., Bn),
        B1 < B2, ..., Bn-1 < Bn, Bn/2 < Sentinel, Sentinel < Bn/2+1.
sentinel(Sentinel, A1, ..., An, B1, ..., Bn) :- Sentinel < 0,
        ring(A1, ..., An, B1, ..., Bn),
        B1 < B2, ..., Bn-1 < Bn, Bn < Sentinel.
sentinel(Sentinel, A1, ..., An, B1, ..., Bn) :- Sentinel > 0,
        ring(A1, ..., An, B1, ..., Bn),
        B1 < B2, ..., Bn-1 < Bn, B1 > Sentinel.

ring(A1, ..., An, B1, ..., Bn) :- B1 = A1, ..., Bn = An.
ring(A1, ..., An, B1, ..., Bn) :- A1 > A2,
        ring(A2, ..., An, A1, B1, ..., Bn).
ring(A1, ..., An, B1, ..., Bn) :-
        ring(An, A1, ..., An-1, B1, ..., Bn).
```

Figure 3.12: Code of `sentinel/m` program.

Table 3.7: Run time (ms) of tabling and Mod TCLP($\mathbb{L}at$) for `analyze/1`.

		Tabling	Mod TCLP($\mathbb{L}at$)
`takeuchi/m` ($m = n+1$)	n=8	31.44	**8.09**
	n=6	13.75	**5.85**
	n=3	**2.42**	3.12
`sentinel/m` ($m = 2n+1$)	n=8	1375.13	**9.23**
	n=6	218.93	**6.53**
	n=4	30.99	**4.56**

Table 3.7 shows the run time results of analyzing `takeuchi/m` parameterized by the dimension of the function, n ($m = n+1$), and `sentinel/m` parameterized by n, the length of the ring ($m = 2n+1$). In both examples the analysis with the TCLP version of the interpreter is faster than the analysis with the interpreter without constraints: the latter has to evaluate each permutation completely in the recursive predicates, while the former can suspend and save computation time using results from a previous, more general, call.

Let us examine an example. For a variable `A`, let us write $\mathtt{A}^{abs}$ to represent $\mathtt{A} \sqsubseteq abs$. On the one hand, when an initial goal $\mathtt{ring(A_1}^{top}, \ldots, \mathtt{A_n}^{top}, \mathtt{B_1}^{top}, \ldots, \mathtt{B_n}^{top}\mathtt{)}$ is interpreted by the TCLP analyzer, the first clause of `ring/2n` produces the first answer. Then the interpreter continues with the second clause, interprets the goal $\mathtt{A_1} > \mathtt{A_2}$ and starts the evaluation of $\mathtt{ring(A_2}^{num}, \ldots, \mathtt{A_n}^{top}, \mathtt{A_1}^{num}, \mathtt{B_1}^{top}, \ldots, \mathtt{B_n}^{top}\mathtt{)}$. Since $num \sqsubseteq top$, this new call entails the previous one and TCLP suspends this execution.

Table 3.8: Run time (ms) of tabling and Mod TCLP($\mathbb{L}at$) for (un)constrained calls for `analyze/1` .

		Tabling		Mod TCLP($\mathbb{L}at$)	
		constraints before call	unconst. call	constraints before call	unconst. call
`sentinel/m` ($m=2n+1$)	n=8	**749.38**	1375.13	**5.29**	9.23
	n=6	**98.80**	218.93	**3.31**	6.53
	n=4	**6.53**	30.99	**2.85**	4.56

Then the interpreter continues with the third clause, which starts the evaluation of $\texttt{ring(}\texttt{A}_\texttt{n}{}^{top},\texttt{A}_1{}^{top},\ldots,\texttt{A}_{\texttt{n}-1}{}^{top},\texttt{B}_1{}^{top},\ldots,\texttt{B}_\texttt{n}{}^{top}\texttt{)}$, and TCLP also suspends the execution. Since the generator does not have more clauses to evaluate, TCLP resumes the suspended execution with the previously obtained answer. Each consumer produces a new answer but since they are at least as particular as the previous one, they are discarded.

On the other hand when the initial goal is evaluated in the tabling interpreter with $\texttt{A}_1{}^{top},\ldots,\texttt{A}_\texttt{n}{}^{top},\texttt{B}_1{}^{top},\ldots,\texttt{B}_\texttt{n}{}^{top}$ as entry substitution, the first answer is also produced. Then the interpreter continues with the second clause, interprets the goal $\texttt{A}_1 > \texttt{A}_2$ and starts the evaluation of the recursive call with the entry substitution $\texttt{A}_1{}^{num},\ldots,\texttt{A}_\texttt{n}{}^{top},\texttt{B}_1{}^{num},\ldots,\texttt{B}_\texttt{n}{}^{top}$. However, tabling does not suspend the execution because it is not a *variant* call of the previous one, which results in increased computation time.

Table 3.8 shows the results of analyzing `sentinel/m` in two different scenarios: without any constraints in the abstract substitution of the variables, or adding the constraint `Sentinel` $\sqsubseteq +$ before the analysis. Adding that domain restriction reduces analysis times by approximately the same ratio in both cases. Note that this compares two scenarios which are possible both for TCLP and for tabling without constraints. Additionally, the TCLP-based analyzer would be able to take into account constraints *among* variables, which would not be directly possible using tabling without constraints.

3.4 Discussion

We have presented an approach to include constraint solvers in logic programming systems with tabling. Our main goal is making the addition of new constraint solvers easier while taking full advantage of entailment between constraint stores. In order to achieve this, we determined the services that a constraint solver should provide to a tabling engine. This interface has been designed to give the constraint solver freedom to implement them. To validate our design, we have interfaced one solver previously written in C, CLP($\mathbb{D}_{\leq}$), two existing classical solvers, CLP($\mathbb{Q}/\mathbb{R}$), and a new solver, CLP($\mathbb{L}at$), and we have found the integration to be easy — certainly easier than with other designs —, validating the usefulness of the capabilities that our system provides.

We evaluated its performance in a series of benchmarks. In some of them large savings are attained w.r.t. non-tabled/tabled executions, even taking into account the penalty to pay for the additional flexibility and modularity. We are in any case confident that there is still ample space to improve the efficiency of the implementation, since in the current implementation we gave more importance to the cleanliness of the code and the design.

Chapter 4

Incremental Evaluation of Aggregates using Tabled CLP

The previous chapters show that keeping only the more general answers not only improves performance, reduces memory requirements, and improves performance but also preserves correctness. In this chapter, we show that for some applications only the aggregation of those answers is needed and we present ATCLP, a framework to incrementally compute the aggregation of elements in a lattice, that improves the efficiency of certain programs without losing correctness. The framework is based on a new intended semantics for lattice-based aggregates, consistent with the LFP semantics, that guaranties soundness and completeness and improves termination properties. The ATCLP framework is implemented using an extended version of Mod TCLP that allows the combination of answers. Its design allows not only the definition of arbitrary lattice-based aggregates, but also the definition of aggregates that do not fit into a lattice structure.

As we show in Chapter 2, using Mod TCLP, it is possible to synthesize answers to a logic programming query without generating all possible answers, i.e., by applying an operational semantics that includes mechanisms to avoid repeating (some) redundant computations. The operations that take a series of records in a database table or answers to a query, and synthesize a result using them are called aggregates. The maximum, minimum, the set of all answers, the number of (different) solutions, and the average of the solutions are well-known examples of aggregates. They can be straightforwardly computed by gathering all the solutions and then computing the aggregate as dictated by its definition. However, this can be suboptimal or have drawbacks. An example would be returning the shortest path in a graph: the search can be stopped whenever the current path is longer than the shortest one found so far. In other, more involved cases, the computation of the aggregate may recursively involve the aggregate itself, and computing first a model without aggregates with a fixpoint procedure and then applying aggregation may be impossible or unsound.

In this chapter we introduce, implement, and evaluate a consistent least-fixed point semantics for a class of common aggregates derived from an interpretation of their meaning in a lattice. This interpretation makes it possible to give them a consistent least fixed point semantics. We observe that existing implementations for tabled logic programming (Santos Costa et al., 2012; Swift and Warren, 2012; Zhou, 2012) have been extended to provide the machinery necessary to compute aggregates incrementally: tabling needs to store the answers returned by the different branches of the computation, which is a first step towards computing aggregates (e.g., the so-called *modes* makes it possible to incrementally compute some specific aggregates). However, while being very helpful in some situations, a careful examination of their behavior reveals inconsistencies with the LFP semantics (Vandenbroucke et al., 2016) which makes reasoning about simple programs unsound. We, therefore, added the necessary support in the form of syntax and underlying infrastructure to incrementally compute aggregates based on the answers that are added to the table. In particular, we extend the answer management strategy of Mod TCLP with a more flexible answer management strategy in order to make it possible the combination of answers.

In Section 4.1 we motivate and present an intended semantics, for lattice-based aggregates that is consistent with the LFP semantics and supports programs with non-stratified aggregates. In Section 4.2 we characterize the lattice-based aggregates that are consistent with the intended semantics. In Section 4.3 we describe the design of the interface provided to define aggregates, we give some examples, and we sketch some implementation details of the framework In Section 4.4 we explain how to implement some non-lattice aggregates, whose execution may not align with LFP semantics. In Section 4.5 we evaluate the expressiveness and performance of ATCLP versus Prolog and tabling and finally, in Section 4.6, we offer some conclusions.

4.1 Motivation

Tabling engines that implement *mode-directed tabling* (Guo and Gupta, 2008; Zhou et al., 2010) and/or *answer subsumption* (Swift and Warren, 2010) can use policies other than being a variant to decide whether an answer should be stored or discarded. These are expressed by specifying the *modes* of some arguments: `dist(_, _, min)` specifies the (aggregate) mode `min` for the third argument, and the query `?- dist(a, Y, D)` will terminate also for acyclic graphs because only the shortest distance will be returned. These systems usually included a predefined, small collection of aggregates with ad-hoc implementations.

```
p(3).
p(2).
p(1) :- p(2).
p(0) :- p(3).
```

(a) Program $\mathscr{P}$.

$$p(x) \leftarrow 3 \leq x.$$
$$p(x) \leftarrow 2 \leq x.$$
$$p(x) \leftarrow 1 \leq x \wedge p(2).$$
$$p(x) \leftarrow 0 \leq x \wedge p(3).$$

(b) A constraint-based intended meaning of $\mathscr{P}_{min}$.

Figure 4.1: Code and interpretation of a predicate aggregated using minimum.

However, as shown elsewhere (Kemp and Stuckey, 1991; Pelov et al., 2007; Vandenbroucke et al., 2016), reconciling the standard least fixpoint semantics for logic programming with a sensible semantics for aggregates in LP is not straightforward.

Example 4.1. Let us consider the standard LP program $\mathscr{P}$ in Fig. 4.1a, a variant of an example from (Vandenbroucke et al., 2016). Its model would be {p(0), p(1), p(2), p(3)}. Let us call $\mathscr{P}_{min}$ the program that minimizes the argument to predicate p/1; its model ought to be {p(0)}. That is a type of aggregate, since all answers for $\mathscr{P}$ need to be taken into account to produce a single answer for $\mathscr{P}_{min}$.[1] However, a straightforward reduction of the model of $\mathscr{P}$ to generate the model of $\mathscr{P}_{min}$ is at odds with the standard least fixpoint semantics: in order to apply the immediate consequence operator T_P of the fixpoint semantics, for p(0) to be in the model of $\mathscr{P}_{min}$, p(3) needs to be also in that model, but we had stated before that the model or $\mathscr{P}_{min}$ only contains p(0).

The lattice semantics for stratified programs presented in (Vandenbroucke et al., 2016) provides an interpretation for programs with aggregates consistent with the expected aggregated answers. However, it distinguishes between predicates that imply and are implied by aggregated values in such as way that the atoms subsumed by the aggregated answer are only true *during* the evaluation of the aggregated predicates: p(3) is true (and used) while computing ?- p(X). However, the query ?- p(3) fails. That is somewhat misleading, and reasoning, transforming, debugging, etc. with a programming language having that semantics can be challenging.

On top of that, the behavior of implementations featuring aggregates is sometimes erratic. XSB and B-Prolog return p(1) for the query ?- p(X) to $\mathscr{P}_{min}$, while Yap, which uses batch scheduling,[2] returns, on backtracking, p(3), p(2), and p(1) the *first time* the query is issued, and only p(1) in subsequent calls. These issues, already mentioned in (Vandenbroucke et al., 2016), point to the need for a better semantics and consistent implementations. We will present here what we think is an alternative,

[1]This is usually marked with a program declaration that specifies which predicate and which argument is to be minimized. We delay introducing its precise syntax and just assume it appears in $\mathscr{P}_{min}$.

[2]Batch scheduling returns answers as soon as they are found.

defensible meaning for a class of aggregates that is compatible with the least fixpoint semantics.

Under the assumption that `p(0)` is the expected result for the query `?- p(X).` to the program $\mathscr{P}_{min}$, then we can expect that for any atom `p(`x`)` in the model of $\mathscr{P}$, the constraint $0 \leq x$ holds. From this, we assume that minimizing the parameter to `p/1` redefines its meaning to be: `p(`k_m`)` is in the model of $\mathscr{P}_{min}$ iff for all k_i such that `p(`k_i`)` is in the model of $\mathscr{P}$, the constraint $k_m \leq k_i$ holds. In other words, we link the computation of the minimum with the solution of the set of constraints $k_m \leq k_1 \wedge k_m \leq k_2 \wedge \cdots \wedge k_m \leq k_n$ where $m \in \{1, \ldots, n\}$ and we posit `p(`k_m`)` as the only atom of the model of $\mathscr{P}_{min}$. With the knowledge that the minimum is induced by the constraint $\leq$, the value k_m acts as a representative of the set of constraints. Fig. 4.1b shows a CLP program whose (constraint) semantics model that of $\mathscr{P}_{min}$ (but whose model is a set of constraints, rather than a canonical representative, as in our case).

Under this semantics, the clause `p(0) :- p(3)` can be used without lack of consistency: if the model `{p(0)}` is assumed to mean `p(`x`)` s.t. $0 \leq x$, the atom `p(3)` is consistent with that because $0 \leq 3$. Therefore, `p(3)` can be used to support `p(0)`. Furthermore, the query `?- p(3)` would also succeed.

Additionally, this makes the behavior of the program under modifications more reasonable. Let us assume that we add the clause `q :- p(2)` to $\mathscr{P}$. Its evaluation would return the model {`p(0), p(1), p(2), p(3), q`}. If we are asked to evaluate the corresponding $\mathscr{P}_{min}$, we would expect the model {`p(0), q`}. However, with the proposal in (Vandenbroucke et al., 2016), the model would be just `{p(0)}`, while with our proposal `p(2)` would be entailed by `p(0)` and therefore `q` would be part of the model.

One effect of this interpretation is to extend the model of $\mathscr{P}_{min}$ to include some atoms that were not in $\mathscr{P}$. The model the latter `{p(0), p(1), p(2), p(3)}` and the intended meaning of the model of $\mathscr{P}_{min}$ is `{p(`x`)` $\mid 0 \leq x$`}`. Therefore, the query `?- p(5)` to $\mathscr{P}_{min}$ should also succeed. This may seem odd, but let us note that by aggregating (on the minimum, in this case) we are effectively loosing information about $\mathscr{P}$: there is an infinite number of different programs similar to $\mathscr{P}$ whose model is `{p(0)}`. When `p(0)` is returned as representative, we implicitly "forget" information about the model of $\mathscr{P}$, and it makes sense to assume it to be as general as possible — i.e., `{p(`x`)` $\mid 0 \leq x$`}`.

In the rest of the chapter we will explore the properties that should be satisfied by aggregate operations so that they can follow a semantics similar to what we have presented, and we will sketch an implementation that behaves according to this semantics.

4.2 Aggregates as Lattice Operations

Several useful aggregates can be expressed based on the repeated application of a selection or composition operation. Moreover, the elements on which this operation is defined can often be put in a lattice structure, with the base operation on which the aggregate is defined being intimately related to operations or relations in the lattice itself. This requires a notion of partial order between the results computed by the aggregates, but in turn gives flexibility on how the aggregate can be computed. In particular, the aggregate can be the result of a fixpoint computation on the lattice. As an intuitive example, the minimum y of a finite set of elements S ordered by a relation $\leq$ can be computed by iterating over the elements of S and selecting the y s.t. $\neg\exists x \cdot x \neq y \wedge x \leq y$.

4.2.1 Entailment-Based Aggregates

The simplest type of aggregate functions can be defined using only the $\sqsubseteq$ relation of the lattice. We will refer them as "aggregates based on entailment" since in our framework, the relation $\sqsubseteq$ will be treated very similarly to how constraint entailment is treated in constraint solvers.

Definition 4.1 (Entailment-Based Aggregates)**.** Given a partial order relation $\sqsubseteq$ (i.e., a relation that is reflexive, anti-symmetric, and transitive) over a multi-set[3] S that induces a lattice structure on it, the aggregate of S emanating from $\sqsubseteq$, denoted as $Agg_{\sqsubseteq}\sqsubseteq$, is the set of maximals of S w.r.t. $\sqsubseteq$:

$$Agg_{\sqsubseteq}\sqsubseteq(S) = \{x \in S \mid \neg\exists\, y \in S \cdot y \neq x \ \wedge \ y \sqsubseteq x\}$$

The minimum and the maximum are widely used entailment-based aggregates: the minimum of a multiset is the least upper bound of the lattice induced by $\leq$ (resp., the maximum). The minimum of a set S, as an aggregate based on $\leq$, is defined as:

$$\min(S) = Agg_{\sqsubseteq}\leq(S) = \{x \in S \mid \neg\exists\, y \in S \cdot y \neq x \ \wedge \ y \leq x\}$$

Note that, in general, different multisets can have the same aggregate: $Agg_{\sqsubseteq}\leq(\{2,3,4,5,6\}) = Agg_{\sqsubseteq}\leq(\{2,3,4\}) = \{2\}$ or, on the other direction, a single aggregate can correspond to many initial sets. For example, $Agg_{\sqsubseteq}\leq(S) = \{2\}$ would be valid for any set S such that $\forall x \in S \cdot 2 \leq x$. As noted before, we adopt the view that an aggregate *represents* the S that meets this definition.

[3]This definition is usually based on a set instead of a multi-set. The reason to choose explicitly a multi-set will be clear in Section 4.4, when we apply our implementation to operations that cannot be embedded in a lattice.

Since $\leq$ among numbers is a total order, the minimum is unique and its aggregate is a singleton: $Agg_{\sqsubseteq\leq}(\{2,3,4\}) = \{2\}$. This is not always the case. Let us assume that our domain is a pair of numbers (x,y) and we define

$$(a_1,a_2) \leq_p (b_1,b_2) \ \leftrightarrow\ a_1 \leq b_1 \ \wedge\ a_2 \leq b_2$$

Then, $Agg_{\sqsubseteq\leq_p}(\{(4,4),(4,2),(3,3)\}) = \{(4,2),(3,3)\}$. This example can be seen as a particular case of the aggregation of a tuple of n ($n > 1$) elements with a specific entailment-based aggregate for each tuple element. Note that the aggregation of tuples with multiple elements that are aggregated using different operators can return a set of tuples that are not comparable with each other. In a similar way, the Pareto Frontier (Pareto, 1964) is the set of Pareto-optimal solutions for multi-objective optimization problem, according to different objective functions. By using aggregates instead of objective functions, we can define a relation between the elements (i.e., we define the preferred element) instead of defining objective functions that have to be maximized/minimized. Therefore, it is possible to encode Pareto frontiers in our setting by defining a relationship $\sqsubseteq_{PF}$ as $x \sqsubseteq_{PF} y \leftrightarrow f_1(x) \leq f_1(y) \wedge f_2(x) \leq f_2(y)$ for two optimization function f_1 and f_2.

It is interesting to note that some policies commonly used to determine whether an answer (or a call) is or not a repetition of previously seen answer / call, such as `variant` or `subsumption`, can be expressed as multi-objective entailment-based aggregates.

4.2.2 Join-Based Aggregates

Some applications benefit from aggregates that require operations richer than the entailment, because they, for example, have to generate new elements based on previous elements (see Chapter 5). For these cases, we propose using an aggregate similar to that in Def. 4.1, but using the *join* operation instead of the entailment.

Definition 4.2 (Join-Based Aggregates)**.** Given a join-semilattice domain D with a join operation $\sqcup$ (that is commutative, associative, and idempotent), the aggregated value of a multi-set $S \in D$ over $\sqcup$, denoted as $Agg_{\sqcup}\sqcup$, is the least upper bound of S w.r.t. $\sqcup$:

$$Agg_{\sqcup}\sqcup(S) = LUB_{\sqcup}(S)$$

There are two main differences w.r.t. entailment-based aggregates: $Agg_{\sqcup}\sqcup(S)$ returns a single answer (not a set of answers) and the returned answer may **not** belong to S. In a logic program, the result of using a join-based aggregate may not be a logical consequence of the program without aggregates. For example, let us define an (infinite) lattice whose elements are pairs of natural numbers and where the join operation is defined as $(a_1,b_1) \sqcup_{\min} (a_2,b_2) = (\min(a_1,a_2),\min(b_1,b_2))$. For a set $S = \{(a_i,b_i)\}, i = 1\dots n$, the aggregation over this join operator is:

$$Agg_{\sqcup}\sqcup_{\min}(S) = LUB_{\sqcup_{\min}}(\{(a_i,b_i) \in S\}) = (min(a_i),min(b_i)) \text{ for } i = 1\dots n$$

A simple call of this operation may yield a value outside the set to which it is applied: $Agg_{\sqcup}\sqcup_{\min}(\{(4,4),(4,2),(3,3)\}) = (3,2)$. In a similar way to what happened with the entailment-based aggregates, the meaning of the atoms derived from a join-based aggregate are expected to represent a sub-lattice through a minimal (w.r.t. $\sqsubset_{\min}$) representative, i.e., $\forall(x,y) \in \{(4,4),(4,2),(3,3)\} \cdot (x,y) \sqsubseteq_{\min} (3,2)$ for a suitable defined $\sqsubseteq_{\min}$ based on $\sqcup_{\min}$: $x \sqsubseteq_{\min} y \leftrightarrow y = y \sqcup_{\min} x$.

4.3 The ATCLP Framework

We will now present some implementation details of the ATCLP framework. As mentioned in Section 4.1, with the ideas presented in Section 4.2 we aim at working around some of the inconsistencies found in systems such as XSB, B-Prolog, and Yap, which also implement a notion of aggregates using tabling. We will first discuss here the ATCLP interface used to declare aggregated predicates and how the predicates on which these aggregates are based are written. We will then sketch the implementation of the framework and how Mod TCLP was extended to make it possible to filter and combine answers.

Our system is built upon the infrastructure that Mod TCLP uses to handle tabling with constraints (see Chapter 3). A compelling reason to do so is that many key operations are alike: from an implementation point of view, entailment and join in a lattice can be handled similarly to how entailment (including removing redundant answers) and joins (answer merging) are managed in a tabled constraint system.

4.3.1 Design of the ATCLP Interface

ATCLP provides a directive to declare which argument(s) of which predicate(s) are to be aggregated and an interface to specify at user-level how these arguments have to be filtered or combined.

The ATCLP framework is activated with the directive `:- use_package(tclp_aggregates)`, which loads the runtime ATCLP library and installs a compile-time translation (Cabeza and Hermenegildo, 2000) to convert the programs with aggregate declarations into the core language of Mod TCLP (see

Table 4.1: Encoding example of entailment-based aggregates.

Aggregate	Code for entailment checking
minimum among numbers	`=<`
maximum among numbers	`>=`
enclosing interval	`interval(A1-A2,B1-B2):- A1=<B1, A2>=B2.`
containing set	`set(A,B):- ord_subset(B,A).`
index / variant	`variant`
answer subsumption	`sub(A,B):- instance(B,A).`
Pareto-frontier(Op)	`frontier(Op,As,Bs):- maplist(Op,As,Bs).`
n Pareto-frontier(Ops)	`n_frontier([],[],[]).` `n_frontier([Op|Ops],[A|As],[B|Bs]):-` `Op(A,B), n_frontier(Ops,As,Bs).`

Chapter 3). Aggregated predicates are declared with directives similar to those used by mode-directed tabling. For a predicate `p/n`, the general form is `:- aggregate p(mode`$_1$`,...,mode`$_n$`)`, where `mode`$_i$ denotes the aggregate used for the i^{th} argument. `mode`$_i$ can be either `entail(agg`$_i$`)` for entailment-based aggregates or `join(agg`$_i$`)` for join-based aggregates. The predicates `agg`$_i$ that implement the entailment and join operation are expected to behave as follows:

Entailment: `agg`$_i$ is expected to be a predicate of arity 2 `agg(A, B)` that succeeds iff `A` is more general than `B`, i.e., $B \sqsubseteq_{agg} A$.

Join: `agg`$_i$ is expected to be a predicate of arity 3 `agg(A, B, New)` that computes the LUB of `A` and `B` and leaves in `New`, i.e., $New = A \sqcup B$.

The arguments that are not aggregated are denoted using the mode `'_'`. These will be evaluated under variant tabling, which means that extra answers that are equal modulo variable renaming will be removed. Additionally, since exploring several examples strongly suggest that the cases where join- and entailment-based aggregates are used together seldom appear in practice, we provide two shorter notations: `:- agg_entail p(agg`$_1$`,...,agg`$_n$`)` for entailment-based aggregates, and `:- agg_join p(agg`$_1$`,...,agg`$_n$`)` for join-based aggregates. The markers `agg`$_i$ have the same meaning as before.

Examples of Entailment-Based Aggregates The flexibility of ATCLP makes it possible to very concisely encode many (if not all) aggregate functions that have been proposed in the database and logic programming literature. These are available in many implementations, sometimes by ad-hoc, systems-level libraries. We note that we aim at providing user-level code for aggregates and leave at systems-level only the infrastructure operations. Table 4.1 gives implementations for some common aggregates that can be defined based on entailment.

Some aggregates, such as minimum among numbers, can be expressed with already existing predicates, and thus they do not require specific additional code. That is, for example, the case of the minimum: in order to aggregate `p/1` as shown in Fig. 4.1a, we would use directly the directive `:- agg_entail p(=<)`. The maximum (included for completeness) would of course be similar.

The *enclosing interval* aggregate deals with continuous intervals: an interval $[A_1, A_2]$ is represented by a term `A1-A2`. The entailment checks, for any two intervals, that one completely contains the other: interval `B` entails interval `A` if `A` contains `B`. Therefore, the aggregate retains, for a set of intervals, only those that are not contained in any of the others. Note that this builds on the (infinite) lattice of intervals, and it is a case where the aggregate may not be a singleton: for the set of initial answers $\{[1,5],[2,4],[0,3],[-1,4]\}$, its aggregate would be $\{[1,5],[-1,4]\}$, and the answers would be returned one by one on backtracking. This, and similar aggregates on intervals, are useful to code applications and frameworks using Allen's algebra of intervals (Allen, 1983), such as the event calculus (Mueller, 2014; Shanahan, 1999).

The aggregate *containing set* is similar to *enclosing interval* but using discrete sets instead. It is directly built on the data structures used by Richard O'Keefe *ordered sets* library (ordered lists), that is available in many Prolog systems. Entailment is set inclusion and, similarly to the *enclosing interval* aggregate, the aggregate can return several answers.

index and *subsumption* are defined using library predicates. The aggregate *index* removes repeated answers (modulo variable renaming) while *subsumption* keeps only the most general answers.

As we mentioned in Section 4.2.1, it is possible to generalize the $\leq_p$ example by taking tuples of an arbitrary number of components. `frontier(Op)` is an advanced example that uses the higher-order capabilities of Ciao to write a parametric aggregate. Tuples are represented using lists, where the order applied to the elements of the list is determined by the parameter `Op` in the declaration itself. For a predicate `p/1` that returns lists of integers of a fixed length, the directive `:- agg_entail p(frontier(=<))` will make predicate `p/1`, defined as

```
p([4,4]).  p([4,2]).  p([3,3]).
```

work as follows:

```
?- p(R).
  R = [4,2] ;
  R = [3,3]
```

Table 4.2: Encoding examples of join-based aggregates.

Aggregate	Code

least upper bound	`lub(A,B):- lub(A,B,A).`
	`lub(a,b,c).   lub(a,c,c).   lub(a,d,d).`
	`lub(b,a,c).   lub(b,c,c).   lub(b,d,d).`
	`lub(c,d,d).   lub(X,X,X).`

widest enclosing interval	`interval(A1-A2,B1-B2,C1-C2):-`
	`      (A1=<B1 -> C1=A1; C1=B1),`
	`      (A2>=B2 -> C2=A2; C2=B2).`

set	`set(A,B,C):- ord_union(A,B,C).`

It is also possible to define an *n*-generic Pareto Frontier where each tuple component is compared differently. The directive `:- agg_entail p(n_frontier([=<,>=, sub]))` together with the definition of `n_frontier/3`, will apply a different entailment check to every member of the list: for a predicate of the form `p([A, B, C])`, `A`'s will be compared using `=<`, `B`'s are compared using `>=`, and `C`'s will be compared using `sub`sumption. While this can be built by wrapping all arguments in a single structure and writing a specific entailment check predicate that applies a different check to every element in the structure, ATCLP frees the programmer from having to write such glue code.

Example of Join-Based Aggregates Join-based aggregates require the definition of a predicate to compute the LUB of two values and also the definition of a predicate to check entailment, which ATCLP uses to improve termination. When a new answer `A` entails a previous one `B`, joining existing answers with `A` will generate answers that are entailed by previously generated answers. Therefore, both `A` and the answers that joining it may have generated are superfluous and we can discard `A` immediately.

Table 4.2 shows the implementation of some join-based aggregates. The `lub` aggregate, proposed in (Vandenbroucke et al., 2016) to join the values from the lattice `{a, b, c, d}` with $a \sqsubseteq c$, $b \sqsubseteq c$, and $c \sqsubseteq d$, is implemented by defining `agg/2` to check entailment and `agg/3` to compute the join. The entailment check is defined here in terms of the join operation (i.e., `lub(A, B) :- lub(A, B, A)`). Although entailment can be defined so for all lattice-based aggregates, ATCLP requires both the join and the entailment check to be defined explicitly. This makes it possible to write entailment check predicates that may be more efficient than those directly based on join.

Additionally, this design decision makes it possible to select the entail-based version of join-based aggregates. For example, the aggregates `interval` and `set` which Table 4.1 presents as entail-based aggregates can be also be considered as join-based aggregates

```
path(X,Set):-
    setof(Y, path_(X,Y), Set).
path_(X,Y):- edge(X,Y).
path_(X,Y):- edge(X,Z), path_(Z,Y).
```

(a) LP version.

```
:- agg_join path(_,set).
path(X,[Y]):- edge(X,Y).
path(X,Ys) :- edge(X,Z),
              path(Z,Ys).
```

(b) Version with aggregates.

```
edge(a,b).
edge(b,c).
edge(b,a).
edge(c,d).
```

(c) Cyclic graph.

Figure 4.2: Code of `path/2` set of reachable nodes from a given node.

by defining a join operator.

> **Example 4.2.** Let us consider a program to compute the set of nodes that are reachable from a given node in a graph. Fig. 4.2 shows, on the left, a simple Prolog program and, on the right, an ATCLP program using the `set` aggregate. While both seem to have the same expressiveness, the Prolog program would loop for graphs with cycles and cannot answer some queries that the ATCLP program can (see at the end of this example). Adding tabling to the Prolog program helps in this particular case, but note that mixing all-solution predicates and tabling does not always work. In particular, when table predicates and all-solution predicates call each other, the suspension and resumption mechanism of tabling interacts with the usual failure- and assert-driven implementations of `setof/3` (and similar) predicates. This example returns the set of nodes as an ordered list without repetitions: for the query `?- path(a, L)`, it answers `L=[a, b, c, d]`. Moreover, if we want to know from which nodes we can reach a set of nodes, the query `?- path(X, [a, d])` returns `X=a` and `X=b` under ATCLP, which neither Prolog nor tabling can if `setof/3` is used.

4.3.2 Implementation Sketch

Programs with aggregates are transformed at compile-time into another program that uses Mod TCLP as underlying infrastructure.

Mod TCLP: As we explained in Chapter 3, Mod TCLP is a tabling engine that handles constraints natively. It can use constraint entailment to perform suspension and to save and return only the most general answers to a query. Its modularity comes from a generic interface between the tabling infrastructure and the constraint solver that defines what operations must be provided by the solver. By extending the code that deals with these solver operations, we *piggybacked* on the existing TCLP engine to compute aggregates as previously described.

Program transformation: The program in Fig. 4.3 shows the result of the program transformation when applied to the program in Fig. 4.1a after adding the directive

```
1  :- include(aggregate_rt).
2  :- table '$p'/1.
3
4  p(Arg1) :- put(V1,(=</2,F1)),
5            '$p'(V1),
6           ( var(Arg1) -> Arg1 = F1
7           ; '$entail'(=<,Arg1,F1) ).
8  '$p'(V):- get(V,(=</2,A)), A=3.
9  '$p'(V):- get(V,(=</2,A)), A=2.
10 '$p'(V):- get(V,(=</2,A)), A=1, p(2).
11 '$p'(V):- get(V,(=</2,A)), A=0, p(3).
12
13 '$entail'(=<,B,A) :- =<(A,B).
```

Figure 4.3: Transformed code for the minimization of `p/1`.

`:- agg_entail p( =< )` to minimize the argument of `p/1`.

The predicate `p/1` is rewritten to call an auxiliary predicate `'$p'/1` where the arguments to aggregate are substituted by attributed variables (Holzbaur, 1992). Their attributes are tuples of the form `(Mode`$_i$`,F`$_i$`)`, where `Mode`$_i$ is the name of the predicate that encodes the entailment check or computes the join of values and `F`$_i$ is a fresh variable where the aggregated value will be collected. Calls with attributed variables are captured by the tabling engine and their execution is redirected to the constraint interface for Mod TCLP. By catching these calls, the engine can retrieve the answers and apply the appropriate aggregation predicate. The auxiliary predicate mimics the original one, but the actual call arguments are retrieved from the attributed variables with `get/2`. Once the auxiliary predicate collects the aggregated answer, it is either returned (if called with an unbound variable) or checked for entailment against the value in the corresponding argument (line 7). `'$entail'/3` is a bridge predicate automatically generated by the compiler to give access at runtime to the user-provided entailment check:

```
'$entail'(Agg,B,A) :- Agg(A,B).
```

where `Agg` will be statically instantiated to the name of the predicate that implements the base operations of the aggregate (see line 14 of Fig. 4.3.) Similarly, the compiler generates bridge predicates to access the join operation:

```
'$join'(Agg,A,B,New) :- Agg(A,B,New).
```

They are used to invoke the right entailment or join predicate for each aggregate.

ATCLP Internals The TCLP tabling engine calls interface predicates initially designed to be provided by a constraint solver. When this interface is used by the aggregate library, their implementation is always the same and is provided by the library runtime — see Fig. 4.4, where we made the simplifying assumption that we are aggregating over a single variable. This implementation merely recovers information related to which aggregate is being used and which variables are involved, and passes it to and from the join and entailment operations.

ATCLP uses two objects: the variable corresponding to the call argument on which the aggregation is performed (`V`) and how this aggregation is performed. The latter

```
store_projection(V, (Agg/T,A))                            :- get(V, (Agg/T,A)).
call_entail((_, _), (_, B))                               :- var(B).
answer_compare((Agg/T,A), (Agg/T,B),'=<')                 :-'$entails'(Agg,A,B),!.
answer_compare((Agg/T,A), (Agg/T,B), '>')                 :-'$entails'(Agg,B,A),!.
answer_compare((Agg/3,A), (Agg/3,B),'$new'((Agg/3,New))) :-'$join'(Agg,A,B,New).
apply_answer(V, (Agg/T,B))                                :- get(V,(Agg/T,A)), A=B.
```

Figure 4.4: Simplified ATCLP interface with the constraint tabling engine.

corresponds to the attribute mentioned before: a tuple `(Mode, A)` where `Mode` is of the form `Agg/2` or `Agg/3` for an entailment or join aggregate, respectively, and `Agg` is instantiated to the name of the corresponding predicate, and `A` is the original runtime variable that was used to invoke the predicate to be aggregated. There are three main phases in the execution of ATCLP:

Call entailment: for a new call `NewCall`, the TCLP engine invokes `store_projection(+V,-(Agg/T,A))` to retrieve the information corresponding to the arguments to aggregate (`V`, in this case). Then, `call_entail/2` is invoked to check whether the current value on `A` entails the value of a previous generator. When computing aggregates, this check will always succeed because the program transformation makes `A` be a fresh variable.

Answer entailment: the TCLP engine invokes `store_projection(+V,-(Agg/T,A))` to retrieve the information corresponding to a new answer. It then invokes `answer_compare(+(Agg/T,A),+(Agg/T,B),-Res)` to compare this new answer `A` against a previous answer `B` and returns `Res='=<'` or `Res='>'` if $A \sqsubseteq_{Agg} B$ or $B \sqsubset_{Agg} A$, respectively. This result is used to detect entailment with existing answers and either discard a new, more particular answer, or to remove existing answers from the table. If entailment fails and there is a join aggregate defined, a new lattice point $New{=}A \sqcup_{Agg} B$ is computed and returned with `Res='$new'(New)`. `New` is added to the answer table and `A` and `B` are removed. Otherwise, `answer_compare/3` fails and the new answer is stored in the answer table of the generator.

Answer consumption: In tabled constraints, answers from a generator may not be directly applicable to a consumer: if the environment of the consumer is more restrictive than that of the generator, the generator's answers have to be filtered by applying the constraints in the consumer environment to generate compatible answers. As we mentioned before, the program transformation introduces fresh variables for the parameters on which we want to aggregate (see line 4 in Fig. 4.3), so when the TCLP engine calls `apply_answer(+V,+(Agg/T,B))`, it will just return the aggregated answer. Later on, if the aggregated predicate was called with a ground terms then, `'$entail'/3` is invoked to check entailment (see line 7 in Fig. 4.3) and it succeeds if the ground term is subsumed by the aggregated answer.

```
1  new_answer :-
2      answer_lookup_table(V, Ans),
3      store_projection(V, A),
4      (   projstore_As(Ans, List_Ans),
5          member(B, List_Ans),
6          answer_compare(A, B, Res),
7          (    Res == '=<'                 % Discard answer A
8          ;    Res ==  '>',                % Remove answer B
9               remove_answer(B),
10              fail
11         ;    Res == '$new'(New),         % Save New answer
12              remove_answer(B),
13              save_answer(Ans, New)
14         ), !
15     ;   save_answer(Ans, A)               % Save answer A
16     ), !, fail.
```

Figure 4.5: Extended implementation of `new_answer/0`.

4.3.3 Adapting the Answer Management of TCLP

The answer management strategies originally proposed in Mod TCLP only perform entailment check and does not allow answer merging (Schrijvers et al., 2008). To make it possible to combine answers for the join-based aggregates, we extended the predicate `new_answer/0` that is used by Mod TCLP to collect the stored answers. Fig. 4.5 shows the current implementation of `new_answer/0`, where lines 11 to 13 were added to make it possible to merge the answers `A` and `B` in a new answer `New` by returning `Res='$new'(New)`.

The tabling engine invokes `new_answer/0` to collect the answers of a generator. First, it retrieves `Ans`, the generator's answer table (line 2). Then, `store_projection/2` returns in `A` the representation of the aggregated argument `V` of the current answer. In lines 4 to 5 it retrieves one by one the previous answers B_i stored in the answer table and invokes `answer_compare/3` to compare/combine them. As we mentioned before there are three cases:

- if `Res == '=<'`, `A` entails a previous answer, and then it is discarded and the tabling engine continues the evaluation.
- if `Res == '>'`, a previous answer B_i entails `A` and B_i is removed (line 9). Then `answer_compare/3` backtracks to retrieve the next previous answer B_{i+1} (line 5).
- if `Res == '$new'(New)`, the answers `A` and `B` has been merged in `New`. The previous answer `B` is removed (line 12) and the merged answer `New` is stored (line 13).

Table 4.3: Encoding examples of non-lattice aggregates.

Aggregate	Code	
first or nt	`first(_,_):- true.`	
last	`last(_,_):- fail.`	`last(_,B,B).`
all solutions	`all(_,_):- fail.`	
threshold(Epsilon)	`threshold(Epsilon,A,B):- A < Epsilon*B.`	
addition	`add(_,_):- fail.`	`add(A,B,C):- C is A+B.`
multiplication	`mlt(_,_):- fail.`	`mlt(A,B,C):- C is A*B.`

Finally, for the cases where `member/2` (line 5) fails trying to reclaim a previous answer B_i (i.e., the answer store is empty or all the previous answers has been checked) then the current answer `A` is stored (line 15).

4.4 Non-Lattice Aggregates

We have presented aggregates that are defined over lattices where the entailment operation is reflexive, anti-symmetric, and transitive and the join operation is commutative, associative, and idempotent. However, there are common aggregates that cannot be expressed with operations on a lattice: counting answers, adding up numeric answers, and computing an average, among many others. As a consequence, their execution may not completely align with LFP semantics, but they can notwithstanding be implemented using ATCLP with the understanding that they will not obey some of the properties for lattice aggregates.

Table 4.3 shows some examples. The aggregate `first`, presented in YAP and equivalent to the mode `nt` (meaning *non tabled*) in B-Prolog, returns the first answer found, and the aggregate `last` returns the last answer found. These aggregates do not fit into a lattice structure because by definition they are sensitive to the solution order. Furthermore, the encoding of the join operation for `last/3` uses the fact that its second argument corresponds to the current answer, and the first argument corresponds to the aggregated value, which in this case is the last but one answer.

The *all answers* aggregate returns all the answers by stating that no answer is entailed by any other answer, e.g., by making the entailment check to always fail. This will keep answers that are redundant even modulo variable renaming and is useful to, for example, determine whether repeated answers are being generated, which may point to possible problems in the code. These would go unnoticed if the standard elimination of

```
edge(a,b,0.2).
edge(a,c,0.3).
edge(a,d,0.0002).
edge(b,d,0.0003).
edge(c,c,0.2).
edge(c,e,0.8).
```

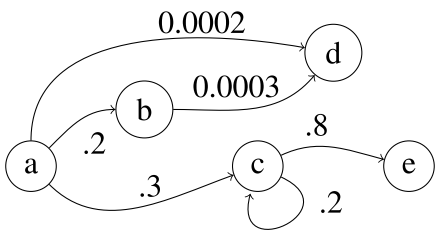


Figure 4.6: Graph for the random walk problem.

```
:- use_package(tclp_aggregates).

:- agg_join reach(_,sum).
reach(N,P) :- path(a,N,P).

sum(_,_) :- fail.
sum(A, B, C) :- C is A + B.
```

```
:- agg_entail path(_,_,thr(0.001)).
path(X,Y,P) :- edge(X,Y,P).
path(X,Y,P) :- edge(X,Z,P1),
               path(Z,Y,P2),
               P is P1 * P2.

thr(Epsilon, A, B) :- A < Epsilon * B.
```

Figure 4.7: Complete encoding for the random walk problem.

variant answers is used.

The `threshold` aggregate, parametrized by `Epsilon`, can be used to discard a value `A` whose relative value w.r.t. a value `B` falls below `Epsilon`. It does not define a lattice structure because it is not reflexive. Finally, `add` and `mlt`, whose names should be self-explanatory, are not aligned with the LFP semantics because their join operator is not idempotent and entailment is not defined (i.e, entailment should always fail since its semantics is based on multi-sets, rather than sets).

Example 4.3 (Random walk)**.** Let us see now an additional, more complex, but more interesting, example using non-lattice aggregates. Let us consider a possibly (cyclic) graph where each edge has a transition probability (Fig. 4.6). We want to compute the probability `P` of reaching a node `N` from some source node `a` considering random walks/paths from `a` to `N`. `P` comes from adding the transition probabilities of all possible paths from `a` to `N`. The probability of a path is computed as the multiplication of the probability associated to every edge in the path. On the other hand, in a cyclic graph we have an unbound number of different path of unbounded length (corresponding to traversing the cycles an unbound number of times). Note that the path probability decreases with every new edge added to the path, and therefore with every time a cycle is traversed.

A feasible approach is to *discard* paths when their contribution goes below a certain user-defined threshold. With a somewhat ad-hoc reading of this condition, we can say that new solutions with a difference small enough w.r.t. existing solutions *entail* these previous solutions and therefore they might not to be taken into account. This can be expressed in our framework by defining another aggregate that decides, via entailment, when further advancing in a path does not contribute enough.

To implement this approach, we use `add` to compute the addition of the different path probabilities and an entailment aggregate `threshold(Epsilon)` to determine whether paths should be added (if they differ from previously computed paths less than `Epsilon`).

Fig. 4.7 shows the ATCLP code. For each node `N`, `reach(_, sum)` aggregates in its second argument the sum of the transition probabilities of the paths from node `a` to node `N`. Note that we want to add all distances; therefore we define the entailment of `add` to be always false. The `threshold` aggregate, denoted by `thr(Epsilon)`, discards paths between `X` and `Y` whose relative contribution to the final result w.r.t. the contribution of previous path falls below `Epsilon=0.001`. The results for the query `?- reach(N,P)` are:

```
N = d, P = 0.00026 ? ;
N = e, P = 0.2999039999999999 ? ;
```

where the probability of ending the random walk at node `d`, `P=0.00026`, coincides with the expected probability (does not traverse cycles and no path has been discarded) while the probability of ending at `e` is an approximation of the correct value (`0.3`).

4.5 Experimental Evaluation

We will now evaluate the expressiveness and performance of ATCLP and we will compare ourselves with what arguably are the closest languages, e.g., classical Prolog and tabled Prolog. ATCLP is implemented as part of Ciao Prolog and it is available at `http://www.ciao-lang.org`. The examples and benchmarks presented in this chapter are available at `http://www.cliplab.org/papers/tplp2020-atclp/`. All the experiments were performed on a Mac OS X 10.13.6 laptop with a 2 GHz Intel Core i5 and all times are given in milliseconds.

Our first evaluation will use an implementation of the well-known minimax algorithm applied to (an extended version of) *TicTacToe*. Our starting point is the Prolog version from (Bratko, 2001), available at Appendix A.1 for the reader's convenience. It relies on `bagof/3` to collect all the possible movements from a position and selects the best one. The expressiveness of ATCLP makes it possible for the core minimax procedure (Fig. 4.8) to be considerably more compact than the equivalent Prolog code.

The ATCLP code selects the best movement by applying the `best` aggregate which discards movements with worst (resp., best, depending on the current player) score. Gathering solutions and keeping track of the best one so far is done transparently. Note

```
:- agg_entail minimax(_, first, best).
minimax(Pos, NextPos, (Pos,Val)) :-
   move(Pos, NextPos),
   minimax(NextPos, _, (NextPos,Val)).
minimax(Pos, Pos, (Pos,Val)) :-
   \+ move(Pos, _), utility(Pos,Val).

% Chose first best option
first(_,_) :- true.
% Minimizing
best( (Pos,ValA), (Pos,ValB) ) :-
    min_to_move(Pos),
    ValA =< ValB.
% Maximizing
best( (Pos,ValA), (Pos,ValB) ) :-
    max_to_move(Pos),
    ValA >= ValB.
```

Figure 4.8: Minimax algorithm in ATCLP used in a TicTacToe implementation.

Table 4.4: Run time (ms) and memory usage (between parentheses, in Mb) for TicTacToe.

	LP	tabling	ATCLP
3x3	1051	**167** (2)	359 **(1)**
4x4$^{(a)}$	$>$ 5 min	**10166** (130)	15194 **(30)**
4x4$^{(b)}$	$>$ 5 min	out of mem.	**134918** **(252)**

that we are using two aggregates in the same predicate: `best`, to minimize/maximize scores, and `first` to keep only the first solution found among those with the same score.[4]

We compared the execution time and memory usage in two scenarios: determining the best initial movement in a 3×3 board and determining the best movement in a 4×4 board, starting in two different positions. In all three cases the remaining game tree was completely explored. The results (Table 4.4) show that the Prolog version is the slowest, with the tabling version being faster than the ATCLP version for a couple of cases. However, the ATCLP version uses less table memory (between parentheses, in Mb). This is because viewing aggregates as constraints automatically stops the search as soon as the value of an aggregate is worse than a previously found one. That makes the ATCLP version to terminate for cases where regular tabling runs out of memory.

The second benchmark is the *Game* problem presented in the LP/CP contest of ICLP 2015 (`http://picat-lang.org/lp_cp_pc/Games.html`). The problem can be seen as a graph traversal where the movements represent a decision regarding whether to repeat the same game or play a different one. There are two parameters to optimize: `M`, the remaining money, and `F`, the fun we have had (which can be negative). The final goal is to have as much fun as possible, for which one has to keep as much money as possible. Fig. 4.9 shows the core of the algorithm, where we again want to stress its succinctness.

[4]Using `first` is up to some point an arbitrary decision, since there is no additional information to prefer one given movement among those tagged as best.

```
:- agg_entail total_fun(>=),
                reach(_,_,>=,>=).

total_fun(F) :-
    reach(initial,end,_,F).

reach(GameA,GameB,M,F) :-
    edge(GameA,GameB,M,F).
reach(GameA,GameB,Mf,Ff) :-
    reach(GameA,GameZ,M1,F1),
    edge(GameZ,GameB,M2,F2),
    Ff is F1 + F2,
    Mm is M1 + M2, Mm >= 0,
    ( cap(Cap), Mm > Cap
      -> Mf is Cap
       ; Mf is Mm         ).
```

Figure 4.9: Core algorithm for the *Game* benchmark.

We developed three comparable versions of a program to solve this problem using Prolog, tabling, and ATCLP. Table 4.5 shows that the ATCLP on-the-fly aggregate computation performs better than either Prolog or tabling, since ATCLP does not try to evaluate states where `M` and `F` are worse than in states already evaluated.

ATCLP performs better than tabling in *Games*, but this is not the case in *TicTacToe*. That is because in the tabled version of *TicTacToe* the use of `findall/3`, to aggregate the possible movements and select the best one, is sound, while in the tabled version of *Games* not. Note that the tabled version of *TicTacToe* (encoding available in Appendix A.1) memorizes the best movement by tabling `best/3`, and therefore, it behaves as the ATCLP version avoiding re-computations. However, in the tabled version of *Games* it is not possible to aggregate and memorize the more convenient intermediate states by using `findall/3` in `reach/4` because it interleaves recursive calls (encoding available in Appendix A.2). It is important to note that ATCLP can be used in any situation, but the correctness of using using `findall/3` under tabling depends on the program because tabling `findall/3` in the presence of recursion produces wrong results. Let us consider the program from (Swift et al., 2016):

```
:- table t/1.
t(X) :- findall(Y, t(Y), X).
t(0).
```

The query `?- t(X)`, which does not terminate in Prolog, terminates under tabling using Ciao and XSB, but it will return two answers: `X=[]` and `X=0`. This result is hard to defend semantically, because `X=[]` is not a valid answer and it is missing the answer generated by the first clause (which is infinite). Trying to avoid this wrong execution, Swi Prolog raises a "permission to append" runtime error inside tabled evaluations. XSB provides the predicate `tfindall/3` to call tabled predicates which, also in runtime, throws an error indicating that a call to `tfindall/3` is non-stratified.

Table 4.5: Run time (ms) comparison for *Game* with different scenarios.

	LP	**tabling**	**ATCLP**
game_data_01	8062.49	14.66	**2.89**
game_data_02	$>$ 5 min.	37.59	**4.87**
game_data_03	$>$ 5 min.	1071.26	**19.61**
game_data_04	$>$ 5 min.	4883.00	**23.21**

4.6 Discussion

We have presented a framework to implement aggregates defined over a lattice structure so that the returned values are representatives of a class of points in the lattice. We have also given them an intuitive meaning that makes answers to queries be consistent the usual least fixpoint semantics of logic programming. We provide a clean, easy-to-use interface so that final users can define the basic operations on which the aggregates are built. Notwithstanding, a library of common aggregates is provided with the implementation of the framework.

We validated the flexibility and expressiveness of our framework through several examples. We also evaluated their performance in a couple of benchmarks, which showed a positive balance between memory consumption and execution speed.

Chapter 5

Abstract Interpretation Fixpoint using Tabled CLP

In this chapter, we present a real application of Tabled Constraint Logic Programming using Mod TCLP and the extension introduced in the previous chapter to combine answers. We used Mod TCLP to adapt PLAI, the state-of-the-art abstract interpreter, and evaluate the benefits of tabling w.r.t. Prolog. Note that in 1987 it was already shown the relationship between tabling and abstract interpretation. PLAI is included in CiaoPP, an analyzer and optimizer suite for logic programs, part of the Ciao development environment. We preserved the interface of PLAI with the rest of the system to make it possible a fear comparison. We evaluated the performance by analysing several programs using different abstract domains and the complexity in terms of lines of code. This is, to our best knowledge, the first comparison with these characteristics. In the adapted version using Mod TCLP, the tabling engine drives the fixpoint computation using semantic equivalence*, and the TCLP interface invokes the* LUB *operator of the abstract domains to combine the abstract substitutions. That reduces the lines of code to one third and improves the performance in most of the benchmarks.*

Abstract interpretation is a theory for approximation of the semantics of programs. The semantics of a logic program is its least Herbrand model, and it defines the set of atoms that are logical consequences of the program. The key idea of abstract interpretation is to over-approximate the execution of a program using an abstraction of the concrete semantics of the program. Abstract interpretation has always been seen as one of the most clear applications of tabled logic programming. It requires a fixpoint procedure, often implemented using memo tables and dependency tracking, which play a role very similar to the internal data structures that tabling engines need to detect repeated calls, store and reuse answers, and check for termination.

As we mentioned in Chapter 4, the integration of tabling with constraint solvers has been proposed to improve abstract interpreters by invoking the constraint engine to aggregate the abstract substitutions of different clauses, but the relationship between abstract interpretation and tabling was recognized very early. *Extension tables* (Dietrich, 1987) were proposed to record results from the execution of predicates and turn intensional definitions into extensional definitions. Their applications included "improving the termination and completeness characteristics of depth-first evaluation strategies in the presence of recursion". The idea of extension tables were applied as the embryo of SLG resolution and the XSB system. At the same time, abstract interpretation was then viewed as inefficient, and as part of the efforts to make it a practical technique to implement analyzers, tables, but also other ideas such as dependency tracking, were used (Warren et al., 1988), thus making it clear that a common underlying technology could be used in both types of systems. These components, independently available in tabling systems, were used to build abstract interpreters:

- The possibilities offered by OLDT (Tamaki and Sato, 1986) were used in (Kanamori and Kawamura, 1993) to explore its application in abstract interpretation. Using type inference as the guiding example, it suggests certain changes to OLDT and concludes that it is feasible to do abstract interpretation with OLDT. The paper neither describes an implementation nor reports performance, but it states that the abstract interpreter was implemented and was available.
- In (Warren, 1999) an abstract interpreter written in XSB is presented as one of the applications of tabled Prolog.
- Other abstract interpreters has been used as a benchmark to compare different implementations and/or scheduling strategies of tabling (Demoen and Sagonas, 1998; Freire et al., 2001).
- Advanced tabled systems and techniques have been proposed to implement more efficient abstract interpreters by using the *least upper bound* operator (Schrijvers et al., 2008) to combine answers, numeric constraint solvers (Chico de Guzmán et al., 2012) to implement the Octagon domain, and the *partial order answer subsumption with abstraction* (Swift and Warren, 2010) for cases where, e.g., the program computed does not have a finite model.

However, surprisingly none of them reports performance evaluation w.r.t. implementations without tabling. To the best of our knowledge, the only one is a framework (Janssens and Sagonas, 1998), based on abstract compilation, that executes the abstract version of the program under analysis, together with domain-dependent abstract operations, which is evaluated using the tabling system XSB and compared with the AMAI and PLAI systems (Janssens et al., 1995; Muthukumar and Hermenegildo, 1992). Both systems use abstract interpreters written in Prolog without tabling, but they rely on very different underlying technologies, and with different representations for the abstract domains. From that evaluation, the paper concludes that tabling is a viable infrastructure for abstract interpretation, but concedes that the PLAI fixpoint algorithm was the most efficient abstract interpreter for logic programming available at the moment. The very different underlying infrastructure makes it difficult to use these results to draw meaningful conclusions.

In this chapter we use Mod TCLP to exploit the synergy of the integration of tabling and constraint solvers in abstract interpretation to adapt PLAI. PLAI is the fixpoint algorithm implemented in the program analysis, optimization, and transformation tool CiaoPP (Hermenegildo et al., 2005, 2012), available at `www.ciao-lang.org`. The resulting re-implementation preserves the interface with the rest of CiaoPP in order to compare some indicators of code complexity (e.g., comparing lines of code, with the assumption that the tabled version is essentially a subset of the original version) and performance on a completely equal footing. This is, to our best knowledge, the first comparison that has these characteristics.

5.1 The PLAI algorithm

We assume that the reader is familiar with the basic principles of abstract interpretation (Bruynooghe, 1991; Cousot and Cousot, 1977; Nielson et al., 2005). The PLAI algorithm used by the abstract interpreter of CiaoPP for static analysis extends the fixpoint algorithms proposed by (Bruynooghe, 1991) with the optimizations described in (Muthukumar and Hermenegildo, 1990). In logic programming, all possible concrete substitutions in the program (i.e., terms to which the variables in that program will be bound at run-time for a given query) can be infinite, which gives rise to an infinite execution tree. The core idea of PLAI is to represent this infinite execution tree by an abstract and-or tree using abstract substitutions to finitely represent the possibly infinite sets of substitutions in the concrete domain. The set of all possible abstract substitutions that a variable can be bound to is the *abstract domain* which is usually a complete lattice (or a complete partial order of finite height).

5.1.1 Domains in PLAI

PLAI is domain-independent: new abstract domains can be easily implemented and integrated by using a common interface. The operations required by the interface are:

- $\lambda' \sqcup \lambda''$, which gives the LUB of the abstract substitutions λ' and λ''. The LUB operation is defined in terms of the $\sqsubseteq$ relation of the abstract domain.

- `call_to_entry(p(`$\vec{u}$`),C,`λ`)`, where `C` is a clause and `p(`$\vec{u}$`)` is a call. It gives an abstract substitution describing the effects on `vars(C)` of unifying `p(`$\vec{u}$`)` with `head(C)` given an abstract substitution λ for the variables in $\vec{u}$.

- `exit_to_success(`λ`, p(`$\vec{u}$`), C, `β`)` which returns an abstract substitution describing the effect of execution `p(`$\vec{u}$`)` against clause `C`. For this, the variables of the abstract substitution β are renamed taking into account the unification with the terms in `head(C)` and the variables in `p(`$\vec{u}$`)`, and a new abstract substitution is returned updating λ with the new information.

- `extend(`λ`,`λ'`)` which extends abstract substitution λ to incorporate the information in λ' in a way that it is still consistent.

- `project_in(`$\vec{u}$`,`λ`)` which extends the abstract substitution λ so that it refers to all the variables in $\vec{u}$.

- `project_out(`$\vec{u}$`,`λ`)` which restricts the abstract substitution λ to refer only to the variables in $\vec{u}$.

For additional examples of abstract domains integrated in CiaoPP, we refer the reader to (Bueno et al., 2004; Muthukumar and Hermenegildo, 1989; Vaucheret and Bueno, 2002).

5.1.2 And-Or trees and substitutions

In PLAI, the abstract and-or tree is constructed using a top-down driven strategy (instead of a bottom-up computation) so that the computation is restricted to what is required for the given query. In the resulting and-or tree, an *and-node* is a clause head `h` whose children are the literals in its body, $\mathtt{p}_1,\ldots,\mathtt{p}_n$, and an *or-node* is a literal, $\mathtt{p}_i$, whose children are the heads $\mathtt{h}_1,\ldots,\mathtt{h}_m$ of the clauses that unify with $\mathtt{p}_i$. Its construction starts with the abstract call substitution for the query. Then, abstract substitutions at all points of the abstract and-or tree are computed and finally, the success substitution for the query is computed.

Inside a clause, abstract substitutions at every point are denoted depending on their position among its literals. Given a clause $\mathtt{h}$ `:-` $\mathtt{p}_1,\ldots,\mathtt{p}_n$, let λ_i and λ_{i+1} be the abstract substitutions to the left and right of the subgoal p_i, $1 \leq i \leq n$. Then, λ_i and λ_{i+1} are, respectively, the *abstract call substitution* and the *abstract success substitution* for the subgoal p_i. The projection of λ_1 on *vars*(`h`) is the *abstract entry substitution*, β_{entry}, of the given clause, and, similarly, the projection of λ_{n+1} on *vars*(`h`) is its *abstract exit substitution*, β_{exit}. The abstract substitutions for a clause are computed as follows:

Algorithm 1: `entry_to_exit`: Compute exit substitution from entry substitution.

```
Data: A clause C of the form h($\vec{u}$):-p_1($\vec{u}_1$),...,p_m($\vec{u}_m$); an entry substitution $\beta_{entry}$
Result: An exit substitution $\beta_{exit}$
$\lambda_1$ := project_in($vars(C)$,$\beta_{entry}$)
for i := 1 to m do
    $\lambda_{i+1}$ := call_to_success(p_i($\vec{u}_i$),$\lambda_i$)
end
return project_out($\vec{u}$,$\lambda_{m+1}$)
```

Algorithm 2: `call_to_success`: Compute success substitution from call substitution.

```
Data: A goal p($\vec{u}$); an abstract call substitution $\lambda_{call}$
Result: A success substitution $\lambda_{success}$
$\lambda_{proj}$ := project_out($\vec{u}$,$\lambda_{call}$)
$\lambda'$ := $\perp$
for each clause C which unifies with p($\vec{u}$) do
    $\beta_{exit}$ := entry_to_exit(C, call_to_entry(p($\vec{u}$),C,$\lambda_{proj}$))
    $\lambda'$ := $\lambda' \sqcup$ exit_to_success($\lambda_{proj}$,p($\vec{u}$),C,$\beta_{exit}$)
end
return extend($\lambda_{call}$,$\lambda'$)
```

- Exit substitution from the entry substitution (Algorithm 1): Given a clause `h :- p`$_1,\ldots,$`p`$_n$ and an entry substitution β_{entry} for the clause head h, the call substitution λ_1 for p_1 is computed by simply adding to β_{entry} an abstraction for the variables in the clause that do not appear in the head. The success substitution for p_1 is λ_2, and it is computed as explained below (essentially, by repeating this same process for the clauses which unify with p_1). $\lambda_3,\ldots,\lambda_{n+1}$ are computed similarly. The exit substitution β_{exit} for this clause is the projection of λ_{n+1} onto $\vec{u}$, the variables in h.

- Success substitution from the call substitution (Algorithm 2): Given a call substitution λ_{call} for a subgoal p, let $h_1,\ldots,h_m$ be the heads of clauses that unify with p. Compute the entry substitutions $\beta 1_{entry},\ldots,\beta m_{entry}$ for these clauses. Compute their exit substitutions $\beta 1_{exit},\ldots,\beta m_{exit}$ as explained above. Compute the success substitutions $\lambda 1_{success},\ldots,\lambda m_{success}$ from the exit substitutions corresponding to these clauses. At this point, all different success substitutions can be considered for the rest of the analysis, or a single success substitution $\lambda_{success}$ for subgoal p computed by means of an aggregation operation for $\lambda 1_{success},\ldots,\lambda m_{success}$. This aggregate is the least upper bound (LUB), denoted by $\sqcup$, of the abstract domain.

Note that these two procedures are mutually recursive and would not finish in case of mutually recursive calls. They merely describe how abstract substitutions are generated for the case of literals in a body (by carrying success abstract substitutions to call abstract substitutions) and how entry and exit substitutions of several clauses are composed together. For the general case of recursive predicates, where repeated calls and termination have to be detected, PLAI implements a fixpoint algorithm that we sketch below.

5.1.3 PLAI's fix point algorithm

The core idea of PLAI's fixpoint algorithm (Muthukumar and Hermenegildo, 1990) is that the subtree corresponding to the abstract interpretation of a node with a recursive predicate p should be finite. If the abstract domain is finite, a predicate p can only have a finite number of distinct call substitutions and therefore the subtree can only have a finite number of occurrences of nodes that have a variant of p and which themselves have subtrees. In addition to that, all other nodes in the subtree with the same predicate name p and with the same call substitutions (modulo variable renaming) use the approximate value of the success substitution computed previously for the root node of the subtree labeled with p, and hence they do not have any descendent nodes.

Based on this idea, the fixpoint algorithm iteratively refines the approximate values of the success substitution of the recursive predicate p as follows:

- First, it computes an approximate value of the projected success substitution using the LUB of the projected success substitutions corresponding to the **non-recursive** clauses of p. This provides an initial, hopefully non-empty, abstract substitution that is fast to compute (it does not need to check for repeated calls or termination) and accelerates the convergence of the fixpoint algorithm. In practice, it can be delegated to a specialized version of Algorithms 1 and 2 restricted to non-recursive calls / clauses. These can be determined beforehand by a reachability analysis based on strongly connected components.

- Then, it traverses the (finite) subtree corresponding to p in a depth-first fashion. When an entry-exit combination is needed for a call to p having the same call substitution (modulo variable renaming), the existing approximation is used. For a call to p with a different call substitution, a new (nested) fixpoint computation is started. When the analysis returns to the root of the subtree, the success substitution for p is updated as the LUB of the previous value and the value just computed from the recursive clauses of p.

- If there is a change in the success substitution for p, the depth-first traversal is restarted using the new success substitution, which is used for the subtree nodes corresponding to p that have a compatible call substitution. These depth-first traversal iterations can take place only a bounded number of times, since the

LUB operation is monotonic and the abstract substitutions form a lattice of finite height.[1] Therefore, a fixpoint will be reached in a finite number of steps.

- If there is no change in the success substitution for the root node of the subtree of `p` for a given call substitution, then the analysis of that subtree is complete (for that call substitution) and the fixpoint computation of the predicate `p` terminates.

For recursive predicates called from within recursive predicates, the dependencies between nested calls have to be recorded to restart the traversal of the subtrees containing predicate calls whose success substitution has been updated.

5.2 Implementations of the PLAI Algorithm: Prolog vs. Tabling

We will now describe more in depth how the PLAI algorithm is implemented in CiaoPP (sketch available in Appendix B.2) and highlight the differences w.r.t. the version that uses Tabled CLP (sketch available in Appendix B.1).

5.2.1 PLAI in CiaoPP

The implementation of `call_to_success` is the entry point, as it relates the entry and exit substitutions of a call (in particular, of the top-level call). During the analysis of a goal $\texttt{p}(\vec{u})$, and for each clause that unifies with $\texttt{p}(\vec{u})$, the predicate `call_to_success` invokes `entry_to_exit` which, for each subgoal in the body of the clause, invokes again `call_to_success`. The abstract interpreter is able to stop the evaluation of a part of the program and move to another part to evaluate calls to other predicates. The implementation of PLAI is optimized to accelerate the convergence of the fixpoint and reduce the computation by reusing previous results, among other techniques.

The PLAI algorithm is based on the construction of an and-or tree, described in Section 5.1, with the nodes representing the predicate calls visited during the analysis. To construct this tree, `call_to_success` identifies each goal with its corresponding and/or node and with the specialized version of its father (i.e., the version of the literal that originated the call) and carries around a list with the nodes on which the current goal depends. The analysis starts with a query (a goal) and a call substitution. With this information, `call_to_success` creates the root node of the tree and the list of clauses

[1]While it is true that abstract domains can be infinite, if convergence is not reached after some time, a widening operation changes the representation of the abstract substitutions to a coarser domain that has more chances to converge (or is sure to converge, if it is finite).

that unify with the goal. If the goal corresponds to a non-recursive predicate, it computes the success substitution which is asserted in a memo-table to reuse the result later on. Otherwise, the goal corresponds to a recursive predicate and it is dealt with by the fixpoint algorithm: first, it evaluates the non-recursive clauses obtaining an approximation of the success substitution and, after this, it starts the fixpoint computation.

During the fixpoint computation, for a goal with a given call substitution:

- If complete information has been already inferred and saved, `call_to_success` reuses it, to avoid re-computations.
- If it is already inside a fixpoint computation (some parent started a fixpoint with the same call), `call_to_success` reuses the approximation stored for this call, to avoid entering loops.
- If an analyzed call depends on other nodes whose fixpoint are not completed yet, two cases are treated:
  - If the information on which the predicate depends is updated, a local fixpoint computation is started.
  - Otherwise, nothing is done.

  To decide whether updated information for a node is available, the information inferred for it has a version number:
  - When the information on a node is updated, its version number is increased by one.
  - When a node uses information from another node, it stores the version of that information in the list of nodes on which it depends.

  Version numbers are used to detect updates of the information on which a node analysis depend. If the version number of the last information used from a node does not match its current version number, there has been an update that needs to be propagated.

When the fixpoint computation finishes and the list of dependent nodes is empty, the current information for this call is asserted. Otherwise, if this list is not empty, the information remains flagged as an approximation and the fixpoint restarts. As it can easily be seen, while the algorithm can be conceptually not too complex, its implementation is cumbersome and at points costly, since many interactions are done through the database using identifiers for program points.

5.2.2 The PLAI Algorithm in TCLP

The PLAI code using tabling is a simplification of the corresponding Prolog implementation. The main points that were changed are:

- The handling of dependencies among nodes and the detection of termination in the fixpoint computation, that were explicit in the Prolog version, are now transferred to the underlying fixpoint of the tabling engine.
- The calculation of the LUB of the abstract substitutions generated by different clauses unifying with a call is done via lattice-based constraint aggregation (which is in turn built upon tabling).

5.2.2.1 Internal Database and Dependencies

In the Prolog implementation, the information related to the abstract substitutions is kept in a dynamic database relating code, program points, entry/exit substitutions, and dependencies. This makes it globally accessible and allows it to survive across backtracking and calls, so that it does not need to be carried around the program and be rebuilt every time there is a change in the substitution at a program point.

However, making the abstract interpreter update that information, switch among calls, and re-analyze calls needs accessing and updating this database, which is costly and mixes declarative and imperative styles. On top of that, the CiaoPP implementation has been fine-tuned during many years to avoid unnecessary (re-)analyses and minimize the overhead of accessing the database. All of these optimizations cause the code to have to deal with specific cases for the sake of performance, hence adding to its complexity. But despite the involved implementation, this machinery mimics, at Prolog level, an infrastructure similar to a tabling engine, but specialized for a given program —the abstract interpreter— and with optimizations specific for the task at hand.

This bookkeeping becomes unnecessary when using a tabling-based implementation. An abstract interpreter written using tabling and equipped with the capability to detect when two syntactically different substitutions represent the same object, can automatically take care of termination, suspend analysis when repeated calls are detected, and resume them when new information is available — all of it as part of the normal execution of a tabled program, without having to explicitly update and check dependencies.

That makes the code much simpler (no dependencies, lists of pending goals, resuming, etc. need to be explicitly coded) and shorter (we have obtained a threefold reduction in code size). On the other hand, the tabling engine is generic and cannot decide which suspension and/or resumption policy is better for a particular application. We on purpose chose to (a) keep the TCLP code simple and not include any specific heuristic

```
call_to_success(SgKey,Call,Proj,Sg,Sv,AbsInt,Succ) :-
        call_to_success_fixpoint(SgKey,Sg, st(Sv,Call,Proj,AbsInt,Prime) ),
        each_extend(Sg,Prime,AbsInt,Sv,Call,Succ).

:- use_package(tclp_aggregate).
:- table call_to_success_fixpoint(_,_,abst_lub).
call_to_success_fixpoint(SgKey,Sg, st(Sv,Call,Proj,AbsInt,Prime) ) :-
        trans_clause(SgKey,_,Clause),
        do_nr_cl(Clause,Sg,Sv,Call,Proj,AbsInt,Prime).
call_to_success_fixpoint(SgKey,Sg, st(Sv,_Call,Proj,AbsInt,Prime) ) :-
        \+ trans_clause(SgKey,_,_),
        apply_trusted0(Proj,SgKey,Sg,Sv,AbsInt,_ClId,Prime).
```

Figure 5.1: Implementation of `call_to_success/7` under the TCLP framework

in the code, (b) not to reimplement an analyzer from scratch, but simplify existing code, and (c) keep exactly the same interfaces (both those offered to the rest of CiaoPP and those required by the fixpoint code) so that the TCLP-based abstract interpreter can interoperate with the rest of the CiaoPP machinery as a drop–in replacement with close to zero effort. For these and other reasons, our performance figures (Section 5.3) are a lower bound of what could be achieved.

As an example, the implementation of `call_to_success/13` in Prolog checks several cases: if the call being analyzed is complete, under evaluation in a fixpoint, a call to a recursive predicate, a call to a non-recursive predicate, etc. to update information accordingly. It eventually invokes `proj_to_prime_nr/9`, which starts the fixpoint computation itself, and which recursively calls `call_to_success/13`. `call_to_success/13` has eight clauses and `proj_to_prime_nr/9` has six clauses (see Appendix B.2 or the corresponding file at `http://www.cliplab.org/papers/tclp-plai-iclp2019`).

In the tabling implementation, the underlying engine and the calls to the abstract domain operations through the constraint solver interface take care of these cases and dependencies. This makes the implementation of `call_to_success` have just **one** clause (Fig. 5.1). The counterpart to `proj_to_prime_nr/9` (which we renamed `call_to_success_fixpoint/3` for clarity) has just two clauses: one for user predicates and another one for library and builtin predicates.

Additionally, the use of tabling makes it unnecessary to save explicitly all the intermediate substitutions, database identifiers for calls and program points, dependencies among goals, etc. This reduces the number of arguments, and `call_to_success` went from thirteen used in Prolog:

```
call_to_success(RFlag,SgKey,Call,Proj,Sg,Sv,AbsInt,ClId,Succ,List,F,N,Id)
```

to seven in the tabling-based implementation:

```
call_to_success(SgKey,Call,Proj,Sg,Sv,AbsInt,Succ)
```

```
call_entail(abst_lub, st(Sv,_,ProjA,AbsInt,_), st(Sv,_,ProjB,AbsInt,_) ) :-
        identical_abstract(AbsInt,ProjA,ProjB).
answer_entail(abst_lub, st(Sv,_,_,AbsInt,PrimeA), st(Sv,_,_,AbsInt,PrimeB) ) :-
        less_or_equal(AbsInt,PrimeA,PrimeB).
answer_join(abst_lub,st(Sv,_,_,Abs, A), st(Sv,_,_,Abs, B), st(Sv,_,_,Abs,New) ) :-
        compute_lub(Abs,[A,B],New).
apply_answer(abst_lub, st(Sv,_,_,AbsInt,Prime), st(Sv,_,_,AbsInt,Prime) ).
```

Figure 5.2: Code of the operator `abst_lub` under the TCLP framework

5.2.2.2 Deciding Termination and Computing the LUB

In the PLAI algorithm, the different exit substitutions obtained from the clauses that unify with a given call are combined using the LUB operator of the abstract domain (Algorithm 2): exit substitutions $\beta_{i\,exit}$, for every clause C_i are joined to return the success substitution $\lambda_{success}$.

The CiaoPP implementation uses `bagof/3` to collect all the clauses in a list and then traverses it and analyzes every clause to create another list of abstract substitutions that are joined with the LUB. This processing is conceptually simple, but its implementation obscures the code with low-level operations, does not match the idea of having an interpreter executing on an abstract domain, and requires database accesses to retrieve the substitution applicable at that point.

In our implementation, the use of lattice-based aggregates with the tabling engine (see Chapter 4) simplifies the code. The `abst_lub` identifier in line 6 of Fig. 5.1 is the name of an interface that has several missions: determine suspension of calls, detect termination of the fixpoint, and perform aggregation of abstract substitutions. In the same line, the underscores state that the corresponding arguments are to be checked for equality (necessary to decide whether a fixpoint has been reached) using the *variant* policy, i.e., syntactical equality modulo variable renaming.

The implementation of the interface named `abst_lub` in Fig. 5.2 tells the tabling engine how to treat the argument selected previously with this identifier. In particular, the tabling engine checks the corresponding arguments for equality by calling `call_entail/3`. In our case, two abstract substitutions are termed equal if the abstract domain implementation (`identical_abstract/3`) decides so. This makes it possible to detect that two different representations correspond to the same object in the lattice and, if so, suspend a call or retrieve saved answers for it.

The code in Fig. 5.2 also aggregates the results returned in the third argument (the abstract substitutions) by joining them with the LUB of the lattice. The tabling engine calls `answer_entail/3` to decide whether a new answer (a substitution) is or not more general than an existing answer (`less_or_equal/3`). If its not comparable, `answer_join/4` (which in turn invokes `compute_lub/3`) is called to compute the LUB of a previous answer and the new one. With these definitions, lines 7 to 12 in Fig. 5.1 contain **all** the code necessary to return the exit substitution of a call w.r.t. all its matching clauses. The implementation of the LUB operation (`abs_lub`, Fig. 5.2) is based on the operations provided by the abstract domain implementation.

This code also performs an incremental computation of the LUB as follows: upon success, the first answer, corresponding to the exit substitution $\beta 1_{exit}$, is stored in the answer table of the tabled predicate. Let us call this stored answer β_{exit}. For the subsequent exit substitutions $\beta\, i_{exit}, i > 1$, there are two possible cases: if the saved substitution is more general ($\beta\, i_{exit} \sqsubseteq \beta_{exit}$), then $\beta\, i_{exit}$ is discarded; otherwise we make $\beta_{exit} = \beta_{exit} \sqcup \beta\, i_{exit}$.

5.2.2.3 Connecting Abstract Substitutions with Lattice-Based Aggregates

The TCLP system handles entailment, aggregation, etc. by delegating operations to an underlying constraint solver using a fixed interface (see Chapter 3). Since we purposely did not change the representation of the CiaoPP abstract domains (they are used in other parts of the system), we constructed a bridge between these abstract domains and the interface that TCLP expects.

The original entry point of the fixpoint, `proj_to_prime_nr/9` (renamed as `call_to_success_fixpoint/3` in the TCLP implementation), now tabled, is automatically rewritten (by the package `tclp_aggregate`) to call an auxiliary predicate that, at run time, substitutes the arguments carrying abstract substitutions by attributed variables (Holzbaur, 1992) that simulate having a constrained variable. Their attributes are tuples that contain (a) the identifier (`abst_lub`, in our example) that determines the interface to be used and (b) the abstract substitution and ancillary information necessary by the abstract interpreter.

When one operation of the tabling engine involves a call with attributed variables, the engine checks if it has an attribute with contents it recognizes. If so, it calls the corresponding predicate from the interface that, in our case, operates on the substitution stored in the attributes.

5.3 Evaluation

Besides simplifying code, the implementation of PLAI using TCLP gives performance advantages in many cases. These come mainly because part of the bookkeeping related to dependencies, saving the analysis state when restarting the analysis of a dependent call, checking for termination, etc. are handled at a lower level. On the other hand, the implementation currently in CiaoPP, as commented before, has been fine-tuned and specialized during many years to minimize the overhead of the fixpoint implementation, so that a large proportion of the analysis time is spent in abstract domain-related operations. On top of that, the CiaoPP domain representation and abstract domain operations are designed to work well with its current architecture and coding decisions (e.g. saving and retrieving from the dynamic databases) and are suboptimal for a tabling-based implementation: for example, redundant data is manipulated and/or stored. As commented earlier, we did not change any of these so the TCLP fixpoint can seamlessly interact with the rest of the CiaoPP tool, exposing and using exactly the same interfaces.

Even with these constraints, we observed speedups when analyzing most programs from a benchmark set. We used the *Groundness* and *Sharing+Freeness* (Muthukumar and Hermenegildo, 1991) domains due to their relevance (e.g., for program optimization and correctness of parallelization). *Groundness* (see Table 5.1 for performance results) determines if some program variable will be bound to a ground term. This is useful to derive modes, optimize unification, and improve the precision of the *Sharing+Freeness* analysis, among others.

Sharing+Freeness (see Table 5.2) determines if two (or more) program variables may be bound to terms sharing a common variable. It is useful to determine, for example, whether running two goals in parallel may try to bind the same variable, thus causing races and compromising correctness. The benchmarks used are standard programs that have been previously used to evaluate CiaoPP.

All the experiments in this chapter were performed on a Linux 5.0.0-13-generic machine with an Intel Core i7 at 1.80GHz with 16Gb of memory and using `gcc 8.3.0` to compile the abstract machine of Ciao Prolog. In all cases, every program was analyzed 40 times and the 10 worst times were discarded, both when using the tabling and the Prolog implementation, to try to minimize the effect of spurious interruptions, O.S. scheduling, etc. that can introduce noise in the execution. The remaining times were averaged. All the code and the system under evaluation is available at `http://www.cliplab.org/papers/tclp-plai-iclp2019`.

The average speedups in each table were calculated by adding up the (averaged) execution times for all the benchmarks and dividing the *Prolog* time by the *TCLP* time. This shows that, on average, the analysis with the *Groundness* domain speeds up a bit more than 30%, while the analysis with the *Sharing+Freeness* has experienced, on average, a slight slowdown (about 3%).

By looking at every benchmark in isolation, we can observe that the speedups differ greatly among them. We have sorted the benchmarks according to the speedup to appreciate better the differences. In both cases, only a small part of the benchmarks (three) experienced a slowdown, and even in these cases, the maximum slowdown was about 10%. In the case of *Sharing+Freeness*, the slowest analysis corresponded as well to the largest execution time (larger than the rest of the benchmarks combined). We want to note that this benchmark (zebra) is probably not a representative of a typical

Table 5.1: Performance comparison: CiaoPP fixpoint in Prolog and TCLP (*Groundness* domain).

	Speedup	TCLP (ms)	Prolog (ms)
fibf_alt	1.60	**0.29**	0.46
aiakl	1.56	**2.45**	3.82
boyer	1.50	**7.31**	10.97
pv_queen	1.46	**0.74**	1.07
subst	1.41	**0.25**	0.35
pv_gabriel	1.37	**3.65**	4.99
rdtok	1.32	**7.03**	9.25
mmatf	1.24	**0.31**	0.39
hanoi	1.22	**0.53**	0.65
revf_lin	1.20	**0.27**	0.32
append	1.20	**0.17**	0.20
rev_lin	1.19	**0.26**	0.31
prefix	1.16	**0.27**	0.31
revf	1.15	**0.32**	0.37
pv_plan	1.15	**1.94**	2.23
sublist_app	1.14	**0.24**	0.27
reverse	1.14	**0.38**	0.43
flatten	1.13	**0.55**	0.62
palindro	1.12	**0.34**	0.38
fact	1.08	**0.25**	0.27
rotate	1.06	**0.46**	0.49
maxtree	0.98	0.63	**0.61**
zebra	0.92	1.38	**1.26**
browse	0.89	1.76	**1.57**
AVG	1.31	31.78	41.59

Table 5.2: Performance comparison: CiaoPP fixpoint in Prolog and TCLP (*Sh*+*Fr* domain).

	Speedup	TCLP (ms)	Prolog (ms)
fact	1.30	**0.26**	0.33
pv_queen	1.23	**1.21**	1.49
mmatf	1.17	**0.51**	0.60
mmatrix	1.15	**0.53**	0.61
prefix	1.14	**0.46**	0.52
revf	1.12	**0.47**	0.53
revf_lin	1.10	**0.39**	0.43
reverse	1.10	**0.39**	0.43
rev_lin	1.10	**0.38**	0.42
rotate	1.06	**0.72**	0.76
pv_pg	1.01	**2.67**	2.70
append	0.98	1.11	**1.09**
sublist_app	0.96	0.87	**0.84**
zebra	0.91	16.34	**14.80**
AVG	0.97	26.31	25.55

program, as it is a combinatorial problem with many free variables in a single clause, some of which are aliased with each other.

The source of the speed difference is not easy to determine. A profile of the number of fixpoint calls in Prolog vs. fixpoint calls, entailment checks, joins, etc. in the TCLP version does not seem to show a correlation with the observed speedups. We therefore conjecture that the shape and size of the abstract substitution, and the relative cost of checking entailment, has to be explored to have a better explanation of the differences observed.

5.4 Discussion

We have presented a re-implementation of PLAI, a fixpoint computation algorithm for abstract interpretation, using tabled constraint logic programming. The resulting code is considerably shorter than the current Prolog implementation of PLAI in CiaoPP (one-third of its size) and much simpler: all the bookkeeping necessary to keep track of dependencies between predicates, analysis restarting, etc. is in charge of the tabling engine, which increases the maintainability of the implementation of PLAI.

We have evaluated its performance using several benchmarks and abstract domains, and compared it with the original implementation in CiaoPP. In most cases, the TCLP implementation showed improved performance, sometimes with a speedup of 60%. In a few cases there was a small slowdown, which we think is a reasonable price to pay for the added code clarity, especially taking into account that there is room for improvement in the current implementation.

Among the immediate future plans, we want to experiment re-implementing the abstract domains with an optimized representation of the abstract substitutions, and also use constraint logic programming techniques to propagate the effects of updates. We also expect that, using constraints, we will be able to define widening heuristics independently of the fixpoint algorithm thereby increasing the resulting flexibility, precision and performance w.r.t. the state of the art.

Acknowledgements

We would like to thank Maximiliano Klemen, who helped us understand the intricacies of the CiaoPP implementation of PLAI. Thanks are also due to Manuel Hermenegildo, who gave us very valuable feedback on the manuscript and also a historical account on the early relationship between tabling and efficient abstract interpreters.

Part II

Constraint Answer Set Programming

Chapter 6

Constraint Answer Set Programming without Grounding

Constraint Answer Set Programming is a promising paradigm thanks to its ability to incorporate non-monotonic reasoning. Most of the proposed CASP systems require a grounding phase that removes the variables and the links among them and also causes a combinatorial explosion in the size of the program. In this chapter, we present s(CASP), a goal-directed non-monotonic reasoner whose top-down execution strategy avoids the grounding phase and computes (partial) stable models of CASP programs while retaining the logical and constrained variables during the execution and in the answer sets. It is implemented in Ciao Prolog and its interpreter lets Prolog take care of all operations that it can handle natively, especially those involving constraints. We designed a generic interface to plug-in different constraint domains, and a generic forall *algorithm to evaluate goals with variables constrained under arbitrary constraint domains such as a new solver for disequality constrains and CLP($\mathbb{Q}$/$\mathbb{R}$). We show through several examples its enhanced expressiveness and improved performance w.r.t. state-of-the-art (C)ASP and (C)LP systems.*

As we explained in Section 1.2.2, incorporating constraint in Answer Set Programming systems is not straightforward because most of them require a grounding phase. During the grounding phase of the programs, the variables are grounded and, therefore, the constrains linking them disappear. The proposals to work around this issue require explicit hooks in the language, limit the range of admissible constraint domains the places where constraints can appear, and the type (or number) of models that can be returned. On the other hand, there are top-down execution models for ASP, such as s(ASP) (Marple et al., 2017a), a goal-directed SLD resolution-like procedure. s(ASP) evaluates programs under the ASP semantics without a grounding phase either before or during execution. Additionally, s(ASP) supports predicates and thus retains logical variables both during execution and in the answer sets.

In this chapter we propose the integration of ASP with constraint by incorporating constraints into the s(ASP) execution model. We have extended s(ASP)'s execution model to make its integration with generic constraint solvers possible. The resulting execution model and system, called s(CASP), makes it possible to express constraints on variables and extends s(ASP)'s in the same way that CLP extends Prolog's execution model. Thus, s(CASP) inherits and generalizes the execution model of s(ASP) while remaining parametric w.r.t. the constraint solver. Due to its basis in s(ASP), s(CASP) avoids grounding the program and the concomitant combinatorial explosion. s(CASP) can also handle answer set programs that manipulate arbitrary data structures as well as reals, rationals, etc.

The s(CASP) system has been implemented in Ciao Prolog by integrating Holzbaur's CLP($\mathbb{Q}$) (Holzbaur, 1995), a linear constraint solver over the rationals.[1] To validate the advantages of s(CASP) we used it to solve a series of problems that would cause infinite recursion in other top-down systems, but which in s(CASP) finitely finish, as well as others that require constraints over dense and/or unbound domains. Thus, s(CASP) is able to solve problems that cannot be straightforwardly solved in other systems. We show, through several examples, its enhanced expressiveness w.r.t. ASP, CLP, and other ASP systems featuring constraints. We briefly discuss s(CASP)'s efficiency: on some benchmarks it can outperform mature, highly optimized ASP systems.

6.1 ASP and s(ASP)

ASP (Brewka et al., 2011; Gelfond and Lifschitz, 1988) is a logic programming and modelling language. An ASP program Π is a finite set of *rules*. Each rule $r \in \Pi$ is of the form:

$$a \leftarrow b_1 \land \ldots \land b_m \land not\ b_{m+1} \land \ldots \land not\ b_n.$$

where a and $b_1, \ldots, b_n$ are atoms and *not* corresponds to *default* negation. An atom is an expression of form $p(t_1, \ldots, t_n)$ where p is a predicate symbol of arity n and t_i, are *terms*. An atom is *ground* if no variables occur in it. The set of all *constants* appearing in Π is denoted by C_Π. The head of rule r is $h(r) = \{a\}$[2] and the body consists of positive atoms $b^+(r) = \{b_1, \ldots, b_m\}$ and negative atoms $b^-(r) = \{b_{m+1}, \ldots, b_n\}$. Intuitively,

[1]Note that while we used CLP($\mathbb{Q}$) in this chapter, CLP($\mathbb{R}$) could also have been used.

[2]Disjunctive ASP programs (i.e., programs with disjunctions in the heads of rules) can be transformed into non-disjunctive ASP programs by using *default* negation (Ji et al., 2016).

rule r is a justification to *derive* that a is true if all atoms in $b^+(r)$ have a derivation and no atom in $b^-(r)$ has a derivation. An interpretation I is a subset of the program's Herbrand base and it is said to satisfy a rule r if $h(r)$ can be derived from I. A model of a set of rules is an interpretation that satisfies each rule in the set. An answer set of a program Π is a minimal model (in the set-theoretic sense) of the program

$$\Pi^I = \{h(r) \leftarrow b^+(r) \mid r \in \Pi, b^-(r) \cap I = \emptyset\}$$

which is called the *Gelfond-Lifschitz reduct* of Π with respect to I (Gelfond and Lifschitz, 1991). The set of all answer sets of Π is denoted by $AS(\Pi)$. ASP solvers which compute the answer sets of non-ground programs use the above semantics by first applying, to each rule $r \in \Pi$, all possible substitution from the variables in r to elements of C_Π (this procedure is called *grounding*). To make this grounding possible, the rules of the program should be *safe*, i.e., all variables that appear in a rule have to appear in some positive literal in the body. The rule is termed *unsafe* otherwise.

A difference between ASP and Prolog-style (i.e., SLD resolution-based) languages is the treatment of negated literals. Negated literals in a body are treated in ASP using their logical semantics based on computing stable models. The *negation as failure* rule of Prolog, SLDNF resolution (Clark, 1978), makes a negated call succeed (respectively, fail) iff the non-negated call fails (respectively, succeeds). To ensure soundness, SLDNF has to be restricted to ground calls, as a successful negated goal cannot return bindings. However, SLDNF increases the cases of non-termination w.r.t. SLD.

s(ASP) (Marple et al., 2017a,b) is a top-down, goal-driven interpreter of ASP programs written in Prolog (`http://sasp-system.sourceforge.net`). The top-down evaluation makes the *grounding* phase unnecessary. The execution of an s(ASP) program starts with a *query*, and each answer is the resulting *mgu* of a successful derivation, its justification, and a (partial) stable model. This partial stable model is a subset of the ASP stable model (Gelfond and Lifschitz, 1988) including only the literals necessary to support the query with its output bindings.[3]

Example 6.1. Assuming an extended Herbrand Base.
Given the program below:

```
married(john).                          :- not married(X).
```

most ASP systems are not able to compute its stable model (not even an empty one), because the global constraint is unsafe. On the other hand, s(ASP) is able to compute queries to programs with unsafe rules by assuming that the unsafe variables take values in an *extended Herbrand Universe*, and not just that of the terms which can

[3]Note that the subset property holds only when the Gelfond–Lifschitz transformation is applied assuming an *extended Herbrand Base* obtained by extending the set of constants in the program, C_Π, with an infinite number of new elements.

be constructed from the symbols in the program. Therefore, using this alternative semantics `:- not married(X).` corresponds to $\neg\exists x.\neg married(x) \equiv \forall x.married(x)$ and since the program only has evidence of one married individual (`john`), there is no stable model (i.e., it cannot be derived that all possible individuals are married). However, if we add the (unsafe) fact `married(X)` (i.e., $\forall x.married(x)$) to the program, the resulting stable model will be {`married(X)`} — every element of the universe is married.

s(ASP) has two additional relevant differences w.r.t. Prolog: first, s(ASP) resolves negated atoms *not* l_i against *dual rules* of the program (Section 6.1.1), instead of using negation as failure. This makes it possible for a non-ground negated call `not p(X)` to return the results for which the positive call `p(X)` would fail. Second, and very important, the dual program is **not** interpreted under SLD semantics: a number of very relevant changes related to how loops are treated (see later) are introduced.

6.1.1 Dual of a Logic Program

The dual of a predicate `p/1` is another predicate that returns the `X` such that `p(X)` is not true. It is used to give a constructive answer to a goal `not p(X)`. The dual of a logic program is another logic program containing the dual of each predicate in the program (Alferes et al., 2004). To synthesize the dual of a logic program P we first obtain Clark's completion (Clark, 1978), which assumes that the rules of the program completely capture all possible ways for atomic formulas to be true, and then we apply De Morgan's laws:

1. For each literal p/n that appears in the head of a rule, choose a tuple $\vec{x}$ of n distinct, new variables $x_1, \ldots, x_n$.

2. For each i-th rule of a predicate p/n of the form $p_i(\vec{t_i}) \leftarrow B_i$, with $i = 1, \ldots, k$, make a list $\vec{y_i}$ of all variables that occur in the body B_i but do not occur in the head $p_i(\vec{t_i})$, add $\exists\vec{y_i}$ to the body and rename the variables that appear in the head $\vec{t_i}$ with the tuple $\vec{x}$, obtained in the previous step, resulting in a predicate representing $\forall\vec{x}\ (p_i(\vec{x}) \leftarrow \exists\vec{y_i}\ B_i)$. Note that $\vec{x}$ are local, fresh variables. This step captures the standard semantics of Horn clauses.

3. With all these rules and using Clark's completion, we form the sentences:

$$\begin{array}{lcl} \forall\vec{x}\ (\ p(\vec{x}) & \longleftrightarrow & p_1(\vec{x}) \vee \cdots \vee p_k(\vec{x})\ ) \\ \forall\vec{x}\ (\ p_i(\vec{x}) & \longleftrightarrow & \exists\vec{y_i}\ (b_{i.1} \wedge \ldots \wedge b_{i.m} \wedge \neg\, b_{i.m+1} \wedge \ldots \wedge \neg\, b_{i.n})\ ) \end{array}$$

4. Their semantically equivalent duals $\neg p/n$, $\neg p_i/n$ are:

$$\forall \vec{x}\ (\ \neg p(\vec{x}) \quad \longleftrightarrow \quad \neg(p_1(\vec{x}) \vee \cdots \vee p_k(\vec{x}))\ )$$
$$\forall \vec{x}\ (\ \neg p_i(\vec{x}) \quad \longleftrightarrow \quad \neg\ \exists \vec{y}_i\ (b_{i.1} \wedge \ldots \wedge b_{i.m} \wedge \neg\ b_{i.m+1} \wedge \ldots \wedge \neg\ b_{i.n})\ )$$

5. Applying De Morgan's laws we obtain:

$$\forall \vec{x}(\ \neg p(\vec{x}) \quad \longleftrightarrow \quad \neg p_1(\vec{x}) \wedge \ldots \wedge \neg p_k(\vec{x})\ )$$
$$\forall \vec{x}\ (\neg p_i(\vec{x}) \quad \longleftrightarrow \quad \forall \vec{y}_i\ (\neg b_{i.1} \vee \cdots \vee \neg\ b_{i.m} \vee\ b_{i.m+1} \vee \cdots \vee\ b_{i.n})\ )$$

which generates a definition for $\neg p(\vec{x})$ and a separate clause with head $\neg p_i(\vec{x})$ for each positive or negative literal $b_{i.j}$ in the disjunction. Additionally, a construction to implement the universal quantifier introduced in the body of the dual program is necessary (Section 6.1.3).

Definitions for the initially negated literals $\neg b_{i.m+1} \ldots \neg b_{i.n}$ and for each of the *new* negated literals $\neg b_{i.1} \ldots \neg b_{i.m}$ are similarly synthesized. At the end of the chain, unification has to be negated to obtain disequality, e.g., $x = y$ is transformed into $x \neq y$ (Section 6.1.2).

Example 6.2.
Given the program below:

```
p(0).
p(X) :- q(X), not t(X,Y).
q(1).
t(1,2).
```

the resulting dual program is:

```
1  not p(X) :- not p1(X), not p2(X).
2  not p1(X) :- X \= 0.
3  not p2(X) :-
4     forall(Y, not p2_(X,Y)).
5  not p2_(X,Y) :- not q(X).
6  not p2_(X,Y) :- q(X), t(X,Y).
7  not q(X) :- not q1(X).
8  not q1(X) :- X \= 1.
9  not t(X,Y) :- not t1(X,Y).
10 not t1(X,Y) :- X \= 1.
11 not t1(X,Y) :- X=1, Y \= 2.
```

For efficiency, the generation of the dual diverges slightly from the previous scheme. The dual of a body $B \equiv l_1 \wedge \ldots$ is the disjunction of its negated literals $\neg B \equiv \neg l_1 \vee \ldots$, which generates independent clauses in the dual program. To avoid redundant answers, every clause for a negated literal $\neg l_i$ includes calls to any positive literal l_j with $j < i$. E.g., clause 6 from the previous program, `not p2(X,Y) :- q(X), t(X, Y)`, would only need to be `not p2(X,Y) :- t(X, Y)`. However, the literal `q(X)` is included to avoid exploring solutions already provided by clause 5, `not p2(X,Y) :- not q(X)`. The same happens with clauses 10 and 11.

Algorithm 3: *forall*

```
1  forall receives V, a variable name, and Goal, a callable goal.
2  V starts unbound
3  Execute Goal.
4  if Goal succeeded then                                    Let us check the bindings of V
5      if V is unbound then forall succeeds                   Goal's success is independent of V
6      else if V is bound, then backtrack to step 4 and try other clauses
7      else                           V has been constrained to be different from a series of values
8          Re-execute Goal, successively substituting the variable V with each of these
             values
9          if Goal succeeds for each value then forall succeeds
10         else forall fails             There is at least one value for which Goal is not true
11     end
12 else forall fails                  There are infinitely many values for which Goal is not true
```

6.1.2 Constructive Disequality

Unlike Prolog's *negation as failure*, disequality in s(ASP), denoted by "\=" , represents the constructive negation of the unification and is used to construct answers from negative literals. Intuitively, `X \= a` means that `X` can be any term not unifiable with `a`. In the implementation reported in (Marple et al., 2017a) a variable can only be disequality-constrained against ground terms, and the disequality of two compound terms may require backtracking to check all the cases: `p(1, Y) \= p(X, 2)` first succeeds with `X \= 1` and then, upon backtracking, with `Y \= 2`.

The former restriction reduces the range of valid programs, but this does not seem to be a problem in practice: since positive literals are called before negative literals in the dual program, the number of cases where this situation may occur is further reduced. Since this is orthogonal to the implementation framework, it can be improved upon separately. The second characteristic impacts performance, but can again be ameliorated with a more involved implementation of disequality which carries a disjunction of terms.

6.1.3 The `forall` Algorithm

In (Marple et al., 2017a) the universal quantifier is evaluated by `forall(V, Goal)` which checks if `Goal` is true for all the possible values of `V`. When `forall/2` succeeds, the evaluation continues with the quantified variable unbound. Multiple quantified variables are handled by nesting: $\forall v_1, v_2.Goal$ is executed as `forall(V1,forall(V2, Goal))`. The underlying idea is to verify that for any solution with `V \= a` (for some `a`), `Goal` also succeeds with `V=a` (Algorithm 3).

Example 6.3.
Consider the following program with the dual rule for `p/0`:

```
p :- not q(X).
q(X) :- X=a.
q(X) :- X \= a.
```

```
not p :- forall(X, not p1(X)).
not p1(X) :- q(X).
```

Under the query `?- not p`, the interpreter will execute `forall(X, not p1(X))` with `X` unbound. First, `not p1(X)` is executed and calls `q(X)`, succeeding with `X=a`. Then, since `X` is bound, the interpreter backtracks and succeeds with `X \= a` (second clause of `q/1`). Now, since `X` is constrained to be different from `a`, the interpreter re-executes `not p1(X)` with `X=a` which succeeds (first clause of `q/1`). Since there are no more constrained values to be checked, the evaluation of the query finishes with success. Note that leaving `X` unbound after the success of `forall(X, p(X))` is consistent with the interpretation that the answer set {`p(X)`} corresponds to $\forall x . p(x)$.

6.1.4 Non-Monotonic Checking Rules

Non-monotonic rules are used by s(ASP) to ensure that partial stable models are consistent with the global constraints of the program. Given a consistency rule of the form $\forall \vec{x}(p_i(\vec{x}) \leftarrow \exists \vec{y}\, B_i \wedge \neg p_i(\vec{x}))$, and in order to avoid contradictory rules of the form $p_i(\vec{a}) \leftarrow \neg p_i(\vec{a})$, all stable models must satisfy that at least one literal in B_i is false (i.e., $\neg B_i$) or, for the values $\vec{a}$ where B_i is true, $p_i(\vec{a})$ can be derived using another rule. To ensure that the partial stable model is consistent, the s(ASP) compiler generates, for each consistency rule, a rule of the form:

$$\forall \vec{x}(\; chk_i(\vec{x}) \;\longleftrightarrow\; \forall \vec{y}_i(\neg B_i \vee p(\vec{x})\;)\;)$$

To ensure that each sub-check (chk_i) is satisfied, the compiler introduces into the program the rule $nmr\_check \leftarrow chk_1 \wedge \ldots \wedge chk_k$, which is transparently called after the program query.

Example 6.4.
Given the program below:

```
:- not s(1, X).
```

```
p(X):- q(X), not p(X).
```

the resulting *NMR* check rules are:

```
nmr_check :-
    chk1,
    forall(A, chk2(A)).
```

```
chk1 :- forall(X,s(1,X)).
chk2(X) :- not q(X).
chk2(X) :- q(X), p(X).
```

6.1.5 Handling Loops

Finally, in order to break infinite loops, s(ASP) uses three techniques to deal with *odd loops over negation*, *even loops over negation*, and *positive loops* (Gupta et al., 2007; Marple et al., 2017a).

Top-down evaluations may enter loops. Several techniques, notably tabling, have been used to enhance the termination properties of LP systems. This is more relevant in s(ASP) because the presence of negation introduces new types of loops:

- **Odd loop over negation**: it occurs when a cycle in the call graph contains an odd number of intervening negations. These loops are important because they place global constraints which restrict which literals can appear in a model. s(ASP) ensures that these global constraints are satisfied by introducing *non monotonic rules* (Section 6.1.4). The odd loops are detected with a static analysis of the call graph checking the number of negations between recursive calls.

  Example 6.5.
  The rules below, which are equivalent if `p/0` can not be added to the model by another rule, generate odd loops and force the stable model to satisfy $\neg\ q(a)$.

  ```
  p :- q(a), not p.                    :- q(a).
  ```

  Run-time check of odd loops When, during the execution, a call unifies with its negation in the call path, the execution fails and backtracks. Had it succeeded, it would have introduced a contradiction, and therefore the resulting partial stable model would have been discarded.

- **Even loop over negation**: This happens when a call unifies with an ancestor in the call path and there is an even, non-zero, number of intervening negated calls between them. In this case, the execution succeeds assuming that the recursive call (partially) supports the *negation* of those calls. The spirit underlying this assumption is similar to coinductive SLD resolution (Gupta et al., 2007), used to compute the greatest fixpoint of a program. Note that the Gelfond–Lifschitz method computes the fixpoint of the residual program, which is between the least fixpoint (computed by a top-down execution) and the greatest fixpoint. This assumption is safe because in cases where the evaluation tries to make this recursive call *true*, the *non monotonic rules* and the run-time detection of odd loops will discard the model.

  Example 6.6.
  Consider the next program (with its dual) and the query `?- p(a).`

```
p(X) :- not q(X).

q(X) :- not p(X).
q(b).

not p(X) :- not p1(X).
not p1(X) :- q(X).
not q(X) :- not q1(X), not q2(X).
not q1(X) :- p(X).
not q2(X) :- X \= b.
```

The call path `p(a)`$\leadsto$`not q(a)`$\leadsto$`not q1(a)`$\leadsto$`p(a)`, resulting from the query, shows that assuming `p(a)` we support both negated calls (i.e., `not q(a)` and `not q1(a)`). Note that `not q(a)` is only partially supported because it succeeds only if also `not q2(X)` succeeds. Therefore, while the query `?- p(a)` succeeds, the query `?- p(b)` fails.

- **Positive loops**: when a call unifies with an ancestor in the call path and there are no intervening negative calls between them, the original s(ASP) fails to avoid infinite loops. However, this behaviour compromises completeness and soundness. We work around this by checking that the call and its ancestor are equal (Section 6.2.2).

Example 6.7.
The next program generates infinitely many answers to the query `?- nat(X).`

```
nat(0).
nat(X) :- nat(Y), X is Y+1.
```

However, if it fails, when the recursive call `nat(Y)` unifies with its ancestor in the call path (i.e., the query), it loses completeness as it only returns the answer `X=0`, and therefore, due to the presence of negation, it also loses soundness.

6.2 s(CASP): Design and Implementation

S(CASP) (available together with the benchmarks used in this chapter at `https://gitlab.software.imdea.org/joaquin.arias/sCASP`) extends s(ASP) by computing partial stable models of programs with constraints. This extension makes the following contributions:

- The interpreter is reimplemented in Ciao Prolog (Hermenegildo et al., 2012). The driving design decision of this reimplementation is to let Prolog take care of all operations that it can handle natively, instead of interpreting them. Therefore, a large part of the environment for the s(CASP) program is carried implicitly in the Prolog environment. Since s(CASP) and Prolog shared many characteristics (e.g., the behavior of variables), this results in flexibility of implementation (see the

Table 6.1: Run-time (ms) of s(CASP) and s(ASP) for different programs.

	s(CASP)	**s(ASP)**
hanoi(8,T)	**1,528**	13,297
queens(4,Q)	**1,930**	20,141
One hamicycle	**493**	3,499
Two hamicycle	**3,605**	18,026

interpreter code sketched in Figure 6.1 and in full in Appendix C.1) and gives a large performance improvement. Table 6.1 shows run-time comparison of s(ASP) vs s(CASP) using ASP programs without constraint. All the experiments in this chapter were performed on a MacOS 10.13 machine with an Intel Core i5 at 2GHz.

- A new solver for disequality constraints.
- The definition and implementation of a generic interface to plug-in different constraint solvers. This required, in addition to changes to the interpreter, changes to the compiler which generates the dual program. This interface has been used, in this chapter, to connect both the disequality constraint solver and the CLP($\mathbb{Q}$) solver.
- The design and implementation of *C-forall* (Algorithm 4), a generic algorithm which extends the original *forall* algorithm (Algorithm 3) with the ability to evaluate goals with variables constrained under arbitrary constraint domains. In addition to being necessary to deal with constraints, this extension generalizes and clarifies the design of the original one.

6.2.1 s(CASP) Programs

An s(CASP) program is a finite set of rules of the form:

$$a \leftarrow c_a \wedge b_1 \wedge \ldots \wedge b_m \wedge not\ b_{m+1} \wedge \ldots \wedge not\ b_n.$$

where the difference w.r.t. an ASP program is c_a, a simple constraint or a conjunction of constraints. A query to an s(CASP) program is of the form $\leftarrow\ c_q \wedge l_1 \wedge \ldots \wedge l_n$, where c_q is also a simple constraint or a conjunction of constraints. The semantics of s(CASP) extends that of s(ASP) following (Jaffar and Maher, 1994). During the evaluation of an s(CASP) program, the interpreter generates constraints whose consistency w.r.t. the

```
??(Query) :-
   solve(Query,[],Mid),
   solve_goal(nmr_check,Mid,Out),
   print_just_model(Out).
solve([], In, ['$success'|In]).
solve([Goal|Gs], In, Out) :-
   solve_goal(Goal, In, Mid),
   solve(Gs, Mid, Out).
solve_goal(Goal, In, Out) :-
   user_defined(Goal), !,
   pr_rule(Goal, Body),
   solve(Body, [Goal|In], Out).
solve_goal(Goal, In, Out) :-
   call(Goal),
   Out=['$success',Goal|In].
```

Figure 6.1: (Very abridged) Code of the s(CASP) interpreter.

current constraint store is checked by the *constraint solver*. The existence of variables both during execution and in the final models is intuitively justified by adopting an approach similar to that of the S-semantics (Gabbrielli and Levi, 1991).

6.2.2 The Interpreter and the Disequality Constraint Solver

The s(CASP) interpreter carries the environment (the call path and the model) implicitly and delegates to Prolog all operations that Prolog can do natively, such as handling the bindings due to unification, the unbinding due to backtracking, and the operations with constraints, among others. The clauses of the program, their duals, and the NMR-checks are created by the compiler by generating rules of the predicate `pr_rule(Head,Body)`, where `Head` is an atom and `Body` is the list of literals. While the s(CASP) interpreter performs better than s(ASP), little effort has been invested in optimizing it, and there is ample room for improvement.

Figure 6.1 shows a highly simplified sketch of the code that implements the interpreter loop in s(CASP), where:

- `??(+Query)` receives a query and prints the successful path derivations.
- `solve(+Goals,+PathIn,-PathOut)` reproduces SLD resolution.
- `solve_goal(+Goal,+PathIn,-PathOut)` evaluates the user-defined predicates and hands over to Prolog the execution of the builtins using `call/1`. The `PathOut` argument encodes the derivation tree in a list.

Every `'$success'` constant denotes the success of the goals in the body of a clause and means that one has to go up one level in the derivation tree. Several `'$success'` constants in a row mean, accordingly, that one has to go up the same number of levels.

In s(CASP), constructive disequality is handled by a disequality constraint solver, called CLP($\neq$), implemented using attributed variables that makes disequality handling transparent to the user code. The current implementation of CLP($\neq$) does not address the restrictions described in Section 6.1.2; however, as mentioned before, since the solver is independent of the interpreter, its improvements are orthogonal to the core implementation of s(CASP).

The interpreter checks the call path before the evaluation of user-defined predicates to prevent inconsistencies and infinite loops (Marple et al., 2017a), as we mentioned before in Section 6.1.5. The call path is a list constructed with the calls, and the bindings of the variables in these calls are automatically updated by Prolog.

- When a positive loops occurs, the interpreter fails only if the looping goal and its ancestor are equal (i.e., `p(X) :- ...,p(X)`). Termination properties are enhanced if a tabling system featuring variant calls or entailment (see Chapters 2 and 3) is used as implementation target, so that all programs with a finite grounding or with the constraint-compact property terminate.
- However, when the current call is equal to an already-proven ancestor, the evaluation succeeds to avoid its re-computation and to reduce the size of the justification tree.

6.2.3 Integration of Constraint Solvers in s(CASP)

Holzbaur's CLP($\mathbb{Q}$) (Holzbaur, 1995) solver was integrated in the current implementation of s(CASP). Since the interpreter already deals with the CLP($\neq$) constraint solver, only two details have to be taken in consideration:

- The compiler is extended to support CLP($\mathbb{Q}$) relations $\{<,>,=,\geq,\leq,\neq\}$ during the construction of the dual program and the NMR rules.
- Since it is not possible to decide at compile time whether equality will be called with CLP($\mathbb{Q}$) or Herbrand variables, its dual \= is extended to decide at run-time whether to call the CLP($\mathbb{Q}$) solver or the disequality solver.

Finally, to make integrating further constraint solvers easier, the operations that the s(CASP) interpreter requires from the CLP($\mathbb{Q}$) solver are encapsulated in a single module that provides the interface between the interpreter and the constraint solver. Additional constraint solvers only need to provide the same interface.

6.2.4 The `C-forall` Algorithm

Extending s(ASP) to programs with constraints requires a generalization of *forall* (Algorithm 3) which we will call *C-forall* (Algorithm 4). A successful evaluation of

Algorithm 4: C-forall

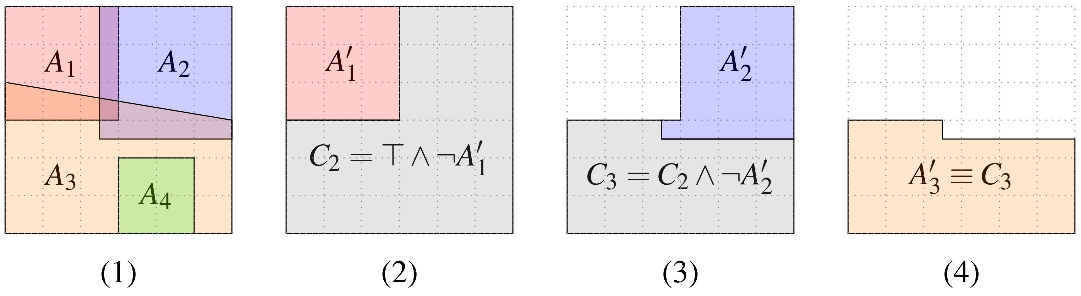
```
1  C-forall receives a variable V, a callable goal Goal, and a constraint store, $C_i$ ($i = 1$).
2  V starts unbound.                          The constraint store of V is empty, $C_{V.i} = \top$
3  Execute Goal with $C_i$ as the current constraint store. Its first answer constraint store is $A_1$
4  if the execution of Goal succeeds then      Check $A_{V.i}$, the domain of V in the answer constraint
5      if $A_{V.i} \equiv C_{V.i}$ then          There was no refinement in the domain of V
6          C-forall succeeds                  V is not relevant for the success of Goal
7      else                                   The domain of V has been restricted, $A_{V.i} \sqsubset C_{V.i}$
8          $C_{i+1} = C_i \wedge A_{\overline{V}.i} \wedge \neg A_{V.i}$    Remove from V the elements for which Goal succeeds
9          Return to step 3 and re-execute Goal under $C_{i+1}$
                                              Check whether Goal is true for the rest of the elements of V
10     end
11 else C-forall fails                        There is a non-empty domain for which Goal is not true
```

Goal in s(CASP) returns, on backtracking, a (potentially infinite) sequence of models and answer constraint stores $A_1, A_2, \ldots$. Each A_i relates variables and constants by means of constraints and bindings (i.e., syntactical equality constraints). The execution of `forall(V,Goal)` is expected to determine if Goal is true for all possible values of V in its constraint domain.

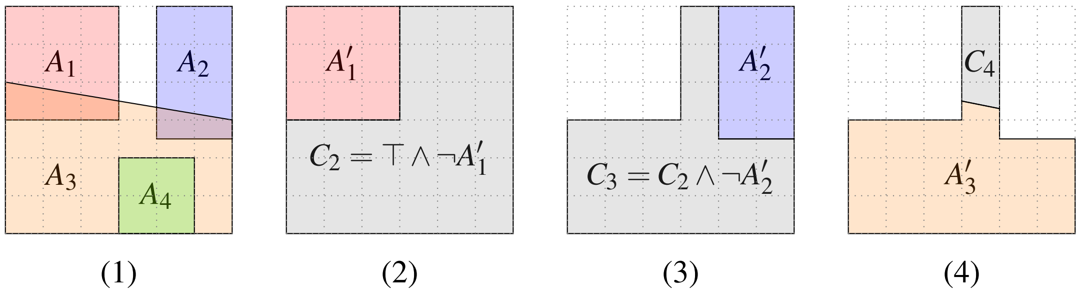
In what follows we will use $\overline{\texttt{V}}$ to denote the set variables in Goal that are not V: *vars*(Goal) = $\{\texttt{V}\} \cup \overline{\texttt{V}} \wedge \texttt{V} \notin \overline{\texttt{V}}$. The core idea is to iteratively narrow the store C under which Goal is executed by selecting **one** answer A and re-executing Goal under the constraint store $C \wedge A_{\overline{\texttt{V}}} \wedge \neg A_{\texttt{V}}$, where $A_{\texttt{V}}$ is the projection of A on V and $A_{\overline{\texttt{V}}}$ is the projection of A on $\overline{\texttt{V}}$. The iterative execution finishes with a positive or negative outcome.

Example 6.8. *C-forall* terminates with success.
Figure 6.2a shows an example where the answers $A_1, \ldots, A_4$ to Goal cover the whole domain, represented by the square. Therefore, *C-forall* should succeed. The answer constraints that the program can generate are depicted on picture (1). For simplicity in the pictures, we will assume that the answers A_i only restrict the domain of V, so it will not be necessary to deal with V and $\overline{\texttt{V}}$ separately since $A_{\overline{\texttt{V}}}$ will always be empty, and therefore $A_{\texttt{V}.i} = A_i$. Picture (2) shows the result of the first iteration of *C-forall* starting with $C_1 = \top$: answer A_1 is more restrictive than C_1 and therefore $C_2 = C_1 \wedge \neg A_1$ (in grey) is constructed. Picture (3) shows the result of the second iteration: the domain is further reduced. Finally, in picture (4) the algorithm finishes successfully because $A_3 \equiv C_3$, i.e., A_3 covers the remaining domain. Note that we did not need to generate A_4.

Termination for an infinite number of answer sets The previous example points to a nice property: even if there were an infinite number of answer sets to Goal, as long as



(a) *C-forall* terminates with success.



(b) *C-forall* terminates with failure.

Figure 6.2: Two *C-forall* evaluation examples.

a finite subset of them covers the domain of `V` and this subset can be finitely enumerated by the program, the algorithm will finish. This is always true for constraint-compact constraint domains, such as disequality over a finite set of constants or the gap-order constraints (Revesz, 1993). Note that this happens as well in the next example, where *C-forall* fails.

Example 6.9. *C-forall* terminates with failure.
Figure 6.2b shows an example where the answer constraints do not cover the domain and therefore *C-forall* ought to fail. Again, we assume that the answers A_i only restrict the domain of `V`. Picture (1) depicts the answer constraints that `Goal` can generate. Note the gap in the domain not covered by the answers. Pictures (2) to (4) proceed as in the previous example. Picture (4) shows the final step of the algorithm: the execution of `Goal` under the store $C_4 = C_3 \wedge \neg A'_3$ fails because the solution A_4 of `Goal` does not have any element in common with C_4, and then *C-forall* also fails.

Figure 6.3 shows a sketch of the code that implements *C-forall* in the s(CASP) interpreter, written in Prolog/CLP. In this setting, `Goal` carries the constraint stores C_i and the answer stores A_i implicitly in its execution environment. We know that the interpreter

```
forall(V, Goal) :-
   empty_store(Store),           % V has no attached constraints
   eval_forall(V, Goal, [Store]). % start the evaluation of Goal
eval_forall(_, _, []).           % it's done, forall succeeds
eval_forall(V,Goal,[Store|Sts]):-
   copy(V, Goal, NV, NGoal),     % copy to keep V unbound
   apply(NV, V, Store),          % add the constraint to NV
   once(NGoal),                  % if fails, the forall fails
   dump(NV, V, AnsSt),           % project the answer store
   (   equal(AnsSt, Store)       % if there is no refinement in NV
   ->  true                      % then, it's done, continue
   ;   dual(AnsSt, AnsDs),       % else, the answer's dual/duals
       add(AnsDs, Store, NSt),   % is/are added to Store
       eval_forall(V, Goal, NSt) % to evaluate Goal
   ),
   eval_forall(V, Goal, Sts).    % continue the evaluation
```

Figure 6.3: Implementation of `forall/2` in the s(CASP) interpreter.

will call `forall(V,Goal)` with a fresh, unconstrained `V`, because the executed code is generated by the s(CASP) compiler. Therefore, the projection of C_1 onto `V` is an empty constraint store, which we introduce explicitly to start the computation.

The call `copy(V, Goal, NV, NGoal)` copies `Goal` in `NGoal` sharing only $\overline{\texttt{V}}$, while `V` is substituted in `NGoal` by a fresh variable, `NV`. In the main body of `eval_forall/3`, `Store` always refers to `V`, while `NGoal` does not contain `V`, but `NV`. The call `apply(NV,V, Store)` takes the object `Store` and makes it part of the global store but substituting `V` for `NV` so that the execution of `NGoal` can further constrain `NV` while `V` remains untouched. Note, however, that in the first iteration, `NV` will always remain unconstrained, since the constraint store that `apply(NV,V, Store)` applies to it is empty ($C_{\texttt{V}.1} = \top$). However, in the following iterations, `Store` will contain the successive constraint stores $C_{\texttt{V}.i+1}$.

When `once(NGoal)` succeeds, the constraint store $C_i \wedge A_{\overline{\texttt{V}}.i}$ is implicit in the binding of $\overline{\texttt{V}}$. Therefore, the execution of `eval_forall(V,Goal,Store)` carries this constraint store implicitly because `Goal` and `NGoal` share $\overline{\texttt{V}}$. Finally, the predicate `dump(NV,V, AnsSt)` projects the constraint store after the execution of `NGoal` on `NV`, rewrites this projection to substitute `NV` for `V`, and leaves the final result in `AnsSt`, generating $A_{\texttt{V}.i}$. Note that, in some sense, it is transferring constraints in the opposite direction to what `dump/3` did before. If the call `equal(AnsSt, Store)` succeeds, it means that $A_{\texttt{V}.i} \equiv C_{\texttt{V}.i}$ and therefore the `forall` succeeds (for the branch that was being explored, see below).

Otherwise, we have to negate the projection of the answer onto `V`, i.e., construct $\neg A_{\texttt{V}.i}$. The negation of a conjunction generates a disjunction of constraints and most constraint solvers cannot handle disjunctions natively. Therefore, the predicate `dual(AnsSt,AnsDs)` returns in `AnsDs` a list with the components $\neg A_{\texttt{V}.i.j}$ of this disjunction, $j = 1, 2, \ldots, \mathit{length}(\texttt{AnsDs})$. Then, `add(AnsDs,Store, NSt)` returns in `NSt` a list of stores, each of which is the conjunction of `Store` with one of the components of the disjunction in `AnsDs`, i.e., a list of $C_{\mathtt{V}.i} \wedge \neg A_{\mathtt{V}.i.j}$, for a fixed i. There may be cases where this conjunction is inconsistent; `add/3` captures them and returns only the components which are consistent. Note that if a conjunction $C_{\mathtt{V}.i} \wedge \neg A_{\mathtt{V}.i.j}$ is inconsistent, it means that $\neg A_{\mathtt{V}.i.j}$ has already been (successfully) checked.

Each of the resulting constraint stores will be re-evaluated by `eval_forall/3`, where `apply/3` will apply them to a new variable `NV`, in order to complete the implicit construction of C_{i+1} before the execution of `once(NGoal)`. `forall/2` finishes with success when there are no pending constraint stores to be processed (line 4).

Example 6.10. `C-forall` execution negating a constraint conjunction.
Given the program below, consider the evaluation of `forall(A, p(A))`:

```
p(X) :- X #>= 0, X #=< 5.
p(X) :- X #> 1.
p(X) :- X #< 3.
p(X) :- X #< 1.
```

In the first iteration $C_1 = \top$. The first answer is $A_1 = \{X \geq 0 \wedge X \leq 5\}$, which is more restrictive than C_1, so we compute $\neg A_1 = \{X < 0 \vee X > 5\}$. First, `p/1` is evaluated with $C_{2.a} = \{\top \wedge X < 0\}$ obtaining $A_{2.a} = \{X < 0\}$ using the third clause. Since $A_{2.a} \equiv C_{2.a}$, we are done with $C_{2.a}$. But we also have to evaluate `p/1` with $C_{2.b} = \{\top \wedge X > 5\}$. Using the second clause, $A_{2.b} = \{X > 5\}$ is obtained and since $A_{2.b} \equiv C_{2.b}$, the evaluation succeeds.

6.3 Examples and Evaluation

The expressiveness of s(CASP) allows the programmer to write programs / queries that cannot be written in [C]ASP without resorting to a complex, unnatural encoding. Additionally, the answers given by s(CASP) are also more expressive than those given by ASP. This arises from several points:

- s(CASP) inherits from s(ASP) the use of unbound variables during the execution and in the answers. This makes it possible to express constraints more compactly and naturally (e.g., ranges of distances can be written using constraints)
- s(CASP) can use structures / functors directly, thereby avoiding the need to encode them unnaturally (e.g., giving numbers to Hanoi movements to represent what in a list is implicit in the sequence of its elements).
- The constraints and the goal-directed evaluation strategy of s(CASP) makes it possible to use direct algorithms and to reduce the search space (e.g., by putting bounds on a path's length).

```
1  valid_stream(P,Data) :-
2       stream(P,Data),
3       not cancelled(P, Data).
4
5  cancelled(P, Data) :-
6       higher_prio(P1, P),
7       stream(P1, Data1),
8       incompt(Data, Data1).
9
10 higher_prio(PHi, PLo) :-
11      PHi #> PLo.
12 incompt(p(X), q(X)).
13 incompt(q(X), p(X)).
14
15 stream(1,p(X)).
16 stream(2,q(a)).
17 stream(2,q(b)).
18 stream(3,p(a)).
```

Figure 6.4: Code of a stream data reasoner under s(CASP).

6.3.1 Stream Data Reasoning

Let us assume that we deal with data streams, some of whose items may be contradictory (Arias, 2016). Moreover, different data sources may have a different degree of trustworthiness which we use to prefer a given data item in case of inconsistency. Let us assume that `p(X)` and `q(X)` are contradictory and we receive `p(X)` from source S_1 and `q(a)` from source S_2. We may decide, depending on how reliable are S_1 and S_2, that: (i) `p(X)` is true because S_1 is more reliable than S_2; (ii) `q(a)` is true since S_2 is more reliable than S_1, and for any `X` different from `a` (i.e., `X \= a`), `p(X)` is also true; (iii) or, if both sources are equally reliable, them we have (at least) two different models: one where `q(a)` is true and another where `p(X)` is true.

Figure 6.4 shows the code for a stream data reasoner using s(CASP). Data items are represented as `stream(Priority,Data)`, where `Priority` tells us the degree of confidence in `Data`; `higher_prio(PHi,PLo)` hides how priorities are encoded in the data (in this case, the higher the priority, the more level of confidence); and `incompt/2` determines which data items are contradictory (in this case, `p(X)` and `q(X)`). Note that `p(X)` (for **all** `X`) has less confidence than `q(a)` and `q(b)`, but `p(a)` is an exception, as it has more confidence than `q(a)` or `q(b)`. Lines 1-8, alone, define the reasoner rules: `valid_stream/2` states that a data stream is valid if it is *not cancelled* by another contradictory data stream with more confidence.

The confidence relationship uses constraints, instead of being checked afterwards. *C-forall*, introduced by the compiler in the dual program (available in Appendix C.2.1), will check its consistency. For the query `?- valid_stream(Pr,Data)`, it returns: {`Pr=1,Data=p(A),A\=a, A\=b`} because `q(a)` and `q(b)` are more reliable than `p(X)`; {`Pr=2,Data=q(b)`}; and {`Pr=3,Data=p(a)`}. The justification tree and the model are in Appendix C.2.2.

The constraints and the goal-directed strategy of s(CASP) make it possible to resolve queries without evaluating the whole stream database. For example, the rule `incompt(p(X),q(X))` does not have to be grounded w.r.t. the stream database, and if timestamps were used as trustworthiness measure, for a query such as `?- T #> 10,`

```
duration(load,25).
duration(shoot,5).
duration(wait,36).
spoiled(Armed) :- Armed #> 35.
prohibited(shoot,T) :- T #< 35.

holds(0,St,[]) :- init(St).
holds(F_Time, F_St, [Act|As]) :-
   F_Time #> 0,
   F_Time #= P_Time + Duration,
   duration(Act, Duration),
   not prohibited(Act, F_Time),
   trans(Act, P_St, F_St),
   holds(P_Time, P_St, As).
init(st(alive,unloaded,0)).

trans(load, st(alive,_,_),
            st(alive,loaded,0)).
trans(wait, st(alive,Gun,P_Ar),
            st(alive,Gun,F_Ar)) :-
   F_Ar #= P_Ar + Duration,
   duration(wait,Duration).
trans(shoot, st(alive,loaded,Armed),
             st(dead,unloaded,0)) :-
   not spoiled(Armed).
trans(shoot, st(alive,loaded,Armed),
             st(alive,unloaded,0)) :-
   spoiled(Armed).
```

Figure 6.5: Code of the Yale Shooting problem under s(CAPS).

`valid_stream(T,p(A))` the reasoner would validate streams received after `T=10` regardless how long they extend in the past.

6.3.2 Yale Shooting Scenario

In the spoiling Yale shooting scenario (Janhunen et al., 2017), there is a gun and three possible actions: *load*, *shoot*, and *wait*. If we load the gun and shoot within 35 minutes, the turkey is killed. Otherwise, the gun powder is spoiled. The executable plan must ensure that we kill the turkey within 100 minutes, assuming that we are not allowed to shoot in the first 35 minutes.

The ASP + constraint code, in (Janhunen et al., 2017) and Appendix C.3.1, uses *clingo[DL/LP]*, an ASP incremental solver extended for constraints. The program is parametric w.r.t. the step counter n, used by the solver to iteratively invoke the program with the expected length of the plan. In each iteration, the solver increases n, grounds the program with this value (which, in this example, specializes it for a plan of exactly n actions) and solves it. The execution returns two plans for $n = 3$: {`do(wait,1)`, `do(load,2)`, `do(shoot,3)`} and {`do(load,1)`, `do(load,2)`, `do(shoot,3)`}.

The s(CASP) code (Figure 6.5) does not need a counter. The query `?- T #< 100, holds(T,st(dead,_, _), Actions)`, sets an upper bound to the duration `T` of the plan, and returns in `Actions` the plan with the actions in reverse chronological order: {`T=55, Actions=[shoot, load, load]`}, {`T=66, Actions=[shoot, load, wait]`}, {`T=80, Actions=[shoot, load, load, load]`}, {`T=91, Actions=[shoot, load, load, wait]`}, {`T=91, Actions=[shoot, load, wait, load]`}, {`T=96, Actions=[shoot, load, shoot, wait, load]`}.

```
% Every node must be reachable.
:- node(U), not reachable(U).
reachable(a) :- cycle(V,a).
reachable(V) :- cycle(U,V),
                reachable(U).

% Only one edge to each node.
:- cycle(U,W), cycle(V,W), U \= V.

% Only one edge from each node.
cycle(U,V) :-
  edge(U,V), not other(U,V).
other(U,V) :-
  node(U), node(V), node(W),
  edge(U,W), V \= W, cycle(U,W).

travel_path(S,Ln,Cycle) :-
  path(S,S,S,Ln,[],Cycle).
path(_,X,Y,D,Ps,[X,[D],Y|Ps]) :-
  cycle_dist(X,Y,D).
path(S,X,Y, D, Ps,Cs) :-
  D #= D1 + D2,
  cycle_dist(Z,Y,D1), Z \= S,
  path(S,X,Z,D2,[[D1],Y|Ps],Cs).

edge(X,Y) :- distance(X,Y,D).
cycle_dist(U,V,D) :-
  cycle(U,V), distance(U,V,D).

node(a).         node(b).
node(c).         node(d).

distance(b,c,31/10).
distance(c,d,L):-
    L #> 8, L #< 21/2.
distance(d,a,1).
distance(a,b,1).
distance(a,d,1).
distance(c,a,1).
distance(d,b,1).
```

Figure 6.6: Code of the Traveling Salesman problem under s(CASP).

6.3.3 The Traveling Salesman Problem (TSP)

Let us consider a variant of the traveling salesman problem (visiting every city in a country only once, starting and ending in the same city, and moving between cities using the existing connections) where we want to find out only the Hamiltonian cycles whose length is less than a given upper bound. Solutions for this problem, with comparable performance, using ASP and CLP(*FD*) appear in (Dovier et al., 2005) (also available at Appendix C.4.1 and Appendix C.4.2). The ASP encoding is more compact, even if the CLP(*FD*) version uses the non-trivial library predicate `circuit/1`, which does the bulk of the work. We will show that s(CASP) is more expressive also in this problem.

Finding the (bounded) path length in ASP requires using a specific, ad-hoc builtin that accesses the literals in a model and calls it from within a global constraint. Using *clasp* (Hölldobler and Schweizer, 2014), it would be as follows:

```
cycle_length(N) :- N = #sum [cycle(X,Y) : distance(X, Y, C) = C].
:- cycle_length(N), N >= 10.      % Cycles whose length is less than 10
```

where `#sum` is a builtin aggregate operator that here is used to add the distances between nodes in some Hamiltonian cycle.

The s(CASP) code in Figure 6.6 solves this TSP variant by modeling the Hamiltonian cycle in a manner similar to ASP and using a recursive predicate, `travel_path(S,Ln, Cycle)`, that returns in `Cycle` the list of nodes in the circuit (with the distance between

```
#show move/3. %s(CASP) directive

hanoi(N,T):-
    move_(N,0,T,a,b,c).
move_(N,Ti,Tf,Pi,Pf,Px) :-
    N #> 1, N1 #= N - 1,
    move_(N1,Ti,T1,Pi,Px,Pf),
    move_( 1,T1,T2,Pi,Pf,Px),
    move_(N1,T2,Tf,Px,Pf,Pi).
move_(1,Ti,Tf,Pi,Pf,_) :-
    Tf #= Ti + 1,
    move(Pi,Pf,Tf).

move(Pi,Pf,T):-
    not negmove(Pi,Pf,T).
negmove(Pi,Pf,T):-
    not move(Pi,Pf,T).
```

Figure 6.7: Code of the Towers of Hanoi problem under s(CASP).

every pair of nodes also in the list), starting at node `S`, and the total length of the circuit in `Ln`.

This example highlights the marriage between ASP encoding (to define models of the Hamiltonian cycle using the `cycle/2` literal) and traditional CLP (which uses the available `cycle/2` literals to construct paths and return their lengths). Note as well that we can define node distances as intervals (line 35) using a dense domain (rationals, in this case). This would not be straightforward (or even feasible) if only CLP(*FD*) was available: while CLP(*FD*) can encode CLP($\mathbb{Q}$), the resulting program would be cumbersome to maintain and much slower than the CLP($\mathbb{Q}$) version, since Gaussian elimination has to be replaced by enumeration, which actually compromises completeness (and, in the limit, termination). Additionally, in our proposal, constraints can appear in bindings and as part of the model. For example, the query `?- D #< 10, travel_path(b,D, Cycle)` returns the model {`D=61/10, Cycle=[b, [31/10], c, [1], a, [1], d, [1], b]`}. For reference, Appendix C.4.3 shows the complete output.

6.3.4 Towers of Hanoi

We will not explain this problem here as it is widely known. Let us just remind the reader that solving the puzzle with three towers (the standard setup) and n disks requires at least $2^n - 1$ movements.

Known ASP encodings, for a *standard* solver, set a bound to the number of moves that can be done, as proposed in (Gebser et al., 2008) (available for the reader's convenience at Appendix C.5.1, for `7` disks and up to `127` movements) or for an *incremental* solver, increasing the number n of allowed movements (from the *clingo 5.2.0* distribution, also available at Appendix C.5.2).

s(CASP)'s top down approach can use a CLP-like control strategy to implement the well-known Towers of Hanoi algorithm (Figure 6.7). Predicate `hanoi(N,T)` receives in `N` the number of disks and returns in `T` the number of movements needed to solve the

Table 6.2: Run-time (ms) of s(CASP) and clingo (standard and incremental) for `hanoi/2` with n disks.

	s(CASP)	**clingo 5.2.0** *standard*	**clingo 5.2.0** *incremental*
$n = 7$	**479**	3,651	9,885
$n = 8$	**1,499**	54,104	174,224
$n = 9$	**5,178**	191,267	$>$ 5 min

puzzle. The resulting partial stable model will contain all the movements and the time in which they have to be performed. For reference, Appendix C.5.3 shows the partial stable model for `?- hanoi(7,T).`

Table 6.2 compares execution time (in milliseconds) needed to solve the Towers of Hanoi with `n` disks by s(CASP) and *clingo 5.2.0* with the *standard* and *incremental* encodings. s(CASP) is orders of magnitude faster than both clingo variants because it does not have to generate and test all the possible plans; instead, as mentioned before, it computes directly the smallest solution to the problem. The standard variant is less interesting than s(CASP)'s, as it does not return the minimal number of moves — it merely checks if the problem can be solved in a given number of moves. The incremental variant is by far the slowest, because the program is iteratively checked with an increasing number of moves until it can be solved.

6.4 Discussion

We have reported on the design and implementation of s(CASP), a top-down system to evaluate constraint answer set programs, based on s(ASP). Its ability to express non-monotonic programs *à la* ASP is coupled with the possibility of expressing control in a way similar to traditional logic programming — and, in fact, a single program can use both approaches simultaneously, achieving the best of both worlds. We have also reported a very substantial performance increase w.r.t. the original s(ASP) implementation. Thanks to the possibility of writing pieces of code with control in mind, it can also beat state-of-the-art ASP systems in certain programs.

Chapter 7

Modeling and Reasoning in Event Calculus using s(CASP)

In this chapter we present an AI application of Constraint Answer Set Programming using s(CASP). Event Calculus (EC) is a sound formalism for modelling commonsense reasoning, which is essential for building AI systems featuring human-like reasoning. Logic programs that implement EC can not be easily executed directly by Prolog, and implementations using ASP and/or SAT solvers are limited to Discrete Event Calculus (DEC). Using s(CASP) the resulting translation is more expressive thanks to its ability to represent classical *and* default *negation. Additionally, it is able to represent continuous change (e.g., time and other physical quantities) thanks to the integration of constraint solvers over dense, unbound domains such as CLP($\mathbb{Q}$). We present two simple examples to highlight the expressiveness of the resulting reasoner, which besides allowing deductive reasoning also paves the way to abductive reasoning. We also show that s(CASP) with dense domains outperforms a state-of-the-art DEC reasoner executed in a mature highly optimized ASP system.*

The ability to model continuous characteristics of the world is essential for Commonsense Reasoning (CR) in many domains that require dealing with continuous change: time, the height of a falling object, the gas level of a car, the water level in a sink, etc. Event Calculus (EC) is a formalism based on many-sorted predicate logic (Kowalski and Sergot, 1989; Mueller, 2014) that can represent continuous change and capture the commonsense law of inertia, whose modeling is a pervasive problem in CR. In EC, time-dependent properties and events are seen as objects and reasoning is performed on the truth values of properties and the occurrences of events at a point in time.

Answer Set Programming (ASP) has also been used to model the Event Calculus (Lee and Palla, 2012, 2019). But as we mentioned before, classical implementations of EC under ASP are limited to variables ranging over discrete and bound domains and use mechanisms such as *grounding* and SAT solving to find out models (called *answer sets*) of ASP programs. However, EC reasoning often needs variables ranging over dense domains (e.g., those involving time or physical quantities) to faithfully represent the properties of these domains.

In this chapter we use s(CASP) as the underlying reasoning infrastructure to model and reason in Event Calculus. As we explain in Chapter 6, the s(CASP) system is an implementation of Constraint Answer Set Programming over first-order predicates which combines ASP and constraints. It features predicates, constraints among non-ground variables, uninterpreted functions, and, most importantly, a top-down, query-driven execution strategy. These features make it possible to return answers with non-ground variables (possibly including constraints among them) and to compute partial models by returning only the fragment of a stable model that is necessary to answer a given query. Thanks to its interface with constraint solvers, sound non-monotonic reasoning with constraints is possible.

The EC reasoner based on s(CASP) achieves more conciseness and expressiveness than other related implementations. This is because continuous quantities can be faithfully modeled as dense domains, while in other proposals (Lee and Palla, 2019; Mellarkod et al., 2008) such quantities had to be discretized, and therefore, they lose precision or even soundness. Additionally, in our approach the amalgamation of ASP and constraints and its realization in s(CASP) is considerably more natural: under s(CASP), answer set programs are executed in a goal-directed manner so constraints encountered along the way are collected and solved dynamically as execution proceeds — this is very similar to the way in which Prolog has been extended with constraints. The implementation of other ASP systems featuring constraints is considerably more complex.

7.1 Motivation and Related Work

Previous work translated *discrete* EC into ASP (Lee and Palla, 2012, 2019) by reformulating the EC models as first-order stable models and translating the (almost universal) formulas of EC into a logic program that preserves stable models. Given a finite domain, EC2ASP (and its evolution, F2LP) compile (discrete) Event Calculus formulas into ASP programs (Lee and Palla, 2012, 2019). This translation scheme relies on two facts: second order circumscription and first order stable model semantics coincide on canonical formulas, and almost-universal formulas can be transformed into a logic program while preserving the stable models. As a result, computing models of Event Calculus descriptions can be done by computing the stable models of an appropriately generated program.

Clearly, approaches featuring discrete domains cannot faithfully handle continuous quantities such as time. In addition, because of their reliance on SAT solvers to find the stable models, they can only handle *safe* programs. In contrast, the s(CASP) system (see Chapter 6), because of its direct support for predicates with arbitrary terms, constructive negation, and the novel *forall mechanism*, program safety is not a requirement. Thus, s(CASP) can model Event Calculus axioms much more directly and elegantly.

The approaches mentioned above assume discrete quantities and do not support reasoning about continuous time or change. As long as SAT-based ASP systems are used to model Event Calculus, continuous fluents cannot be straightforwardly expressed since they require unbounded, dense domains for the variables. The work closest to incorporating continuous time makes use of SMT solvers. In this approach, constraints are incorporated into ASP and the grounded theory is executed using an SMT solver (Lee and Meng, 2013a). However, this approach has not been directly applied to modeling the Event Calculus. The closest tool chain is ASPMT2SMT (Bartholomew and Lee, 2014) that uses *gringo* to partially ground the ASPMT theories and generate constraints that are processed by *Z3*. However, regular, discrete ASP variables are at the heart of the model, and these are grounded and used to generate the constraints. Therefore, if these discrete variables approximate continuous variables in the model, the constraints generated will only approximate the conditions of the original problem and therefore their solutions will also be an approximation (or a subset) of the solutions for the real problem. In other words, the initial discretization done for the ASP variables will be propagated via the generated constraints to the final solutions that will, in the best case, be a discretized version of the actual solutions. As an example, if time is discretized, the solutions to the model will suffer from this discretization.

Other ASP-based approaches to deal with planning in continuous domains include, for example, action languages (Gelfond and Lifschitz, 1993) which were developed to model the elements of natural language that are used to describe the effects of actions, and PDDL+ (Fox and Long, 2002), which was developed to allow reasoning with continuous time-dependent effects. Action languages have been implemented using answer set programming (Gelfond and Kahl, 2014) and there have been extensions of action languages to accommodate time: for example, the action language C+ has been extended to accommodate continuous time (Lee and Meng, 2013a). PPDL+ models temporal behavior in terms of the initiation and termination of processes, which in turn act upon the numeric components of states. Processes are initiated and terminated instantaneously by actions or exogenous events. Continuous changes are made by concurrent processes. In PDDL+, reasoning is monotonic and thus the degree of elaboration tolerance is low. There are implementations of PDDL+ using constraint answer set programming (CASP) (Balduccini et al., 2016) though these have not been applied to modeling the Event Calculus and requires the use of discrete variables to model some quantities, e.g., time.

EC can be written as a (Horn-clause) logic program, but it cannot be executed directly by Prolog (Shanahan, 2000), as it lacks some necessary features, such as constructive negation, deduction of negative literals, and (to some extent) detection of infinite failure (Mueller, 2008a). A common approach is to write a metainterpreter specific to the EC variant at hand. This can be as complex as writing a (specialized) theorem prover or, more often, a specialized interpreter whose correctness is difficult to ascertain (see the code at (Chittaro and Montanari, 1996)). Therefore, some Prolog implementations of EC do not completely formalize the calculus or implement a reduced version. In our case, we leverage on the capabilities of s(CASP) to provide constructive, sound negation, negative rule heads, and loop detection (see Chapter 6).

7.2 Event Calculus

EC is a formalism for reasoning about events and change (Mueller, 2014), of which there are several axiomatizations. There are three basic, mutually related, concepts in EC: *events*, *fluents*, and *time points* (see Fig. 7.1a). An event is an action or incident that may occur in the world: for instance, a person dropping a glass is an event. A fluent is a time-varying property of the world, such as the altitude of a glass. A time point is an instant in time. Events may happen at a time point; fluents have a truth value at any time point or over an interval, and their truth values are subject to change upon the occurrence of an event. In addition, fluents may have (continuous) quantities associated with them when they are true. For example, the event of dropping a glass initiates the fluent that captures that the glass is falling, and perhaps its height above the ground, and the event of holding a glass terminates the fluent that the glass is falling. An EC description consists of a universal theory and a domain narrative. The universal theory is a conjunction of EC axioms and the domain narrative consists of the causal laws of the domain and the known events and fluent properties.

Circumscription (McCarthy, 1980) is applied to EC domain narratives to minimize the extension of predicates and has two effects: the only events that happen are those defined and the only effects of events are those defined.

The original EC (OEC) was introduced by Kowalski and Sergot in 1986 (Kowalski and Sergot, 1989). OEC has sorts for event occurrences, fluents, and time periods. In this chapter we use the Basic Event Calculus (BEC) formulated by Shanahan (Mueller, 2008a). Fig. 7.1b summarizes the seven BEC axioms. An explanation of these axioms follows:

- **Axiom BEC1**. A fluent f is stopped between time points t_1 and t_2 iff it is terminated or released by some event e that occurs after t_1 and before t_2.
- **Axiom BEC2**. A fluent f is started between time points t_1 and t_2 iff it is initiated or released by some event e that occurs after t_1 and before t_2.

Predicate	**Meaning**
$InitiallyN(f)$	fluent f is false at time 0
$InitiallyP(f)$	fluent f is true at time 0
$Happens(e,t)$	event e occurs at time t
$Initiates(e,f,t)$	if e happens at time t, f is true and not released from the commonsense law of inertia after t
$Terminates(e,f,t)$	if e occurs at time t, f is false and not released from the commonsense law of inertia after t
$Releases(e,f,t)$	if e occurs at time t, f is released from the commonsense law of inertia after t
$Trajectory(f_1,t_1,f_2,t_2)$	if f_1 is initiated by an event that occurs at t_1, then f_2 is true at t_2
$StoppedIn(t_1,f,t_2)$	f is stopped between t_1 and t_2
$StartedIn(t_1,f,t_2)$	f is started between t_1 and t_2
$HoldsAt(f,t)$	fluent f is true at time t

(a) BEC predicates (e = event, f, f_1, f_2 = fluents, t, t_1, t_2 = timepoints).

BEC1. $StoppedIn(t_1,f,t_2) \equiv$
$$\exists e,t\ \ (\, Happens(e,t) \land t_1 < t < t_2 \land (\, Terminates(e,f,t) \lor Releases(e,f,t)\,)\,)$$

BEC2. $StartedIn(t_1,f,t_2) \equiv$
$$\exists e,t\ \ (\, Happens(e,t) \land t_1 < t < t_2 \land (\, Initiates(e,f,t) \lor Releases(e,f,t)\,)\,)$$

BEC3. $HoldsAt(f_2,t_2) \leftarrow$
$$Happens(e,t_1) \land Initiates(e,f_1,t_1) \land Trajectory(f_1,t_1,f_2,t_2) \land \neg StoppedIn(t_1,f_1,t_2)$$

BEC4. $HoldsAt(f,t) \leftarrow \quad InitiallyP(f) \land \neg StoppedIn(0,f,t)$

BEC5. $\neg HoldsAt(f,t) \leftarrow \quad InitiallyN(f) \land \neg StartedIn(0,f,t)$

BEC6. $HoldsAt(f,t_2) \leftarrow$
$$Happens(e,t_1) \land Initiates(e,f,t_1) \land t_1 < t_2 \land \neg StoppedIn(t_1,f,t_2)$$

BEC7. $\neg HoldsAt(f,t_2) \leftarrow$
$$Happens(e,t_1) \land Terminates(e,f,t_1) \land t_1 < t_2 \land \neg StartedIn(t_1,f,t_2)$$

(b) BEC axioms.

Figure 7.1: Formalization of Basic Event Calculus from (Mueller, 2014).

- **Axiom BEC3**. A fluent f_2 is true at time t_2 if a fluent f_1 initiated at t_1 does not finish before t_2 and it makes fluent f_2 be true.[1]

- **Axiom BEC4**. A fluent f is true at time t if it is true at time 0 and is not stopped on or before t.

- **Axiom BEC5**. A fluent f is false at time t if it is false at time 0 and it is not started on or before t.

- **Axiom BEC6**. A fluent f is true at time t_2 if it is initiated at some earlier time t_1 and it is not stopped before t_2.

- **Axiom BEC7**. A fluent f is false at time t_2 if it is terminated at some earlier time t_1 and it is not started on or before t_2.

7.3 From Event Calculus to s(CASP)

7.3.1 Modeling EC with s(CASP)

Two key factors contribute to s(CASP)'s ability to model Event Calculus: the preservation of non-ground variables during the execution and the integration with constraint solvers.

Treatment of variables in s(CASP): Thanks to the usage of non-ground variables, s(CASP) is able to directly model Event Calculus axioms that would otherwise require "unsafe" rules. In classical ASP, a rule is safe when every variable that appears in its head or in a negated literal in its body also appears in a positive literal in the body of the rule, and it is unsafe otherwise. Safety guarantees that every variable can be grounded. For example, BEC4 is unsafe since parameter t, which appears in the head, does not appear in a positive literal in the body (i.e., it only appears in $\neg StoppedIn(0, f, t)$). A SAT-based ASP solver such as *clingo* (Gebser et al., 2014) will not be able to directly process unsafe rules like this. However, the top-down execution strategy of s(CASP) makes it possible to keep logical variables both during execution and in answer sets and therefore free (logical) variables can be handled in heads and in negated literals.

Integration with constraint solvers: The s(CASP) system has a generic interface to enable plugging in constraint solvers. s(CASP) currently uses Holzbaur's CLP(Q) linear constraint solver (Holzbaur, 1995), that supports the constraints $<, >, =, \neq, \leq, \geq$. As we saw, the definitions and axioms of BEC require inequality comparisons over time

[1]For implementation convenience, and without loss of expressiveness, we assume that argument t_2 in $Trajectory(f_1, t_1, f_2, t_2)$ is not a time difference w.r.t. t_1, but an absolute time after t_1.

```
%% BEC1
stoppedIn(T1,F,T2) :-
        T1 #< T, T #< T2,
        terminates(E,F,T),
        happens(E,T).
stoppedIn(T1,F,T2) :-
        T1 #< T, T #< T2,
        releases(E,F,T),
        happens(E,T).
%% BEC2
startedIn(T1,F,T2) :-
        T1 #< T, T #< T2,
        initiates(E,F,T),
        happens(E,T).
startedIn(T1,F,T2) :-
        T1 #< T, T #< T2,
        releases(E,F,T),
        happens(E,T).
%% BEC3
holdsAt(F2,T2) :-
        initiates(E,F1,T1),
        happens(E,T1),
        trajectory(F1,T1,F2,T2),
        not stoppedIn(T1,F1,T2).
%% BEC4
holdsAt(F,T) :-
        0 #< T,
        initiallyP(F),
        not stoppedIn(0,F,T).
%% BEC5
-holdsAt(F,T) :-
        0 #< T,
        initiallyN(F),
        not startedIn(0,F,T).
%% BEC6
holdsAt(F,T) :-
        T1 #< T,
        initiates(E,F,T1),
        happens(E,T1),
        not stoppedIn(T1,F,T).
%% BEC7
-holdsAt(F,T) :-
        T1 #< T,
        terminates(E,F,T1),
        happens(E,T1),
        not startedIn(T1,F,T).
%% Consistency
:- -holdsAt(F,T), holdsAt(F,T).
```

Figure 7.2: BEC axioms modeled in s(CASP)

points, and the ability of s(CASP) to make use of constraint solvers makes it ideal to model continuous time in EC.

7.3.2 Translating the BEC Axioms into s(CASP)

Our translation of the BEC axioms into s(CASP) is similar to that of the systems EC2ASP and F2LP (Lee and Palla, 2012, 2019), but we differ in some key aspects that improve performance and are relevant for expressiveness: *the treatment of rules with negated heads, the possibility of generating unsafe rules,* and *the use of constraints over rationals*. We describe below, with the help of a running example, the translation that turns logic statements (as found in BEC) into an s(CASP) program. The code corresponding to the translations of the axioms of BEC in Fig. 7.1b can be found in Fig. 7.2. s(CASP) code follows the syntactical conventions of logic programming: constants (including function names) and predicate symbols start with a lowercase letter and variables start with an uppercase letter. In addition, logic constraints are written as constraints in s(CASP), (e.g., `#<`) to make it clear that they do not correspond to Prolog's arithmetic comparisons:

- **Atoms and Constants:** Their names are preserved. *Uniqueness of Names* (Shanahan, 1999) is assumed by default (and enforced) in logic programming.
- **Constraints:** Predicates that represent constraints (e.g., on time) are directly translated to their counterparts in s(CASP). E.g., $t_1 < t_2$ becomes `T1 #< T2`, which is handled by CLP(Q), one of the available constraint solvers. The translation (and s(CASP) itself) is parametric on the constraint domain.
- **Definitions:** The axiomatization of BEC uses definitions of the form $D(x) \equiv \exists y B(x,y)$, where $B(x,y)$ is a conjunction of (negated) atoms, disjunctions of atoms, and constraints (e.g., BEC1). The use of definitions makes it easier to build conceptual blocks out of basic predicates. However, for performance reasons we treat them as if they were written as $\forall x(D(x) \leftarrow \exists y B(x,y))$, following (Lee and Palla, 2019). Intuitively, if we ignore the truth value of D in the (partial) models that s(CASP) generates, the models returned using implication and/or equivalence are the same, and the literal D can be ignored because if were expanded where it is used, it would have disappeared. Additionally, s(CASP) internally performs Clark's completion (Clark, 1978) to the s(CASP) program, and therefore, we can assume that s(CASP) rules expresses all possible ways in which heads can be true.
- **Rules with Positive Heads:** A rule (e.g., BEC6)

  $$\forall x(H(x) \leftarrow \exists y(A(y) \land \neg B(x,y) \land x < y))$$

  where $x < y$ is a constraint, is translated into

  ```
  h(X) :- X #< Y, a(Y), not b(X,Y).
  ```

  s(CASP) performs left-to-right evaluation, and since constraint solvers are deterministic, constraining variables as soon as possible helps reduce the size of the search tree.
- **Rules with Negated Heads:** BEC rules 5 and 7 infer negated heads $\neg HoldsAt(f,t)$ while rules 4 and 6 infer positive heads $HoldsAt(f,t)$, i.e., they follow, respectively, the scheme

  $$\forall x(H(x) \leftarrow \exists y A(x,y)) \ \land \ \forall x(\neg H(x) \leftarrow \exists y B(x,y))$$

  The standard approach to translate rules with negated heads is to convert them into global constraints (Lee and Palla, 2012):

  ```
  :- b(X,Y), h(X).
  ```

  Our approach is to define instead a rule for the literal `-h(X)` that captures the explicit evidence that `h(X)` is false:

  ```
  -h(X) :- b(X,Y).
  ```

```
happens(turn_on, 2).
happens(turn_off, 4).
happens(turn_on, 6).

initiates(turn_on, on, T).
terminates(turn_off, on, T).
trajectory(on, T1, red, T2) :-
      T1 #< T2, T2 #< T1+1.
trajectory(on, T1, green, T2) :-
      T2 #>= T1+1.
releases(turn_on, red, T).
releases(turn_on, green, T).
```

Figure 7.3: Narrative of the light scenario modeled in s(CASP)

which makes it possible to call `-h(X)` in a top-down execution. This construct was termed *classical* negation in (Marple et al., 2017a) and behaves as a regular predicate, except that the s(CASP) compiler, to ensure that `-h(X)` and `h(X)` cannot be simultaneously true, automatically adds the global constraint `:- -h(X), h(X)` Therefore, s(CASP) can detect an inconsistency (and will return an empty model) if both $HoldsAt(f,t)$ and $\neg HoldsAt(f,t)$ can be simultaneously derived from an BEC narrative. Since circumscription is not applied to the EC theory, not being able to derive $HoldsAt(f,t)$ does not immediately determine that its negation is true. We will see how this is connected with the translation of the narrative.

- **Rules with Disjunctive Bodies:** A rule (e.g., BEC1)

  $$\forall x[H(x) \leftarrow \exists y(\ (A(x,y) \vee B(x,y)) \wedge C(x,y)\ )]$$

  is translated into two separate clauses:

  ```
  h(X) :- a(X,Y), c(X,Y).
  h(X) :- b(X,Y), c(X,Y).
  ```

7.3.3 Translation of the Narrative

The definition of a given scenario (its *narrative* part) states the basic actions and effects using the predicates in Fig. 7.1a. EC assumes circumscription of the predicates defined in the *narrative*: the events (resp., effects) known to occur are the only events (resp., effects) that occur. Note that this is automatic in s(CASP), since it produces the Clark's completion of s(CASP) programs when generating the dual program. In addition, global constraints can restrict the admissible states of the system.

Every basic BEC predicate $P(x)$ (where P can stand for an event occurrence, an effect of an event on a fluent, etc.) is translated into an s(CASP) rule $P(x) \leftarrow \gamma$, where γ states **all** the cases where $P(x)$ is true. In many cases, these are *facts*, but in other cases γ captures the conditions for $P(x)$ to hold.

Let us consider example 14 in (Mueller, 2014), which reasons about turning a light switch on and off. Fig. 7.3 shows the encoding of this example under s(CASP).

- **Events:** The description below (translated in lines 1-3 of Fig. 7.3):

$$\begin{aligned} Happens(e,t) \;\equiv\;& (e = TurnOn \wedge (t = 2 \vee t = 6)) \;\vee \\ & (e = TurnOff \wedge t = 4) \end{aligned}$$

  states that the *TurnOn* event will happen at time $t = 2$ and $t = 6$, and that *TurnOff* will happen at $t = 4$.

- **Event effects:** When the event *TurnOn* happens, the light is put in *on* status; similarly, when the event *TurnOff* happens, the *on* status of the light is terminated. In both cases, this can happen at any time t (lines 5 and 6 in Fig. 7.3)

- **Release from Inertia:** When turned on, the light emits red light within the first second, and then green light is emitted. *Trajectory* expresses how this change depends on the time elapsed since an event occurrence. The *Trajectory* formula has the shape $P(x) \leftarrow \gamma$, as we need to state the (time) conditions for the fluent to become activated (see lines 7-10 in Fig. 7.3). *Releases* states that the color of the light is released from the commonsense law of inertia. After a fluent is released, its truth value is not determined by BEC and it can change. Thus, there may be models in which the fluent is true, and models in which the fluent may be false. Releasing a fluent (see lines 11 and 12 in Fig. 7.3) frees it up so that other axioms in the domain description can be used to determine its truth value, thus allowing us to represent continuous change of the fluent.

- **State Constraints:** State constraints usually contain $HoldsAt(f,t)$ or $\neg HoldsAt(f,t)$ and represent restrictions on the models. In our running example, a light cannot be red and green at the same time: $\forall t. \neg(HoldsAt(Red,t) \wedge HoldsAt(Green,t))$. This is translated as `:- holdsAt(red,T), holdsAt(green,T)` Adding this constraint to the program in Fig. 7.3 does not change its models. However, if we change line 8 stating that the light is red for 2 seconds (i.e., `T2 #< T1+2`), the state constraint is violated and therefore there are no valid models.

- **A Note on using $\neg HoldsAt(f,t)$ in γ:** The basic BEC predicates may depend on what the BEC theory can deduce, e.g., γ may depend on $HoldsAt(f,t)$ or $\neg HoldsAt(f,t)$ (see Fig. 7.4). $HoldsAt(f,t)$ can be invoked directly, but $\neg HoldsAt(f,t)$ ought to be called using classical negation, e.g., `-holdsAt(F,T)`. The reason is that since BEC does not apply circumscription to its axioms, we can deduce only the truth (or falsehood) of a predicate when we have direct evidence of either of them — i.e., what the positive (`holdsAt(F,T)`) and negative (`-holdsAt(F,T)`) heads provide.

```
1  #include bec_theory.
2
3  max_level(10):- not max_level(16).
4  max_level(16):- not max_level(10).
5
6  initiallyP(level(0)).
7  happens(overflow,T).
8  happens(tapOn,5).
9
10 initiates(tapOn,filling,T).
11 terminates(tapOff,filling,T).
12 initiates(overflow,spilling,T):-
13    max_level(Max),
14    holdsAt(level(Max), T).
15 releases(tapOn,level(0),T):-
16    happens(tapOn,T).
17
18 trajectory(filling,T1,level(X2),T2):-
19    T1 #< T2, X2 #= X+T2-T1,
20    max_level(Max), X2 #=< Max,
21    holdsAt(level(X),T1).
22 trajectory(filling,T1,level(overflow),T2):-
23    T1 #< T2, X2 #= X+T2-T1,
24    max_level(Max), X2 #> Max,
25    holdsAt(level(X),T1).
26 trajectory(spilling,T1,leak(X),T2):-
27    holdsAt(filling, T2),
28    T1 #< T2, X #= T2-T1.
```

Figure 7.4: Encoding of an Event Calculus narrative with continuous change

7.3.4 Continuous Change: A Complete Encoding

We consider now an example from (Shanahan, 1999): a water tap fills a vessel, whose water level is subject to continuous change. When the level reaches the bucket rim, it starts spilling. We will present the main ideas behind its encoding (Fig. 7.4) and will show some queries we can ask about its state and behavior.

- **Continuous Change:** The fluent *Level*(x) represents that the water is at level x in the vessel. The first *Trajectory* formula (lines 18-21) determines the time-dependent value of the *Level*(x) fluent,[2] which is active as long as the *Filling* fluent is true and the rim of the vessel is not reached. Additionally, the second *Trajectory* formula (lines 22-25) allows us to capture the fact that the water reached the rim of the vessel and overflowed.

- **Triggered Fluent:** The fluent *Spilling* is triggered (lines 12-14) when the water level reaches the rim of the vessel. As a consequence, the *Trajectory* formula in lines 26-28 starts the fluent *Leak*(x) and captures the amount of water leaked while the fluent *Spilling* holds.

- **Different Worlds:** The clauses in lines 3-4 force the vessel capacity to be either 10 or 16, i.e., they create two possible worlds/models: {`max_level(10)`, `not max_level(16)`, ...} and {`max_level(16)`, `not max_level(10)`, ...}. The same mechanism can be used to state whether an event happens or not. For this, a keyword `#abducible` is provided as a shortcut in s(CASP). We will use it in the *Abduction* subsection later on.

[2]For simplicity the amount of water filled/leaked correspond directly to how long the water has been pouring in / spilling from the vessel.

7.4 Examples and Evaluation

The benchmarks used in this section are available as part of the s(CASP) distribution at `https://gitlab.software.imdea.org/ciao-lang/sCASP/`. They were run on a MacOS 10.14.3 laptop with an Intel Core i5 at 2GHz.

Deduction: Deduction determines whether a state of the world is possible given a theory (in our case, BEC) and an initial narrative. We can perform deduction in BEC for the previous examples through queries to the corresponding s(CASP) program. For the lights scenario (Fig. 7.3):

`?- holdsAt(on,3)` succeeds: it deduces that the light is on at time 3.

`?- -holdsAt(on,5)` succeeds: the light is not on at time 5.

`?- holdsAt(F,3)` is true in one stable model containing `holdsAt(green,3)` and `holdsAt(on,3)`, meaning that at time 3, the light is on and green.

In the water level scenario (Fig. 7.4) we can make queries involving time and the water level:

`?- holdsAt(level(H),15/2)` is true when `H=5/2`.

`?- holdsAt(level(5/2),T)` is true when `T=15/2`.

Note that, as explained with more detail in the *Evaluation* subsection below, s(CASP) can operate and answer correctly queries involving rationals without having to modify the original program to introduce domains for the relevant variables or to *scale* the constants to convert rationals into integers.

Abduction: Abductive reasoning tries to determine a sequence of events/actions that reaches a final state. In the case of ASP, actions are naturally captured as the set of atoms that are true in a model which includes the initial and final states and are consistent with BEC. For the water scenario, (Fig. 7.4), let us assume we want to reach water level 14 at time 19. The query `?- holdsAt(level(14),19)`will return a single model with a vessel size of 16 and the rest of the atoms in the model capturing what must (not) happen to reach this state.

More interesting abductive tasks can be performed: adding the line `#abducible happens(tapOff,U)` to the program, we specify that it is possible (but not necessary) for the tap to close at some time `U`. As we mentioned in Section 7.3.4, this directive is translated into code that creates different worlds/models. The query `?- holdsAt(spilling, T)` determines if the water may overspill and under which conditions. s(CASP) returns two models:

- One containing `T > 15, holdsAt(spilling,T), happens(tapOn,5), 5 < U < 15, not happens(TapOff,U), max_level(10)` meaning that the water will spill at `T=15` if the vessel has a capacity of 10, the tap is open at `T=5`, and it is **not** closed between times 5 and 15.
- Another similar model, with the water spilling at `T=21` in a vessel with capacity of 16 and where the tap was not closed before `U=21`.

Note that s(CASP) determined the truth value of *Happens* and, more importantly, performed constraint solving to infer the time ranges during which some events ought (and ought not) to take place, represented by the negated atoms in the models inferred by constructive negation. Since all relevant atoms have a time parameter, they actually represent a *timed plan*. Due to the expressiveness of constraints, this plan contains information on time points when events must (not) happen and also on time *windows* (sometimes in relation with other events) during which events must (not) take place. Note that it would be impossible to (finitely) represent this interval with ground atoms, as it corresponds to an infinite number of points.

Evaluation: Comparing directly our implementation of BEC in s(CASP) with implementations in other systems is not easy: most previous systems implemented *discrete* Event Calculus (DEC) and they do not support continuous quantities. One of them is F2LP (Lee and Palla, 2019), an ASP-based system that according to (Lee and Palla, 2012) outperforms *DEC reasoner* (Mueller, 2008b), reported by (Lee and Palla, 2012) as the more efficient SAT-based implementation. F2LP is a tool that executes DEC by turning first order formulas under the stable model semantics into a logic program w.o. constraints that is evaluated using an ASP solver.

We compared the light scenario in Fig. 7.3 running under s(CASP) with the F2LP translation under *clingo 5.1.1*, the current version of the state-of-the-art ASP system. Since the directive `#domain` is no longer available in *clingo*, we had to adapt the translation of F2LP adding `timestep(1..10)` and `timestep/1` to make the clauses safe (Appendix D.1). While under s(CASP) we can reason about time points in an unbounded continuous domain, the previous encoding of F2LP will make time belong to the integers from 1 to 10. Therefore, since the light is red for $t > 2, t < 3$ and for $t > 6, t < 7$, there are no integer time points from 1 to 10 when the emitted light is red. I.e., for the query `?- holdsAt(red,T)` the execution under *clingo* fails and the execution under s(CASP) returns the constraint `T #> 2, T #< 3` and `T #> 6, T #< 7`.

Table 7.1: Run time (ms) comparison for the light scenario.

Queries	s(CASP)	F2LP+clingo
`?- holdsAt(red,6.9).`	216	**73**
`?- holdsAt(red,6.99).`	**217**	8,798
`?- holdsAt(red,6.999).`	**213**	> 5 min.

In order to find at what time point the light red is on under *clingo*, we had to modify the program generated by F2LP to refine the `timestep` domain with `timestep(1..10*P) :- precision(P)`, where the new predicate `precision(P)` makes it possible to have a finer grain for the possible values of `timestep` by increasing the value of `P`. E.g, for `P=10` it is possible to check if the light is red at time $t = 6.9$ by querying `?- holdsAt(red,69)`, for `P=100` it is possible to check for $t = 6.99$ by querying `?- holdsAt(red,699)`, and so on. This modification (Appendix D.2) obfuscates the resulting encoding (and for more complex narratives it would be harder or even infeasible) and also impact negatively its performance. Table 7.1 shows that additional precision in the F2LP encoding (to handle each of the queries) increases the execution run-time of *clingo* by orders of magnitude. On the other hand, s(CASP) does not have to adapt its encoding/queries and its performance does not change.

7.5 Discussion

We showed how Event Calculus can be modeled in s(CASP), a goal-directed implementation of constraint answer set programming with predicates, with much fewer limitations than other approaches. s(CASP) can capture the notion of continuous time (and, in general, fluents) in Event Calculus thanks to its grounding-free top-down evaluation strategy. It can also represent complex models and answer queries in a flexible manner thanks to the use of constraints.

The main contribution of the chapter is to show how Event Calculus can be directly modeled using s(CASP), an ASP system that seamlessly supports constraints. The modeling of Event Calculus using s(CASP) is more elegant and faithful to the original axioms compared to other approaches such as F2LP, where time has to be discretized. While other approaches such as ASPMT do support continuous domains, their reliance on SMT solvers makes their implementation really complex as associations among variables are lost during grounding. The use of s(CASP) brings other advantages: for example, the justification for the answers to a query is obtained for free, since in a query-driven system, the justification is merely the trace of the proof. Likewise, explanations for observations via abduction are also generated for free, thanks to the goal-directed, top-down execution of s(CASP).

To the best of our knowledge, this approach is the only one that faithfully models continuous-time Event Calculus under the stable model semantics. All other approaches discretize time and thus do not model EC in a sound manner. Our approach supports both deduction and abduction with little or no additional effort.

The work reported in this chapter can be seen as the first serious application of s(CASP) (see Chapter 6). It illustrates the advantages that goal-directed ASP systems have over grounding and SAT solver-based ones for certain applications. Our future work includes applying the s(CASP) system to solving planning problems where a generated plan must obey real-time constraints.

Chapter 8

Conclusions and Future Work

This thesis presents approaches to improving constraint logic programming by addressing the main research challengers described in Chapter 1: (i) extend theoretical foundations and provide a flexible Tabled Constraint Logic Programming framework, (ii) design a goal-directed non-monotonic reasoner to compute Constraint Answer Set Programming.

Although other frameworks have been proposed for Tabled Constraint Logic Programming, they limited the use of entailment and/or projection and they do not provide a clear interface to facilitate the integration of different constraint solvers. Moreover, mainly due to those limitations, they have not been used in real applications. Our proposal provides a modular interface that we used to re-implement the fixpoint algorithm of CiaoPP, an analyzer and optimizer suite for logic programs, part of the Ciao development environment.

Concerning the evaluation of Constraint Answer Set Programs we re-implemented and extended a goal-directed non-monotonic reasoner that does not require the grounding phase. The re-implemented interpreter lets Prolog handle natively operations such as those involving constraint and the extended compiler and *forall* algorithm make it possible the evaluation of CASP programs using arbitrary constraint domains and, therefore, it does not restrict the constraint domains or the type and number of models that can be computed.

Let us review the contributions of this thesis, advanced in Chapter 1.3, in the light of the evaluation results and other validation showed throughout the previous chapters.

8.1 Mod TCLP: Summary

The first contributions are the foundations and implementation of Mod TCLP, a generic framework that consists in a modular framework that facilitates the integration of different constraint solvers through a simple interface, and performs a full implementation of the entailment and projection of constraint stores. In particular, we made the specific following contributions:

- We extended the theoretical basis of Tabled Constraint Logic Programming for a top-down execution in Chapter 2. We proved soundness and completeness of TCLP under a more efficient answer management strategy, and we formulated refined termination properties by stating additional conditions of programs/queries using non constraint-compact constraint domains to ensure termination (including the Herbrand domain).

- We designed and implemented Mod TCLP, a modular framework that provides a simple interface to integrate different constraint solvers and fully implement the call/answer entailment phase of TCLP. Details of Mod TCLP and the integration of different constraint solver are described in Chapter 3.

- We validated the flexibility of Mod TCLP and its performance with the integration of a solver previously written in C, CLP($\mathbb{D}_{\leq}$), two existing classical solvers, CLP($\mathbb{Q}$/$\mathbb{R}$), and a new solver, CLP($\mathbb{L}at$). In some of the benchmarks large savings are attained w.r.t. non-tabled/tabled executions, even taking into account the penalty to pay for the additional flexibility and modularity.

- In Chapter 4 we proposed the use of Mod TCLP to implement a framework for incremental evaluation of lattice-based aggregates under a new semantics consistent with the fixpoint semantics.

- Finally, in Chapter 5 we re-implemented PLAI, a fixpoint computation algorithm for abstract interpretation using Mod TCLP, which, to the best of our knowledge, is the first real application of Tabled Constraint Logic Programming. The resulting encoding size is one-third of the original code and, since the dependencies between predicates and analysis restarts are kept by the tabling engine, the TCLP version is much simpler than the current implementation in CiaoPP. Additionally, we evaluate its performance by analysing several programs with two relevant abstract domains (Groundness and Sharing+Freeness) and, in most cases, the TCLP implementation showed improved performance (with speedups of up to 1.6$\times$).

8.2 s(CASP): Summary

We addressed the limitation of bottom-up implementations of Constraint Answer Set Programming with s(CASP), a goal-directed, non-monotonic reasoner which computes CASP programs without grounding. In particular, we have made the following contributions:

- We designed and implemented s(CASP), a top-down system to compute CASP programs based on s(ASP). Its ability to evaluate non-monotonic programs is coupled with the possibility of writing algorithms using the execution strategies similar to Prolog's. The details of s(CASP) model and implementation are described in Chapter 6.
- We validated the expressiveness of s(CASP) w.r.t. ASP, CLP, and other ASP systems featuring constraints with several examples, and showed it has better performance than highly optimized ASP systems in Chapter 6.
- Finally, in Chapter 7 we described a framework to model and reason in Event Calculus, the first serious application of s(CASP). This framework can capture the notion of continuous time (and, in general, continuous fluents) thanks to the grounding-free top-down evaluation strategy of s(CASP). To the best of our knowledge, this approach is the only one that faithfully models continuous time calculus under the stable model semantics. Note that all other approaches we are aware of discretize time and thus do not model the EC accurately and faithfully.

8.3 Directions for Future Research

The resulting proposals, presented in this thesis, contribute to improving the expressiveness and performance of Constraint Logic Programming and the applications presented constitute an evidence of a step forward. Nevertheless, we have already identified some future research directions and issues that, in some cases, we have started to explore.

- The first main future work line is the extension of s(CASP) with tabling featuring call and answer entailment. A preliminary attempt to incorporate tabling uses Mod TCLP to detect repeated call by checking entailment between their respective call paths (due to the non-monotonic reasoning subsumed calls may belong to different models). To reduce the memory footprint, it would require a more compact representation of the call path and/or tracking only the active derivation.

The extended theoretical foundation, in terms of soundness, completeness, and termination, of Tabled Constraint Logic Programming, and the modular design of Mod TCLP, pave the way to new research directions:

- Design and implement a TCLP interface for CLP($\mathbb{FD}$) (Díaz and Codognet, 1993; Dincbas et al., 1988; Van Hentenryck, 1989), a constraint solver over finite domains. CLP($\mathbb{FD}$) is widely used to model discrete problems such as scheduling, planning, packing, and timetabling. The implementation is not straightforward due to the cost of expressing projection and check entailment inside CLP($\mathbb{FD}$) (Carlson et al., 1994). But as seen in Section 2.6, we can use partial projections to check call entailment. In the case of CLP($\mathbb{FD}$), the partial projection could use the domains of the constrained variables of the call and discard the constraints between these variables.

- Evolve CLP($\mathbb{L}at$), the constraint solver over lattices, to allow reasoning on ontologies, a TCLP-based reasoner would be able to feature automatic reuse of properties from more general concepts and to combine and/or aggregate answers into a more general element of the ontology.

- Explore richer, faster, and more flexible implementations of abstract interpretation-based analyzers. We already show the improvement in terms of simplicity of PLAI under Mod TCLP w.r.t. its Prolog version. The next steps would be:
  - A re-implementation of the abstract domains, with an optimized representation of the abstract substitutions, that uses CLP techniques to propagate the effects of updates.
  - The use of constraints to define widening heuristics independently of the fixpoint algorithm in order to improve the flexibility, precision, and performance w.r.t. the state-of-the-art implementation available in CiaoPP.
  - Implement a backward analysis based on the tabled fixpoint. Apply it on static analysis, verification of energy/resources consumption, optimization of dynamic scheduling (concurrency), testing and/or debugging.

Additionally, as pointed out before, the implementation of s(CASP) can still be substantially improved:

- Static analysis can be used to optimize the compilation of non-monotonic check rules, being able to interleave them with the top-down execution strategy to discard models as soon as they are shown inconsistent. That can be done using the Groundness abstract domain, extended to handle dual rules and the c-forall.

- The disequality constraint solver should be improved to handle the pending cases (see Section 6.1.2). Its integration with the tabling engine will improve the performance by suspending more particular calls and reusing previous results.

- Dependency analysis could be used to improve the generation of the dual programs. The application of partial evaluation can remove (part of) the overhead brought about by the interpreting approach.

- Finally, another significant research direction includes applying s(CASP) system to solving planning problems where a generated plan must obey time constraints over continuous domains.

Funding Acknowledgments

This research was partially supported by EIT Digital (`https://eitdigital.eu`), MINECO project TIN2015-67522-C3-1-R (TRACES), Comunidad de Madrid project S2013/ICE-2731 *N-Greens Software*, Comunidad de Madrid project S2018/TCS-4339 *BLOQUES-CM*, co-funded by EIE Funds of the European Union, and USA NFS projects NSF IIS 1718945 and NSF IIS 1423419.

Bibliography

Alferes, J. J., Pereira, L. M., and Swift, T. (2004). **Abduction in Well-Founded Semantics and Generalized Stable Models via Tabled Dual Programs**. *Theory and Practice of Logic Programming* 4.4, pp. 383–428 (cited on p. 118).

Allen, J. F. (1983). **Maintaining Knowledge about Temporal Intervals**. *Communications of the ACM* 26.11, pp. 832–843 (cited on p. 85).

Alviano, M., Faber, W., Greco, G., and Leone, N. (2012). **Magic Sets for Disjunctive Datalog Programs**. *Artificial Intelligence* 187, pp. 156–192 (cited on p. 5).

Arias, J. (2016). **Tabled CLP for Reasoning over Stream Data**. In: *Technical Communications of the 32nd Int'l. Conference on Logic Programming*. Vol. 52. Doctoral Consortium. OASIcs, pp. 1–8 (cited on p. 131).

Arias, J., Chen, Z., Carro, M., and Gupta, G. (2019a). **Modeling and Reasoning in Event Calculus Using Goal-Directed Constraint Answer Set Programming**. In: *Pre-Proc. of the 29th Int'l. Symposium on Logic-based Program Synthesis and Transformation* (cited on p. 9).

Arias, J. and Carro, M. (2016). **Description and Evaluation of a Generic Design to Integrate CLP and Tabled Execution**. In: *18th Int'l. ACM Symposium on Principles and Practice of Declarative Programming*. ACM Press, pp. 10–23 (cited on p. 7).

— (2019a). **Description, Implementation, and Evaluation of a Generic Design for Tabled CLP**. *Theory and Practice of Logic Programming* 19.3, pp. 412–448 (cited on p. 7).

— (2019b). **Evaluation of the Implementation of an Abstract Interpretation Algorithm using Tabled CLP**. *Theory and Practice of Logic Programming* 19.5-6. Special Issue on ICLP'19, pp. 1107–1123 (cited on p. 8).

— (2019c). **Incremental Evaluation of Lattice-Based Aggregates in Logic Programming Using Modular TCLP**. In: *21st Int'l. Symposium on Practical Aspects of Declarative Languages*. Ed. by J. J. Alferes and M. Johansson. Vol. 11372. LNCS. Springer, pp. 98–114 (cited on p. 8).

Arias, J., Carro, M., Salazar, E., Marple, K., and Gupta, G. (2018). **Constraint Answer Set Programming without Grounding**. *Theory and Practice of Logic Programming* 18.3-4. Special Issue on ICLP'18, pp. 337–354 (cited on p. 8).

Arias, J., Carro, M., Chen, Z., and Gupta, G. (2019b). **Constraint Answer Set Programming without Grounding and its Applications**. In: *3rd Int'l. Workshop on the Resurgence of Datalog in Academia and Industry (Datalog 2.0)*. Ed. by M. Alviano and A. Pieris. Vol. 2368. CEUR-WS, pp. 22–26 (cited on p. 9).

Balduccini, M. and Lierler, Y. (2017). **Constraint Answer Set Solver EZCSP and why Integration Schemas Matter**. *Theory and Practice of Logic Programming* 17.4, pp. 462–515 (cited on pp. 5, 6).

Balduccini, M., Magazzeni, D., and Maratea, M. (2016). **PDDL+ Planning via Constraint Answer Set Programming**. In: *9th Workshop on Answer Set Programming and Other Computing Paradigms*. http://arxiv.org/abs/1609.00030 (cited on p. 139).

Banbara, M., Kaufmann, B., Ostrowski, M., and Schaub, T. (2017). **Clingcon: The Next Generation**. *Theory and Practice of Logic Programming* 17.4, 408–461 (cited on p. 6).

Bartholomew, M. and Lee, J. (2014). **System aspmt2smt: Computing ASPMT Theories by SMT Solvers**. In: *14th European Conference on Logics in Artificial Intelligence*. Vol. 8761. LNCS. Springer, pp. 529–542 (cited on p. 139).

Baselice, S. and Bonatti, P. A. (2010). **A Decidable Subclass of Finitary Programs**. *Theory and Practice of Logic Programming* 10.4-6, pp. 481–496 (cited on pp. 2, 6).

Baselice, S., Bonatti, P. A., and Criscuolo, G. (2009). **On Finitely Recursive Programs**. *Theory and Practice of Logic Programming* 9.2, pp. 213–238 (cited on pp. 2, 6).

Bratko, I. (2001). **Prolog programming for artificial intelligence**. Pearson education (cited on pp. 93, 171).

Brewka, G., Eiter, T., and Truszczyński, M. (2011). **Answer Set Programming at a Glance**. *Communications of the ACM* 54.12, pp. 92–103 (cited on p. 116).

Bruynooghe, M. (1991). **A Practical Framework for the Abstract Interpretation of Logic Programs**. *Journal of Logic Programming* 10, pp. 91–124 (cited on p. 99).

Bueno, F., Lopez-Garcia, P., and Hermenegildo, M. V. (2004). **Multivariant Non-Failure Analysis via Standard Abstract Interpretation**. In: *7th Int'l. Symposium on Functional and Logic Programming*. Vol. 2998. LNCS. Springer-Verlag, pp. 100–116 (cited on p. 100).

Cabeza, D. and Hermenegildo, M. V. (2000). **A New Module System for Prolog**. In: *International Conference on Computational Logic, CL2000*. LNAI 1861. Springer-Verlag, pp. 131–148 (cited on p. 83).

Calimeri, F., Cozza, S., and Ianni, G. (2007). **External Sources of Knowledge and Value Invention in Logic Programming**. *Annals of Mathematics and Artificial Intelligence* 50.3-4, pp. 333–361 (cited on p. 5).

Carlson, B., Carlsson, M., and Diaz, D. (1994). **Entailment of Finite Domain Constraints**. In: *11th International Conference on Logic Programming*. The MIT Press, pp. 339–353 (cited on p. 156).

Charatonik, W., Mukhopadhyay, S., and Podelski, A. (2002). **Constraint-Based Infinite Model Checking and Tabulation for Stratified CLP**. In: *18th Int'l. Conference on Logic Programming*. LNCS, pp. 115–129 (cited on pp. 4, 45).

Chico de Guzmán, P., Carro, M., Hermenegildo, M. V., and Stuckey, P. (2012). **A General Implementation Framework for Tabled CLP**. In: *15th Int'l. Symposium on Functional and Logic Programming*. Ed. by T. Schrijvers and P. Thiemann. Vol. 7294. LNCS. Springer Verlag, pp. 104–119 (cited on pp. 4, 25, 46, 60, 62, 64, 98).

Chittaro, L. and Montanari, A. (1996). **Efficient Temporal Reasoning in the Cached Event Calculus**. *Computational Intelligence* 12, pp. 359–382 (cited on p. 140).

Clark, K. L. (1978). **Negation as Failure**. In: *Logic and Data Bases*. Ed. by H. Gallaire and J. Minker. Springer, pp. 293–322 (cited on pp. 117, 118, 144).

Cousot, P. and Cousot, R. (1977). **Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints**. In: *4th Int. ACM Symposium on Principles of Programming Languages*. ACM Press, pp. 238–252 (cited on pp. 71, 99).

Cui, B. and Warren, D. S. (2000). **A System for Tabled Constraint Logic Programming**. In: *Int'l. Conference on Computational Logic*. Vol. 1861. LNCS. Springer, pp. 478–492 (cited on pp. 4, 25, 42, 45, 48, 64, 66).

Dal Palù, A., Dovier, A., Pontelli, E., and Rossi, G. (2009). **GASP: Answer Set Programming with Lazy Grounding**. *Fundamenta Informaticae* 96.3, pp. 297–322 (cited on pp. 2, 6).

Dawson, S., Ramakrishnan, C. R., and Warren, D. S. (1996). **Practical Program Analysis Using General Purpose Logic Programming Systems – A Case Study**. In: *Proc. 9th ACM Conference on Programming Language Design and Implementation*. New York, USA: ACM Press, pp. 117–126 (cited on p. 4).

Demoen, B. and Sagonas, K. (1998). **CAT: The Copying Approach to Tabling**. In: *Programming Language Implementation and Logic Programming*. Vol. 1490. Lecture Notes in Computer Science. Springer-Verlag, pp. 21–35 (cited on p. 98).

Díaz, D. and Codognet, P. (1993). **A Minimal Extension of the WAM for `clp(FD)`**. In: *10th International Conference on Logic Programming*. Budapest. MIT press, pp. 774–790 (cited on p. 156).

Dietrich, S. W. (1987). **Extension Tables: Memo Relations in Logic Programming**. In: *Fourth IEEE Symposium on Logic Programming*, pp. 264–272 (cited on p. 98).

Dincbas, M., Hentenryck, P. V., Simonis, H., and Aggoun, A. (1988). **The Constraint Logic Programming Language CHIP**. In: *2nd International Conference on Fifth Generation Computer Systems*, pp. 249–264 (cited on p. 156).

Dovier, A., Formisano, A., and Pontelli, E. (2005). **A Comparison of CLP(FD) and ASP Solutions to NP-Complete Problems**. In: *21st International Conference on Logic Programming*. Springer, pp. 67–82 (cited on pp. 133, 204).

Emden, M. H. van and Kowalski, R. A. (1976). **The Semantics of Predicate Logic as a Programming Language**. *Journal of the ACM* 23, pp. 733–742 (cited on pp. 2, 20).

Falaschi, M., Levi, G., Martelli, M., and Palamidessi, C. (1989). **Declarative Modeling of the Operational Behaviour of Logic Programs**. *Theoretical Computer Science* 69, pp. 289–318 (cited on pp. 19–21).

Fox, M. and Long, D. (2002). **PDDL+: Modeling continuous time dependent effects**. In: *Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*. Vol. 4, p. 34 (cited on p. 139).

Freire, J., Swift, T., and Warren, D. S (2001). **Beyond Depth-First: Improving Tabled Logic Programs through Alternative Scheduling Strategies**. In: *International Symposium on Programming Language Implementation and Logic Programming*. LNCS 1140. Springer-Verlag, pp. 243–258 (cited on p. 98).

Frigioni, D., Marchetti-Spaccamela, A., and Nanni, U. (1998). **Fully Dynamic Shortest Paths and Negative Cycles Detection on Digraphs with Arbitrary Arc Weights**. In: *6th International European Symposium on Algorithms*, pp. 320–331 (cited on p. 60).

Gabbrielli, M. and Levi, G. (1991). **Modeling Answer Constraints in Constraint Logic Programs**. In: *Proc. 8th Int'l Conference on Logic Programming*, pp. 238–252 (cited on pp. 20, 125).

Gange, G., Navas, J. A., Schachte, P., Søndergaard, H., and Stuckey, P. J. (2013). **Failure Tabled Constraint Logic Programming by Interpolation**. *Theory and Practice of Logic Programming* 13.4-5, pp. 593–607 (cited on p. 4).

Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., and Thiele, S. (2008). **A User's Guide to gringo, clasp, clingo, and iclingo**. Available at: `http://potassco.sourceforge.net`. Accessed on December, 2019 (cited on pp. 134, 206).

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. (2014). **Clingo = ASP + Control: Preliminary Report**. *arXiv preprint arXiv:1405.3694* (cited on p. 142).

Gelfond, M. and Kahl, Y. (2014). **Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach**. Cambridge University Press (cited on p. 139).

Gelfond, M. and Lifschitz, V. (1988). **The Stable Model Semantics for Logic Programming**. In: *5th International Conference on Logic Programming*, pp. 1070–1080 (cited on pp. 2, 5, 116, 117).

— (1991). **Classical Negation in Logic Programs and Disjunctive Databases**. *New Generation Computing* 9.3/4, pp. 365–386 (cited on p. 117).

— (1993). **Representing Action and Change by Logic Programs**. *The Journal of Logic Programming* 17.2-4, pp. 301–321 (cited on p. 139).

Genaim, S., Codish, M., and Howe, J. (2001). **Worst-Case Groundness Analysis Using Definite Boolean Functions**. *Theory and Practice of Logic Programming* 1.05, pp. 611–615 (cited on p. 72).

Guo, H.-F. and Gupta, G. (2008). **Simplifying Dynamic Programming via Mode-directed Tabling**. *Software: Practice and Experience* 1, pp. 75–94 (cited on p. 78).

Gupta, G., Bansal, A., Min, R., Simon, L., and Mallya, A. (2007). **Coinductive Logic Programming and its Applications**. In: *23rd Int'l. Conference on Logic Programming*. Springer, pp. 27–44 (cited on p. 122).

Hermenegildo, M. V., Puebla, G., Bueno, F., and Lopez-Garcia, P. (2005). **Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor)**. *Science of Computer Programming* 58.1–2, pp. 115–140 (cited on p. 99).

Hermenegildo, M. V., Bueno, F., Carro, M., Lopez-Garcia, P., Mera, E., Morales, J., and Puebla, G. (2012). **An Overview of Ciao and its Design Philosophy**. *Theory and Practice of Logic Programming* 12.1–2, pp. 219–252 (cited on pp. 46, 99, 123).

Hölldobler, S. and Schweizer, L. (2014). **Answer Set Programming and clasp, a Tutorial**. In: *Young Scientists' International Workshop on Trends in Information Processing*, pp. 77–95 (cited on p. 133).

Holzbaur, C. (1992). **Metastructures vs. Attributed Variables in the Context of Extensible Unification**. In: *Int'l. Symposium on Programming Language Implementation and Logic Programming*. LNCS 631. Springer Verlag, pp. 260–268 (cited on pp. 88, 108).

— (1995). **OFAI CLP(Q,R) Manual, Edition 1.3.3**. Tech. rep. TR-95-09. Vienna: Austrian Research Institute for Artificial Intelligence (cited on pp. 57, 116, 126, 142).

Jaffar, J. and Maher, M. (1994). **Constraint Logic Programming: A Survey**. *Journal of Logic Programming* 19/20, pp. 503–581 (cited on pp. 2, 18–21, 124).

Janhunen, T., Kaminski, R., Ostrowski, M., Schellhorn, S., Wanko, P., and Schaub, T. (2017). **Clingo goes Linear Constraints over Reals and Integers**. *Theory and Practice of Logic Programming* 17.5-6, pp. 872–888 (cited on pp. 6, 132, 203).

Janssens, G. and Sagonas, K. (1998). **On the Use of Tabling for Abstract Interpretation: An Experiment with Abstract Equation Systems**. In: *Tabulation in Parsing and Deduction* (cited on p. 98).

Janssens, G., Bruynooghe, M., and Dumortier, V. (1995). **A Blueprint for an Abstract Machine for Abstract Interpretation of (Constraint) Logic Programs**. In: *ILPS*, pp. 336–350 (cited on p. 99).

Ji, J., Wan, H., Wang, K., Wang, Z., Zhang, C., and Xu, J. (2016). **Eliminating Disjunctions in Answer Set Programming by Restricted Unfolding**. In: *25th Int'l. Joint Conference on Artificial Intelligence*, pp. 1130–1137 (cited on p. 116).

Kanamori, T. and Kawamura, T. (1993). **Abstract Interpretation Based on OLDT Resolution**. *Journal of Logic Programming* 15, pp. 1–30 (cited on p. 98).

Kanellakis, P. C., Kuper, G. M., and Revesz, P. Z. (1995). **Constraint Query Languages**. *Journal of Computer and System Sciences* 51.1, pp. 26–52 (cited on pp. 3, 45).

Kemp, D. B. and Stuckey, P. J. (1991). **Semantics of Logic Programs with Aggregates**. In: *Int'l. Simposium on Logic Programming*. Citeseer, pp. 387–401 (cited on p. 79).

Knuth, D. E. (1991). **Textbook Examples of Recursion**. *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pp. 207–230 (cited on p. 72).

Kowalski, R. (1979). **Algorithm = Logic + Control**. *Communications of the ACM* 22.7, pp. 424–436 (cited on p. 1).

Kowalski, R. and Sergot, M. (1989). **A Logic-Based Calculus of Events**. In: *Foundations of knowledge base management*. Springer, pp. 23–55 (cited on pp. 137, 140).

Lee, J. and Meng, Y. (2013a). **Answer Set Programming Modulo Theories and Reasoning about Continuous Changes**. In: *23rd Int'l. Joint Conference on Artificial Intelligence*, pp. 990–996 (cited on p. 139).

— (2013b). **Answer Set Programming Modulo Theories and Reasoning about Continuous Changes**. In: *IJCAI 2013*, pp. 990–996.

Lee, J. and Palla, R. (2012). **Reformulating the Situation Calculus and the Event Calculus in the General Theory of Stable Models and in Answer Set Programming**. *Journal of Artificial Intelligence Research* 43, pp. 571–620 (cited on pp. 138, 143, 144, 149).

— (2019). **F2LP: Computing Answer Sets of First Order Formulas**. http://reasoning.eas.asu.edu/f2lp/. Accessed on December, 2019 (cited on pp. 138, 143, 144, 149).

Lifschitz, V. (2008). **What Is Answer Set Programming?** In: *23rd National Conference on Artificial Intelligence*. Vol. 3. AAAI Press, pp. 1594–1597 (cited on p. 2).

Lloyd, J. (1994). **Practical Advantages of Declarative Programming**. In: *Proc. Joint Conference on Declarative Programming GULP-PRODE'94*, pp. 18–30 (cited on p. 1).

Maier, D. and Warren, D. S. (1988). **Computing with Logic: Logic Programming with Prolog**. Menlo Park, CA 94025: Benjamin/Cummings Publishing Co., Inc. (cited on p. 3).

Marple, K., Salazar, E., and Gupta, G. (2017a). **Computing Stable Models of Normal Logic Programs Without Grounding**. *CoRR* eprint arXiv:1709.00501 (cited on pp. 2, 6, 116, 117, 120, 122, 126, 145).

Marple, K., Salazar, E., Chen, Z., and Gupta, G. (2017b). **The s(ASP) Predicate Answer Set Programming System**. *The Association for Logic Programming Newsletter* (cited on p. 117).

Marriott, K. and Stuckey, P. J. (1993). **Semantics of Constraint Logic Programs with Optimization**. *LOPLAS* 2.1-4, pp. 197–212 (cited on p. 2).

— (1998). **Programming with Constraints: an Introduction**. MIT Press (cited on p. 62).

McCarthy, J. (1980). **Circumscription - A Form of Non-Monotonic Reasoning**. *Artificial Intelligence* 13.1-2, pp. 27–39 (cited on p. 140).

Mellarkod, V. S., Gelfond, M., and Zhang, Y. (2008). **Integrating Answer Set Programming and Constraint Logic Programming**. *Annals of Mathematics and Artificial Intelligence* 53.1-4, pp. 251–287 (cited on p. 138).

Mueller, E. T. (2008a). **Chapter 17: Event Calculus**. In: *Handbook of Knowledge Representation*. Ed. by F. van Harmelen, V. Lifschitz, and B. Porter. Vol. 3. Foundations of AI. Elsevier, pp. 671 –708 (cited on p. 140).

Mueller, E. T. (2008b). **Discrete event calculus reasoner documentation**. *Software documentation, IBM Thomas J. Watson Research Center, PO Box* 704. Available at: `http://decreasoner.sourceforge.net/`. Accessed on December, 2019 (cited on p. 149).

Mueller, E. T. (2014). **Commonsense reasoning: an event calculus based approach**. Morgan Kaufmann (cited on pp. 85, 137, 140, 141, 145).

Muthukumar, K. and Hermenegildo, M. (1989). **Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation**. In: *1989 North American Conference on Logic Programming*. MIT Press, pp. 166–189 (cited on p. 100).

— (1990). **Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs**. Technical Report ACT-DC-153-90. Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759 (cited on pp. 99, 102).

— (1991). **Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation**. In: *8th Int'l. Conference on Logic Programming*. MIT Press, pp. 49–63 (cited on p. 109).

— (1992). **Compile-time Derivation of Variable Dependency Using Abstract Interpretation**. *Journal of Logic Programming* 13.2/3. Ed. by S. Debray, pp. 315–347 (cited on p. 99).

Nielson, F., Nielson, H. R., and Hankin, C. (2005). **Principles of Program Analysis**. Second Ed. Springer (cited on p. 99).

Pareto, V. (1964). **Cours d'économie politique**. Vol. 1. Librairie Droz (cited on p. 82).

Pelov, N., Denecker, M., and Bruynooghe, M. (2007). **Well-Founded and Stable Semantics of Logic Programs with Aggregates**. *Theory and Practice of Logic Programming* 3, pp. 301–353 (cited on p. 79).

Ramakrishna, Y., Ramakrishnan, C., Ramakrishnan, I., Smolka, S., Swift, T., and Warren, D. (1997). **Efficient Model Checking Using Tabled Resolution**. In: *Computer Aided Verification*. Vol. 1254. LNCS. Springer Verlag, pp. 143–154 (cited on p. 4).

Ramakrishnan, I., Rao, P., Sagonas, K., Swift, T., and Warren, D. (1995). **Efficient Tabling Mechanisms for Logic Programs**. In: *12nd Int'l. Conference on Logic Programming*. MIT Press, pp. 697–711 (cited on pp. 48, 50).

Revesz, P. Z. (1993). **A Closed-Form Evaluation for Datalog Queries with Integer (Gap)-Order Constraints**. *Theoretical Computer Science* 116.1, pp. 117–149 (cited on pp. 4, 37, 128).

Robinson, J. A. (1965). **A Machine Oriented Logic Based on the Resolution Principle**. *Journal of the ACM* 12.23, pp. 23–41 (cited on p. 2).

Santos Costa, V., Rocha, R., and Damas, L. (2012). **The YAP Prolog system**. *Theory and Practice of Logic Programming* 1-2, pp. 5–34 (cited on p. 78).

Schrijvers, T., Demoen, B., and Warren, D. S. (2008). **TCHR: a Framework for Tabled CLP**. *Theory and Practice of Logic Programming* 4, pp. 491–526 (cited on pp. 4, 19, 42, 43, 45, 46, 64, 90, 98).

Shanahan, M. (1999). **The Event Calculus Explained**. In: *Artificial Intelligence Today*. Springer, pp. 409–430 (cited on pp. 85, 144, 147).

— (2000). **An Abductive Event Calculus Planner**. *The Journal of Logic Programming* 44.1-3, pp. 207–240 (cited on p. 140).

Sterling, L. and Shapiro, E. (1994). **The Art of Prolog**. MIT Press, second edition (cited on pp. vii, 2).

Swift, T. and Warren, D. S. (2010). **Tabling with Answer Subsumption: Implementation, Applications and Performance**. In: *Logics in Artificial Intelligence*. Vol. 6341, pp. 300–312 (cited on pp. 4, 15, 17, 78, 98).

— (2012). **XSB: Extending Prolog with Tabled Logic Programming**. *Theory and Practice of Logic Programming* 1-2, pp. 157–187 (cited on pp. 4, 78).

Swift, T., Warren, D. S., Sagonas, K., Freire, J., Rao, P., Cui, B., Johnson, E., Castro, L. de, Marques, R. F., Saha, D., Dawson, S., and Kifer, M. (2016). **The XSB System Version 3.7.x. Volume 1: Programmer's Manual**. Available at: `http://xsb.sourceforge.net/manual1/manual1.pdf`. Accessed on December, 2019. Stony Brook University, Universidade Nova de Lisboa, XSB Inc., and Coherent Knowledge Systems, Inc. (cited on p. 95).

Tamaki, H. and Sato, M. (1986). **OLD Resolution with Tabulation**. In: *3rd Int'l. Conference on Logic Programming*. Vol. 111. London: Springer-Verlag, pp. 84–98 (cited on pp. 2, 3, 98).

Toman, D. (1995). **Top-Down beats Bottom-Up for Constraint Based Extensions of Datalog**. In: *1995 Int'l Symposium on Logic Programming*. Ed. by J. W. Lloyd. MIT Press, pp. 98–112 (cited on p. 4).

— (1997a). **Constraint Databases and Program Analysis Using Abstract Interpretation**. In: *Constraint Databases and Their Applications*. Vol. 1191. LNCS. Springer, pp. 246–262 (cited on pp. 4, 45).

— (1997b). **Memoing Evaluation for Constraint Extensions of Datalog**. *Constraints* 2.3/4, pp. 337–359 (cited on pp. 4, 13, 19, 20, 25, 34, 36–38, 45).

Van Hentenryck, P. (1989). **Constraint Satisfaction in Logic Programming**. MIT Press (cited on p. 156).

Vandenbroucke, A., Pirog, M., Desouter, B., and Schrijvers, T. (2016). **Tabling with Sound Answer Subsumption**. *Theory and Practice of Logic Programming, 32nd Int'l. Conference on Logic Programming* 5-6, pp. 933–949 (cited on pp. 78–80, 86).

Vaucheret, C. and Bueno, F. (2002). **More Precise yet Efficient Type Inference for Logic Programs**. In: *9th Int'l. Static Analysis Symposium*. Vol. 2477. LNCS. Springer-Verlag, pp. 102–116 (cited on p. 100).

Warren, D. S. (1992). **Memoing for Logic Programs**. *Communications of the ACM* 35.3, pp. 93–111 (cited on pp. 2, 3).

Warren, D. S. (1999). **Programming in Tabled Prolog**. https://www3.cs.stonybrook.edu/~warren/xsbbook/book.html. Unpublished manuscript. Accessed on December, 2019 (cited on p. 98).

Warren, R., Hermenegildo, M., and Debray, S. K. (1988). **On the Practicality of Global Flow Analysis of Logic Programs**. In: *5th Int'l. Conference and Symposium on Logic Programming*. MIT Press, pp. 684–699 (cited on pp. 4, 98).

Zhou, N. (2012). **The Language Features and Architecture of B-Prolog**. *Theory and Practice of Logic Programming* 1-2, pp. 189–218 (cited on p. 78).

Zhou, N., Kameya, Y., and Sato, T. (2010). **Mode-Directed Tabling for Dynamic Programming, Machine Learning, and Constraint Solving**. In: *Int'l. Conference on Tools with Artificial Intelligence*. 2. IEEE, pp. 213–218 (cited on p. 78).

Zou, Y., Finin, T., and Chen, H. (2005). **F-OWL: An Inference Engine for Semantic Web**. In: *Formal Approaches to Agent-Based Systems*. Vol. 3228. LNCS. Springer Verlag, pp. 238–248 (cited on p. 4).

Appendix A

Incremental Evaluation of Aggregates

A.1 Prolog and tabling encoding of Minimax

The next figure shows the Prolog encoding for the minimax algorithm applied to (an extended version of) TicTacToe described in Section 4.5. The encoding for the minimax algorithm is from (Bratko, 2001). The tabling version tabled the predicate `best/3` adding the directive `:- tabled best/3`.

```
% Pos has successors
minimax(Pos, BestNextPos, Val) :-
        findall(NextPos, move(Pos, NextPos), NextPosList),
        best(NextPosList, BestNextPos, Val), !.
% Pos has no successors
minimax(Pos, _, Val) :-
        utility(Pos, Val).

% There is no more position to compare
best([Pos], Pos, Val) :-
        minimax(Pos, _, Val), !.
% There are other positions to compare
best([Pos1 | PosList], BestPos, BestVal) :-
        minimax(Pos1, _, Val1),
        best(PosList, Pos2, Val2),
        betterOf(Pos1, Val1, Pos2, Val2, BestPos, BestVal).

betterOf(Pos0, Val0, _, Val1, Pos0, Val0) :-   % Pos0 better than Pos1
        min_to_move(Pos0),                         % MIN to move in Pos0
        Val0 > Val1, !.                            % MAX prefers the greater value
betterOf(Pos0, Val0, _, Val1, Pos0, Val0) :-   % Pos0 better than Pos1
        max_to_move(Pos0),                         % MAX to move in Pos0
        Val0 < Val1, !.                            % MIN prefers the lesser value
betterOf(_, _, Pos1, Val1, Pos1, Val1).        % Otherwise Pos1 better than Pos0
```

A.2 Tabling encoding of Game

The next figure shows the tabling encoding for the game problem described in Section 4.5. The tabling version tabled the predicate `reach/4` but as we mentioned before, it is not possible to optimize it with an intermediate predicate that using `findall/3` would collects the optimal paths, because its interleaving recursion with `reach/4` will produce wrong results.

```
:- module(game_tabling_01, _).

:- use_package(tabling).

total_fun(Fun) :-
        findall(F, reach(initial, end, _, F), Fs)
        max_list(Fs,Fun).

max_list([X|Xs],Max) :- max_list_(Xs,X,Max).
max_list_([],Max,Max).
max_list_([X|Xs],Prev,Max) :- X > Prev, max_list_(Xs,X,Max).
max_list_([X|Xs],Prev,Max) :- X =< Prev, max_list_(Xs,Prev,Max).

:- tabled reach/4.
reach(GameA, GameB, Tf, Ff) :-
        reach(GameA, GameZ, T1, F1),
        edge(GameZ, GameB, T2, F2),
        Ff is F1 + F2,
        Tm is T1 + T2, Tm >= 0,
        cap(Cap), (Tm > Cap -> Tf is Cap ; Tf is Tm).
reach(GameA, GameB, T, F) :-
        edge(GameA, GameB, T, F).

edge(initial,      g(1, no),     T,  0) :- cap(T).
edge(g(Game, yes), end,          0,  0) :- num_g(Game).
edge(g(Game, _),   g(Game, yes), -1, F) :- fun(Game, F).
edge(g(Game, yes), g(Game1, no), T,  0) :-
        Game1 is Game + 1, num_g(Games), Game1 =< Games, refill(T).
```

Appendix B

Abstract Interpretation Fixpoint

B.1 PLAI Algorithm Using TCLP

In this appendix we include, for reference for the reviewers, the code corresponding to the reimplementation of PLAI using TCLP. It is not expected to be used to understand the code (we did not add any facility or improve its functionality), but rather to compare the code length and complexity with that of the original PLAI in CiaoPP, which we include in Appendix B.2. Therefore, we have removed the comments that appear in the original files. The files with comments can be accessed at `http://www.cliplab.org/papers/tclp-plai-iclp2019`.

```
/*              Copyright (C)1990-2019 UPM-CLIP                          */

:- module(fixpo_plai_tabling,
        [
            query/8,
            init_fixpoint/0,
            cleanup_fixpoint/1,
            entry_to_exit/9
        ],
        [assertions, datafacts]).

% Ciao library
:- use_module(engine(io_basic)).

:- use_module(library(aggregates), [bagof/3, (^)/2]).
:- use_module(library(lists), [member/2, append/3]).
:- use_module(library(terms_vars), [varset/2]).
:- use_module(library(terms_check)).
:- use_module(library(sets), [merge/3, ord_subtract/3]).
:- use_module(library(sort), [sort/2]).
:- use_module(library(messages)).
:- use_module(library(write)).

% CiaoPP library
:- use_module(ciaopp(preprocess_flags), [current_pp_flag/2, set_pp_flag/2]).

% Plai library
:- use_module(ciaopp(plai/fixpo_ops), [inexistent/2, variable/2, bottom/1,
        singleton/2, fixpoint_id_reuse_prev/5, fixpoint_id/1, fixp_id/1,
        each_abs_sort/3,
        % each_concrete/4,
        each_extend/6, each_project/6, each_exit_to_prime/8, each_unknown_call/4,
        each_body_succ_builtin/12, body_succ_meta/7, reduce_equivalent/3,
        each_apply_trusted/7,   widen_succ/4,   decide_memo/6,clause_applies/2,
        abs_subset_/3]).

:- use_module(ciaopp(plai/domains)).
:- use_module(ciaopp(plai/trace_fixp), [fixpoint_trace/7, cleanup/0]).
:- use_module(ciaopp(plai/plai_db),
        [ complete/7, memo_call/5, memo_table/6, cleanup_plai_db/1, patch_parents/6 ]).
:- use_module(ciaopp(plai/psets), [update_if_member_idlist/3]).
:- use_module(ciaopp(plai/re_analysis), [erase_previous_memo_tables_and_parents/4]).
:- use_module(ciaopp(plai/transform), [body_info0/4, trans_clause/3]).
:- use_module(ciaopp(plai/apply_assertions_old),
        [ apply_trusted0/7,
          cleanup_trusts/1 ]).

:- doc(author,"Joaquin Arias").

:- doc(module,"This module adapt the implementation of the top-down
        fixpoint algorithm of PLAI, under TCLP with aggregates and an
        extension which also check call entailment.").

init_fixpoint.

cleanup_fixpoint(_AbsInt).

%-------------------------------------------------------------------------%
% call_to_success(+,+,+,+,+,+,-)                                          %
%-------------------------------------------------------------------------%

call_to_success(SgKey,Call,Proj,Sg,Sv,AbsInt,Succ) :-
        call_to_success_fixpoint(SgKey,Sg, st(Sv,Call,Proj,AbsInt,Prime)),
        each_extend(Sg,Prime,AbsInt,Sv,Call,Succ).

%%%%%%%%%%%% TCLP interface %%%%%%%%%%%%
 :- use_package(tclp_aggregate).
 :- table call_to_success_fixpoint(_,_,abst_lub).

call_entail(abst_lub, st(V,_,ProjA,AbsInt,_), st(V,_,ProjB,AbsInt,_)) :-
        identical_abstract(AbsInt,ProjA,ProjB), !.

answer_entail(abst_lub, st(V,_,_,AbsInt,PrimeAs), st(V,_,_,AbsInt,PrimeBs),1) :-
```

```
        singleton(PrimeA,PrimeAs),
        singleton(PrimeB,PrimeBs),
        less_or_equal(AbsInt,PrimeA,PrimeB), !.

answer_join(abst_lub,st(V,_,_,AbsInt,PrimeAs), st(V,_,_,AbsInt,PrimeBs),
                                                 st(V,_,_,AbsInt,PrimeNews)) :-
        singleton(PrimeA,PrimeAs),
        singleton(PrimeB,PrimeBs),
        singleton(PrimeNew,PrimeNews),
        compute_lub(AbsInt,[PrimeA,PrimeB],PrimeNew), !.

apply_answer(abst_lub, st(V,_,_,Ab,A), st(V,_,_,Ab,B)) :- A = B.

call_to_success_fixpoint(SgKey,Sg,st(Sv,Call,Proj,AbsInt,Primes)) :-
        trans_clause(SgKey,_,Clause),
        do_nr_cl(Clause,Sg,Sv,Call,Proj,AbsInt,Primes).
call_to_success_fixpoint(SgKey,Sg,st(Sv,_Call,Proj,AbsInt,Primes)) :-
        \+ trans_clause(SgKey,_,_),
        apply_trusted0(Proj,SgKey,Sg,Sv,AbsInt,_ClId,Prime),
        singleton(Prime,Primes).

do_nr_cl(Clause,Sg,Sv,Call,Proj,AbsInt,Primes):-
        Clause = clause(Head,Vars_u,K,Body),
        clause_applies(Head,Sg), !,
        varset(Head,Hv),
        sort(Vars_u,Vars),
        ord_subtract(Vars,Hv,Fv),
        process_body(Body,K,AbsInt,Sg,Hv,Fv,Vars_u,Head,Sv,Call,
                                                        Proj,Primes,_Id).
do_nr_cl(_Clause,_Sg,_Sv,_Call,_Proj,_AbsInt,[[]]).

process_body(Body,K,AbsInt,Sg,Hv,_Fv,_,Head,Sv,Call,Proj,LPrime,_Id):-
        Body = g(_,[],'$built'(_,true,_),'true/0',true), !,
        singleton(Prime,LPrime),
        call_to_success_fact(AbsInt,Sg,Hv,Head,K,Sv,Call,Proj,Prime,_Succ).
process_body(Body,K,AbsInt,Sg,Hv,Fv,Vars_u,Head,Sv,_,Proj,Prime,Id):-
        call_to_entry(AbsInt,Sv,Sg,Hv,Head,K,Fv,Proj,Entry,ExtraInfo),
        singleton(Entry,LEntry),
        entry_to_exit(Body,K,LEntry,Exit,[],_,Vars_u,AbsInt,Id),
        each_exit_to_prime(Exit,AbsInt,Sg,Hv,Head,Sv,ExtraInfo,Prime).

%-----------------------------------------------------------------------%
% entry_to_exit(+,+,+,-,+,-,+,+,+)                                      %
%-----------------------------------------------------------------------%

entry_to_exit((Sg,Rest),K,Call,Exit,OldList,NewList,Vars_u,AbsInt,NewN):- !,
        body_succ(Call,Sg,Succ,OldList,IntList,Vars_u,AbsInt,K,NewN,_),
        entry_to_exit(Rest,K,Succ,Exit,IntList,NewList,Vars_u,AbsInt,NewN).
entry_to_exit(true,_,Call,Call,List,List,_,_,_):- !.
entry_to_exit(Sg,Key,Call,Exit,OldList,NewList,Vars_u,AbsInt,NewN):-
        body_succ(Call,Sg,Exit,OldList,NewList,Vars_u,AbsInt,Key,NewN,_),
        true.
```

```

body_succ(Call,_Atom,Succ,List,List,_HvFv_u,_AbsInt,_ClId,_ParentId,no):-
        bottom(Call), !,
        Succ = Call.
body_succ(Call,Atom,Succ,List,NewList,HvFv_u,AbsInt,ClId,ParentId,Id):-
        Atom=g(Key,Sv,Info,SgKey,Sg),
        body_succ_(Info,SgKey,Sg,Sv,HvFv_u,Call,Succ,List,NewList,AbsInt,
                   ClId,Key,ParentId,Id).

body_succ_(Info,SgKey,Sg,Sv,HFv,Call,Succ,L,NewL,AbsInt,ClId,Key,PId,Id):-
        Info = [_|_], !,
        split_combined_domain(AbsInt,Call,Calls,Domains),
        map_body_succ(Info,SgKey,Sg,Sv,HFv,Calls,Succs,L,NewL,Domains,
                      ClId,Key,PId,Id),
        split_combined_domain(AbsInt,Succ,Succs,Domains).
body_succ_(Info,SgKey,Sg,Sv,HFv,Call,Succ,L,NewL,AbsInt,ClId,Key,PId,Id):-
        body_succ0(Info,SgKey,Sg,Sv,HFv,Call,Succ,L,NewL,AbsInt,
                   ClId,Key,PId,Id).

map_body_succ([],_SgKey,_Sg,_Sv,_HFv,[],[],L,L,[],_ClId,_Key,_PId,no).
map_body_succ([I|Info],SgKey,Sg,Sv,HFv,[Call|Calls],[Succ|Succs],L,NewL,
              [AbsInt|Domains],ClId,Key,PId,Id):-
        body_succ0(I,SgKey,Sg,Sv,HFv,Call,Succ,L,_NewL,AbsInt,
                   ClId,Key,PId,_Id), !,
        map_body_succ(Info,SgKey,Sg,Sv,HFv,Calls,Succs,L,NewL,Domains,
                   ClId,Key,PId,Id).

body_succ0('$var',SgKey,Sg,_Sv_u,HvFv_u,Calls,Succs,List0,List,AbsInt,
               _ClId,F,_N,_Id):-
        !,
        ( Calls=[Call],
          concrete(AbsInt,Sg,Call,Concretes),
          concretes_to_body(Concretes,SgKey,AbsInt,B)
        -> meta_call(B,HvFv_u,Calls,[],Succs,List0,List,AbsInt,_ClId,_Id,_Ids)
         ; List=List0,
           each_unknown_call(Calls,AbsInt,[Sg],Succs) % Sg is a variable
        ).
body_succ0('$meta'(T,B,_),SgKey,Sg,Sv_u,HvFv_u,Call,Succ,List0,List,AbsInt,
           _ClId,_F,_N,_Id):-
        !,
        meta_call(B,HvFv_u,Call,[],Exits,List0,List,AbsInt,ClId,Id,_Ids),
        ( body_succ_meta(T,AbsInt,Sv_u,HvFv_u,Call,Exits,Succ) ->
          true
        ; % for the trusts, if any:
          varset(Sg,Sv_r), % Sv_u contains extra vars (from meta-term)
                           % which will confuse apply_trusted
          body_succ0(nr,SgKey,Sg,Sv_r,HvFv_u,Call,Succ,[],_List,AbsInt,
                     _ClId,_F,_N,_Id0)
        ).
body_succ0('$built'(T,Tg,Vs),SgKey,Sg,Sv_u,HvFv_u,Call,Succ,List0,List,AbsInt,
           _ClId,_F,_N,_Id):-
        !,
        List=List0,
        sort(Sv_u,Sv),
        each_body_succ_builtin_(Call,AbsInt,T,Tg,Vs,SgKey,Sg,Sv,HvFv_u,Succ).
body_succ0(_RFlag,SgKey,Sg,Sv_u,HvFv_u,Call,Succ,_List0,_List,AbsInt,
           _ClId,_F,_N,_Id):-
        sort(Sv_u,Sv),
        each_call_to_success(Call,SgKey,Sg,Sv,HvFv_u,AbsInt,Succ).

%% predicate adapted from fixpo_ops
each_body_succ_builtin_([],_,_T,_Tg,_,_,_Sg,_Sv,_HvFv_u,[]).
each_body_succ_builtin_([Call|Calls],AbsInt,T,Tg,Vs,SgKey,Sg,Sv,HvFv_u,[Succ|Succs]):-
        project(AbsInt,Sg,Sv,HvFv_u,Call,Proj),
        body_succ_builtin(T,AbsInt,Tg,Vs,Sv,HvFv_u,Call,Proj,Succ),!, %% Doamin call
        each_body_succ_builtin_tabling_(Calls,AbsInt,T,Tg,Vs,SgKey,Sg,Sv,HvFv_u,Succs).

each_call_to_success([Call],SgKey,Sg,Sv,HvFv_u,AbsInt,Succ):-
        !,
        project(AbsInt,Sg,Sv,HvFv_u,Call,Proj),
        call_to_success(SgKey,Call,Proj,Sg,Sv,AbsInt,Succ).

each_call_to_success(LCall,SgKey,Sg,Sv,HvFv_u,AbsInt,LSucc):-
        each_call_to_success0(LCall,SgKey,Sg,Sv,HvFv_u,AbsInt,
                              LSucc).

each_call_to_success0([],_SgK,_Sg,_Sv,_HvFv,_AbsInt,[]).
each_call_to_success0([Call|LCall],SgKey,Sg,Sv,HvFv_u,AbsInt,
                      LSucc):-
        project(AbsInt,Sg,Sv,HvFv_u,Call,Proj),
        call_to_success(SgKey,Call,Proj,Sg,Sv,AbsInt,LSucc0),
        append(LSucc0,LSucc1,LSucc),
        each_call_to_success0(LCall,SgKey,Sg,Sv,HvFv_u,AbsInt,
                              LSucc1).

meta_call([],_HvFv_u,Call,[],Call,List,List,_AbsInt,_ClId,_Id,[]).
meta_call([Body|Bodies],HvFv_u,Call,Succ0,Succ,L0,List,AbsInt,ClId,Id,Ids):-
        meta_call_([Body|Bodies],HvFv_u,Call,Succ0,Succ,L0,List,AbsInt,ClId,Id,Ids).
meta_call_([Body|Bodies],HvFv_u,Call,Succ0,Succ,L0,List,AbsInt,ClId,Id,Ids):-
        meta_call_body(Body,ClId,Call,Succ1,L0,L1,HvFv_u,AbsInt,Id,Ids0),
        widen_succ(AbsInt,Succ0,Succ1,Succ2),
        append(Succ0,Succ1,Succ2),
        append(Ids0,Ids1,Ids),
        meta_call_(Bodies,HvFv_u,Call,Succ2,Succ,L1,List,AbsInt,ClId,Id,Ids1).
meta_call_([],_HvFv_u,_Call,Succ,Succ,List,List,_AbsInt,_ClId,_Id,[]).

meta_call_body((Sg,Rest),K,Call,Exit,OldList,NewList,Vars_u,AbsInt,PId,CIds):-
        !,
        CIds=[Id|Ids],
        body_succ(Call,Sg,Succ,OldList,IntList,Vars_u,AbsInt,K,PId,Id),
        meta_call_body(Rest,K,Succ,Exit,IntList,NewList,Vars_u,AbsInt,PId,Ids).
meta_call_body(true,_,Call,Call,List,List,_,_,_,_,[no]):- !.
meta_call_body(Sg,Key,Call,Exit,OldList,NewList,Vars_u,AbsInt,PId,[Id]):-
        body_succ(Call,Sg,Exit,OldList,NewList,Vars_u,AbsInt,Key,PId,Id).
```

```

concretes_to_body([],_SgKey,_AbsInt,[]).
concretes_to_body([Sg|Sgs],SgKey,AbsInt,[B|Bs]):-
        body_info0(Sg:SgKey,[],AbsInt,B),
        concretes_to_body(Sgs,SgKey,AbsInt,Bs).

%------------------------------------------------------------------------%
% query(+,+,+,+,+,+,+,-)                                                 %
%------------------------------------------------------------------------%

:- doc(query(AbsInt,QKey,Query,Qv,RFlag,N,Call,Succ),
        "The success pattern of @var{Query} with @var{Call} is
        @var{Succ} in the analysis domain @var{AbsInt}. The predicate
        called is identified by @var{QKey}. The goal @var{Query} has
        variables @var{Qv}.").


query(AbsInt,QKey,Query,Qv,_RFlag,_N,Call,Succ) :-
        project(AbsInt,Query,Qv,Qv,Call,Proj),
        call_to_success(QKey,Call,Proj,Query,Qv,AbsInt,Succ), !.

query(_AbsInt,_QKey,_Query,_Qv,_RFlag,_N,_Call,_Succ):-
        % should never happen, but...
        error_message("SOMETHING HAS FAILED!").
```

B.2 PLAI Algorithm Using Ciao Prolog

We include here the Ciao Prolog implementation of PLAI. As mentioned before, we have removed the comments from the file since the goal of this appendix it to make it easier for the reader to compare the Ciao Prolog code w.r.t. the code using TCLP, which we include in B.1. The original version is available at `http://www.cliplab.org/papers/tclp-plai-iclp2019`.

```
/*              Copyright (C)1990-2002 UPM-CLIP                          */

:- module(fixpo_plai_with_comments,
        [ query/8,
          init_fixpoint/0,
          cleanup_fixpoint/1,
          entry_to_exit/9
        ],
        [assertions, datafacts]).

% Ciao library
:- use_module(library(aggregates), [bagof/3, (^)/2]).
:- use_module(library(lists), [member/2, append/3]).
:- use_module(library(terms_vars), [varset/2]).
:- use_module(library(sets), [merge/3, ord_subtract/3]).
:- use_module(library(sort), [sort/2]).
:- use_module(library(messages)).

% CiaoPP library
:- use_module(ciaopp(preprocess_flags), [current_pp_flag/2, set_pp_flag/2]).

% Plai library
:- use_module(ciaopp(plai/fixpo_ops), [inexistent/2, variable/2, bottom/1,
        singleton/2, fixpoint_id_reuse_prev/5, fixpoint_id/1, fixp_id/1,
        each_abs_sort/3,
        % each_concrete/4,
        each_extend/6, each_project/6, each_exit_to_prime/8, each_unknown_call/4,
        each_body_succ_builtin/12, body_succ_meta/7, reduce_equivalent/3,
        each_apply_trusted/7,   widen_succ/4,   decide_memo/6,clause_applies/2,
        abs_subset_/3]).

:- use_module(ciaopp(plai/domains)).
:- use_module(ciaopp(plai/trace_fixp), [fixpoint_trace/7, cleanup/0]).
:- use_module(ciaopp(plai/plai_db),
        [ complete/7, memo_call/5, memo_table/6, cleanup_plai_db/1, patch_parents/6 ]).
:- use_module(ciaopp(plai/psets), [update_if_member_idlist/3]).
:- use_module(ciaopp(plai/re_analysis), [erase_previous_memo_tables_and_parents/4]).
:- use_module(ciaopp(plai/transform), [body_info0/4, trans_clause/3]).
:- use_module(ciaopp(plai/apply_assertions_old),
        [ apply_trusted0/7,
          cleanup_trusts/1 ]).

:- doc(author,"Kalyan Muthukumar").
:- doc(author,"Maria Garcia de la Banda").
:- doc(author,"Francisco Bueno").

:- doc(module,"This module implements the top-down fixpoint
        algorithm of PLAI, both in its mono-variant and multi-variant
        on successes versions. It is always multi-variant on calls.
        The algorithm is parametric on the particular analysis domain.").


:- data '$depend_list'/3.
:- data ch_id/2.

:- data approx/6.
:- data fixpoint/6.
:- data fixpoint_variant/6.
:- data approx_variant/7.

init_fixpoint:-
        retractall_fact(approx(_,_,_,_,_,_)),
        retractall_fact(fixpoint(_,_,_,_,_,_)),
        retractall_fact('$depend_list'(_,_,_)),
        retractall_fact(ch_id(_,_)),
        retractall_fact(fixpoint_variant(_,_,_,_,_,_)),
        retractall_fact(approx_variant(_,_,_,_,_,_)),
        trace_fixp:cleanup.

cleanup_fixpoint(AbsInt):-
        cleanup_plai_db(AbsInt),
        cleanup_trusts(AbsInt),
        retractall_fact(fixp_id(_)),
        asserta_fact(fixp_id(0)), % there is no way to recover this
        init_fixpoint.            % if several analysis coexist!

approx_to_completes(AbsInt):-
        current_fact(approx(SgKey,Sg,Proj,Prime,Pid,Fs),Ref),
        asserta_fact(complete(SgKey,AbsInt,Sg,Proj,Prime,Pid,Fs)),
        erase(Ref),
        fail.
approx_to_completes(AbsInt):-
        current_fact(approx_variant(_Id,Pid,SgKey,Sg,Proj,Prime,Fs),Ref),
        asserta_fact(complete(SgKey,AbsInt,Sg,Proj,Prime,Pid,Fs)),
        erase(Ref),
        fail.
approx_to_completes(_AbsInt).


%------------------------------------------------------------------------%
% call_to_success(+,+,+,+,+,+,+,-,-,+,+,+)                               %
%------------------------------------------------------------------------%

call_to_success(_RFlag,SgKey,Call,Proj,Sg,Sv,AbsInt,_ClId,Succ,List,F,N,Id):-
        % ClId = number identifying the clause?... for an entry point is 0...
        % F = program point of the call. clauseId+/0 for an entry call
        current_fact(complete(SgKey,AbsInt,Subg,Proj1,Prime1,Id,Fs),R),
        identical_proj(AbsInt,Sg,Proj,Subg,Proj1), !,
        patch_parents(R,complete(SgKey,AbsInt,Subg,Proj1,Prime1,Id,Ps),F,N,Ps,Fs),
        List = [],
        each_abs_sort(Prime1,AbsInt,Prime),
        each_extend(Sg,Prime,AbsInt,Sv,Call,Succ).
call_to_success(r,SgKey,Call,Proj,Sg,Sv,AbsInt,_ClId,Succ,List,F,N,Id) :-
        current_fact(approx(SgKey,Subg,Proj1,Prime1,Id,Fs),Ref),
        identical_proj(AbsInt,Sg,Proj,Subg,Proj1), !,
        each_abs_sort(Prime1,AbsInt,TempPrime),
        current_fact('$depend_list'(Id,SgKey,IdList)),
        call_to_success_approx(SgKey,Subg,Call,Proj,Proj1,Sg,Sv,AbsInt,F,N,Fs,
                               Id,Ref,IdList,Prime1,TempPrime,List,Prime),
        each_extend(Sg,Prime,AbsInt,Sv,Call,Succ).
call_to_success(r,SgKey,Call,Proj,Sg,Sv,AbsInt,_ClId,Succ,List,F,N,Id):-
        current_fact(fixpoint(SgKey,Subg,Proj1,Prime1,Id,Fs),Ref),
        identical_proj(AbsInt,Sg,Proj,Subg,Proj1), !,
        patch_parents(Ref,fixpoint(SgKey,Subg,Proj1,Prime1,Id,Ps),F,N,Ps,Fs),
        current_fact(ch_id(Id,Num)),
        List = [Id/Num],
        each_abs_sort(Prime1,AbsInt,Prime),
        each_extend(Sg,Prime,AbsInt,Sv,Call,Succ).
call_to_success(_RFlag,SgKey,Call,Proj,Sg,Sv,AbsInt,_ClId,Succ,List,F,N,Id):-
        current_pp_flag(variants,on),
        current_fact(complete(SgKey,AbsInt,Subg,Proj1,Prime1,_Id1,_Fs),_R),
        identical_proj_1(AbsInt,Sg,Proj,Subg,Proj1,Prime1,Prime2), !,
        format("call to success tipe _RFlag SgKey",[]),
        ( current_pp_flag(reuse_fixp_id,on) ->
            fixpoint_id_reuse_prev(SgKey,AbsInt,Sg,Proj,Id)
        ;
            fixpoint_id(Id)
        ),
        each_abs_sort(Prime2,AbsInt,Prime),
        List = [],
        asserta_fact(complete(SgKey,AbsInt,Sg,Proj,Prime,Id,[(F,N)])),
        each_extend(Sg,Prime,AbsInt,Sv,Call,Succ).
call_to_success(r,SgKey,Call,Proj,Sg,Sv,AbsInt,_ClId,Succ,List,F,N,Id) :-
        current_pp_flag(variants,on),
        current_fact(approx(SgKey,Subg,Proj1,Prime1,Id1,Fs),Ref),
        identical_proj_1(AbsInt,Sg,Proj,Subg,Proj1,Prime1,Prime2), !,
        each_abs_sort(Prime2,AbsInt,TempPrime),
        current_fact('$depend_list'(Id1,SgKey,IdList)),
        call_to_success_approx_variant(SgKey,Subg,Call,Proj,Proj1,Sg,Sv,AbsInt,F,N,Fs,
                                       Id,Id1,Ref,IdList,Prime1,TempPrime,List,Prime),
        each_extend(Sg,Prime,AbsInt,Sv,Call,Succ).
call_to_success(r,SgKey,Call,Proj,Sg,Sv,AbsInt,_ClId,Succ,List,F,N,Id):-
        current_pp_flag(variants,on),
        current_fact(fixpoint(SgKey,Subg,Proj1,Prime1,Id1,_Fs),_Ref),
        identical_proj_1(AbsInt,Sg,Proj,Subg,Proj1,Prime1,Prime2), !,
        (
            current_fact(fixpoint_variant(Id1,Id,SgKey,Sgv,Projv,Fsv),Refv),
            identical_proj(AbsInt,Sg,Proj,Sgv,Projv) ->
            patch_parents(Refv,fixpoint_variant(Id1,Id,SgKey,Sgv,Projv,Ps),F,N,Ps,Fsv)
        ;
            (
                current_pp_flag(reuse_fixp_id,on) ->
                fixpoint_id_reuse_prev(SgKey,AbsInt,Sg,Proj,Id)
            ;
                fixpoint_id(Id)
            ),
            asserta_fact(fixpoint_variant(Id1,Id,SgKey,Sg,Proj,[(F,N)]))
        ),
        each_abs_sort(Prime2,AbsInt,Prime),
        current_fact(ch_id(Id1,Num)),
        List = [Id1/Num],
        each_extend(Sg,Prime,AbsInt,Sv,Call,Succ).
call_to_success(r,SgKey,Call,Proj,Sg,Sv,AbsInt,_ClId,Succ,List,F,N,Id) :-
        init_fixpoint0(SgKey,Call,Proj,Sg,Sv,AbsInt,F,N,[(F,N)],Id,List,Prime),
        each_extend(Sg,Prime,AbsInt,Sv,Call,Succ).
call_to_success(nr,SgKey,Call,Proj,Sg,Sv,AbsInt,ClId,Succ,[],F,N,Id):-
        ( current_pp_flag(reuse_fixp_id,on) ->
            fixpoint_id_reuse_prev(SgKey,AbsInt,Sg,Proj,Id)
        ;
```

```
            fixpoint_id(Id)
        ),
        proj_to_prime_nr(SgKey,Sg,Sv,Call,Proj,AbsInt,ClId,Prime,Id),
        asserta_fact(complete(SgKey,AbsInt,Sg,Proj,Prime,Id,[(F,N)])),
        each_extend(Sg,Prime,AbsInt,Sv,Call,Succ).

call_to_success_approx(SgKey,Subg,_Call,Proj,Proj1,Sg,_Sv,_AbsInt,F,N,Fs,
                             Id,Ref,IdList,Prime1,TempPrime,List,Prime):-
        not_modified(IdList), !,
        patch_parents(Ref,approx(SgKey,Subg,Proj1,Prime1,Id,Ps),F,N,Ps,Fs),
        Prime = TempPrime,
        List = IdList.
call_to_success_approx(SgKey,_Subg,Call,Proj,_Proj1,Sg,Sv,AbsInt,F,N,Fs,
                             Id,Ref,_IdList,_Prime1,TempPrime,List,Prime):-
        erase(Ref),
        init_fixpoint_(SgKey,Call,Proj,Sg,Sv,AbsInt,F,N,Fs,Id,
                             TempPrime,List,Prime).

aproxs_to_fixpoint_variant(Id):-
        current_fact(approx_variant(Id,Idv,SgKey,Sgv,Projv,_Primev,Fs),Ref),!,
        erase(Ref),
        asserta_fact(fixpoint_variant(Id,Idv,SgKey,Sgv,Projv,Fs)),
        aproxs_to_fixpoint_variant(Id).
aproxs_to_fixpoint_variant(_).


call_to_success_approx_variant(SgKey,_Subg,_Call,Proj,_Proj1,Sg,_Sv,AbsInt,F,N,_Fs,
                             Id,Id1,_Ref,IdList,_Prime1,TempPrime,List,Prime):-
        not_modified(IdList), !,
        (
            current_fact(approx_variant(Id1,Id,SgKey,Sgv,Projv,Primev,Fsv),Refv),
            identical_proj(AbsInt,Sg,Proj,Sgv,Projv) ->
            patch_parents(Refv,approx_variant(Id1,Id,SgKey,Sgv,Projv,Primev,Ps),F,N,Ps,Fsv)
        ;
            (
                current_pp_flag(reuse_fixp_id,on) ->
                fixpoint_id_reuse_prev(SgKey,AbsInt,Sg,Proj,Id)
            ;
                fixpoint_id(Id)
            ),
            asserta_fact(approx_variant(Id1,Id,SgKey,Sg,Proj,TempPrime,[(F,N)]))
        ),
        Prime = TempPrime,
        List = IdList.
call_to_success_approx_variant(SgKey,Subg,Call,Proj,Proj1,Sg,Sv,AbsInt,F,N,Fs,
                             Id,Id1,Ref,_IdList,Prime1,_TempPrime,List,Prime):-
        (
            current_fact(approx_variant(Id1,Id,SgKey,Sgv,Projv,_Primev,Fsv),Refv),
            identical_proj(AbsInt,Sg,Proj,Sgv,Projv) ->
            erase(Refv),
            ( member((F,N),Fsv) -> NewFs = Fsv ; NewFs = [(F,N)|Fsv] %)
        ;
```

```
                (
                    current_pp_flag(reuse_fixp_id,on) ->
                    fixpoint_id_reuse_prev(SgKey,AbsInt,Sg,Proj,Id)
                ;
                    fixpoint_id(Id)
                ),
                NewFs = [(F,N)]
            ),
            aprox s_to_fixpoint_variant(Id1),
            erase(Ref),
            asserta_fact(fixpoint_variant(Id1,Id,SgKey,Sg,Proj,NewFs)),
            varset(Subg,Subv),
            init_fixpoint_(SgKey,Call,Proj1,Subg,Subv,AbsInt,F,N,Fs,Id1,
                                Prime1,List,Prime0),
            each_exit_to_prime(Prime0,AbsInt,Sg,Subv,Subg,Sv,(no,Proj),Prime).

init_fixpoint0(SgKey,Call,Proj,Sg,Sv,AbsInt,F,N,Fs,Id,List,Prime):-
        init_fixpoint2(SgKey,Call,Proj,Sg,Sv,AbsInt,F,N,Fs,Id,List,Prime).

init_fixpoint1(SgKey,_Call,Proj,Sg,_Sv,AbsInt,F,N,_Fs0,Id,List,Prime):-
        current_fact(complete(SgKey,AbsInt,Subg,Proj1,Prime1,Id,Fs),R),
        identical_proj(AbsInt,Sg,Proj,Subg,Proj1), !,
        patch_parents(R,complete(SgKey,AbsInt,Subg,Proj1,Prime1,Id,Ps),F,N,Ps,Fs),
        List = [],
        each_abs_sort(Prime1,AbsInt,Prime).
init_fixpoint1(SgKey,Call,Proj,Sg,Sv,AbsInt,F,N,_Fs0,Id,List,Prime):-
        current_fact(approx(SgKey,Subg,Proj1,Prime1,Id,Fs),Ref),
        identical_proj(AbsInt,Sg,Proj,Subg,Proj1), !,
        each_abs_sort(Prime1,AbsInt,TempPrime),
        current_fact('$depend_list'(Id,SgKey,IdList)),
        call_to_success_approx(SgKey,Subg,Call,Proj,Proj1,Sg,Sv,AbsInt,F,N,Fs,
                                Id,Ref,IdList,Prime1,TempPrime,List,Prime).
init_fixpoint1(SgKey,_,Proj,Sg,_Sv,AbsInt,F,N,_Fs0,Id,List,Prime):-
        current_fact(fixpoint(SgKey,Subg,Proj1,Prime1,Id,Fs),Ref),
        identical_proj(AbsInt,Sg,Proj,Subg,Proj1), !,
        patch_parents(Ref,fixpoint(SgKey,Subg,Proj1,Prime1,Id,Ps),F,N,Ps,Fs),
        current_fact(ch_id(Id,Num)),
        List = [Id/Num],
        each_abs_sort(Prime1,AbsInt,Prime).
init_fixpoint1(SgKey,_Call,Proj,Sg,_Sv,AbsInt,F,N,_Fs0,Id,List,Prime):-
        current_pp_flag(variants,on),
        current_fact(complete(SgKey,AbsInt,Subg,Proj1,Prime1,_Id1,_Fs),_R),
        identical_proj_1(AbsInt,Sg,Proj,Subg,Proj1,Prime1,Prime2), !,
        ( current_pp_flag(reuse_fixp_id,on) ->
            fixpoint_id_reuse_prev(SgKey,AbsInt,Sg,Proj,Id)
        ;
            fixpoint_id(Id)
        ),
        each_abs_sort(Prime2,AbsInt,Prime),
        List = [],
        asserta_fact(complete(SgKey,AbsInt,Sg,Proj,Prime,Id,[(F,N)])).
init_fixpoint1(SgKey,Call,Proj,Sg,Sv,AbsInt,F,N,_Fs0,Id,List,Prime):-
```

```
        current_pp_flag(variants,on),
        current_fact(approx(SgKey,Subg,Proj1,Prime1,Id1,Fs),Ref),
        identical_proj_1(AbsInt,Sg,Proj,Subg,Proj1,Prime1,Prime2), !,
        each_abs_sort(Prime2,AbsInt,TempPrime),
        current_fact('$depend_list'(Id1,SgKey,IdList)),
        call_to_success_approx_variant(SgKey,Subg,Call,Proj,Proj1,Sg,Sv,AbsInt,F,N,Fs,
                                       Id,Id1,Ref,IdList,Prime1,TempPrime,List,Prime).
init_fixpoint1(SgKey,_,Proj,Sg,_Sv,AbsInt,F,N,_Fs0,Id,List,Prime):-
        current_pp_flag(variants,on),
        current_fact(fixpoint(SgKey,Subg,Proj1,Prime1,Id1,_Fs),_Ref),
        identical_proj_1(AbsInt,Sg,Proj,Subg,Proj1,Prime1,Prime2), !,
        (
            current_fact(fixpoint_variant(Id1,Id,SgKey,Sgv,Projv,Fsv),Refv),
            identical_proj(AbsInt,Sg,Proj,Sgv,Projv) ->
            patch_parents(Refv,fixpoint_variant(Id1,Id,SgKey,Sgv,Projv,Ps),F,N,Ps,Fsv)
        ;
            (
                current_pp_flag(reuse_fixp_id,on) ->
                fixpoint_id_reuse_prev(SgKey,AbsInt,Sg,Proj,Id)
            ;
                fixpoint_id(Id)
            ),
            asserta_fact(fixpoint_variant(Id1,Id,SgKey,Sg,Proj,[(F,N)]))
        ),
        each_abs_sort(Prime2,AbsInt,Prime),
        current_fact(ch_id(Id1,Num)),
        List = [Id1/Num].
init_fixpoint1(SgKey,Call,Proj,Sg,Sv,AbsInt,F,N,Fs,Id,List,Prime):-
        init_fixpoint2(SgKey,Call,Proj,Sg,Sv,AbsInt,F,N,Fs,Id,List,Prime).

init_fixpoint2(SgKey,Call,Proj,Sg,Sv,AbsInt,F,N,Fs,Id,List,Prime):-
        ( current_pp_flag(reuse_fixp_id,on) ->
            fixpoint_id_reuse_prev(SgKey,AbsInt,Sg,Proj,Id)
        ;
            fixpoint_id(Id)
        ),
        asserta_fact(ch_id(Id,1)),
        proj_to_prime_r(SgKey,Sg,Sv,Call,Proj,AbsInt,TempPrime,Id),
        init_fixpoint_(SgKey,Call,Proj,Sg,Sv,AbsInt,F,N,Fs,Id,
                                     TempPrime,List,Prime).

init_fixpoint_(SgKey,Call,Proj,Sg,Sv,AbsInt,F,N,Fs,Id,Prime0,List,Prime):-
        normalize_asub0(AbsInt,Prime0,TempPrime),
        asserta_fact(fixpoint(SgKey,Sg,Proj,TempPrime,Id,Fs)),
        bagof(X, X^(trans_clause(SgKey,r,X)),Clauses),!,
        fixpoint_compute(Clauses,SgKey,Sg,Sv,Call,Proj,
                                   AbsInt,_LEntry,TempPrime,Prime1,Id,TempList),
        each_apply_trusted(Proj,SgKey,Sg,Sv,AbsInt,Prime1,Prime),
        current_fact(fixpoint(SgKey,Sg,_,_,Id,Fs2),Ref),
        erase(Ref),
        ( current_fact('$depend_list'(Id,SgKey,_),RefDep) ->
          erase(RefDep)
        ; true
        ),
        update_if_member_idlist(TempList,Id,AddList),
        ( member((F,N),Fs2) -> NewFs = Fs2 ; NewFs = [(F,N)|Fs2] ),
        decide_approx(AddList,Id,NewFs,AbsInt,SgKey,Sg,Proj,Prime),
        List = AddList.

widen_call(AbsInt,SgKey,Sg,F1,Id0,Proj1,Proj):-
        ( current_pp_flag(widencall,off) -> fail ; true ),
        widen_call0(AbsInt,SgKey,Sg,F1,Id0,[Id0],Proj1,Proj), !.

widen_call0(AbsInt,SgKey,Sg,F1,Id0,Ids,Proj1,Proj):-
        widen_call1(AbsInt,SgKey,Sg,F1,Id0,Ids,Proj1,Proj).
widen_call0(AbsInt,SgKey,Sg,F1,Id0,Ids,Proj1,Proj):-
        current_pp_flag(widencall,com_child),
        widen_call2(AbsInt,SgKey,Sg,F1,Id0,Ids,Proj1,Proj).

widen_call1(AbsInt,SgKey,Sg,F1,Id0,Ids,Proj1,Proj):-
        current_fact(fixpoint(SgKey0,Sg0,Proj0,_Prime0,Id0,Fs0)),
        ( SgKey=SgKey0,
          % same program point:
          member((F1,_NewId0),Fs0)
        -> Sg0=Sg,
           abs_sort(AbsInt,Proj0,Proj0_s),
           abs_sort(AbsInt,Proj1,Proj1_s),
           widencall(AbsInt,Proj0_s,Proj1_s,Proj)
         ; % continue with the parents:
           member((_F1,NewId0),Fs0),
           \+ member(NewId0,Ids),
           widen_call1(AbsInt,SgKey,Sg,F1,NewId0,[NewId0|Ids],Proj1,Proj)
        ).

widen_call2(AbsInt,SgKey,Sg,F1,_Id,_Ids,Proj1,Proj):-
        current_fact(complete(SgKey,AbsInt,Sg0,Proj0,_Prime0,_Id0,Fs0)),
        member((F1,Id0),Fs0),
        Sg0=Sg,
        same_fixpoint_ancestor(Id0,[Id0],AbsInt),
        abs_sort(AbsInt,Proj0,Proj0_s),
        abs_sort(AbsInt,Proj1,Proj1_s),
        widencall(AbsInt,Proj0_s,Proj1_s,Proj).

same_fixpoint_ancestor(Id0,_Ids,_AbsInt):-
        current_fact(fixpoint(_SgKey0,_Sg0,_Proj0,_Prime0,Id0,_Fs0)), !.
same_fixpoint_ancestor(Id0,_Ids,_AbsInt):-
        current_fact(approx(_SgKey0,_Sg0,_Proj0,_Prime0,Id0,_Fs0)), !.
same_fixpoint_ancestor(Id0,Ids,AbsInt):-
        current_fact(complete(_SgKey0,AbsInt,_Sg0,_Proj0,_Prime0,Id0,Fs0)),
        member((_F1,Id),Fs0),
        \+ member(Id,Ids),
        same_fixpoint_ancestor(Id,[Id|Ids],AbsInt).

fixpoint_variants_update(Id,AbsInt,Sg,Prime):-
```

```
        current_fact(fixpoint_variant(Id,Idv,SgKey,Sgv,Projv,Fs),Ref),!,
        erase(Ref),
        varset(Sg,Hv),
        varset(Sgv,Hvv),
        each_exit_to_prime(Prime,AbsInt,Sgv,Hv,Sg,Hvv,(no,Projv),Prime2),
        asserta_fact(complete(SgKey,AbsInt,Sgv,Projv,Prime2,Idv,Fs)),
        fixpoint_variants_update(Id,AbsInt,Sg,Prime).
fixpoint_variants_update(_,_,_,_).

approx_variants_update(Id,AbsInt,Sg,Prime):-
        current_fact(fixpoint_variant(Id,Idv,SgKey,Sgv,Projv,Fs),Ref),!,
        erase(Ref),
        varset(Sg,Hv),
        varset(Sgv,Hvv),
        each_exit_to_prime(Prime,AbsInt,Sgv,Hv,Sg,Hvv,(no,Projv),Prime2),
        asserta_fact(approx_variant(Id,Idv,SgKey,Sgv,Projv,Prime2,Fs)),
        approx_variants_update(Id,AbsInt,Sg,Prime).
approx_variants_update(_,_,_,_).

decide_approx([],Id,Fs,AbsInt,SgKey,Sg,Proj,Prime):- !,
        current_fact(ch_id(Id,_),Ref3),
        erase(Ref3),
        % Not needed for correctness: only book-keeping
        % update_depend_list_approx(Id,AbsInt),
        asserta_fact(complete(SgKey,AbsInt,Sg,Proj,Prime,Id,Fs)),
        (
            current_pp_flag(variants,on) ->
            each_abs_sort(Prime,AbsInt,Prime_s),
            fixpoint_variants_update(Id,AbsInt,Sg,Prime_s)
        ;
            true
        ).
decide_approx(AddList,Id,Fs,_AbsInt,SgKey,Sg,Proj,Prime):-
        asserta_fact('$depend_list'(Id,SgKey,AddList)),
        asserta_fact(approx(SgKey,Sg,Proj,Prime,Id,Fs),_),
        (
            current_pp_flag(variants,on) ->
            each_abs_sort(Prime,AbsInt,Prime_s),
            approx_variants_update(Id,AbsInt,Sg,Prime_s)
        ;
            true
        ).

not_modified([]).
not_modified([Id/N|List]):-
        current_fact(ch_id(Id,N)), !,
        not_modified(List).

proj_to_prime_nr(SgKey,Sg,Sv,Call,Proj,AbsInt,_ClId,LPrime,Id) :-
        bagof(X, X^(trans_clause(SgKey,nr,X)),Clauses), !,
        proj_to_prime(Clauses,SgKey,Sg,Sv,Call,Proj,AbsInt,LPrime1,Id),
        compute_clauses_lub(AbsInt,Proj,LPrime1,LPrime).
```

```
proj_to_prime_nr(SgKey,Sg,Sv,_Call,Proj,AbsInt,ClId,LPrime,_Id) :-
        apply_trusted0(Proj,SgKey,Sg,Sv,AbsInt,ClId,Prime), !,
        singleton(Prime,LPrime).
proj_to_prime_nr(_SgKey,Sg,Sv,Call,_Proj,AbsInt,_ClId,LSucc,_Id) :-
        % In Java programs, mode and type information is known for any method.
        % Therefore, in case of a method with unavailable code we can still
        % infer useful information.
        ( current_pp_flag(prog_lang,java) ->
          unknown_call(AbsInt,Sg,Sv,Call,Succ),
          singleton(Succ,LSucc)
        ;
          fail
        ).
proj_to_prime_nr(SgKey,_Sg,_Sv,_Call,_Proj,_AbsInt,ClId,Bot,_Id) :-
        bottom(Bot),
        inexistent(SgKey,ClId).

proj_to_prime_r(SgKey,Sg,Sv,Call,Proj,AbsInt,Prime,Id) :-
        bagof(X, X^(trans_clause(SgKey,nr,X)),Clauses), !,
        proj_to_prime(Clauses,SgKey,Sg,Sv,Call,Proj,AbsInt,Prime,Id).
proj_to_prime_r(_SgKey,_Sg,_Sv,_Call,_Proj,_AbsInt,Bot,_Id):-
        bottom(Bot).

proj_to_prime(Clauses,SgKey,Sg,Sv,Call,Proj,AbsInt,Prime,Id) :-
        proj_to_prime_loop(Clauses,Sg,Sv,Call,Proj,AbsInt,ListPrime0,Id),
        reduce_equivalent(ListPrime0,AbsInt,ListPrime1),
        each_apply_trusted(Proj,SgKey,Sg,Sv,AbsInt,ListPrime1,Prime).

proj_to_prime_loop([],_,_,_,_,_,[],_).
proj_to_prime_loop([Clause|Rest],Sg,Sv,Call,Proj,AbsInt,Primes,Id):-
        do_nr_cl(Clause,Sg,Sv,Call,Proj,AbsInt,Primes,TailPrimes,Id),!,
        proj_to_prime_loop(Rest,Sg,Sv,Call,Proj,AbsInt,TailPrimes,Id).

do_nr_cl(Clause,Sg,Sv,Call,Proj,AbsInt,Primes,TailPrimes,Id):-
        Clause = clause(Head,Vars_u,K,Body),
        clause_applies(Head,Sg), !,
        varset(Head,Hv),
        sort(Vars_u,Vars),
        ord_subtract(Vars,Hv,Fv),
        process_body(Body,K,AbsInt,Sg,Hv,Fv,Vars_u,Head,Sv,Call,
                                                        Proj,LPrime,Id),
        append_(LPrime,TailPrimes,Primes).
do_nr_cl(_Clause,_Sg,_Sv,_Call,_Proj,_AbsInt,Primes,Primes,_Id).

append_([Prime],TailPrimes,Primes):- !, Primes=[Prime|TailPrimes].
append_(LPrime,TailPrimes,Primes):- append(LPrime,TailPrimes,Primes).

process_body(Body,K,AbsInt,Sg,Hv,Fv,_,Head,Sv,Call,Proj,LPrime,Id):-
        Body = g(_,[],'$built'(_,true,_),'true/0',true), !,
        Help=(Sv,Sg,Hv,Fv,AbsInt),
        singleton(Prime,LPrime),
        call_to_success_fact(AbsInt,Sg,Hv,Head,K,Sv,Call,Proj,Prime,_Succ),
        ( current_pp_flag(fact_info,on) ->
          call_to_entry(AbsInt,Sv,Sg,Hv,Head,K,[],Prime,Exit,_),
          decide_memo(AbsInt,K,Id,no,Hv,[Exit])
        ;
          true
        ).
process_body(Body,K,AbsInt,Sg,Hv,Fv,Vars_u,Head,Sv,_,Proj,Prime,Id):-
        call_to_entry(AbsInt,Sv,Sg,Hv,Head,K,Fv,Proj,Entry,ExtraInfo),
        singleton(Entry,LEntry),
        entry_to_exit(Body,K,LEntry,Exit,[],_,Vars_u,AbsInt,Id),
        each_exit_to_prime(Exit,AbsInt,Sg,Hv,Head,Sv,ExtraInfo,Prime).

fixpoint_compute(Clauses,SgKey,Sg,Sv,Call,Proj,AbsInt,LEntryInf,
                 Prime0,Prime,Id,List) :-
        fixpoint_compute_(Clauses,SgKey,Sg,Sv,Call,Proj,AbsInt,LEntryInf,
                          Prime0,Prime1,Id,List),
        compute_clauses_lub(AbsInt,Proj,Prime1,Prime).

fixpoint_compute_(Clauses,SgKey,Sg,Sv,Call,Proj,AbsInt,LEntryInf,
                 TempPrime,Prime,Id,List) :-
        compute(Clauses,SgKey,Sg,Sv,Call,Proj,AbsInt,LEntryInf,
                 TempPrime,Prime1,Id,[],NewList,Flag),
        fixpoint(NewList,Flag,Clauses,SgKey,Sg,Sv,Call,Proj,AbsInt,LEntryInf,
                 Prime1,Prime,Id,List), !.

fixpoint([],_,_,_,_,_,_,_,_,_,Prime1,Prime,_,List):- !,
        Prime = Prime1,
        List = [].
fixpoint(NewList,Flag,_,_,_,_,_,_,_,_,Prime1,Prime,_,List):-
        var(Flag),!,
        Prime = Prime1,
        List = NewList.
fixpoint(_,_,Clauses,SgKey,Sg,Sv,Call,Proj,AbsInt,LEntryInf,Prime1,Prime,Id,List):-
        fixpoint_compute_(Clauses,SgKey,Sg,Sv,Call,Proj,AbsInt,LEntryInf,
                 Prime1,Prime,Id,List).

% some domains need normalization to perform the widening:
normalize_asub0(AbsInt,Prime0,Prime):-
        current_pp_flag(widen,on), !,
        normalize_asub(AbsInt,Prime0,Prime).
normalize_asub0(_AbsInt,Prime,Prime).

compute([],_,_,_,_,_,_,[],Prime,Prime,_,List,List,_).
compute([Clause|Rest],SgKey,Sg,Sv,Call,Proj,AbsInt,[EntryInf|LEntryInf],
                          TempPrime,Prime,Id,List,NewList,Flag) :-
        do_r_cl(Clause,SgKey,Sg,Sv,Proj,AbsInt,EntryInf,Id,List,IntList,
                                   TempPrime,NewPrime,Flag),
        compute(Rest,SgKey,Sg,Sv,Call,Proj,AbsInt,LEntryInf,NewPrime,Prime,
                                   Id,IntList,NewList,Flag).

do_r_cl(Clause,SgKey,Sg,Sv,Proj,AbsInt,EntryInf,Id,OldL,List,TempPrime,
                                                NewPrime,Flag):-
        Clause=clause(Head,Vars_u,K,Body),
        clause_applies(Head,Sg), !,
        erase_previous_memo_tables_and_parents(Body,AbsInt,K,Id),
        varset(Head,Hv),
        reuse_entry(EntryInf,Vars_u,AbsInt,Sv,Sg,Hv,Head,K,Proj,Entry,ExtraInfo),
        singleton(Entry,LEntry),
        entry_to_exit(Body,K,LEntry,Exit,OldL,List,Vars_u,AbsInt,Id),
        each_exit_to_prime(Exit,AbsInt,Sg,Hv,Head,Sv,ExtraInfo,Prime1),
        widen_succ(AbsInt,TempPrime,Prime1,NewPrime),
        decide_flag(AbsInt,TempPrime,NewPrime,SgKey,Sg,Id,Proj,Flag).

do_r_cl(_,_,_,_,_,_,_,_,List,List,Prime,Prime,_).

widen_succ_off(AbsInt,Prime0,Prime1,LPrime):-
        current_pp_flag(multi_success,on), !,
        reduce_equivalent([Prime0,Prime1],AbsInt,LPrime).
widen_succ_off(AbsInt,Prime0,Prime1,Prime):-
        singleton(P0,Prime0),
        singleton(P1,Prime1),
        singleton(P,Prime),
        compute_lub(AbsInt,[P0,P1],P).

reuse_entry(EntryInf,Vars_u,AbsInt,Sv,Sg,Hv,Head,K,Proj,Entry,ExtraInfo):-
        var(EntryInf), !,
        sort(Vars_u,Vars),
        ord_subtract(Vars,Hv,Fv),
        call_to_entry(AbsInt,Sv,Sg,Hv,Head,K,Fv,Proj,Entry,ExtraInfo),
        EntryInf = (Entry,ExtraInfo).
reuse_entry(EntryInf,_Vars_u,_AbsInt,_Sv,_Sg,_Hv,_Head,_K,_Proj,Entry,ExtraInfo):-
        EntryInf = (Entry,ExtraInfo).

decide_flag(AbsInt,TempPrime,NewPrime,_SgKey,_Sg,_Id,_Proj,_Flag):-
        abs_subset_(NewPrime,AbsInt,TempPrime), !.
decide_flag(_AbsInt,TempPrime,NewPrime,SgKey,Sg,Id,Proj,Flag):-
        Flag = notend,
        merge_(NewPrime,TempPrime,LPrime),
        current_fact(fixpoint(SgKey,Sg,_,_,Id,Fs),Ref),
        erase(Ref),
        asserta_fact(fixpoint(SgKey,Sg,Proj,LPrime,Id,Fs)),
        current_fact(ch_id(Id,Num),Ref3),
        erase(Ref3),
        Num1 is Num+1,
        asserta_fact(ch_id(Id,Num1)).

merge_([NewPrime],_TempPrime,LPrime):- !, LPrime=[NewPrime].
merge_(NewPrime,TempPrime,LPrime):-
        merge(NewPrime,TempPrime,LPrime).

%-------------------------------------------------------------------------%
% entry_to_exit(+,+,+,-,+,-,+,+,+)                                        %
%-------------------------------------------------------------------------%

```

```
entry_to_exit((Sg,Rest),K,Call,Exit,OldList,NewList,Vars_u,AbsInt,NewN):- !,
        body_succ(Call,Sg,Succ,OldList,IntList,Vars_u,AbsInt,K,NewN,_),
        entry_to_exit(Rest,K,Succ,Exit,IntList,NewList,Vars_u,AbsInt,NewN).
entry_to_exit(true,_,Call,Call,List,List,_,_,_):- !.
entry_to_exit(Sg,Key,Call,Exit,OldList,NewList,Vars_u,AbsInt,NewN):-
        body_succ(Call,Sg,Exit,OldList,NewList,Vars_u,AbsInt,Key,NewN,_),
        decide_memo(AbsInt,Key,NewN,no,Vars_u,Exit),!.

body_succ(Call,Atom,Succ,List,List,HvFv_u,AbsInt,_ClId,ParentId,no):-
        bottom(Call), !,
        Succ = Call,
        Atom=g(Key,_Av,_I,_SgKey,_Sg),
        asserta_fact(memo_table(Key,AbsInt,ParentId,no,HvFv_u,Succ)).
body_succ(Call,Atom,Succ,List,NewList,HvFv_u,AbsInt,ClId,ParentId,Id):-
        Atom=g(Key,Sv,Info,SgKey,Sg),
        body_succ_(Info,SgKey,Sg,Sv,HvFv_u,Call,Succ,List,NewList,AbsInt,
                   ClId,Key,ParentId,Id),
        decide_memo(AbsInt,Key,ParentId,Id,HvFv_u,Call).

body_succ_(Info,SgKey,Sg,Sv,HFv,Call,Succ,L,NewL,AbsInt,ClId,Key,PId,Id):-
        Info = [_|_], !,
        split_combined_domain(AbsInt,Call,Calls,Domains),
        map_body_succ(Info,SgKey,Sg,Sv,HFv,Calls,Succs,L,NewL,Domains,
                      ClId,Key,PId,Id),
        split_combined_domain(AbsInt,Succ,Succs,Domains).
body_succ_(Info,SgKey,Sg,Sv,HFv,Call,Succ,L,NewL,AbsInt,ClId,Key,PId,Id):-
        body_succ0(Info,SgKey,Sg,Sv,HFv,Call,Succ,L,NewL,AbsInt,
                   ClId,Key,PId,Id).

map_body_succ([],_SgKey,_Sg,_Sv,_HFv,[],[],L,L,[],_ClId,_Key,_PId,no).
map_body_succ([I|Info],SgKey,Sg,Sv,HFv,[Call|Calls],[Succ|Succs],L,NewL,
              [AbsInt|Domains],ClId,Key,PId,Id):-
        body_succ0(I,SgKey,Sg,Sv,HFv,Call,Succ,L,_NewL,AbsInt,
                   ClId,Key,PId,_Id), !,
        map_body_succ(Info,SgKey,Sg,Sv,HFv,Calls,Succs,L,NewL,Domains,
                   ClId,Key,PId,Id).

body_succ0('$var',SgKey,Sg,_Sv_u,HvFv_u,Calls,Succs,List0,List,AbsInt,
           ClId,F,_N,Id):-
        !,
        ( Calls=[Call],
          concrete(AbsInt,Sg,Call,Concretes),
          concretes_to_body(Concretes,SgKey,AbsInt,B)
        -> fixpoint_id(Id),
           meta_call(B,HvFv_u,Calls,[],Succs,List0,List,AbsInt,ClId,Id,Ids),
           assertz_fact(memo_call(F,Id,AbsInt,Concretes,Ids))
         ; Id=no,
           List=List0,
           variable(F,ClId),
           each_unknown_call(Calls,AbsInt,[Sg],Succs) % Sg is a variable
        ).
body_succ0('$meta'(T,B,_),SgKey,Sg,Sv_u,HvFv_u,Call,Succ,List0,List,AbsInt,
            ClId,F,N,Id):-
        !,
        ( current_pp_flag(reuse_fixp_id,on) ->
           ( Call=[C]
             -> sort(Sv_u,Sv),
                project(AbsInt,Sg,Sv,HvFv_u,C,Proj),
                fixpoint_id_reuse_prev(SgKey,AbsInt,Sg,Proj,Id)
              ; true
           )
        ;
            fixpoint_id(Id)
        ),
        meta_call(B,HvFv_u,Call,[],Exits,List0,List,AbsInt,ClId,Id,_Ids),
        ( body_succ_meta(T,AbsInt,Sv_u,HvFv_u,Call,Exits,Succ) ->
           ( Call=[C] ->
               sort(Sv_u,Sv),
               project(AbsInt,Sg,Sv,HvFv_u,C,Proj),
               each_project(Exits,AbsInt,Sg,Sv,HvFv_u,Prime),
               asserta_fact(complete(SgKey,AbsInt,Sg,Proj,Prime,Id,[(F,N)]))
             ; true
           )
        ; % for the trusts, if any:
          varset(Sg,Sv_r), % Sv_u contains extra vars (from meta-term)
                           % which will confuse apply_trusted
          body_succ0(nr,SgKey,Sg,Sv_r,HvFv_u,Call,Succ,[],_List,AbsInt,
                     ClId,F,N,Id0),
          retract_fact(complete(SgKey,AbsInt,Sg,Proj,Prime,Id0,Ps)),
          asserta_fact(complete(SgKey,AbsInt,Sg,Proj,Prime,Id,Ps))
        ).
body_succ0('$built'(T,Tg,Vs),SgKey,Sg,Sv_u,HvFv_u,Call,Succ,List0,List,AbsInt,
            _ClId,F,N,Id):-
        !,
        Id=no,
        List=List0,
        sort(Sv_u,Sv),
        each_body_succ_builtin(Call,AbsInt,T,Tg,Vs,SgKey,Sg,Sv,HvFv_u,F,N,Succ).
body_succ0(RFlag,SgKey,Sg,Sv_u,HvFv_u,Call,Succ,List0,List,AbsInt,
           ClId,F,N,Id):-
        sort(Sv_u,Sv),
        each_call_to_success(Call,RFlag,SgKey,Sg,Sv,HvFv_u,AbsInt,ClId,
                             Succ,List0,List,F,N,Id).

each_call_to_success([Call],RFlag,SgKey,Sg,Sv,HvFv_u,AbsInt,ClId,Succ,L0,L,
                     F,N,Id):-
        !,
        project(AbsInt,Sg,Sv,HvFv_u,Call,Proj),
        call_to_success(RFlag,SgKey,Call,Proj,Sg,Sv,AbsInt,ClId,Succ,L1,F,N,Id),

        merge(L1,L0,L).
each_call_to_success(LCall,RFlag,SgKey,Sg,Sv,HvFv_u,AbsInt,ClId,LSucc,L0,L,
                     F,N,Id):-
        each_call_to_success0(LCall,RFlag,SgKey,Sg,Sv,HvFv_u,AbsInt,ClId,
                                LSucc,L0,L,F,N,Id).

each_call_to_success0([],_Flag,_SgK,_Sg,_Sv,_HvFv,_AbsInt,_,[],L,L,_F,_N,_NN).
each_call_to_success0([Call|LCall],RFlag,SgKey,Sg,Sv,HvFv_u,AbsInt,ClId,
                      LSucc,L0,L,F,N,NewN):-
        project(AbsInt,Sg,Sv,HvFv_u,Call,Proj),
        call_to_success(RFlag,SgKey,Call,Proj,Sg,Sv,AbsInt,ClId,LSucc0,L1,F,N,_),
        merge(L0,L1,L2),
        append(LSucc0,LSucc1,LSucc),
        each_call_to_success0(LCall,RFlag,SgKey,Sg,Sv,HvFv_u,AbsInt,ClId,
                                LSucc1,L2,L,F,N,NewN).

meta_call([],_HvFv_u,Call,[],Call,List,List,_AbsInt,_ClId,_Id,[]).
meta_call([Body|Bodies],HvFv_u,Call,Succ0,Succ,L0,List,AbsInt,ClId,Id,Ids):-
        meta_call_([Body|Bodies],HvFv_u,Call,Succ0,Succ,L0,List,AbsInt,ClId,Id,Ids).

meta_call_([Body|Bodies],HvFv_u,Call,Succ0,Succ,L0,List,AbsInt,ClId,Id,Ids):-
        meta_call_body(Body,ClId,Call,Succ1,L0,L1,HvFv_u,AbsInt,Id,Ids0),
        widen_succ(AbsInt,Succ0,Succ1,Succ2),
        append(Succ0,Succ1,Succ2),
        append(Ids0,Ids1,Ids),
        meta_call_(Bodies,HvFv_u,Call,Succ2,Succ,L1,List,AbsInt,ClId,Id,Ids1).
meta_call_([],_HvFv_u,_Call,Succ,Succ,List,List,_AbsInt,_ClId,_Id,[]).

meta_call_body((Sg,Rest),K,Call,Exit,OldList,NewList,Vars_u,AbsInt,PId,CIds):-
        !,
        CIds=[Id|Ids],
        body_succ(Call,Sg,Succ,OldList,IntList,Vars_u,AbsInt,K,PId,Id),
        meta_call_body(Rest,K,Succ,Exit,IntList,NewList,Vars_u,AbsInt,PId,Ids).
meta_call_body(true,_,Call,Call,List,List,_,_,_,[no]):- !.
meta_call_body(Sg,Key,Call,Exit,OldList,NewList,Vars_u,AbsInt,PId,[Id]):-
        body_succ(Call,Sg,Exit,OldList,NewList,Vars_u,AbsInt,Key,PId,Id).

concretes_to_body([],_SgKey,_AbsInt,[]).
concretes_to_body([Sg|Sgs],SgKey,AbsInt,[B|Bs]):-
        body_info0(Sg:SgKey,[],AbsInt,B),
        concretes_to_body(Sgs,SgKey,AbsInt,Bs).

%------------------------------------------------------------------------%
% query(+,+,+,+,+,+,+,-)                                                 %
%------------------------------------------------------------------------%

:- doc(query(AbsInt,QKey,Query,Qv,RFlag,N,Call,Succ),
        "The success pattern of @var{Query} with @var{Call} is
         @var{Succ} in the analysis domain @var{AbsInt}. The predicate
         called is identified by @var{QKey}, and @var{RFlag} says if it
         is recursive or not. The goal @var{Query} has variables @var{Qv},
         and the call pattern is uniquely identified by @var{N}.").

query(AbsInt,QKey,Query,Qv,RFlag,N,Call,Succ) :-
        project(AbsInt,Query,Qv,Qv,Call,Proj),
        call_to_success(RFlag,QKey,Call,Proj,Query,Qv,AbsInt,0,Succ,_,N,0,Id), !,
        approx_to_completes(AbsInt).

query(_AbsInt,_QKey,_Query,_Qv,_RFlag,_N,_Call,_Succ):-
        % should never happen, but...
        error_message("SOMETHING HAS FAILED!").
```

Appendix C

s(CASP)

C.1 s(CASP) interpreter

The next figure shows a sketch of the s(CASP) interpreter's code implemented in Ciao Prolog.

```
??(Query) :-
  solve(Query,[],Mid),
  solve_goal(nmr_check,Mid,Just),
  print_just_model(Just).

solve([],In,['$success'|In]).
solve([Goal|Gs],In,Out) :-
  solve_goal(Goal,In,Mid),
  solve(Gs,Mid,Out).

solve_goal(Goal,In,Out) :-
  user_defined(Goal),!,
  check_loops(Goal,In,Out).
solve_goal(Goal,In,Out) :-
  Goal=forall(Var,G),!,
  forall(V,G,In,Out).
solve_goal(Goal,In,Out) :-
  call(Goal),
  Out=['$success',Goal|In].

check_loops(Goal,In Out) :-
  type_loop(Goal,In,Loop),
  solve_loop(Loop,Goal,In,Out).

solve_loop(odd,_,_,_) :- fail.
solve_loop(pos,_,_,_) :- fail.
solve_loop(eve,G,In,[chs(G)|In]).
solve_loop(pro,G,In,[pro(G)|In]).
solve_loop(cont,G,In,Out) :-
  pr_rule(G, Body),
  solve(Body,[G|In],Out).

forall(V,Goal,In,Out) :-
  empty_store(Store),
  eval_forall(V,Goal,[Store],In,Out).
eval_forall(_,_,[],In,In).
eval_forall(V,Goal,[Store|Sts],In,Out) :-
  copy(V,Goal, NV,NGoal),
  apply(NV, V,Store),
  solve([NGoal],In,['$success'|Out_1]),
  dump(NV, V,AnsSt),
  (  equal(AnsSt,Store)
  -> Out_2 = Out_1
  ;  dual(AnsSt,AnsDs),
     add(AnsDs,Store,NSt),
     eval_forall(V,Goal,NSt,Out_1,Out_2)
  ),
  eval_forall(V,Goal,Sts,Out_2,Out).
```

C.2 Stream Data Reasoning Example

C.2.1 s(CASP) encoding of stream.pl

The next figure shows the code of `stream.pl` with the dual program and the NMR generated by the extended compiler of s(CASP).

```
valid_stream(P,Data) :-
    stream(P,Data),
    not cancelled(P,Data).

cancelled(P,Data) :-
    higher_prio(P1,P),
    stream(P1,Data1),
    incompt(Data,Data1).

higher_prio(PHi,PLo) :-
    PHi #> PLo.

incompt(p(X),q(X)).
incompt(q(X),p(X)).

stream(1,p(X)).
stream(2,q(a)).
stream(2,q(b)).
stream(3,p(a)).

not incompt1(A,_,X) :-
    A \= p(X).
not incompt1(A,B,X) :-
    A=p(X),
    B \= q(X).

not incompt1(A,B) :-
    forall(X,not incompt1(A,B,X)).

not incompt2(A,_,X) :-
    A \= q(X).
not incompt2(A,B,X) :-
    A=q(X),
    B \= p(X).

not incompt2(A,B) :-
    forall(X,not incompt2(A,B,X)).

not incompt(A,B) :-
    not incompt1(A,B),
    not incompt2(A,B).

not higher_prio1(PHi,PLo) :-
    PHi #=< PLo.

not higher_prio(A,B) :-
    not higher_prio1(A,B).

not cancelled1(P,_,P1,_) :-
    not higher_prio(P1,P).
not cancelled1(P,_,P1,Data1) :-
    higher_prio(P1,P),
    not stream(P1,Data1).
not cancelled1(P,Data,P1,Data1) :-
    higher_prio(P1,P),
    stream(P1,Data1),
    not incompt(Data,Data1).

not cancelled1(P,Data) :-
    forall(P1,forall(Data1,not
cancelled1(P,Data,P1,Data1))).

not cancelled(A,B) :-
    not cancelled1(A,B).

not stream1(A,_,X) :-
    A \= 1.
not stream1(A,B,X) :-
    A=1,
    B \= p(X).

not stream1(A,B) :-
    forall(X,not stream1(A,B,X)).

not stream2(A,_) :-
    A \= 2.
not stream2(A,B) :-
    A=2,
    B \= q(a).

not o_stream3(A,_) :-
        A \= 2.
not o_stream3(A,B) :-
        A=2,
        B \= q(b).

not stream4(A,_) :-
    A \= 3.
not stream4(A,B) :-
    A=3,
    B \= p(a).

not stream(A,B) :-
    not stream1(A,B),
    not stream2(A,B),
    not stream3(A,B),
    not stream4(A,B).

not valid_stream1(P,Data) :-
    not stream(P,Data).
not valid_stream1(P,Data) :-
    stream(P,Data),
    cancelled(P,Data).

not valid_stream(A,B) :-
    not valid_stream1(A,B).

not false.

nmr_check.
```

C.2.2 s(CASP) output of stream.pl

The next figure shows the output for the query `?- valid_stream(Pr,Data)` when it is made to the program `stream.pl` (see C.2.1). The output to a query consists of: (i) a justification tree with the successful derivation (note that variables could be free, ground, or constrained); (ii) a model with the positive atoms defined by the program that support the successful derivation; and (iii) the bindings of variables in the query (in this example, the bindings of `Pr` and `Data`). The constraint store active at each call is shown close to each variable.

```
?- valid_stream(Pr, Data).

Answer 1        (in 18.907 ms):

valid_stream(1,p({A \= [a,b]})) :-
   stream(1,p({A \= [a,b]})),
   not cancelled(1,p({A \= [a,b]})) :-
      not o_cancelled1(1,p({A \= [a,b]})) :-
         forall(B,forall(C,not o_cancelled1(1,p({A \= [a,b]}),B,C))) :-
            forall(C,not o_cancelled1(1,p({A \= [a,b]}),{D #=< 1},C)) :-
               not o_cancelled1(1,p({A \= [a,b]}),{D #=< 1},C) :-
                  not higher_prio({D #=< 1},1) :-
                     not o_higher_prio1({D #=< 1},1) :-
                        {D #=< 1} #=< 1.
            forall(C,not o_cancelled1(1,p({A \= [a,b]}),{E #> 3},C)) :-
               not o_cancelled1(1,p({A \= [a,b]}),{E #> 3},F) :-
                  higher_prio({E #> 3},1) :-
                     {E #> 3} #> 1.
                  not stream({E #> 3},F) :-
                     not o_stream1({E #> 3},F) :-
                        forall(G,not o_stream1({E #> 3},F,G)) :-
                        not o_stream1({E #> 3},F,G) :-
                           {E #> 3} \= 1.
                  not o_stream2({E #> 3},F) :-
                     {E #> 3} \= 2.
                  not o_stream3({E #> 3},F) :-
                     {E #> 3} \= 2.
                  not o_stream4({E #> 3},F) :-
                     {E #> 3} \= 3.
         forall(C,not o_cancelled1(1,p({A \= [a,b]}),{H #> 2, H #< 3},C)) :-
            not o_cancelled1(1,p({A \= [a,b]}),{H #> 2, H #< 3},I) :-
               higher_prio({H #> 2, H #< 3},1) :-
                  {H #> 2, H #< 3} #> 1.
               not stream({H #> 2, H #< 3},I) :-
                  not o_stream1({H #> 2, H #< 3},I) :-
                     forall(J,not o_stream1({H #> 2, H #< 3},I,J)) :-
                        not o_stream1({H #> 2, H #< 3},I,J) :-
                           {H #> 2, H #< 3} \= 1.
                  not o_stream2({H #> 2, H #< 3},I) :-
                     {H #> 2, H #< 3} \= 2.
                  not o_stream3({H #> 2, H #< 3},I) :-
                     {H #> 2, H #< 3} \= 2.
                  not o_stream4({H #> 2, H #< 3},I) :-
                     {H #> 2, H #< 3} \= 3.
         forall(C,not o_cancelled1(1,p({A \= [a,b]}),{K #> 1, K #< 2},C)) :-
            not o_cancelled1(1,p({A \= [a,b]}),{K #> 1, K #< 2},L) :-
               higher_prio({K #> 1, K #< 2},1) :-
                  {K #> 1, K #< 2} #> 1.
               not stream({K #> 1, K #< 2},L) :-
                  not o_stream1({K #> 1, K #< 2},L) :-
                     forall(M,not o_stream1({K #> 1, K #< 2},L,M)) :-
                        not o_stream1({K #> 1, K #< 2},L,M) :-
                           {K #> 1, K #< 2} \= 1.
                  not o_stream2({K #> 1, K #< 2},L) :-
                     {K #> 1, K #< 2} \= 2.
                  not o_stream3({K #> 1, K #< 2},L) :-
                     {K #> 1, K #< 2} \= 2.
                  not o_stream4({K #> 1, K #< 2},L) :-
                     {K #> 1, K #< 2} \= 3.
         forall(C,not o_cancelled1(1,p({A \= [a,b]}),2,C)) :-
            not o_cancelled1(1,p({A \= [a,b]}),2,{N \= [q(a),q(b)]}) :-
               higher_prio(2,1) :-
                  2 #> 1.
               not stream(2,{N \= [q(a),q(b)]}) :-
                  not o_stream1(2,{N \= [q(a),q(b)]}) :-
                     forall(O,not o_stream1(2,{N \= [q(a),q(b)]},O)) :-
                        not o_stream1(2,{N \= [q(a),q(b)]},O) :-
                           2 \= 1.
                  not o_stream2(2,{N \= [q(a),q(b)]}) :-
                     2 = 2,
                    {N \= [q(a),q(b)]} \= q(a).
                  not o_stream3(2,{N \= [q(a),q(b)]}) :-
                     2 = 2,
                    {N \= [q(a),q(b)]} \= q(b).
                  not o_stream4(2,{N \= [q(a),q(b)]}) :-
                     2 \= 3.
            not o_cancelled1(1,p({A \= [a,b]}),2,q(a)) :-
               proved(higher_prio(2,1)),
               stream(2,q(a)),
               not incompt(p({A \= [a,b]}),q(a)) :-
                  not o_incompt1(p({A \= [a,b]}),q(a)) :-
                     forall(P,not o_incompt1(p({A \= [a,b]}),q(a),P)) :-
                        not o_incompt1(p({A \= [a,b]}),q(a),{A \= [a,b]}) :-
                           p({A \= [a,b]}) = p({A \= [a,b]}),
                           q(a) \= q({A \= [a,b]}).
                        not o_incompt1(p({A \= [a,b]}),q(a),a) :-
                           p({A \= [a,b]}) \= p(a).
                  not o_incompt2(p({A \= [a,b]}),q(a)) :-
                     forall(P,not o_incompt2(p({A \= [a,b]}),q(a),P)) :-
                        not o_incompt2(p({A \= [a,b]}),q(a),P) :-
                           p({A \= [a,b]}) \= q(P).
            not o_cancelled1(1,p({A \= [a,b]}),2,q(b)) :-
               proved(higher_prio(2,1)),
               stream(2,q(b)),
               not incompt(p({A \= [a,b]}),q(b)) :-
                  not o_incompt1(p({A \= [a,b]}),q(b)) :-
                     forall(Q,not o_incompt1(p({A \= [a,b]}),q(b),Q)) :-
                        not o_incompt1(p({A \= [a,b]}),q(b),{A \= [a,b]}) :-
                           p({A \= [a,b]}) = p({A \= [a,b]}),
                           q(b) \= q({A \= [a,b]}).
                        not o_incompt1(p({A \= [a,b]}),q(b),a) :-
                           p({A \= [a,b]}) \= p(a).
                        not o_incompt1(p({A \= [a,b]}),q(b),b) :-
                           p({A \= [a,b]}) \= p(b).
                  not o_incompt2(p({A \= [a,b]}),q(b)) :-
                     forall(R,not o_incompt2(p({A \= [a,b]}),q(b),R)) :-
                        not o_incompt2(p({A \= [a,b]}),q(b),R) :-
                           p({A \= [a,b]}) \= q(R).
         forall(C,not o_cancelled1(1,p({A \= [a,b]}),3,C)) :-
            not o_cancelled1(1,p({A \= [a,b]}),3,{S \= [p(a)]}) :-
               higher_prio(3,1) :-
                  3 #> 1.
               not stream(3,{S \= [p(a)]}) :-
```

```
                    not o_stream1(3,{S \= [p(a)]}) :-
                       forall(T,not o_stream1(3,{S \= [p(a)]},T)) :-
                          not o_stream1(3,{S \= [p(a)]},T) :-
                             3 \= 1.
                    not o_stream2(3,{S \= [p(a)]}) :-
                       3 \= 2.
                    not o_stream3(3,{S \= [p(a)]}) :-
                       3 \= 2.
                    not o_stream4(3,{S \= [p(a)]}) :-
                       3 = 3,
                      {S \= [p(a)]} \= p(a).
              not o_cancelled1(1,p({A \= [a,b]}),3,p(a)) :-
                 proved(higher_prio(3,1)),
                 stream(3,p(a)),
                 not incompt(p({A \= [a,b]}),p(a)) :-
                    not o_incompt1(p({A \= [a,b]}),p(a)) :-
                       forall(U,not o_incompt1(p({A \= [a,b]}),p(a),U)) :-
                          not o_incompt1(p({A \= [a,b]}),p(a),{A \= [a,b]}) :-
                             p({A \= [a,b]}) = p({A \= [a,b]}),
                             p(a) \= q({A \= [a,b]}).
                          not o_incompt1(p({A \= [a,b]}),p(a),a) :-
                             p({A \= [a,b]}) \= p(a).
                          not o_incompt1(p({A \= [a,b]}),p(a),b) :-
                             p({A \= [a,b]}) \= p(b).
                    not o_incompt2(p({A \= [a,b]}),p(a)) :-
                       forall(V,not o_incompt2(p({A \= [a,b]}),p(a),V)) :-
                          not o_incompt2(p({A \= [a,b]}),p(a),V) :-
                             p({A \= [a,b]}) \= q(V).
add_to_query :- o_nmr_check.

[ valid_stream(1,p({A \= [a,b]})), stream(1,p({A \= [a,b]})), higher_prio({E #> 3},1),
  higher_prio({H #> 2, H #< 3},1), higher_prio({K #> 1, K #< 2},1), higher_prio(2,1),
  stream(2,q(a)), stream(2,q(b)), higher_prio(3,1), stream(3,p(a)), o_nmr_check ]

Pr   =  1,
Data  =  p({A \= [a,b]}) ? ;

Answer 2        (in 49.191 ms):

valid_stream(2,q(b)) :-
   stream(2,q(b)),
   not cancelled(2,q(b)) :-
      not o_cancelled1(2,q(b)) :-
         forall(B,forall(C,not o_cancelled1(2,q(b),B,C))) :-
            forall(C,not o_cancelled1(2,q(b),{A #=< 2},C)) :-
              not o_cancelled1(2,q(b),{A #=< 2},D) :-
            not higher_prio({A #=< 2},2) :-
               not o_higher_prio1({A #=< 2},2) :-
                  {A #=< 2} #=< 2.
      forall(C,not o_cancelled1(2,q(b),{E #> 3},C)) :-
         not o_cancelled1(2,q(b),{E #> 3},F) :-
            higher_prio({E #> 3},2) :-
               {E #> 3} #> 2.
            not stream({E #> 3},F) :-
               not o_stream1({E #> 3},F) :-
                  forall(G,not o_stream1({E #> 3},F,G)) :-
                     not o_stream1({E #> 3},F,G) :-
                        {E #> 3} \= 1.
               not o_stream2({E #> 3},F) :-
                  {E #> 3} \= 2.
               not o_stream3({E #> 3},F) :-
                  {E #> 3} \= 2.
               not o_stream4({E #> 3},F) :-
                  {E #> 3} \= 3.
      forall(C,not o_cancelled1(2,q(b),{H #> 2, H #< 3},C)) :-
         not o_cancelled1(2,q(b),{H #> 2, H #< 3},I) :-
            higher_prio({H #> 2, H #< 3},2) :-
               {H #> 2, H #< 3} #> 2.
            not stream({H #> 2, H #< 3},I) :-
               not o_stream1({H #> 2, H #< 3},I) :-
                  forall(J,not o_stream1({H #> 2, H #< 3},I,J)) :-
                     not o_stream1({H #> 2, H #< 3},I,J) :-
                        {H #> 2, H #< 3} \= 1.
               not o_stream2({H #> 2, H #< 3},I) :-
                  {H #> 2, H #< 3} \= 2.
               not o_stream3({H #> 2, H #< 3},I) :-
                  {H #> 2, H #< 3} \= 2.
               not o_stream4({H #> 2, H #< 3},I) :-
                  {H #> 2, H #< 3} \= 3.
      forall(C,not o_cancelled1(2,q(b),3,C)) :-
         not o_cancelled1(2,q(b),3,{K \= [p(a)]}) :-
            higher_prio(3,2) :-
               3 #> 2.
            not stream(3,{K \= [p(a)]}) :-
               not o_stream1(3,{K \= [p(a)]}) :-
                  forall(L,not o_stream1(3,{K \= [p(a)]},L)) :-
                     not o_stream1(3,{K \= [p(a)]},L) :-
                        3 \= 1.
               not o_stream2(3,{K \= [p(a)]}) :-
                  3 \= 2.
               not o_stream3(3,{K \= [p(a)]}) :-
                  3 \= 2.
```

```
                              not o_stream4(3,{K \= [p(a)]}) :-
                                 3 = 3,
                                {K \= [p(a)]} \= p(a).
                        not o_cancelled1(2,q(b),3,p(a)) :-
                           proved(higher_prio(3,2)),
                           stream(3,p(a)),
                           not incompt(q(b),p(a)) :-
                              not o_incompt1(q(b),p(a)) :-
                                 forall(M,not o_incompt1(q(b),p(a),M)) :-
                                    not o_incompt1(q(b),p(a),M) :-
                                       q(b) \= p(M).
                              not o_incompt2(q(b),p(a)) :-
                                 forall(N,not o_incompt2(q(b),p(a),N)) :-
                                    not o_incompt2(q(b),p(a),{O \= [b]}) :-
                                       q(b) \= q({O \= [b]}).
                                    not o_incompt2(q(b),p(a),b) :-
                                       q(b) = q(b),
                                       p(a) \= p(b).
add_to_query :- o_nmr_check.


[ valid_stream(2,q(b)), stream(2,q(b)), higher_prio({E #> 3},2), higher_
  prio({H #> 2, H #< 3},2), higher_prio(3,2), stream(3,p(a)), o_nmr_check ]

Pr  =  2,
Data  =  q(b) ? ;

Answer 3        (in 1.606 ms):

valid_stream(3,p(a)) :-
   stream(3,p(a)),
   not cancelled(3,p(a)) :-
      not o_cancelled1(3,p(a)) :-
         forall(B,forall(C,not o_cancelled1(3,p(a),B,C))) :-
            forall(C,not o_cancelled1(3,p(a),{A #=< 3},C)) :-
               not o_cancelled1(3,p(a),{A #=< 3},D) :-
                  not higher_prio({A #=< 3},3) :-
                     not o_higher_prio1({A #=< 3},3) :-
                        {A #=< 3} #=< 3.
            forall(C,not o_cancelled1(3,p(a),{E #> 3},C)) :-
               not o_cancelled1(3,p(a),{E #> 3},F) :-
                  higher_prio({E #> 3},3) :-
                     {E #> 3} #> 3.
                  not stream({E #> 3},F) :-
                     not o_stream1({E #> 3},F) :-
                        forall(G,not o_stream1({E #> 3},F,G)) :-
```

```
                           not o_stream1({E #> 3},F,G) :-
                              {E #> 3} \= 1.
                     not o_stream2({E #> 3},F) :-
                        {E #> 3} \= 2.
                     not o_stream3({E #> 3},F) :-
                        {E #> 3} \= 2.
                     not o_stream4({E #> 3},F) :-
                        {E #> 3} \= 3.
add_to_query :- o_nmr_check.

[ valid_stream(3,p(a)), stream(3,p(a)), higher_prio({E #> 3},3), o_nmr_check ]

Pr  =  3,
Data  =  p(a) ? ;

no
```

C.3 Yale Scenario Example

C.3.1 ASP + constraint encoding of yale_shooting_asp.pl

Nest figure shows the spoiled Yale shooting scenario model written in clingo + constraints using multi-shot solving (Janhunen et al., 2017).

```
#include "incmode_lc.lp".
#program base.
action(load).
action(shoot).
action(wait).
duration(load,25).
duration(shoot,5).
duration(wait,36).
unloaded(0).
&sum { at(0) } = 0.
&sum { armed(0) } = 0.

#program step(n).
1 { do(X,n) : action(X) } 1.
&sum { at(n),-1*at(N') } = D :-
     do(X,n),
     duration(X,D),
     N' = n - 1.

loaded(n) :-
     loaded(n-1),
     not unloaded(n).
unloaded(n) :-
     unloaded(n-1),
     not loaded(n).
dead(n) :-
     dead(n-1).

&sum { armed(n) } = 0 :-
     unloaded(n-1).
&sum { armed(n),-1*armed(N') } = D :-
     do(X,n),
     duration(X,D),
     N' = n - 1,
     loaded(N').

loaded(n) :-
     do(load,n).
unloaded(n) :-
     do(shoot,n).
dead(n) :-
     do(shoot,n),
    &sum { armed(n) } <= 35.

:- do(shoot,n), unloaded(n-1).

```

```
#program check(n).
:- not dead(n), query(n).
:- not &sum {at(n)} <= 100, query(n).
:- do(shoot,n), not &sum {at(n)} > 35.
```

C.4 The Traveling Salesman Problem Example

C.4.1 ASP encoding of hamicycle_asp.pl

The next figure shows an ASP program for the Travelling Salesman Problem described in Section 6.3.3. The encoding for the Hamiltonian cycle part is from (Dovier et al., 2005) and the code of #sum is adapted to run using *clingo*. The bound on the total distance is one of the global constraints in the program.

```
1 {cycle(X,Y) : edge(X,Y)} 1 :- node(X).
1 {cycle(Z,X) : edge(Z,X)} 1 :- node(X).

reachable(X) :- node(X), cycle(b,X).
reachable(Y) :- node(X), node(Y), reachable(X), cycle(X,Y).

:- not reachable(X), node(X).

cycle_length(N) :- N = #sum {C: cycle(X,Y), distance(X, Y, C)}.
:- cycle_length(N), N >= 10.      % Cycles whose length is less than 10

edge(X,Y) :- distance(X,Y,D).
cycle_dist(U,V,D) :- cycle(U,V), distance(U,V,D).

node(a).          node(b).           node(c).            node(d).
distance(b,c,3).    %% ASP does not support rationals.
distance(c,d,8).    %% ASP does not support intervals.
distance(d,a,1).          distance(c,a,1).         distance(d,b,1).
distance(a,b,1).          distance(a,d,1).
```

C.4.2 CLP(FD) encoding of hamicycle_clpfd.pl

The next figure shows the program in CLP(*FD*) for the Hamiltonian cycle problem presented in (Dovier et al., 2005), using SICStus Prolog 3.11.2. Note that the library predicate `circuit/1` does the bulk of the work. Its implementation is non-trivial and shares a lot of code with the implementation of *all_different*, and it implicitly imposes that constraint. It does not calculate cycle lengths, but even in this (simplified) case, the code as a whole is much larger that either the ASP or s(CASP) code.

```
hamiltonian_path(Path) :-
    graph(Nodes,Edges),
    length(Nodes,N),
    length(Path,N),
    domain(Path,1,N),
    make_domains(Path,1,Edges,N),
    circuit(Path),
    labeling([ff],Path).

make_domains([],_,_,_).
make_domains([X|Y],Node,Edges,N) :-
    findall(Z,
      member([Node,Z],Edges),Succs),
    reduce_domains(N,Succs,X),
    Node1 is Node + 1,
    make_domains(Y,Node1,Edges,N).

reduce_domains(0,_,_) :- !.
reduce_domains(N,Succs,Var) :-
    N > 0,
    member(N,Succs), !,
    N1 is N - 1,
    reduce_domains(N1,Succs,Var).
reduce_domains(N,Succs,Var) :-
    Var #\= N,
    N1 is N - 1,
    reduce_domains(N1,Succs,Var).
```

C.4.3 s(CASP) output of hamicycle_scasp.pl

The next figure shows the output to the query `?- D #< 10, cycle(a,D, Cycle)` to the program `hamicycle_scasp.pl` (Figure 6.6 in Section 6.3.3).

```
?- D #< 10, travel_path(b, D, Cycle).

Answer 1        (in [2346.489] ms):

[ travel_path(b,61/10,[b,[31/10],c,[1],a,[1],d,[1],b]), path(b,b,b,61/10,[],
  [b,[31/10],c,[1],a,[1],d,[1],b]), cycle_dist(d,b,1), cycle(d,b), edge(d,b),
  distance(d,b,1), node(d), node(b), node(a), edge(d,a), distance(d,a,1),
  other(d,a), node(b), cycle(d,b), node(c), distance(d,b,1), path(b,b,d,51/10,
  [[1],b],[b,[31/10],c,[1],a,[1],d,[1],b]), cycle_dist(a,d,1), cycle(a,d),
  edge(a,d), distance(a,d,1), edge(a,b), distance(a,b,1), other(a,b), node(d),
  cycle(a,d), distance(a,d,1), path(b,b,a,41/10,[[1],d,[1],b],[b,[31/10],c,[1],
  a,[1],d,[1],b]), cycle_dist(c,a,1), cycle(c,a), edge(c,a), distance(c,a,1),
  edge(c,d), distance(c,d,{A #> 8, A #< 21rat2}), other(c,d), node(a), cycle(c,a),
  distance(c,a,1), path(b,b,c,31/10,[[1],a,[1],d,[1],b],[b,[31/10],c,[1],a,[1],
  d,[1],b]), cycle_dist(b,c,31/10), cycle(b,c), edge(b,c), distance(b,c,3.1),
  distance(b,c,31/10), o_nmr_check, reachable(a), cycle(c,a), edge(c,a),
  distance(c,a,1), reachable(b), cycle(d,b), edge(d,b), distance(d,b,1),
  reachable(d), cycle(a,d), edge(a,d), distance(a,d,1), reachable(c), cycle(b,c),
  edge(b,c), distance(b,c,3.1), other(a,a), node(d), other(a,c), node(d),
  other(b,a), node(c), other(b,b), node(c), other(b,d), node(c), other(c,b),
  node(a), other(c,c), node(a), other(d,c), node(b), other(d,d), node(b) ]

Cycle  =  [b,[31/10],c,[1],a,[1],d,[1],b],
D  =  61/10 ?
```

C.5 Towers of Hanoi Example

C.5.1 ASP encoding of toh_asp.pl

The next figure shows an ASP program for the Towers of Hanoi Problem described in Section 6.3.4. The encoding is from (Gebser et al., 2008), part of the *clingo* distribution and is available at `https://github.com/potassco/clingo/tree/master/examples/gringo/toh`.

```
% Instance
peg(a;b;c).
disk(1..7).
init_on(1..7,a).
goal_on(1..7,b).
moves(127).
% Generate
1 { move(D,P,T) : disk(D) : peg(P) } 1 :- moves(M), T = 1..M.
% Define
move(D,T) :- move(D,_,T).
on(D,P,0) :- init_on(D,P).
on(D,P,T) :- move(D,P,T).
on(D,P,T+1) :- on(D,P,T), not move(D,T+1), not moves(T).
blocked(D-1,P,T+1) :- on(D,P,T), disk(D), not moves(T).
blocked(D-1,P,T) :- blocked(D,P,T), disk(D).
% Test
:- move(D,P,T), blocked(D-1,P,T).
:- move(D,T), on(D,P,T-1), blocked(D,P,T).
:- goal_on(D,P), not on(D,P,M), moves(M).
:- not 1 { on(D,P,T) : peg(P) } 1, disk(D), moves(M), T = 1..M.
#hide.
#show move/3.
```

C.5.2 ASP incremental encoding of toh_aspI.pl

The next figure shows an ASP program to incrementally solve the Towers of Hanoi Problem described in Section 6.3.4 using the clingo's inbuild incremental solving mode. The encoding is from (Gebser et al., 2008), part of the *clingo* distribution and is available at `https://github.com/potassco/clingo/tree/master/examples/gringo/toh`.

```
#include <incmode>.

#program base.
peg(a;b;c).
disk(1..7).
init_on(1..7,a).
goal_on(1..7,b).

on(D,P,0) :- init_on(D,P).

#program step(t).
1 {move(D,P,t): disk(D),peg(P)} 1.

move(D,t) :- move(D,P,t).
```

```
on(D,P,t) :- move(D,P,t).
on(D,P,t) :- on(D,P,t-1),
             not move(D,t).
blocked(D-1,P,t) :- on(D,P,t-1).
blocked(D-1,P,t) :- blocked(D,P,t),
                    disk(D).
:- move(D,P,t), blocked(D-1,P,t).
:- move(D,t), on(D,P,t-1), blocked(D,P,t).
:- not 1 { on(D,P,t) } 1, disk(D).

#program check(t).
:- query(t), goal_on(D,P), not on(D,P,t).

#show move/3.
```

C.5.3 s(CASP) output of hanoi.pl

The next figure shows the output to the query `?- hanoi(7,T)` to the program `hanoi.pl` (Figure 6.7 in Section 6.3.4).

```
?- hanoi(7,T).

Answer 1        (in [420.343] ms):

[ hanoi(7,127), move(a,b,1), move(a,c,2), move(b,c,3), move(a,b,4),
  move(c,a,5), move(c,b,6), move(a,b,7), move(a,c,8), move(b,c,9),
  move(b,a,10), move(c,a,11), move(b,c,12), move(a,b,13), move(a,c,14),
  move(b,c,15), move(a,b,16), move(c,a,17), move(c,b,18), move(a,b,19),
  move(c,a,20), move(b,c,21), move(b,a,22), move(c,a,23), move(c,b,24),
  move(a,b,25), move(a,c,26), move(b,c,27), move(a,b,28), move(c,a,29),
  move(c,b,30), move(a,b,31), move(a,c,32), move(b,c,33), move(b,a,34),
  move(c,a,35), move(b,c,36), move(a,b,37), move(a,c,38), move(b,c,39),
  move(b,a,40), move(c,a,41), move(c,b,42), move(a,b,43), move(c,a,44),
  move(b,c,45), move(b,a,46), move(c,a,47), move(b,c,48), move(a,b,49),
  move(a,c,50), move(b,c,51), move(a,b,52), move(c,a,53), move(c,b,54),
  move(a,b,55), move(a,c,56), move(b,c,57), move(b,a,58), move(c,a,59),
  move(b,c,60), move(a,b,61), move(a,c,62), move(b,c,63), move(a,b,64),
  move(c,a,65), move(c,b,66), move(a,b,67), move(c,a,68), move(b,c,69),
  move(b,a,70), move(c,a,71), move(c,b,72), move(a,b,73), move(a,c,74),
  move(b,c,75), move(a,b,76), move(c,a,77), move(c,b,78), move(a,b,79),
  move(c,a,80), move(b,c,81), move(b,a,82), move(c,a,83), move(b,c,84),
  move(a,b,85), move(a,c,86), move(b,c,87), move(b,a,88), move(c,a,89),
  move(c,b,90), move(a,b,91), move(c,a,92), move(b,c,93), move(b,a,94),
  move(c,a,95), move(c,b,96), move(a,b,97), move(a,c,98), move(b,c,99),
  move(a,b,100), move(c,a,101), move(c,b,102), move(a,b,103), move(a,c,104),
  move(b,c,105), move(b,a,106), move(c,a,107), move(b,c,108), move(a,b,109),
  move(a,c,110), move(b,c,111), move(a,b,112), move(c,a,113), move(c,b,114),
  move(a,b,115), move(c,a,116), move(b,c,117), move(b,a,118), move(c,a,119),
  move(c,b,120), move(a,b,121), move(a,c,122), move(b,c,123), move(a,b,124),
  move(c,a,125), move(c,b,126), move(a,b,127) ]

T  =  127 ?
```

Appendix D

Event Calculus

D.1 F2LP encoding of light scenario

```
timestep(0..10).

% If a light is turned on, it will be on:
initiates(turn_on,on,T) :- timestep(T).

% If a light is turned on, whether it is red or green will be released
% from the commonsense law of inertia:
releases(turn_on,red,T) :- timestep(T).
releases(turn_on,green,T) :- timestep(T).

% If a light is turned off, it will not be on
terminates(turn_off,on,T) :- timestep(T).

% After a light is turned on, it will emit red for up to 1 second
% and green after at least 1 second
trajectory(on, T1, red, T2) :-
                     timestep(T1), timestep(T2),
                     T1 < T2, T2 < T1 + 1.
trajectory(on, T1, green, T2) :-
                     timestep(T1), timestep(T2),
                     T2 >= T1 + 1.

%% Actions
happens(turn_on,2).
happens(turn_off,4).
happens(turn_on,6).

%% Query
:- not query.
query :- holdsAt(red,_).
```

D.2 Adapted F2LP translation of light scenario with increased precision

```
timestep(0..10*P) :- precision(P).

% If a light is turned on, it will be on:
initiates(turn_on,on,T) :- timestep(T).

% If a light is turned on, whether it is red or green will be released
% from the commonsense law of inertia:
releases(turn_on,red,T) :- timestep(T).
releases(turn_on,green,T) :- timestep(T).

% If a light is turned off, it will not be on
terminates(turn_off,on,T) :- timestep(T).

% After a light is turned on, it will emit red for up to 1 second
% and green after at least 1 second
trajectory(on, T1, red, T2) :-
                     timestep(T1), timestep(T2), precision(P),
                     T1 < T2, T2 < T1 + (1*P).
trajectory(on, T1, green, T2) :-
                     timestep(T1), timestep(T2), precision(P),
                     T2 >= T1 + (1*P).

%% Actions
happens(turn_on,2*P) :- precision(P).
happens(turn_off,4*P) :- precision(P).
happens(turn_on,6*P) :- precision(P).

%% Query
:- not query.

precision(10).
query :- holdsAt(red,69).
```